Exploring C++

The Adventure Begins

Jason James

January 10, 2025

Volume I
Programming Basics

Brief Contents

Contents

List of Tables

List of Figures

Preface

The first thing to know is that this book is written in a conversational — even whimsical style at times. I'll not be formal unless the topic really calls for it.

Reader Background

I don't require too much background in this book. I expect you are an experienced user of your computer system. That you know your way around the tray/taskbar and can install a new app with the best of them.

I also expect you've had a modicum of math. Being good with algebra manipulations is really helpful to the programming mindset, you see. So the more math you've had beyond College Algebra, the better. My school requires having finished Pre-Calculus, but I think a good algebra student can handle it just fine — especially if you have dabbled at computer programming before.

Styles

There are some color and style conventions used in the book that might be helpful to know up front as well. For instance, different parts of the C++ language are colored differently. You can see a little sample in this chart:

short	*#include*	12'456
int rand()	return	"Welcome"
<iostream>	cout	'\n'

All code samples are rendered in a little box like so:

```
#include <iostream>

using namespace std;

int main()
{
    cout << "\n\t\tWelcome to C++!!!\n\n";

    return 0;
}
```

Note that copy/pasting from such boxes is fraught with peril! You'll end up with lots of crazy miscopied syntax — computer programming talk for punctuation — and have quite the time tracking it down. For longer code samples, I've posted them at the website where the book was found. But for shorter ones, most people consider it a good idea to retype them to build muscle memory and maybe remember it better consciously as well.

But beware a bit of code with a red border:

```
bad goeth here
```

Those are anti-examples — **NOT** to be emulated!

Typography

Sidebars

There are also sidebars with little extra bits of knowledge that you might be able to live without. But why would you want to do that?!

Definitions are given as they are needed but not highlighted in any special way. This highlights that all knowledge is precious — not just that found in a little rounded box! Also, watch for them everywhere — even in footnotes![1]

Also, there are links to online sites/documents. These links look like this one to the awesome website cppreference.com. It is a great place to look up things you've forgotten the details of.[2]

Exercises

I've provided no exercises in this text. There are many example codes that are complete and run just fine, but no explorations. This is because my teaching website alongside this one (craie-programming.org) is filled with programming exercises. Each is separated into semesters and kind and difficulty rating. The semesters at my school are CSC121 and CSC122. The kinds are labs for focusing 1-3 topics at a time and projects for synthesizing 3-10 topics into a cohesive whole. The difficulty is given as a Level where 1 is relatively easy at the time the material is learned and 7 is pretty darn challenging at the time the material is learned. They are great for practice even if you aren't taking my courses so feel free to try them out!

One further note about the prompts as assigned: they are not perfectly clear and tediously laid out on purpose. Part of learning to program is interacting with the prospective 'customer' of the program/application. Their initial product description is likely to be imperfect and require some amount of clarification with them — perhaps even a few rounds of it in some cases. Students of programming need to get practice with this process as well. And who better to gather requirements clarification from than their instructor!

Code Availability

As mentioned above, there are numerous code samples gone over in the text. Most are present in the text in full. A few were much longer and are linked to on the companion site (craie-programming.org/OER). The cutoff is about two pages.

I keep this split on purpose even though the codes in the text cannot generally be copy/pasted out — something about fonts for special characters like underscores and quotes — because I feel it is important to the student's memory to actually type much code for themselves at least in the first two semesters of a typical program of study. This is part of the classical 'muscle memory' tradition found in numerous studies like math, martial arts, etc.

[1] Like this one, but not actually this one. . .

[2] It seems to be where the standards committee for C++ members hang out and help maintain the wiki, so it must have the latest and greatest info, right?

Self-Study

In any significant study of material, there comes a time for self-study. And programming is no exception! Here that takes the form of realizing you have an unanswered question or concern with a topic and making a test program to clear that up or deepen your knowledge. Such programs are typically 10 or so lines long, but can be pages in later topics.[3]

It is a good idea to make such programs regularly and document them well with comments, good variable names and the like, etc. Always make the effort to make your code readable and understandable to whomever may come by it later — even if that someone is just you. Don't underestimate the worth of a good test program in reminding oneself the ways of a feature!

Viewing

I recommend a continuous scroll to keep the flow going from page to page. But it shouldn't look bad in one-page or 2-up modes, either.

Also, in that vein, since this book was produced to be a PDF and not in print, there is no provided index. You've got built-in search in your favorite PDF viewer, so hit those Control/command and F keys![4]

Finally, make sure you check regularly for updates as this is an online document and therefore subject to anytime fixes, additions, or clarifications.

Copyright and License

This work is copyright (©) Jason James but is hereby released under the Creative Commons Open License of Attribution for non-commercial uses only and a share-alike option. That is, you can use this material freely for any purpose that doesn't bring anyone profit, but you must give me credit.

I'd also like to plead with you that if you make changes to the work that you share them back to me so that I may have the chance to consider and possibly incorporate them in my own release. Please email me at 'OER at craie-programming dot org' with any suggestions. Thank you!

Acknowledgements

Here are a very few of the folks to whom I owe a great debt and whose wisdom and service and support helped me make this book you see on your screen:

- my partner Tammy
- my boys Kyle and Caleb
- my Mother
- my coworker Minhua Liu who copy-edited this work several times
- my coworker Carl Molyneaux who has taken over copy-editing after Minhua's retirement
- my students whose struggles led me to refine my craft and come to the point of writing this work[5]

[3] Of course, by then you'll be accustomed to writing such longer programs and it won't seem as daunting as right now. *smile*

[4] In case it wasn't clear, you'll need to either read this in a PDF viewer like Acrobat or Preview or look at it in your web browser.

[5] Not that I'm perfect and know all about my topic or my students. I'm just saying that without coming to understand them with respect to both topical issues and economic issues in learning, I wouldn't have reached the place where I needed

to write this book and release it as a free educational resource.

Part I
Introduction

Chapter 1

Background and Motivation

1.1 Background

1.1.1 Society and Computing

If you think you haven't been influenced by computing, where's that rock you live under? As we look around ourselves, we see computers on every lap, in every pocket, in almost every device we own. Computers make phone calls, take pictures, make toast just that perfect shade of brown, and any number of things that make each day worth living.

Car braking systems, games, Web browsers — the Web itself! All are now controlled by computers to our safety, amusement, and benefit.

To paraphrase a popular meme, if you want to have a profound impact on society, make an app for it!

1.1.2 Jobs in Computing

According to the Occupational Outlook Handbook for Computer and Information Technology put out by the U.S. Bureau of Labor Statistics, the median of computer-oriented job salaries vs median salaries overall was $91250 vs $41950 in May 2020.

But are there jobs in these fields? Yes! The demand for jobs in these fields is expected to go up 13% over the decade from 2020 to 2030. That's about 667600 new jobs. One of them could be yours!

The unemployment rate in computer fields was also around half the national rate in March 2022.[1] Further, there are still many jobs in demand in areas of information technology. So, even if unemployed — it can be only temporary until you've adjusted your tool-set!

[1] 2% vs 3.8%. This is, again, from the U.S. Bureau of Labor Statistics — just harder to ferret out.

1.1.3 Hardware vs. Software

One important thing to distinguish right away is hardware and software. This isn't difficult, but some find it a subtle issue, so let's talk to it briefly.

Hardware is the actual chips and wires and LEDs and such used in making the physical device. This stuff would do nothing without proper software installed.

Software is patterns of 1s and 0s stored in the memory of the hardware that controls the hardware and makes it do something interesting. Software comes in many forms, but that is the topic for a later book. The main thing we are interested in here is distinguishing the physical device from the electronic bits inside that make the hardware do its job.

1.1.4 CPU and OS and User Programs

The primary hardware we are interested in is the CPU or Central Processing Unit. This bit of silicon is in charge of processing instructions to make the rest of the hardware do something interesting. These instructions are loaded in an automated fashion one right after the other from the memory (or RAM — Random Access Memory) of the device until a stop or end style instruction is encountered.

The instructions get into the RAM because a particular bit of software known as the OS or Operating System put them there as part of its job. An OS is key to any hardware and you'll find them all over the place: Windows, macOS, Linux, iOS, Android, etc. One of their functions is to load other software into the RAM and set it going on the CPU when the person using the device — the user — requests it.

Such other software or user programs or applications are usually installed by with the OS or by the user at a later time. They range from web browsers, to editors, to language compilers (see below), to games, and so on.

1.1.5 Distinguishing Programming Languages

It is very helpful to decide what language to use on a particular project if we classify the programming languages available to us in certain ways. Here we present two important ways of classifying programming languages and subsequently find out how they relate to C++.

1.1.5.1 Types of Programming Languages

Traditionally, programming languages have been broken down into four overall categories:

i) procedural ii) object-oriented iii) functional iv) symbolic

Procedural programming is identified by the use of separate procedures which process the information in the program one at a time and coordinate amongst one another whose job is next.

In an object-oriented program the data is the focus of the design and guides the way actions are performed on or between data. It is hard to describe at this point, but we'll be delving into it later in the book.

Functional programming, functions call — or invoke or evaluate — one another much like the procedures in procedural programming, but there is always a value returned from every function. (In procedural programming, there are often no results or side effects from procedures.)

Symbolic (aka logic) programming is for expert systems. Such AI programs try to act as experts in a field to help diagnose problems with solutions.

In addition to these four, a new category has emerged called generic programming. In this type of programming, we see code written that works on arbitrary data rather than data of a specific type. (Again, hard to describe here, but we'll get to it later in the book.)

1.1.5.2 Compiled vs. Interpreted

A program can be run in one of two ways: via an interpreter or directly on the device's hardware. If interpreted, the interpreter that's running on the hardware is used to piece together instructions from a data file — the program — on-the-fly. So the program you are trying to run isn't running directly on the CPU in your computer. It is being translated bit-by-bit into CPU instructions and then run piece-by-piece. If run directly on the hardware, a program can be faster because it won't have to suffer this intermediary translation process as it runs. How?

To run directly on the hardware, we have to first compile the program. Compiling is an extra step between writing a program and running it, but the speed of the running program is well worth the intervening step. During this step, a program is translated from the high-level programming language into a lower-level language — C++, Java, etc. — made entirely of 1s and 0s that the CPU understands how to execute. CPUs don't deal in C++ or Java or anything like that directly, after all.

Interpreted languages are often used for developing a quick mock-up of what a program might look like with dummy routines instead of actual working code when a button is pressed or text entered. Compiled languages are used when speed is of the utmost importance. These include mission-critical things like braking systems on cars and the like.

1.2 Motivation

1.2.1 Why Programming?

Computers are built to be general-purpose these days. There are still what are known as embedded systems which are very specific to a particular task, but mostly we see phones, tablets, laptops, and even the occasional desktop. All of these things are meant to do much more than a single task — often at the same time!

While we could learn to make the hardware that makes all of this possible, that is the subject of another book altogether! We've chosen here to focus on writing programs to make that hardware do certain tasks for us: automating the drudge work that makes our lives tedious and unbearable. Or perhaps we will write a game to make the remainder of our time more enjoyable! Whatever our goal, we can make it happen with the wonderful computers the hardware engineers have made for us by learning to program.

1.2.2 Why C++?

C++ is being used in this book because it is in the top 10 languages in use in this industry no matter what metrics you use to make that judgement. It also uses the popular C language syntax owing to it being a derivative language of C. (For more on the history of C++, see Bjarne Stroustrup's fine book The Design and Evolution of C++.) Many languages mimic this syntax due to the indomitable popularity of C itself. This makes learning other languages easier by dint having already learned a C-like language.

In addition, C++ supports not only procedural programming like its C ancestor language, but also object-oriented programming. In addition, it is growing in its ability to do both generic and functional programming styles. The versatility of this language makes it ideal as a starting point. You can move on from C++ to almost any other language with ease.

As a side-benefit, C++ is also a compiled language which will give you a broader idea of not only the development cycle for software, but also an appreciation for the speed programs can run at.

1.2.3 Why Not Graphical?

A lot of students balk at using a textual interface with programming their first semester or two. They are used to graphical interfaces in all of their daily uses of technology. However, managing such systems is a tremendous chore all its own! We need to learn to use logical thinking patterns to teach a mindless hunk of silicon and plastic to perform tasks for us. This is going to be challenging enough.

In addition, many systems still in use today are non-graphical. All over industry and academia, textual interfaces are, if not popular, prevalent. Even parts of your typical graphical system are textual — input of email, patient diagnoses, descriptions of housing, articles, books, etc.

There are many other reasons to not use a graphical interface in your first semester, but we digress from our purpose here: to learn to program in C++!

1.3 Wrap Up

I think we've clarified just how impactful computers can be in the modern world and how lucrative a career in computing can be. We've also explored basic computer concepts and reasons to study not just programming but the C++ language in particular.

Chapter 2

Getting Started with C++

2.1 An Environment

Some books start with a long-winded section on setting up a C++ environment. But I've chosen to relegate that to an Appendix (A as it turns out). That's where you'll find lots of details about compilers and development environments.[1]

At any rate, this way we can start learning C++ right away and you can go and set that up when you are tired of reading and ready to start doing.

2.2 The main Program

When a program is run on a computer, several steps take place in the background. First the user clicks an icon or types a command at a prompt. This lets the operating system (OS) know that the user wants a program executed. The OS then finds the binary (executable) code for the program on the system drive and loads it up into the central processing unit (CPU) of the computer. This is where all actions that take place in the computer happen — even those of the OS itself!

What, then, does the binary code of a program look like? Well, it is just what it sounds like: a sequence of ones and zeros — binary values — which mean something to the CPU but not much of

[1] Mainly we don't want you using Word™ or Notepad for programming. Plain text is necessary, but there is a need for helping tools beyond what Notepad can offer.

anything to us. The CPU, on the other hand, finds them riveting! It executes small groups of them one after the other until your program has drawn on the screen, taken keyboard and/or mouse input, and generally run its course as to what actions it should have done.

Is this binary code what we'll be writing in this course? Of course not! We'll be writing in a language known as C++. C++ is somewhere between human language and mathematics in complexity. But nowhere near as complicated as a raw sequence of ones and zeros!

2.2.1 The Bare Minimum

What does a C++ program look like, then? The simplest C++ program consists of a single function named main. We call it the main function instead of, say, f as a math student might because it is the main point of the program. It is where all actions performed by the program are planned and plotted and laid out. Without this function, all would be for naught! What could such an important part of a program look like? It is as simple as this:

```
int main()
{
}
```

Technically, this function is missing something, but as a special case, the C++ language standard allows this particular function to leave out its last statement. Were we to include it, it would look like this:

```
int main()
{
    return 0;
}
```

It almost looks like a math function, doesn't it? With the name main instead of f and the parentheses after it like that? But what's the rest? Well, let's compare. Here's a typical math function:

$$f(x) = 3x + 4$$

The math function takes in a value — listed in its parentheses and called x by the function — and is defined to multiply that value by 3 and then add 4. The result of this calculation is the result of the function itself and is understood to be a real number because that is what math typically deals with — numbers with potential decimal places. (As opposed to integers or rational numbers or even letters or logical values. . .)

Comparing this to the main function from C++, we see that the parentheses are empty here. This means that the main function needs no value to start its job — its 'calculation'. However, since most computer programs consist of many steps to coordinate things, we like to enclose them inside a pair of curly braces (the `{}` symbols right after main's `()` and right after the semi-colon on the `return` statement). This allows us to easily visualize the extent of the function. Another visual clue here is that we indent every statement of the main a little more than the curly braces themselves. The amount can vary from program to program, but is typically between 2 and 5 (inclusive of both those numbers).

What's the `int` for? Well, unlike our math function f, the main function of C++ always returns an integer. `int` is short for integer because no-one wanted to type it all out. Why an integer, you say? It seems the OS — remember the OS that loaded the program's binary code into the CPU in the first place? — wants an integer back from the program to indicate how things were during its execution. Kind of like a hotel wants you to give it 4 stars on Yelp™ after your stay except that the OS prefers 0 to 4s. Zero?! Yes, think of it as indicating how many problems we had during our execution or a code for the most horrible problem we had during our execution. 0 is perfect in the eyes of the OS — a successful ending. And, frankly, if our program survives to that last statement, we should get to `return` that 0 to it!

That, then, is what the optional statement is all about: returning that 0 to the OS so it knows how things were while we were running on the CPU. This statement is optional because zero is the most common return value and if left off, that is assumed by the C++ standard.

Is that all there is to C++? Just those 3-4 lines of source code? (Source here to emphasize programming language code vs. binary code.) Of course not! We couldn't make the program do anything but load and stop with that! We generally at least want the program to print a nice message to us on the screen, right? Let's do that, then...

2.2.2 Printing to the User

Wait — who's this 'user'? That's us, of course! Well, you. Whoever is running the program is the user of the program. So if you clicked its icon or typed its name at a command prompt, you are the user. (Command prompt? Just hold on for a couple of paragraphs...)

This next C++ program will print a message on the screen when it is run. But not the graphics screen you are used to. It will print to a special text-only screen called a terminal or command-prompt. Here is the new code:

```
#include <iostream>

using namespace std;

int main()
{
    cout << "\n\t\tWelcome to C++!!!\n\n";

    return 0;
}
```

When we run this program in the terminal, it will print a display like this:

```
$ ./welcome.out

                Welcome to C++!!!

$ _
```

Here the dollar signs represent a typical terminal prompt on a Linux/macOS machine. On Windows a typical prompt ends in a greater than sign instead. Another difference is that on Linux/macOS we have to indicate the executable name completely (the `.out` part) and on Windows the extension can be left off since it is assumed to be `.exe`. Lastly, the Windows computer will assume nothing could go wrong and the Linux/macOS computer is always wary of viruses and Trojans getting in. The `./` in front of the program name makes sure this is less likely to occur by running the `welcome.out` relative to the current directory (folder).

Lesson learned: pay particular attention in lab to how your teacher runs a program so you can do the same when you are alone outside of class! This can vary greatly from one environment to another!

To the original program we've added many items! Let's explore each carefully.

Starting from the middle — a very good place to start! — we see the name `cout`. This is the C++ name for the text screen or terminal. The 'out' part might seem clear but why the 'c'? Is it because it is for C++? No. Oddly, it comes from an older name for the terminal. Historically the screen and keyboard would come in a single piece of hardware with no computer inside. Much bigger than a modern laptop even so, the unit was called a console. (This name has been used by many branches of computing and entertainment over the years to signify different things. But we are talking about computer history here, so...) (In particular, the console was the place where the operator or system administrator sat to enter commands, but that is too much detail for now. See `FOLDOC.org` for more on this topic.) So, `cout` is actually short for 'console output'. (Incidentally, it isn't spelled like an ACRONYM nor is it pronounced 'k-out' but rather pronounced 'see-out'.)

So what is `cout` doing here? Well, it is being used with the insertion operator (the double less-than signs taken together: <<) to insert some text onto the screen. See how the insertion operator looks vaguely like an arrow pointing the way the data is going? That's important for both remembering what symbols to use and to differentiate something later!

The text itself is enclosed in double quotes just like dialogue a person in a book would say. Inside these quotes (and yes, I said double there on purpose; we'll use single quotes for other purposes later as well and you MUST tell them apart!) we see the literal text that should be displayed on screen and some other gibberish with slash characters as well. What are those? These are called escapes. The slash — pardon me, backslash — escapes the next character's meaning and declares a special command instead. The sequence `\n` signals a new line should start here and the sequence `\t` signals that the terminal should jump to the next tab stop. Tab stops are 8 spaces wide on the terminal — not configurable, btw. This pushes our welcome message over in a sort-of pseudo-centering maneuver.

But why does the first blank line before the message take only one `\n` to produce and the blank line after the message takes two `\n` sequences to produce? Noticed that, did you? Very astute! Well, the first one is helped out by the user — you — hitting the Enter/return key at the end of the command line. That not only started the OS loading the program's binary code, but also dropped the cursor (printing position) to the beginning of the next line as well. So we only needed one `\n` to make a blank line there but two afterwards. (The first `\n` after the message dropped to the beginning of the next line but the second one dropped down again leaving the blank line before the next command prompt.)

While there are other escapes to learn, we can see them as they come up later.

So what are those other two lines we added at the top of the program source for? Well, the first grabs a standard library for inputting and outputting information on streams (Input/Output STREAMs, see?). We call this including the library and the `#` is the way C++ denotes this command. (The `#` has many names, btw: octothorpe, pound sign, hashtag, number sign, sharp sign/symbol, etc. In programming, we typically call it the pound sign and call this line a 'pound include' line.) Why the angle brackets (the less-than and greater-than), then? Well, this says that this is, indeed, a standard library rather than one the programmer has supplied. We'll see later how to write our own libraries and *`#include`* them — it's slightly different.

Programming Libraries

A library in programming terms is a collection of code that can be reused in many programs. The C++ standard comes with many libraries pre-specified and your compiler should have come with all of these so never fear!

Just a little depth on this, the `#` is the start of what's called a pre-processor directive. The compiler is actually broken into at least three phases: the pre-processor, the compiler, and the linker. The pre-processor is the part of the compiler that handles certain things like these directives and the removal of comments before the compiler proper takes over translating the C++ code into binary for an executable. The linking phase and more on the pre-processor will come back into focus later in the book (section 4.5.2).

Okay. Then what's that `using` stuff all about? That's a little more complicated and needs a little history to fully understand. In the beginning of programming there wasn't much room on computers for information. Everything had to be as small as possible to fit in. So the names of variables and functions were compacted into typically 6 or fewer letters or digits. The resulting license-plate names were nearly incomprehensible! We longed for more clarity and, as computing progressed, more room was found and names became longer and longer. Nowadays we regularly see variable and function identifiers that are 10-50 characters long. Some prefer them even longer! But C++ says names can be of an unlimited length but are significant only to some limit specified by the local system. (This is often around 250 characters, but let's not get greedy, shall we?)

So, what does that diatribe mean about the `using` line? Well, when names were small, many programmers working on the same project would come up with the same name to represent similar or even dissimilar values in a program. This conflict could cause one part of the code to change another part's values inadvertently. Such mistakes were frequent and painful to track down.

One way C++ thought to avoid such clashes of names was the `namespace`. This idea is to group names into spaces separate from one another. The first `namespace` was called `std` which is short for STanDard as it groups together all the names from the standard libraries. What the using directive is doing, then, is letting every one know that the program is going to use names that belong to the standard `namespace`. If a name is used that is unknown from this program's source, it should be looked for inside that space of names before being reported as an error. One such name is the `cout` we used to display some text.

Is this hullabaloo really necessary? Maybe, maybe not. Some programmers prefer another syntax.[2] Instead of the entire `using` directive, we could have used a simple '`std::`' in front of the `cout`. For this program, such syntax would have been shorter and at least as clear if not clearer. But for more complex programs using many standard library features at once, each being preceded by a separate `std::` would be tedious and messy.

We often choose where in a program to use the `std::` syntax versus where to use a directive syntax. Until we get to a more complete discussion of that, let's just place a single using directive to ease ourselves into programming more simply, shall we?

whew Well, that sure was a lot, let's get to more programming!

2.2.3 Programming in Style

Wait! Before we move on, however, it is important to note more about the style of the program we've written. So far, we've only made a point of indention: indenting each line inside a pair of matched curly braces a set amount of space. But there is more to basic style than indention — lots more![3]

Code Fragments

Code boxes in this section are for fragments of a program. A fragment isn't a whole program that can be run on its own — it is missing something — sometimes a lot! If you can't figure out what is missing from a fragment to make it run, please ask your teacher for help.

Style-schmyle, I hear you say. Well, if it is worth writing now, it will be worth reading later. Also, can you imagine writing hundreds of lines of C++ and trying to many months later adjust its purpose to add a feature the user now realizes they want? It is mind-wracking, at best! In between you've written many thousands of lines of code in C++, Java, Python, and many other languages. How are you going to remember what these few hundred lines of code did? Reading it is a start and good style helps with that.

Two other basic parts of style besides indention are wrapping and comments. Wrapping means taking a statement that is too long for the current line and breaking it at a logical place to take up two lines. But why not just drag the window wider and make the font smaller? That kind of trick only works for so long. And everyone on the project must be able to read this code — not just you! So we typically limit lines to 80-120 characters in length. (Ask your teacher what value is good for presenting code in your class.) When you reach your limit — often represented in the editor by a dropped guideline or just a number in the corner of the window — you should consider hitting Enter/return somewhere sensible and continuing the current line below. But if we did that willy-nilly, it would look crazy, too. So, to make

[2] Basically a way to denote something. It will typically involve punctuation.
[3] Or is it indentation? I can never figure that one out. . .

things more visually obvious, we further indent the wrapped line's continuation by some more space than the original line was indented. How much is a matter of fervent debate and will not be dictated here!

A special case of wrapping is when a length limit is reached inside double-quoted text (aka string). Most languages don't let you just hit Enter/return mid-string and pick up with further indented text on the next line. (Should that extra space and the original indention be part of the display string or not?) Instead, we must close the string with a double quote on the first line and reopen it with another double quote on the next. Make sure to break a string between words for readability and include the inter-word space within one pair of double quotes — not both! This might look like this for our earlier program:

```
cout << "\n\t\tWelcome to "
        "C++!!!\n\n";
```

Note that the space between 'to' and 'C++' is only in the first string — not both. Also note that we didn't have to add a new insertion operator for the second string. This is because such wrapped strings will be automatically joined (called concatenation) in the binary and so don't need a separate inserter (double less than signs; <<).

Lastly, then, comes comments. They are not last to make them clearly of least significance. On the contrary, I leave them last to make sure they are most likely to linger in your mind as you move on.

Comments are notes programmers place to themselves or other programmers who may read the code in the future. Such notes should help the programmer understand what the code is supposed to do in the case of errors that need to be fixed as well as how the code is supposed to be working in the case of new features needing to be added. It is quite the complex task and takes years of practice to write truly effective comments. Some people take technical writing to help hone this skill. Others take a more creative approach. However you write your comments, just make sure others will be able to understand your code when they are done.

But how can comments be placed into a C++ program? There are two basic ways: block comments and end-of-line comments. An end-of-line comment can be placed in code with a pair of slashes — yes, actual slashes this time — and the comment will then run until the end of that physical line of code:

```
int main()  // function takes no input and returns an integer
{
    cout << "\n\t\tWelcome to "    // welcoming the user to
            "C++!!!\n\n";          // their new C++ experience!

    return 0;    // 0 goes back to OS as error count/indicator
}
```

Such comments can get overused as they are so easy to put in, but better too many comments than not enough? Perhaps we could collect the comments together in a block. This can be done by beginning with a slash and an asterisk (star) and ending with the reverse:

```
/*
    This main function takes no input and returns an integer to
    the OS.  The OS takes 0 as an indication of no errors or an
    'all clear' from the program.

    This program welcomes the user to their new C++ experience!
*/
int main()
{
    cout << "\n\t\tWelcome to "
```

```
            "C++!!!\n\n";

    return 0;
}
```

These comments can go on for miles! But try not to drone like this text does. Keep it to the point and clear. Just not too terse to be understood. Leave in the details from your notes or thoughts on how this code came together so it can be re-envisioned by those needing to add features in the future. You can also leave notes on what approaches were tried and failed to work so that new programmers won't try to reinvent the wheel only to find that it has already gone wrong before.

2.3 Data Types and Input

Another key aspect people will expect from a program is interactivity. That is, they'll want to enter their data into the program and get answers back. Luckily `cout` can print any kind of data — not just strings of literal text. But what about reading in the user's data?

To input data from a user, we'll need a place to store it. To do that, the computer demands to know the type of information being stored. So we need to know what information is being read in and tell the computer that somehow. This calls for data types.

A data type is a description to the computer of a set of values, how they are stored in binary form, and how they interact with one another. We'll start with the first two and leave the last for a later section.

2.3.1 Data Types

What data types are already available on the computer? It works with several different classes of numbers, characters that come in as keystrokes, and truth values natively. We've also seen that it can store and display literal text in strings. Those are a little trickier for us to use beyond their literal form, so we'll hold off learning more about the `string` data type until later in the book.

So let's start with numbers: what kinds of numbers does the computer know how to deal with? It knows several kinds of integers and a few approximations of real numbers (the kinds with decimal parts, remember?).

2.3.1.1 Integers

The integers are distinguished by range of values storable and, in particular, whether they can carry a negative sign or not. Following is a chart of the `signed` integer types available on any computer using C++:

Type Name	Minimum	Maximum	Bit Size
`short`	-32'768	32'767	16
`int`	?	?	?
`long`	-2'147'483'648	2'147'483'647	32
`long long`	-9'223'372'036'854'775'808	9'223'372'036'854'775'807	64

Wow! Those values can get really big! Indeed, a 64-bit integer is up to 19 digits long! I've yet to wrap my head around the magnitude of this. The 32-bit integers, on the other hand, are just at 2 billion — a much more palatable quantity.

Digit Grouping

Hey, wait a minute! Why are all the commas single quotes instead? Well, if the computer sees things separate by commas, it thinks they are separate things instead of parts of a whole — always! It doesn't have much of a sense of context, you see. That is, it doesn't know the difference between a comma between numeric groups and a comma between words, for instance. So, to compensate and still allow humans to see the grouping more clearly, the C++ language made it so the computer understands single quotes between groups within the source code. (This won't help the user when inputting values to the program, but it does help programmers reading source code.)

But wait! Don't forget about the `unsigned` versions (those that disallow negative values). Here are the `unsigned` integer types:

Type Name	Minimum	Maximum	Bit Size
`unsigned short`	0	65'535	16
`unsigned int`	0	?	?
`unsigned long`	0	4'294'967'296	32
`unsigned long long`	0	18'446'744'073'709'551'616	64

Whoa! I can't believe it![4]

One thing about the charts demands more investigation. Why are `int` and `unsigned int` full of question marks? That is because they are platform dependent. That is, they depend on the particular hardware and OS combination in use on the machine. This makes them a moving target if you are developing (creating and testing) on one platform but deploying (installing and using) on another. You may be developing on a 64-bit platform, for instance and deploying to a 32-bit or even 16-bit platform. If you use `int` or its `unsigned` counterpart, the range of values that are available will change. This will make all your testing irrelevant and the user upset when certain cases fail to work that you promised would!

To avoid this, I recommend avoiding the raw `int` types and always using the `short` or `long` variants (or `long long` if you need ridiculously large values!). This will be a bit of a headache later because of another decision of the C++ standards committee, but in the long run it is well worth the trouble to meet the user's data expectations and needs.

Type Name Variants

Why 'variant'? Well, `short` is actually just a shortened form of `signed short int`, for instance. The computer knows that you want `signed` numbers — those handling negatives — by default and even defaults to a basic integer as well. Similarly for the `long` and `long long` types. Even the `unsigned` versions could have `int` specified after their names, but why bother when it is understood?[a]

[a] One of your first rules of debugging (finding mistakes): the less you type, the fewer mistakes you can make.

So which type do we use for what? Well, you have to look at your application's particular needs and decide based on the available range of values what is going to be most appropriate. For our purposes in this text, we'll mostly decide between `short` or `long` or their `unsigned` counterparts.

2.3.1.2 Decimal Numbers

What about the other numeric types? Those that approximate the real numbers? Well, those are collectively called the floating-point types. This is because of the way electrical engineers decided to store the binary forms. They allowed the decimal point to float back and forth with exponential/scientific

[4] Okay, I can, but maybe you can't. *grin*

notation and always store the value with a 0 in front of the decimal. This allows them to not store the whole part of the real number at all as it is always a 0!

So what about the rest? Well, let's look at the specifications of how the decimal part (the mantissa) and exponent are stored:

Type Name	Precise Digits	Exponent Maximum
float	9	38
double	17	308
long double	21	4932

Here the number of precise digits is how much of your data is guaranteed to calculate correctly. (See your basic physics or chemistry text for more on precision of calculations.) The exponent maximum tells what power of 10 can be stored safely without overflowing the allotted number of bits used to store each type. This exponent can be used in negative to float the decimal the other direction but with one less magnitude. (So, -37 for float, for instance.)

As you can see, long double is going to be mainly used for cosmological and quantum calculations. float might seem enough for everyday calculations, but it really isn't supported in most hardware (CPUs) these days. So we only use float in special situations like embedded systems or other special processors. That leaves double to be used in everyday calculations. That'll be our floating-point type of choice in all contexts of this book.

2.3.1.3 Characters

What about the characters I mentioned? Well, basically, we can use the char data type to store any single keystroke the user types. This will come in handy for simple queries like yes or no, gender, etc. It is also appropriate for reading much notation the user types like dollar signs, parentheses on coordinates, a letter d for dice rolling notation.[5]

The binary form of char values is known as ASCII — the American Standard Code for Information Interchange. Charts can be found online, but avoid learning lots of numeric codes for letters and such. That's for deep-geeks — not us everyday programmers. Also, it just isn't necessary as the computer knows how to do it and does it automatically. We use the actual letters and such that we want to use and they are translated on the fly.

I only tell you about the ASCII nature of our char storage for two reasons. One, the letters and digits are stored contiguously — right in a row. This makes some comparisons and 'calculations' easier than in other storage systems like EBCDIC which is used on many mainframe machines even today. Two, there are separate entries for the lower and upper case letters. This will make normal comparison of letters and words harder as, for instance, an A is different from an a inside the computer.

I said keystrokes before, but ASCII actually contains a character that cannot be typed at the keyboard as well. It is called the 'null' character and is typed '\0' in source code. This is used as a special value to signal that a particular char memory location hasn't been filled by the user yet.

For those needing more than basic English text and punctuation, there is the data type wchar_t. This type is for wide characters and can hold any data stored in the Unicode format. Sadly, the details of use of this type are beyond the scope of this document.

For some more on these topics, please see Appendix E.

2.3.1.4 Logical Values

Finally, what about those logical values? These are stored in the data type bool. It only has two values: true and false. You might think this would make it good for yes/no questions, but bool doesn't input from the user very well and most users don't think in truth values, anyway. We'll learn more about using this data type in a later chapter.

[5] More on that in section 2.3.5.2!

Now, how do we use these data types to read in the user's input? Well, first we'll have to learn to declare variables of the right data type.

2.3.2 Variables

Variables can't just be used without declaration in C++ like they can in math. In math, all variables are assumed real numbers unless context says otherwise. In C++, there are no assumptions about variable's data types. Therefore, we must learn to declare a variable of a certain type before we can learn how to input data from a user.

The format of a variable declaration is quite simple. Just start with the data type you want to declare and then follow up with one or more variable names (aka identifiers). If more than one variable is to be declared at once, we separate them with commas (as mentioned in the sidebar earlier). As with all statements in C++ (note the `using` directive and `return` statements used so far), a variable declaration isn't over until you place a semi-colon on it.

Let's try it out:

```
short deer_in_park;

long people_in_Chicago;

double gross_pay, net_pay, pay_rate, hours_worked;
char dollar_sign;
```

Here we see several things:

i) identifiers don't have to be single letters

ii) identifiers can have underscores in them

iii) identifiers are case sensitive (upper vs lower case matters!)

We don't use single letters for variable names to avoid the clashes mentioned earlier in the text when describing `namespace`s. Underscores can be used to make more descriptive names with phrases instead of single words. The case sensitivity is often a surprise to students of programming as it seems counterproductive at first. But it can really help distinguish different parts of the program if used consistently and with forethought.

If you don't like the underscores on these longer names, however, there is another popular technique called camel-case. In this technique you use a capital letter for each subsequent word after the first. Instead of separation by underscore, then, you have separation by capitalization:

```
short deerInPark;

long peopleInChicago;

double grossPay, netPay, payRate, hoursWorked;
char dollarSign;
```

We also see in these example code fragments that parts of the code which are somehow logically separate can be separated physically by blank lines. The population variables seem to have nothing to do with one another or with the pay information below and so are separated by blank lines. The dollar sign variable, on the other hand, seems to naturally fit with the pay information and so is not separated from them.

Lastly, let's look more in detail at the comma-separated list of variables storing pay information. Many programmers don't like this style and would prefer us to break the single declaration statement into several separate ones:

```
double grossPay;
double netPay;
double payRate;
double hoursWorked;
```

They point out that this makes it easy to now comment each variable with more details of its nature with end-of-line comments (using the double slashes we learned about above). As a counterpoint, I'd offer that we can do this same thing if we arrange the variables like so:

```
double grossPay,
       netPay,
       payRate,
       hoursWorked;
```

Here we have a single variable declaration spread across several physical lines of the source code. Note again that a declaration statement isn't over until the semi-colon is reached — it doesn't matter how much space is involved or even how many physical lines of the file. And we still have plenty of room for those end-of-line comments! (Also note the extra indention for the wrapped line as discussed before!)

The arguers would then say that we'd have trouble changing the data types later if new insight or knowledge led us to change just one of these variables. I'd tag back that, if our specifications gathering was worthwhile, we'd have grouped these variables together because they were intricately tied by calculations and would never change to be of different types. (This might be a bit of an advanced note at this point in your career, but it never hurts to learn something new, right? *smile*)

Before we go on, though, I should point out that variables can not only be declared, but also initialized (given a first value) at the same time. Some will tell you that you should initialize every variable to something — no matter if you know what they need to be or not.

However, I've been bitten by problems in the past with initializing the variable that controls a loop — a structure that repeats some lines of code until a certain condition is met — too soon and so I recommend to initialize before use instead of when declaring unless you really know a good initial value for that variable.

Whichever side you fall on, there are three ways to initialize a variable to a value. Here are the syntaxes[6] (formats) of how to initialize a variable:

```
short  var1 = 9;
long   var2(42'012'593L);
double var3{15.67};
```

Using an equal sign is simple, comfortable, and conventional. It is not, however, currently in vogue as the number one choice. Neither is the use of parentheses as on the second variable.

The current rage in initialization is the use of curly braces as in the third variable's case.

”Why?” you may ask. It's because it helps avoid narrowing conversions. A narrowing conversion is when you initialize one variable with more data than it can handle.

[6] Yes, that's the plural of syntax. I looked it up!

This might seem silly, but it happens from time to time when we think a variable should be one type and then change our minds later in the design. And the more such problems we can avoid in an automated fashion, the better. For instance:

```
short var = 9.2; // silently
                 // ignored
```

Here, we initialize a short integer with a decimal value. The computer will silently chop off the decimal as it just cannot fit in an integer space. If we had used brace-initialization instead, we'd have gotten a warning or even error that the data would not fit:

Literal Modifiers

Some of you will have noticed the capital L after the second variable's value. This signifies this value as a long integer to the compiler. Without it, the default type for integer values is int and our rather large value may not fit! You can also technically use a lowercase l, but this is avoided because it can be confused with a digit 1 in many fonts. The L notation also works to make a long double out of a double literal.
If we wanted to show a value was particularly unsigned, we could attach a lower or upper case U to it. Also, a float literal can be made by adding an F in lower or upper case to a double.

```
short var{9.2};    // *eek*  Bad init!
```

Another advantage to the brace-initialization pattern is that it can be used to make a default value for the variable:

```
short var{};    // var will be 0
```

Neither other syntax does this. Leaving an equal sign without a value is an error and leaving it off entirely leaves garbage bits from previously running programs in the memory for the variable — a garbage value when interpreted as our data type. With parentheses, you accidentally declare a function instead of a variable! This is quite annoying when you later try to use the variable and find that it isn't one. (More on writing functions other than main in chapter 4.)

2.3.3 Constants

Variables are nice, but they are allowed to change throughout the program. (Hence the name — vari- being a root meaning they can vary.) Sometimes we have data that shouldn't change during a run of the program. These values are, of course, called constants. They can be made in two ways depending on the situation.

Anything can be made constant with the const keyword. This marks a memory location (like a variable or function parameter) to never change during the program run. This can be handy especially for function parameters and we will use it lots after studying that chapter. For now, feel free to apply it to initialized 'variable' 'declarations'. (Those words are in quotes because the memory location will no longer be a vary-able and because initializing a memory location makes this also a definition as well as a declaration.)

We would use it like so:

```
const double PI{3.14159265};    // but see later for a better PI constant!
```

2.3.3.1 constexpr vs. const

The same rule applies for constexpr except that it doesn't work on function parameters. This is a newer keyword but works in slightly different ways. We may talk more about it later, but for now, you can mark memory locations constant with it just as with const:

```
constexpr double PI{3.14159265};    // no, this is not the better PI constant!
```

2.3.3.2 Enumerations

There is another way to make a group of constants whose values may not even matter to us. This is because they are there just to name a series of situations we need to keep track of. This set of constants is referred to as an enumeration and is coded like so:

```
enum WeekDays { THURSDAY, FRIDAY, SATURDAY, SUNDAY, MONDAY,
                TUESDAY, WEDNESDAY };
```

Here the constants values are immaterial and it is just that each one is named and taken care of that is important. (The order here is because of the fact that the computer epoch landed on a Thursday. See our discussion of time in section 2.6.1 for more.)

But, we can also use it to number things from a value we choose:

```
enum MonthNums { January = 1, February, March, April, May, June,
                 July, August, September, October, November,
                 December };
```

Here we number `January` as 1 and the rest are auto-incremented from there. This gives us nice values for printing month numbers to the user later in the program. (If we want names, we need a lot more power. Please see section 3.5.2 and section 3.8.2 for more.)

And, finally, we can use it to number everything just the way we want it:

```
enum MonthDays { Jan_days = 31, Feb_days = 28, Mar_days = 31,
                 Apr_days = 30, May_days = 31, Jun_days = 30,
                 Jul_days = 31, Aug_days = 31, Sep_days = 30,
                 Oct_days = 31, Nov_days = 30, Dec_days = 31 };
```

Here each constant is given a certain value. This is important because the values here are not sequential or even in a code-worthy pattern! So why use the `enum` method here? Why not just make a list of constant or `constexpr short`s? Well, it groups these constants together under a single place and even gives them their own data type![7]

That's right, that name following `enum` in each example is naming a new data type that has only the listed values in it. This lets us declare variables of this type and use these constants with those variables. Sometimes the compiler will even warn when other values are used with the new data type — it depends on the compiler's settings.

2.3.4 Literals

There is one more thing that goes along with constants and variables and those are literals. We've seen a few so far for integers, floating-point values, logical values, and strings. But we haven't seen all of them or any for characters! Let's look them over in the following table:

[7] It also is a little shorter in syntax and we *love* to save typing!

Type Name	Literals
short	N/A
int	5
long	5L, 5l
long long	5LL, 5ll
unsigned short	N/A
unsigned int	5u
unsigned long	5uL, 5ul
unsigned long long	5uLL, 5ull
float	5.F, 5.f
double	5., 5.
long double	5.L, 5.l
char	'5', '\n'
bool	true, false

Note that there are no short-typed literals. All plain integers in source code are assumed to be int. Placing an upper or lower case L on the integer makes it a long integer. Adding a second L/l makes it long long. Adding a U/u makes it unsigned.

For floating-point values — those with a decimal point or scientific notation, they are double by default. To make one into a float, add an F/f. To make one into a long double, add an L/l.

Other than the suffices, the details of the numeric literals are discussed in the appendix D.3. Be warned! They are a little heady and not for the weak at heart.

A char-type literal has to be in single quotes — double quotes for more characters that make a string, you see. This can include escape 'sequences' which count as a single thing. This is very different from a string literal and the two are **NOT** inter-compatible!

We'd already seen the two bool literals, but I've included them for completeness.

2.3.5 User Input

Now that we have variables, we can input data from the user!

Let's start with the name of the screen in C++. Since cout was for console output, what do you think is going to be used for reading data from the other half of the console: the keyboard? That's right! cin is our name for the keyboard in C++ and is short for console input.

To input a value into a variable, we also use a similar syntax as to output. Before we used an inserter or the insertion operator (two less-than symbols side-by-side). This showed the direction of the information flow — from the string to the screen (cout). Now we'll use what's called an extractor or the extraction operator. This is two greater-than symbols side-by-side and again shows the direction the information is flowing — from the keyboard to the variable:

```
double pay_rate;

cout << "What is your hourly rate of pay?  ";
cin >> pay_rate;
```

Here we use a variable called pay_rate to store the user's hourly rate of pay. We can tell this from the prompt and the nicely named variable. What's a prompt, you say? Well, it nudges the user with an appropriate question so they know what to type when the program pauses here. Therefore it is known as a prompt. (It doesn't have to do with timeliness, but with prodding someone into an action.) For this reason, we almost always associate an input with an output to prompt the user.

Note also that cin doesn't begin translating the user's input until they've hit Enter/return. This is the signal that the user is done with their typing and are ready for the program to continue. Until such time, they can use backspace and more typing to edit their input. The arrow keys typically don't work on the console/terminal input.

Some wonder here how the individual keystrokes typed by the user are put together into numbers for us. But not all of us are so inquisitive. So I've put the details of that in Appendix D. To those who go there: Happy investigating! See you back soon!

Sometimes we need a lot of information from the user at once. Toward this end, you can also read more than one value at a time as long as you have a separate variable for each value you want to read:

```
double pay_rate, hours_worked;

cout << "Please enter your hours worked and hourly pay rate:  ";
cin >> hours_worked >> pay_rate;
```

The user must type these values separated by some sort of space (often called whitespace). This can be the Spacebar, the Enter/return key, or the Tab key.[8] (Sometimes there are other spacing keys available, these may be used as well.)

This isn't, however, necessarily a good idea for this situation. But there are other situations that could call for it. We'll see an example of this shortly.

2.3.5.1 Input Failures

Numeric input is pretty robust, but there can be problems. The user can enter anything properly numeric: a leading plus or minus sign and a sequence of digits for both integers and floating-points, a decimal point and another sequence of digits for floating-points, and even scientific notation with E or e, a plus or minus sign, and more digits for floating-points. But, it has to have something of these and a proper combination of them. For instance the e-notation can't be first nor can there be just a plus or minus sign and nothing else. There can be no spaces inside the number. And commas are not allowed, either.

Also, nothing that isn't allowed above can be used during a numeric input: no letters, no other punctuation, no other strange symbols (all of which are actually considered punctuation by the computer). The following input sequence would result in an input failure:

```
What is your hourly rate of pay?  forty-two dollars an hour
```

So what happens when a failure occurs on `cin`? Well, `cin` stops reading and goes into a fail state. From this state, it will not rouse until told all is well again. Doing so requires us to code decision making, but that is a tale for another day (see section 3.6.1.2).

2.3.5.1.1 Not a Failure

One thing that is oddly *not* considered a failure is inputting a negative value into an `unsigned` integer. This merely causes the negative value to wrap around the circle of doom (described in section 2.4.2) to a seemingly arbitrary positive value!

2.3.5.2 char Input

The data type `char` is special during input.[9] While spacing can be put on either side of the character value, it doesn't have to be! This is because each keystroke is exactly one character and so it doesn't take more whitespace to tell the extraction operator that the input has ended. For instance, this code:

[8] Putting multiple extractions on one statement like this is called chaining them together or simply chaining. We can chain insertions on a `cout` as well.

[9] Not as special as `bool` which won't input at all, but definitely different.

```
char dollar_sign;
double price;

cout << "How much is the item you would like to purchase?  ";
cin >> dollar_sign >> price;
```

will work with space after the user's monetary unit or not:

```
How much is the item you would like to purchase?  $ 49.99
```

Works just the same as:

```
How much is the item you would like to purchase?  $49.99
```

Again, the $ being a single keystroke makes the `char` input done as soon as it finds a non-whitespace character. This also makes it ideal for most human notation as humans like to abut their notation against the data tightly:

3d6+2	3:12	2022-16-01
(3.2,-6.4)	(847)555-1234	[4, 12)

2.3.5.2.1 But What If..?

But what if the user forgets their $ on the money? That will be a problem! In the above example, for instance, if the user forgot to type a $ in front of their 49.99, >> would read the `'4'` as the `dollar_sign` variable's value and store just the 9.99 as the `price`. What?! But `'4'` isn't a dollar sign! How can this happen?!

Remember, the computer doesn't really understand human language. When we call a variable `dollar_sign`, the computer just knows that that name associates with a certain block of memory of the right data type. It has no idea what that name means to us or at all. Since it has no significance, any single keystroke that isn't whitespace is just as good as any other. The `'4'` is a single keystroke that isn't whitespace and so is read just fine as the `dollar_sign`.

Don't worry, we'll take efforts to fix this issue, but that'll have to wait until chapter 3...

2.4 Doing Calculations

Now that we can read in the user's data, let's do some calculations on it!

2.4.1 Basic Arithmetic

Arithmetic is built into the C++ language even to the point that it knows the standard order of operations:

Operation	Note
( )	parentheses
* /	multiplication and division
+ -	addition and subtraction

Operations on the same line happen at the same stage in the order seen in the expression from left to right, as usual.[10]

[10] PEMDAS is often misused such that people think multiplication always comes before division and addition before subtraction. This is totally not the case! They just happen from left to right — whichever comes first within each row of the table.

Note also that negation works as a minus sign without anything in front of it: `-x` is the same as `-1*x`.[11] The computer actually understands the difference between `x-y` and just `-y`. (Unlike your poor calculator which needs a separate key for that kind of thing...) This operation happens just before multiplication and division:

Operation	Note
( )	parentheses
-	negation
* /	multiplication and division
+ -	addition and subtraction

2.4.2 Watch Around Corners

Well, not really corners, but what you might think of as ends? Note that the integer types have minimum and maximum values set in stone. Well, what happens arithmetically when we subtract, add, or multiply past that point? Does it just stop and stay at the end? Sadly, no. It turns out that the electrical engineers that designed the system were very clever in reusing the bits of the `signed` integers to extend the `unsigned` integers' ranges. This affects how the extreme boundaries handle over-the-edge cases.

Essentially, what we view as a number line from a minimum to a maximum was bent into a circle and the minimum and maximum are neighbors now. So, when you add 1 to `32'767` in a `signed short` integer, you get `-32'768`. And when you subtract 1 from `0` in an `unsigned long` integer, you get `4'294'967'296uL`.[12]

This is so tricky that some call it the circle of doom. But that seems a bit extremist, don't you think? Just a reason to be careful of edge cases.

2.4.3 Not-So-Basic Arithmetic

Normally humans expect division to produce decimal results. This is what we learn near the end of fourth grade (at least where I grew up; maybe earlier or later for you?). But earlier that year, we learned a different technique (again, your mileage may vary): stopping at the end of the integer and stating any remainder.

So, near the beginning of 4th grade, we learned that 14 frogs divided into groups of 3 had 4 whole groups and 2 frogs left over. But later that year, we'd changed it to $4.\overline{6}$ frogs per group. (Which doesn't really make sense, now does it?)

The computer takes the former approach to dividing integers just in case it doesn't make sense — like dividing frogs into little pieces. If you want a decimal division result, you need to involve floating-point data instead or in addition to the integers. That is:

14 / 3	4
14. / 3	$4.\overline{6}$

That begs the question, how do you get the remainder? Well, they went with a symbol on your keyboard that looks like the division slash but not exactly it: %. It doesn't stand for percentages but the mathematical operation modulo. The thing on the right side of it isn't the divisor any more but the modulus. (But most teachers will be happy if you remember remainder. Ask yours just how meticulous they are on this matter.)

So, then, to get the remainder of frogs we can code:

```
short total_frogs;
```

[11] Except that the multiplication version is a good bit slower...

[12] Assuming the `long` integers are 32-bit, of course. . .

```
cin >> total_frogs;

cout << "Dividing your " << total_frogs << " frogs into groups "
        "of 3 we get " << total_frogs / 3 << "\ngroups and "
     << total_frogs % 3 << " left-over frogs.\n";
```

Modulo takes place in an expression at the same level as multiplication and division:

Operation	Note
( )	parentheses
-	negation
* / %	multiplication, division, and modulo
+ -	addition and subtraction

Note how we wrap longer lines and indent the following lines. Also see how we break the string literal without an extra inserter.

But also remember that this is a fragment missing many things in order to be run properly. It is missing *#include* and using as well as a main structure and even a prompt for the cin!

Make sure you are learning all of these things so that you can try them out on your own in your local environment! Practice makes perfect, they say...

2.4.4 A Helping Hand

As we've seen, when integers divide, they give an integer quotient. But what if we have two variables that are integers and need their decimal quotient instead? We can't just add a decimal point like we did with the 14 frogs earlier. How can we get a decimal answer if both of our dividend and divisor are integers?

Well, perhaps you've heard of a 'casting call' or an actor being 'typecast'? Good. We'll simply apply this technique to data types. We'll still call it 'type casting', but it won't be a bad thing like it is for an actor. It'll be more like the actor being cast into a part.

How do we do it and exactly what's going on here? Let's take a look:

```
short total_pizzas,
      groups;

cin >> total_pizzas;
cin >> groups;

cout << "Dividing your " << total_pizzas << " pizzas into "
     << groups << " groups we get "
     << static_cast<double>(total_pizzas) / groups
     << "\npizzas per group.\n";
```

Now, on the third line of the cout, we see the new syntax for typecasting a variable to act like a new type (just like an actor is asked to act like a new role). The 'cast' part should be clear. But what is the rest? The static part makes sure this cast happens at compile time rather than any other time. This refers to something that isn't moving like in the engineering course statics (as opposed to something that is moving like in the engineering course dynamics).

Then you just put the type you want the data to act like in angle brackets and the variable or expression you want to behave differently in parentheses. The amount of code inside the parentheses is vital! If we had brought in the division by groups, we would have had a whole number quotient after all. The reason is that the division would have been done on the integers *before* the cast to double.

This technique can also be used outside of arithmetic, so watch for it in other contexts!

2.4.5 Storing Results

Sometimes we don't just want a result printed on screen, but actually stored for later use. This might be because we need the result used multiple times. Or it might be because we need it as part of a more complicated calculation.

To do this, we use the assignment operator — a single equal sign:

```
pay_rate = 16.95;   // set hourly pay rate to the right number of $/hr
```

While this looks like the same syntax used to initialize a variable or constant, it is really a different thing to the computer. Note that we aren't declaring the type of the variable here — just setting it to a new value.

Also note that this is not an equation. It does not profess that the two sides are equal. It says to make the left-side variable take on the right-side value. The right side can be a literal value like we have here or a calculation expression. We can do things with this operation like update a variable in place:

```
count = count + 1;    // make count one more than it started
```

This doesn't say `count` is equal to itself plus one, but rather to change the `count` variable to be one more than its current value. The right side is evaluated first and then the result is stored in the left-hand variable.

Thusfar our calculations have been pretty simplistic and so we don't have an example to do this justice, but please keep it in mind as we develop more advanced programs in the future.

2.5 Program Design

Speaking of developing programs, how do we do that?

The process is much like that of doing word problems from a math text except that there are very few numbers given in the problem aside from constant values. Here is a synopsis of the basic process:

```
/*
 * Read the problem statement.  Come to understand what you
 * are asked to do.  Identify variables necessary and name
 * them.  Decide what type of information each will hold.
 * Locate/derive any necessary formulas.
 *
 * Start your code:
 *
 *     #include necessary libraries
 *     using directive
 *     inside main:
 *         declare variables              (declaration statement(s))
 *         greet user  (optional)         (cout statement)
 *         prompt the user                (cout statement) --\___(s)
 *         read inputs                    (cin statement)  --/
 *         calculate answers              (assignment statement(s))
 *         print answers (echo inputs?)   (cout statement(s))
 *         say goodbye  (optional)        (cout statement)
```

```
  */
```

As you can see, the general flow of actions is prescriptive and not subject to much change. However, the details inside each step can become quite complicated. Prompting and reading inputs, for instance, can take quite a bit of work on larger programs. Printing the answers can be pretty intricate as well, if tables or other such formatting is involved. (And what is 'echoing the inputs'? Well, that just means printing the inputs back out to the user as a sort-of verification that we understood them correctly. It doesn't really do anything useful, but it makes many users feel better.)

Note how we declare all the variables at the top of the main function — *inside* the main function, in fact. Make sure not to put any declarations outside the main function. If you do, they are called global variables instead of being local to the main function only. When we start to write more than one function in a single program, global variables would make things much more complicated and we'd like to avoid the potential problems it can cause.

Global constants, however, are acceptable because they cannot be changed. It is the changeability of the variables that makes them troublesome.

2.5.1 An Example

Let's say that a ranger station for the forest service has contacted us to make a helper program to track deer in their park. It seems they need to make predictions about how many deer are going to be in the park in spring given data from last spring and fall. They need these numbers to prepare enough hay to feed the deer over the winter and into the early spring until normal vegetation returns.

We can start by reading in these values and producing a projected growth rate. A growth rate is calculated as either a percent or a multiplicative value. We should probably use the percent for interfacing to the ranger but we'll use the multiplicative form internally for predictive calculations.

With this in mind, we come up with the following variable declarations:

```
short deer_last_fall, deer_last_spring;
double growth_rate_mult, growth_rate_pcent;
```

And we know how to calculate a rate from two population values, right? Just divide:

```
growth_rate_mult = deer_last_spring / deer_last_fall;
```

And converting this to a percent isn't too hard, either, just subtract 100% (aka 1) and multiply by 100:

```
growth_rate_pcent = (growth_rate_mult - 1) * 100;
```

Let's put this all together with input and a report:

```
#include <iostream>

using namespace std;

int main()
{
    short deer_last_fall, deer_last_spring;
    double growth_rate_mult, growth_rate_pcent;

    cout << "\n\t\tWelcome to the Deer Projection Program!\n\n";
    cout << "Please enter last fall's deer population:  ";
    cin >> deer_last_fall;

    cout << "Please enter last spring's deer population:  ";
    cin >> deer_last_spring;

    growth_rate_mult = deer_last_spring / deer_last_fall;
    growth_rate_pcent = (growth_rate_mult - 1) * 100;

    cout << "\nThe growth rate from fall to spring is typically "
         << growth_rate_pcent << "%.\n";

    return 0;
}
```

Now, this is just half the program, but we already have enough to test. Compiling and running it, we find that the result is always something like ±100%. Where are the decimals?

Looking at our calculations in more detail, we find that the multiplicative growth rate is being calculated by dividing two integers! This gives, of course, an integer answer — not a decimal. To solve this problem, we need to typecast one or the other of the populations to `double`.

Why not change their data types to `double` instead? Because we don't want parts of deer running around the park, now do we? If the data types were `double`, the user could inadvertently enter a fractional number of deer. This is not only erroneous, but kinda gross! Typecasting the calculation is definitely the way to go:

```
growth_rate_mult = static_cast<double>(deer_last_spring) / deer_last_fall;
```

With this change in place, our program runs as expected!

Now we can move on to the second phase: projections. The ranger also wanted to know how many deer there might be this coming spring given this fall's numbers — using the previous year's data as a predictor.

To do this calculation, we use our multiplicative growth rate and a new input (this fall's deer population) and get next spring's potential deer population:

```
deer_next_spring = growth_rate_mult * deer_this_fall;
```

Depending on your compiler settings, however, this might give a warning that a `double` is being stored into a `short` memory location and might lose data. This is true, so we can either take the hit or deal with the decimals somehow. Taking the hit can be done without the warning by typecasting to let the compiler know we intend to lose the data:

```
deer_next_spring = static_cast<short>(growth_rate_mult * deer_this_fall);
```

Note the scope (parentheses) of the cast is the entire product. If we had cast just the growth rate, we would have lost all decimals there before the multiplication!

Upon further reflection, however, this seems wrong. What about those partial deer still in their mommy's tummys? don't they deserve to be fed as well? Let's come back once we have the proper library support to tackle this problem.

Until then, practice your skills by taking the idea we just put forth of truncating with a typecast into the program already developed. Don't forget the new input!

Should you put it with the other inputs or after the growth rate report? This is the kind of design flow decision you'll be faced with regularly. Sometimes you have an end user to consult — like the rangers here — but not always. This time we have to rely on our own judgement. Maybe try it both ways to see what seems the most fluid to use.

2.6 Standard Libraries

We've already seen the iostream library's cout and cin objects (an object that represents a real-world entity like the screen or keyboard) and its insertion and extraction operators. But what other libraries are there in the standard and what might we use from them?

In this section we'll explore many of the standard libraries and their capabilities. But our treatment will by no means be exhaustive! If you want a complete list at sometime in the future, try out cppreference.com. They are a great source for reference as they are really thorough and the place the standards committee members seem to hang out. Although not a great place to learn things — they aren't set up for introductory education — they can quickly get you up to speed on something you are familiar with but have lost track of the details on.

2.6.1 Calculating the Time of Day

One task the user will expect of most any device is to be able to display the current time of day to them. This sounds trivial, but is actually quite a bit of work.

We'll start with the ctime library. This library's name starts with that c because it is an ancestral C language library we've inherited. Inside this library is just one thing we'll need: the time function.

The time function reports the number of seconds from a particular point in the past. Unfortunately, this point is rather esoteric: midnight on January 1, 1970. This point in time is known as the computer epoch or just epoch for short. To further complicate things, it is measured not from the local time zone but always from Greenwich, England! That is the zero meridian on the globe, after all. And due to the way computers are manufactured and distributed world-wide, it is far easier to have all of them measure time in the same way rather from their local installed time zone.

So, how do we accomodate for these things? Let's start with the epoch issue and then come back to the GMT (Greenwich Mean Time aka UTC) issue.

Note that the number of seconds since the epoch contains not just today but a large number of days before that.[13] We need to start by removing the seconds that amounted to whole days and just keep the seconds left over that form today. That is, the remaining seconds..? Yes! Modulo to the rescue!

Here's the start of it:

```
constexpr short sec_per_min  = 60,
                min_per_hour = 60,
                sec_per_hour = sec_per_min * min_per_hour,
                hrs_per_day  = 24;
constexpr long  sec_per_day  = static_cast<long>(sec_per_hour)
                               * hrs_per_day;

long sec_today = time(nullptr) % sec_per_day;
```

[13] Years and decades, in fact, but let's not dwell on it.

Wow! That's a lot of code to just get the seconds for today! What's all going on? Well, we start by setting up some constants for the calculation. The basic units are seconds, but we'll also need to know how many of those there are in terms of the other units involved: minutes, hours, and days.

Note how we don't just type in 3600 for the seconds in an hour but let the computer calculate it for us. This is important because many problems occur because of simple typographical errors. Transferring 3600 from your calculator to the source code can turn it into 360 in a flash! Over my years of teaching I've lost track of how many times I've seen this error in students' codes. Letting the computer do it saves us these headaches and makes for more readable code. We can see how the units cancel in the product taken and that makes it more easily verified.

What's happening on the seconds per day, though? Why is it `long` instead of `short` and why the `static_cast`? Well, we are just using normal arithmetic knowledge. In multiplying two two-digit numbers earlier (60 * 60), we know the result is at most four digits long. All four-digit integers fit inside a `short` integer so we are fine. But when we multiply this four-digit number by another two-digit number, we get a potentially six-digit number! That can't fit into a `short` integer and so we have to escalate to `long`.

So why the `static_cast`? Well, when we multiply two `short` integers, the computer is allowed to turn them into `int`-style integers instead. This is because `int` is the fastest type on any given CPU and the standards committee is wanting your code to run as fast as possible. Unfortunately, this means the result is also an `int` and on some systems an `int` can't hold six digits! To protect our result, we typecast one of the `short` integers into a `long` so that the product actually takes place in `long` integer space instead of `int` space. This makes sure there is room for all the digits.[14]

Finally we get to the calculation itself and find more craziness. What is this `nullptr` thing and why is it being sent to the `time` function? Well, `time` takes an argument that can be either `nullptr` or some legitimate address in the system. Since we are not set up to learn about addresses in a computer's memory system (RAM) here, we're going to use the constant `nullptr` to signal that we don't want to use that feature of the function. We just want the seconds returned.

After getting the seconds since the epoch from the `time` function, we mod-off (take a modulo) by the number of seconds in a day to find the seconds that remain after whole days are accounted for.

Now all that remains to do (pardon the pun) is to break these seconds for today into hours, minutes, and leftover seconds. Let's give that a try, shall we:

```
short hour = sec_today / sec_per_hour,
      min  = sec_today % sec_per_hour / sec_per_min,
      sec  = sec_today % sec_per_hour % sec_per_min;
```

This is a little complicated, so let's take it step-by-step. The hour isn't too bad: just divide by the seconds in an hour to get the number of whole hours. But then the minutes needs to start by modding[15] by the seconds in an hour to find out how many seconds didn't form whole hours. Once this is done, we take those seconds and count the number of whole minutes with division. The seconds takes that remaining seconds after hours are counted and mods it by the seconds in a minute to find that remainder.

But we are counting the seconds that don't form whole hours twice. Why not do that just once? Good idea! This kind of calculation is called 'caching the result'. We make a quick helper variable — not to be one of our final outputs — and store the cached result in it. This keeps the computer from redundantly calculating the result over again:[16]

[14] It turns out that this result is just five digits long, but it is too large to fit in any `short` memory space accurately, so...

[15] This is the colloquial form of ”taking a modulo with”. We could also say ”modding off by”.

[16] Although most computers these days are smart enough to just look up the previous result in what is called a register — a tiny piece of memory right on the CPU — rather than repeat the calculation, it never hurts to make sure this is possible by using a cache variable.

```
short hour         = sec_today / sec_per_hour,
      sec_not_hour = sec_today % sec_per_hour,
      min          = sec_not_hour / sec_per_min,
      sec          = sec_not_hour % sec_per_min;
```

Is this lining up of the equal signs really necessary? No. Some people find it pretty and more readable. Others say it is a waste of time to type all those spaces in. I'm letting you (or your teacher) decide, but I'll probably keep doing it this way in this book because I find it more readable and prettier myself.

Now that we've got the time of day calculated, we can print it out for the user:

```
cout << "The time is now " << hour << ':' << min << ':' << sec
     << ".\n";
```

But when we run it — sometimes — we get results like this:

```
The time is now 3:5:8.
```

This doesn't look normal or good. Where are the extra zeros we've come to expect on the minute and second fields? Since leading zeros are insignificant, the computer leaves them off. Why bother, right?

How can we make the computer understand that here the zeros are important to us? This can be done in two ways. Both depend on library help, but one is available with just `iostream` tools which we've already got `#include`d.

What we need to do in either method is to tell the computer to fill in extra space in a printing 'field' with a certain character. A field here is just a fancy name for a piece of data. The terminology comes from designing whole tables of output where each item looks like field or cell from a spreadsheet. So we need to first define a field by its width — how wide should that table column be:

```
cout << "The time is now " << hour << ':';
cout.width(2);
cout << min << ':';
cout.width(2);
cout << sec << ".\n";
```

Here we've set the minute field and the second field to both be 2 characters wide. The syntax is a bit freaky, though, isn't it? When we called[17] the `time` function, it looked almost like using a function in math: name, parentheses, inputs inside. But, of course, we used a strange constant for the input to `time`. *shrug* Here, the second part doesn't look so bad: `width(2)`. But what's with the first bit: `cout.`? Well, it is saying to the `width` function that it should also be doing its work with respect to the `cout` stream.[18] So, in general, when we want to call a special function like `width`, we have to call it (the parentheses and inputs) with respect to (the period or dot operator) an appropriate object (like `cout` here).[19] This part is read, oddly, from right to left which is different from most of the language so far. (Don't worry, there will be more right-to-left pieces later!)

But this still just prints the output like so:

```
The time is now 3: 5: 8.
```

[17] Remember, when we invoke or evaluate a function, we say we've called it. This is a phone metaphor, clearly. The function is on the other end of the phone connection and we provide inputs by telling it what we want it to work with — these are listed inside the parentheses on the call. Then, the function gives us the answer before the call is disconnected.

[18] Stream? Yes, remember that `cout` is the console output stream — hence the `iostream` library name.

[19] Further, `cout` is known as the calling object here. I.E. the object that called the function or the object with respect to which the function was called.

The default fill character is a space when data is too narrow to fill the field width for itself, you see. So we have to tell the computer we want to fill with something else — a digit 0 here:

```
cout << "The time is now " << hour << ':';
cout.fill('0');
cout.width(2);
cout << min << ':';
cout.width(2);
cout << sec << ".\n";
```

The fill has to be a character, of course — not a number and not a string — thus the single quotes and not double quotes.

Why do we call the `fill` function only once and the `width` function twice, though? The fill character is set on a once-and-forever basis whereas the width of a field changes from one field to another and so is automatically reset to 0 — just the width of the data — after every individual print or display. So once the minute has been printed, the field width resets to 0 and the colon isn't widened.

Now the result will finally be the more pleasing:

```
The time is now 3:05:08.
```

Now, in the spirit of section 2.5, let's look at that as a whole program:

```
#include <iostream>
#include <ctime>

using namespace std;

int main()
{
    constexpr short sec_per_min  = 60,
                    min_per_hour = 60,
                    sec_per_hour = sec_per_min * min_per_hour,
                    hrs_per_day  = 24;
    constexpr long  sec_per_day  = static_cast<long>(sec_per_hour)
                                   * hrs_per_day;

    long sec_today = time(nullptr) % sec_per_day;

    short hour         = static_cast<short>(sec_today / sec_per_hour),
          sec_not_hour = static_cast<short>(sec_today % sec_per_hour),
          min          = sec_not_hour / sec_per_min,
          sec          = sec_not_hour % sec_per_min;

    cout << "The time is now " << hour << ':';
    cout.width(2);
    cout.fill('0');
    cout << min << ':';
    cout.width(2);
    cout << sec << ".\n";

    return 0;
}
```

I added `static_cast`s to the `hour` and `sec_not_hour` calculations to suppress warnings about `long` to `short` conversions losing information. It is plain to see that these calculations result in small integers well in the `short` range, so this is safe. (I left them off above because it would have just complicated our discussion of the modulo and integer division. Sorry to mislead you at first. . .)

If you compile this program and run it in your local environment, it should work beautifully. Hopefully, with a couple more examples like this, you'll soon be writing your own programs from either our fragments or even from scratch!

But what about the second library? We'll come back to that shortly.

2.6.2 Beyond Simple Arithmetic

If you have need of math beyond division (and modulo!), say, exponents or trigonometry functions or the like, then you need another library: `cmath`. This is another library we've inherited from our C language ancestry. It contains *many* useful functions from different fields of mathematics. We'll explore a few of them that might prove useful in your current or upcoming courses.

Let's start with a table and then explore some issues with particular functions:

Function	Notes
`pow(base, exponent)`	raise `base` to the `exponent`th power
`sqrt(x)`	take the square root of `x`
`abs(x)`	take the absolute value of `x`
`exp(x)`	take the natural logarithm base (e) to the `x`th power
`log(x)`	take the natural logarithm of `x` (base e)
`log10(x)`	take the common logarithm of `x` (base 10)
`log2(x)`	take the logarithm of `x` (base 2; useful in computing)
`sin(x)`	find the sine of the angle `x`
`cos(x)`	find the cosine of the angle `x`
`tan(x)`	find the tangent of the angle `x`
`asin(x)`	find the angle whose sine is `x`
`acos(x)`	find the angle whose cosine is `x`
`atan(x)`	find the angle whose tangent is `x`
`atan2(y,x)`	find the angle with point (`x`,`y`) on its leading ray
`floor(x)`	give the largest integer less than or equal to `x`
`ceil(x)`	give the smallest integer greater than or equal to `x`
`round(x)`	give the nearest integer to `x`

2.6.2.1 Powers and Logarithms

Why not start the details with the first function: `pow`. This function has two inputs — not like most functions you've experienced in math so far. But they have a clear role and order. You just separate them with a comma (kind of like the declaration of multiple variables of the same type at once). Both of these numbers (in fact all the inputs to the `cmath` functions) are `double`s and so the exponent can be decimal to find roots as well as normal powers.

`sqrt(x)` is like `pow(x, .5)` but faster because it has been optimized for the $\frac{1}{2}$ power. If you really need it, there is also a specialized function for cube roots: `cbrt`.

The `abs` function can be used instead of taking the square root of the square. It isn't alone, either.

There is also an older function you may find in many example codes around the web and elsewhere: `fabs`. This function is so named because it took specifically the absolute value of floating-point numbers. But the committee has since broadened the role of absolute value to all numeric data types.[20]

The `exp` function is fairly self-explanatory. But `log` and `log10` might warrant a little detail. We're not sure why, but they've named ln — the base-e logarithm or natural logarithm — as if it were the base-10 logarithm — aka common logarithm — and then named the common logarithm all funny. It is just a detail that needs to be remembered and it will haunt you in several different languages — not just C and C++.

2.6.2.2 Trigonometry Functions

The trig functions finding the trig values are pretty straight-forward (save for a detail below). But people often are confused as to the names of the inverse trig functions. Why the a instead of an i or a -1 or some such? Isn't inverse sine just sin^{-1} now? Well, yes, but it hasn't always been. Back in the day — when C was being standardized — they were called the 'arc' functions. I.E. arcsine, arccosine, and arctangent.[21] Hence the odd a in their names.

The trig functions, of course, all deal in radians only. But your typical user is focused solely on degrees. For this conversion back and forth, we'll need a π constant. But there won't be one until C++20 makes its way into the mainstream.[22] Until then, we can use the `atan2` function to get a value for π by doing something like this:

```
const double pi = atan2(0, -1);  // the best way to create a pi constant
```

This says find the angle whose leading ray has a point at $(-1, 0)$ — an arbitrary place on the negative x axis. Just note that the x and y are reversed. This has to do with the formula for tangent and some crazy design issues with sending inputs to this function from long ago. We just have to deal with it by memorizing it. *sigh*

2.6.2.3 Rounding Numbers

Finally, let's talk rounding. `floor` is used to round a value down to the previous integer — unless it is already an integer in which case it is unchanged.

The `round` function — for rounding to the nearest integer, on the other hand, is relatively new (C++11). It works well, but before that time, we didn't have a simple function for this need. So older code will often rely on the `floor` function for this purpose.

Mathematical Notation

`floor` and `ceil` are actually regular mathematical functions as well and have interesting notations. `floor` is represented by $\lfloor x \rfloor$. See how the brackets have just lower bars like they are the 'floor' of the room. Similarly `ceil` is represented by $\lceil x \rceil$ where the brackets only have top bars like a ceiling.

The reason is that the two are nearly the same mathematically speaking. One is just the other shifted left half a step. Thus we can get a rounding to the nearest effect by taking the floor of `x` shifted half a step: `floor(x + .5)`. (Remember that shifting a function left requires adding an offset and shifting it right requires subtracting an offset!)

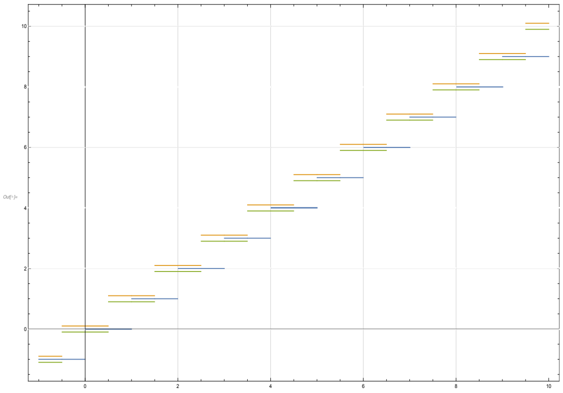
[20] Actually, there were other functions for absolute value in a separate library for just the integer types — like `labs` for `long` integers. This confusion got to be too much to teach those new to C++ so the committee made the name `abs` work for all the numeric types instead of having them all have separate names.

[21] This has to do with the arc at the end of the angle on the unit circle. If you don't get this, don't worry, trigonometry isn't required to complete this book successfully.

[22] There is the `M_PI` constant that many teach, but it isn't found in either the C++ or C standards. It just happens to be there most of the time. Not good to rely on things that aren't even supposed to be there, though. So I recommend the above method.

And if you have C++20, just `#include` the `numbers` library and a `double` version of π is given by `numbers::pi`.

Here's a plot of floor(x) (blue), round(x) (yellow), and floor(x + .5) (green):



I've offset them vertically so they didn't overlap but are instead visibly distinct. As you can see, the round and shifted floor are identical.

Finally, ceil rounds up to the next integer — unless the value is already an integer in which case it is unchanged.

None of them is set up to round to anything but a whole value — no decimal places by default. (This is in contrast to the fact that they all return a double. But some integers can't be stored in even a long so double was used before long long came about.) To round to a decimal value, we need to scale the value in question. Let's say we wanted to find the nearest $\frac{1}{3}$ to the value 3.721. We would divide out all the thirds to make this based at the ones position (as in hundreds, tens, ones digits of a number). Then, once rounded, we'll scale it back out to where the thirds are:

```
round(3.721 * 3) / 3.
```

Note that dividing by $\frac{1}{3}$ is the same as multiplying by 3.

Scaling by fractions is good, but we can reverse it to get other rounding positions as well. We could round to the nearest quarter hour (15 minutes) like this:

```
round(time / 15.) * 15
```

Note that we've put a decimal point on the first 15 to make it a double so that we get decimal places to round. time is undoubtedly an integer and so dividing by just 15 would truncate to the quotient.

So, in general, we can round down, nearest, or up to any position by:

Function Call	Notes
`floor(x / position) * position`	round down to earlier decimal position
`round(x / position) * position`	round to the nearest decimal position
`ceil(x / position) * position`	round up to next decimal position

Either the x or the position must be double for this to work, of course. If the result is expected to be an integer — and it'll fit in an actual integer data type — you can cast the result to the appropriate type:

```
quarter_min = static_cast<short>(round(min / 15.) * 15);
```

2.6.2.3.1 An Example Continued

Now that we have full rounding capabilities, let's go back and tackle those decimal deer, shall we?

We had this calculation causing us moral/ethical issues:

```
deer_next_spring = static_cast<short>(growth_rate_mult * deer_this_fall);
```

Recall that the deer variables were short and the growth rate was a double.

But this was truncating the decimal deer that might be gestating inside the female deer. But now that we have round, floor, and ceil, we can take care of it. We just have to decide which function is right for our circumstances.

The round function seems good at first glance. And we might try it out like this:

```
deer_next_spring = round(growth_rate_mult * deer_this_fall);
```

Don't forget to #include cmath!

But this gives a warning like we got before we added the static_cast! Remember that we said all these functions return a double even though they round to a whole integer. This is because some integers are too large to fit in a long and long long wasn't available when these functions were designed.[23]

So, we'll have to have the static_cast as well as the round call:

```
deer_next_spring = static_cast<short>(round(growth_rate_mult
                                            * deer_this_fall));
```

That gets rid of the warnings on those pickier systems. Is it safe given that some integers are too large to fit even in a long? Well, we are starting with a short number of deer and multiplying by something most likely in $[0, 2]$, so it should be safe enough. If we were really worried, we could change deer_next_spring to long and change the static_cast likewise. This would keep things whole-valued but make sure they are large enough even if we started with a ridiculous number of deer in the park. (I'm not thinking Yosemite here, but a smaller, local type of park.)

Anyway, testing shows this works okay, but there are circumstances where we notice that rounding down occurs. Shouldn't we be feeding those deer that are gestating and not quite ready to come out? They don't eat quite as much, but they do need sustenance!

So, we think of, perhaps, floor next. Well, that one is right out the window because it truncates to the previous integer — if we aren't already one. That definitely won't help the gestating deer babies.

That leaves ceil. This, then, seems perfect! It will round up to the next integer unless we are already an integer. That'll give all those partial deer a fighting chance! The code we end up with, then, is:

```
deer_next_spring = static_cast<short>(ceil(growth_rate_mult
                                           * deer_this_fall));
```

Again, put this change into the original program to run and test. (And don't forget to add an include for cmath for our new call to ceil...)

That's pretty much it for the cmath library. Next up is random number generation!

2.6.3 Random Values

Of course, the computer can't really generate truly random values. However, it can do a darn good job! It can fool even many statistical tests of randomness. The science of how this works is beyond the scope of this book, but if you care to study discrete mathematics or number theory more, that's where you can get into it. We call these sort-of random numbers pseudo-random numbers and their generators pseudo-random number generators (PRNG).

First, we'll need the library cstdlib. This is where we'll find the two functions and the constant we'll need to make random values in the simplest way in C++. (There is a much more complex way, but we'll hold that off until later.)

[23] Well, except for round. It was kept double for sake of symmetry, I'm guessing.

We'll start with the rand function. This function generates a pseudo-random integer between 0 — inclusive — and a constant called RAND_MAX. The constant value may or may not be included in the randomly generated values. It depends on how your library implementer interpreted the original C standard for the function. It was apparently a little fuzzy in its wording. To compensate for not knowing how this was done, we will simply mod-off by a smaller value to make sure we know what the upper bound really is.

2.6.3.1 Integers

First, let's generate random integers in a useful range to our program. We might want random dice rolls or random cards from a deck or any number of such things. We'll generally say that we want to generate an integer value between a and b.[24] We start by modding-off by the number of values in this range. That tells us how many values from 0 to some maximum we need. Then we add the value of a to shift the values to the right starting position.

But how many values are in the range $[a..b]$? That would be $b - a + 1$. This is due to a famous idea known as the fence-post problem. The number of fence segments that can be made with n fence posts is $n - 1$. It takes two posts to make a segment of fence, but one of the posts is reused in the next segment as well. But if we put up, say, 5 posts, there are only 4 segments since the end posts have no mates out to the side to hold up the fence slats.

What does that have to do with the number of values between a and b inclusive? Consider the posts numbered from 1 to n. We just subtracted the one from the other because only one end post was being included. The other one had no following post to connect to. So, when we have a half-inclusive range — $[a..b)$ or $(a..b]$ — we need just subtract the beginning from the end to count the number of discrete values in the range. Note that when we just subtract the values from one another — $b - a$ — you remove not only the values before a, but also the value a itself from the count. (5-3 removes 3 values — not just 1 and 2.) So, if we are being inclusive of the end a, we have to add it back in with a $+1$.[25]

So, all-in-all, we have this code to generate an integer value between a lower-bound a and an upper-bound b inclusive of both ends:

```
rand() % (b - a + 1) + a
```

Note that the rand function itself has no inputs but still needs parentheses next to its name to be called. This pattern will work whether the values are positive, negative, or even 0.

2.6.3.2 Character — ASCII — Values

To generate random letters or punctuation, we'll need to generate random ASCII codes. Since the codes themselves are integers, we'll just use our prior formula. But, since we won't know the codes directly, we'll use a little typecasting magic to get the computer to tell them to us!

```
static_cast<char>(rand() % (static_cast<short>(b)
                            - static_cast<short>(a) + 1)
                  + static_cast<short>(a))
```

Here we've cast each character end-point to a small integer to get its ASCII code so we can do the range and shifting math with them. Then, having generated a random integer in the proper ASCII code range, we cast it back to a char at the end. Voilà!

[24] Mathematically, we're making a value in the range $[a..b]$. The .. denotes a discrete range rather than a continuous one. Continuous ranges are designated with commas.

[25] To complete the picture, we need to subtract one from the difference if neither end is to be included in the range.

2.6.3.3 Logical Values

There are only two logical values to generate: true and false. But, we can generate them in different proportions to one another. For instance, using typecasting, we can easily get a 50:50 split of the values:

```
static_cast<bool>(rand() % 2)
```

This takes a randomly generated 0 or 1 and asks for it to be cast to bool. The 0 always turns into false and anything that isn't all 0 bits — the 1 here — turns into true. (Only 0 is represented by a run of 0 bits. All other numbers need some 1 bits to tell their proper magnitude.)

This realization makes changing the proportions pretty easy. Say we wanted 75 trues to every 25 falses, we'd code:

```
static_cast<bool>(rand() % 4)
```

Here, the 0 still becomes false and the 1, 2, and 3 become true values since they are not all 0 bits.

But how to reverse the proportions? Say we want 75 falses to every 25 trues? Here we need a little help. Let me introduce you to the logical operator !.[26] This operator is unary — it applies to just one thing (or operand). And its purpose is to take the logical opposite of that thing. So, given a true, ! makes false as its answer. And vice versa: false becomes true under a !. (We pronounce this operator 'not' as being 'not true' makes you false and vice versa.)

So, to reverse the proportions, we simply ask for the 'not' of the casted value:

```
!static_cast<bool>(rand() % 4)
```

This gives the requested 75 falses to every 25 trues.

What about proportions that aren't nice integer spreads? We can do that, too, but it takes a little more machinery. Let's come back in a minute...

2.6.3.4 Floating-Point Numbers

To generate random double values, we can take our knowledge of random integers and extend it another step. Keep in mind that the upper-bound of all rand outputs is fuzzily included. So, to make sure we know the upper bound, we'll mod-off by that prospective upper-bound:

```
rand() % RAND_MAX
```

This gives us a value between 0 and RAND_MAX-1 both included. Now that we know the upper bound, we can turn that into a floating-point in the range $[0, 1]$:[27]

```
rand() % RAND_MAX / (RAND_MAX - 1.)
```

Note how we put the decimal point on the subtracted 1 making it a double under the division! And since we've used the largest value that can possibly result from that modulo, we'll have an upper bound of 1 now.

Next, we scale the upper bound to the desired range maximum. This isn't, however, b as you might expect. It is b-a. Remember, this random range still starts at 0 right now. So we are still going to have to add a to shift it into place. That makes the width of the range — in a continuous-like system — b-a:

[26] We could use the keyword not instead, but we use the symbol because it is very common in existing code and you need to recognize it and not freak out when you see it. *smile*

[27] Even though floating-points just approximate real numbers' continuousness, we still use the comma here.

```
rand() % RAND_MAX / (RAND_MAX - 1.) * (b - a) + a
```

I went ahead and included the a-shift, but you see how the $[0, 1]$ range changes to a $[0, (b - a)]$ range, right? And then adding a onto every possible value makes the range $[a, b]$ as desired.

2.6.3.5 Logical Values Revisited

Now that we have random floating-point values in our tool belt, we can use them as probabilities and make more diversely spread logical randomness!

If you haven't taken a statistics or probability course yet, don't worry. We aren't using anything fancy. But everyone should know that all probabilities are in the range $[0, 1]$. There is nothing more likely than 100%. And you can't have something less likely than 0%. (Other percents exist, but they are not probabilities.)

So, given that we want a situation with a probability p of `true` values showing up (and therefore probability $1 - p$ of `false` values), we'll just make a random probability and compare to our desired probability:

```
rand() % RAND_MAX / (RAND_MAX - 1.) <= p
```

What's that funny symbol between p and the random $[0, 1]$ value? That's how C++ programs represent 'less-than or equal to'. We can't do the usual symbol on a plain text screen so we combined a less than and an equal to make ours. Clever, no? Well, we thought it was...

Anyway, this <= test will come out `true` $p * 100\%$ of the time and `false` the rest. Try it with an off-center value like 60%. (If you test with 50%, it is misleading and can lead you to use the wrong comparison.)

2.6.3.6 Why Aren't My Random Values Changing?

Each time you run your program, unfortunately, you'll end up with the same sequence of values — over and over again. They don't appear to be truly random or even kind-of random after all!

That's for repeatability. This is useful in debugging and testing. If results — even random ones — weren't repeatable, it would be mighty difficult to track down a problem!

Toward this end, the `rand` function actually always starts at a certain spot between 0 and `RAND_MAX` and works from there through a sequence of values always guaranteed within those bounds. It has a nice property, too, called a maximal period. This means that it won't repeat a value until all the values in the range are used once.

Once you've debugged your program and are ready to deploy, you need to add one line to your program. You need to call the function `srand`.

This function will 'seed' or start the random sequence produced by `rand` in a different place than normal. How do you tell it a seed, then that will change every time the program runs? Well, what value do we already know that changes regularly — about every second? That's right! The result of `time` from the `ctime` library! Let's use that:

```
srand(time(nullptr));
```

Recall the `nullptr` input that is necessary to tell `time` we don't want its optional behavior — just the seconds returned.

There is one thing that can go wrong here, though. The result of the `time` function is a little different than the type the `srand` function expects to see come in. `srand` expects an `unsigned` integer type and the time function's result is signed. Why would the number of seconds ever advancing from the epoch need to be signed, one might ask? Well, the data type associated with time — named time_t — is also meant to be able to express times that came before the epoch. Of course those would be negative with respect to that 0-time.

So, what can we do? We cast the issue away! Knowing that the actual result of the time function is always positive, we can safely cast it to unsigned without worry or harm. This makes even the pickiest of compilers happy!

```
srand(static_cast<unsigned>(time(nullptr)));
```

And, with that added to your source code, your values should be more visibly random now!

What? You've got a sequence coming out now of all the same value? Where did you put that srand at, exactly? Oh! I see. Don't place it right in front of every rand call. Only place the srand once — at the beginning of the main function. It never needs to be repeated. If you do, at the speeds computers run these days — billions of operations a second — you'd see the same values repeated over and over because you'd be restarting the random sequence over and over billions of times a second.

So always call srand just once — at the beginning of the main function. It can go before or after the variable declarations as it doesn't involve any of your variables. The important thing is to do it just once per program run!

2.6.4 Character Manipulation

Another thing that many programs will need to do is manipulate or somehow 'calculate' with char data. Toward that end, the C language had a whole library. We've inherited it, of course. We call it cctype. The extra c is for its C ancestry. The other c is for char, of course.

So what all is in this library? Let's take a look:

Function	Notes
tolower(c)	return the lowercase form of char c
toupper(c)	return the uppercase form of char c
islower(c)	return true if c is a lowercase letter
isupper(c)	return true if c is a uppercase letter
isalpha(c)	return true if c is a letter
isalnum(c)	return true if c is a letter or digit
isdigit(c)	return true if c is a digit
isxdigit(c)	return true if c is a hexadecimal digit
ispunct(c)	return true if c is punctuation
isspace(c)	return true if c is whitespace
isprint(c)	return true if c is printable to the screen
iscntrl(c)	return true if c is a control code

These, at least, are the heavy hitters of this library. We may not use all of them right away, but they can all be useful from time to time.

These fall into two categories. Let's look at each in turn.

2.6.4.1 Transformation

The to*[28] functions transform the input character into its upper or lower case form if it is a letter and return that. If the input is not a letter, it is returned unchanged. The original input is unharmed in this process.

[28] Here the * indicates multiple matches instead of an actual star or multiplication.

There is one caveat — of course. The value returned is not an actual char. Our C ancestors were so into speed, you see, that they used int heavily! (Remember that it is one of the fastest types on the CPU.) This includes passing char values around as ASCII codes in int guise. So, to store a toupper, for instance, result into a char variable, you'll have to do a little typecasting:

```
char yesno;

cin >> yesno;

yesno = static_cast<char>(toupper(yesno));
```

This has the advantage that it doesn't matter what the user entered — upper or lower case — we can respond to them consistently.

2.6.4.2 Classification

All the other functions there — the is functions — are for classifying a char as one kind or another. The ASCII characters can be split — roughly — into printable and control characters. Control characters are used inside the computer and its communications with other devices to control different processes. These are classified by the iscntrl function.

The printable characters include spacing, letters, digits, and punctuation. (Remember that all those crazy symbols like at signs and octothorpes and such are called punctuation, too.) The isprint function distinguishes all these values at once from the control group. But, we can break down this set into smaller groups as well.

Note that the isalpha function returns true for both upper and lower case letters whereas islower is only true for lower case letters. All these letters are just the standard English alphabet. Sad when we consider the number of alphabets and letters in use all around the globe, really.

The isdigit function works for the standard '0' through '9' characters. isxdigit, though, works for not only these values, but also the values 'A'/'a' through 'F'/'f'. These letters are included because the hexadecimal base is 16 and we only use single position values to indicate 'digits' in a number. So 'A' would represent 10 in this system, 'B' would be 11, etc. up to 'F' would be 15. We need not go higher than this, because 16 would be represented by '10' in base 16. (Indeed, the number '10' always represents the base we are dealing with: 2 in binary, 3 in trinary, 10 in decimal, etc.)

2.6.5 More From iostream

There is more to the iostream library than just cout, cin, <<, and >>.

2.6.5.1 An Example Revisited

To explore, let's revisit the deer projections example from before. It turns out that some of the rangers in testing are giving us problems with the inputs. They don't like stopping at the numbers we've asked for! Some of them want to enter units of 'deer' or worse, long sentences the computer just doesn't know how to deal with. (Poor lonely folks. . .)

To alleviate this, we have two options. One is to jump ahead and use a string-type object (see section 3.8.2). But that seems overkill since it would mean reading in and storing all that garbage the ranger is typing that our program doesn't need or understand. Doing so would waste precious milliseconds and bytes of RAM!

Instead, let's focus on an iostream tool: ignore. This function takes a variety of parameters that can leave even the most steadfast coder confused, so let's take it step-by-step. First, it is called like the fill and width functions from before except with respect to cin instead of cout.

It's first variation is to take no parameters (inputs) at all:

```
cin.ignore();
```

This makes one character disappear from the input stream cin before we move on with the program. Unfortunately for us, this isn't enough for our needs. The rangers are entering at least words if not more — not single characters.

The second variation is to pass an integer and a special character that will be the last character to ignore. This can look like this:

```
cin.ignore(10, '\n');
```

This tells the ignore function to throw away at most 10 characters but stop when a newline is thrown out even if 10 characters haven't been dispatched. This would work for us if the ranger stops typing with something short like 'deer', but if they type much more, it won't be sufficient.

In fact, it isn't generally the thing to do because of a kind of hacking attack known as a buffer-overrun attack. This attack finds out how many characters a particular input (cin) can handle and gives it more than that to make the program break into administrator/superuser mode and give the hacker access to such privileges.

To avoid both these issues, we need a special integer value to send to that first parameter. Luckily, ignore is set up to take such a value! We'll send what's called a flag value — to raise or fly a flag to signal to the ignore function that special circumstances are at play.

The value we need is the maximum possible value for the first parameter. In these circumstances, the ignore function won't wait for that many — ridiculously large number of — characters before stopping. It will understand that this is as close to ∞ as we could code and treat it as a flag to read as many characters as necessary to reach the special stop character — the second parameter.

The unfortunate part of this is the way we access the maximum value. It is a syntax nightmare for new programmers. It starts by needing another library: limits. Then it uses this format:

```
numeric_limits<streamsize>::max()
```

What's with all these angle brackets and colons? Well, let's start at the beginning. The tool numeric_limits informs us about all the properties of numeric types in C++. The type we are interested in is the integer type streamsize. This one is for information related to how many characters can be used in stream contexts like our input stream cin. So, like with static_cast, we put this type in the angle brackets.

The double colon — or colon-colon as it is often read — is similar to the one we learned to avoid with the standard namespace so long ago now. Recall that we could do a using directive (using namespace std;) to avoid doing the syntax std:: in front of every library name we used in our code. That colon-colon (or scope resolution operator) told the compiler to look inside the standard namespace for the definitions of those names instead of in the current code.

This double colon is similar, it turns out that numeric_limits is a group of related information on the numeric types' properties. And so this double colon is saying that the max function is inside that group of related information.

Such a group of related information functions and properties is known as a class. This is the C++ mechanism to make new data types that weren't known by the CPU originally. We'll learn in later chapters how to make our own classes, too.

Anyway, that's quite the mouthful! That's also a lot of typing that could go wrong and cause you to have to fix it and recompile. Since we want to use this after every input (>>), we'll want to make it more palatable to type and read. I recommend a constant with a good name. Perhaps something like this:

```
constexpr streamsize INF_FLAG{numeric_limits<streamsize>::max()};
```

This tells us it is the infinity flag and us using it in the `ignore` first input position will tell us its purpose:

```
cin.ignore(INF_FLAG, '\n');
```

So, that's a lot of changes and I haven't been very clear on where to put them all, so let's look at the whole program once again. Here it is with both the rounding changes and our latest changes for input issues:

```
#include <iostream>
#include <cmath>
#include <limits>

using namespace std;

constexpr streamsize INF_FLAG{numeric_limits<streamsize>::max()};

int main()
{
    short deer_last_fall,      // for creating the projection rate
          deer_last_spring;
    short deer_next_spring,    // for the actual projection calculation
          deer_this_fall;
    double growth_rate_mult,   // the projection rate in multiplicative
           growth_rate_pcent;  // and percent formats

    cout << "\n\t\tWelcome to the Deer Projection Program!\n\n";

    cout << "Please enter last fall's deer population:  ";
    cin >> deer_last_fall;
    cin.ignore(INF_FLAG, '\n');

    cout << "Please enter last spring's deer population:  ";
    cin >> deer_last_spring;
    cin.ignore(INF_FLAG, '\n');

    growth_rate_mult = deer_last_spring / deer_last_fall;
    growth_rate_pcent = (growth_rate_mult - 1) * 100;

    cout << "\nThe growth rate from fall to spring is typically "
         << growth_rate_pcent << "%.\n";

    cout << "\nPlease enter this fall's deer population:  ";
    cin >> deer_this_fall;
    cin.ignore(INF_FLAG, '\n');

    deer_next_spring = static_cast<short>(ceil(growth_rate_mult
                                               * deer_this_fall));
    cout << "\nThen I project the deer population for next spring "
            "will be " << deer_next_spring << ".\n";

    cout << "\nThank you for projecting deer with us today!\n"
            "\nPlease come again!\n";
    return 0;
}
```

If you don't like the name of my constant, you can always change it for your own implementation. I've been known to call it, for instance, UNTIL_YOU_SEE. This reads quite nicely in the ignore call itself: cin, ignore 'until you see' a newline. But others don't like this kind of flippant naming. *shrug* To each their own, I suppose. . .

Anyway, note how the ignore follows every input of a number of deer. This avoids some rangers penchant for entering units or discourse. What about those rangers that don't enter such things? Will it break when they run it? No! It turns out that all input lines end with a newline. That's how cin represents the user hitting Enter/return. So there will always be a '\n' to stop the ignore. *bounce* I just love it when a plan comes together, don't you? *smile*

2.6.5.2 Formatting Output

In addition to the fill and width we saw earlier in the time calculating example, there are many other things you can do to format the output of a program using cout. Most revolve around displaying decimal numbers.

Let's say you had a few decimal numbers in your program that you needed to display in a column — lined up at the decimal points. Let's say further that you've named them poorly as a, b, and c. Let's start off having them hold the values 10, 5.6, and 2.129 respectively. Finally, let's say that the program specification calls for there to be two decimal places on each number in the column. (This will help us line up on the decimal point immensely!)

Our setup so far is this:

```
double a{10}, b{5.6}, c{2.129};

cout << a << '\n' << b << '\n' << c << '\n';
```

This isn't coming out very well at all:

```
10
5.6
2.129
```

But it is a start!

Let's make them all the same width with our old friend width. Given the magnitude of our numbers, we can guess that 7 would be wide enough to hold all our data in this column — even if we should we get more values in future. Now our code looks like this:

```
double a{10}, b{5.6}, c{2.129};

cout.width(7);
cout << a << '\n';
cout.width(7);
```

```
cout << b << '\n';
cout.width(7);
cout << c << '\n';
```

And our output now looks like this:

```
     10
    5.6
  2.129
```

Well, the data now lines up, but not on the decimal point. In fact, our data doesn't all display with decimal points. Nor do they all have the 2 decimal places we were told to display — one even has more!

To fix these issues, we need new helpers. The first is the `cout` function `precision`. This function takes a parameter that sets — wait for it — the precision of all displayed decimal numbers. For normal numbers this amounts to the number of decimal places. For numbers that resort to scientific (E) notation, it means the number of precise digits. (To find out more on 'precise digits', see your local lab science teacher!) Since our numbers are pretty normal — not too large or small, we don't need anything more than:

```
double a{10}, b{5.6}, c{2.129};

cout.precision(2);
cout.width(7);
cout << a << '\n';
cout.width(7);
cout << b << '\n';
cout.width(7);
cout << c << '\n';
```

Now our output looks like this:

```
     10
    5.6
    2.1
```

Wait! What happened to the rest of the third value? The rules for this are ridiculously complicated, but basically, since we didn't specify that the value was to be normal or scientific, `cout` had to go with a mix of the rules. Let's be more specific. Let's set a flag flying that `cout` will interpret as a signal to use a particular style.

Set a what? A flag. A flag in computer terms is a `bool` or even single bit that is `true` or 1 to signal a special circumstance. Setting a flag is making it wave in the breeze — that is, making it `true`/1.

To do so we need two things: a function to 'set' a flag (make it wave in the breeze — make it `true`/1) and a name for our desired flag. Luckily both are provided by the `iostream` library. The function to set a flag is `setf`. The name of the flag is `ios_base::fixed`.[29] This will make the style normal by fixing the decimal point right after the ones place. (Recall that in scientific notation the decimal point is more flexible and we just change the power of 10 to account for this.)

This name seems strange at first, but when you break it down you realize what's going on. The `fixed` constant is actually created inside the `ios_base` `class`. The `::`, you may recall, tells us that one thing is inside another. The syntax of this is read from right to left: "the `fixed` constant is from inside the `ios_base` `class`".

[29] The related constant `ios_base::scientific` is used to change decimal data to always be in scientific notation.

To set it up to print normal numbers (with a fixed decimal point), we code:

```
double a{10}, b{5.6}, c{2.129};

cout.setf(ios_base::fixed);
cout.precision(2);
cout.width(7);
cout << a << '\n';
cout.width(7);
cout << b << '\n';
cout.width(7);
cout << c << '\n';
```

With that our output looks like this:

```
  10.00
   5.60
   2.13
```

Why did the third value change? Well, even without calling the `round` function from `cmath`, `cout` knows to round the displayed number when the variable's value is too long to fit the precision set. (If the value of c had been 2.124 instead, the output would have been 2.12 instead.)

Nice! That looks just fine.

2.6.5.2.1 The Rest of the Story

That's all well and good, but what if there are multiple programmers working on the code and they are all setting different formatting options?

First off, that's just crazy! Multiple programmers on one program? That'd never happen! Actually, that's the way most programs are written. Each programmer takes a part and the project head takes everyone's codes and puts them together.

Second, why would the formatting change? Maybe one part of the program is intended for Federal jurisdiction and the formatting has to change from the way your everyday citizen likes to see it. (*shrug* It can happen. Don't ask so many questions!)

So, what would we do about that? We'd have to learn to play together nicely. To help out, the formatting functions we've studied so far all return the values they were previously using before making your change. How's that help? It tells us what the previous programmer had done and let's us record that value and reset it after we are done!

Let's practice. Let's say the programmer before us had set the filler character to be dashes and we came along and wanted stars. How would we store and reset their filler character? To store the old filler character, we'd need a `char` variable. Then, after we are done with our code, we'd call `fill` again to reset the filler to that value:

```
char old_fill;

old_fill = cout.fill('*');

// do our stuff with stars

cout.fill(old_fill);
```

To make sure this is working as planned, flesh it out to a whole program and try it! Remember all the parts:

```
#include <iostream>

using namespace std;

int main(void)
{
    char old_fill;

    // original programmer sets up dashes for fill
    cout.fill('-');

    cout.width(7);
    cout << 10 << '\n';

    old_fill = cout.fill('*');

    // do our stuff with stars
    cout.width(7);
    cout << 10 << '\n';

    cout.fill(old_fill);

    // original programmer returns and expects dashes to continue!
    cout.width(7);
    cout << 10 << '\n';

    return 0;
}
```

If you try it, you'll indeed get:

```
-----10
*****10
-----10
```

The currently set width is also returned from a call to `width`, but since it only lasts for a single output, this isn't really a concern. The only other things we can change are the `precision` of decimal numbers and the `fixed` flag.

The `precision` is returned as a `streamsize` value. (Recall `streamsize` from our `ignore` usage.) So, to preserve another programmer's settings for `cout`'s `precision`, we'd need one of those and, voila:

```
streamsize old_precision;

old_precision = cout.precision(2);

// do our stuff with 2 precision

cout.precision(old_precision);
```

It looks almost just like the filler character example! Sweet!

But what about the ios_base::fixed flag? Well, that's slightly different. It turns out that there are many potential flags cout can look for and they are all stored as a big group in a single variable. When setf returns the old setting, it returns the whole group at once. To store them for preservation, we have to use the ios_base::fmtflags data type.

I know that one is shocking. First, it is a data type created inside a class — hence the scope resolution operation (::). Second, it has a terrible, mushy name. When the C++ designers were first working on C++ they were but idealistic C programmers. So many old C habits are exhibited in their early efforts. But time heals all wounds and this one is pretty shallow, right?

So, let's look at this preservation pattern:

```
ios_base::fmtflags old_flags;

old_flags = cout.setf(ios_base::fixed);

// do our stuff with fixed notation

cout.flags(old_flags);
```

Wait! Why is the last function flags instead of setf? I thought setf was to 'set flags flying'? Well, it works for individual flags in isolated settings. But the function flags works for groups of flags all at once. Since we stored all of cout's flags together in the old_flags variable, it only makes sense to set them all together with flags instead of setf.

But what about that 'isolated settings' thing I said? Oh, well, when we are programming alone it is safe to use setf like we did. However, when other programmers are around, things can go awry. What if another programmer had set up scientific notation for their part? Then, when we told cout to set the fixed flag, both it and scientific's flag would be flying at the same time! What would cout do then?! It pretty much goes berzerk and pukes up on your screen. *shiver*

Formats Combined

Actually, when both fixed and scientific notations are set on simultaneously, a new format called hexfloat is used. This prints the exact binary form of the decimal value in base 16 (hexadecimal, remember?) form. Crazy, right?

So how do we fix this possibility? Well, we use masks. No, not face masks! What you might call a flag mask. But, since the flags are often stored as single bits, we call them bit masks. The bit mask we need is already defined — yep! you guessed it! — in the ios_base class. It's called floatfield. This is because it encompasses both of the bit 'fields' that deal with floating-point data formatting specifically. (The field terminology will make more sense when we get to writing our own classes. Just use a flash card to remember it. *smile*)

How do we use it? There is another form of the setf function that takes the bit mask as a second parameter. We just call:

```
cout.setf(ios_base::fixed, ios_base::floatfield);
```

That's it? Yep. That's it. Some things are actually relatively simple — even in programming!

2.6.6 Another Point of View

All of the above formatting and preservation of formatting just used the iostream library. But there is another set of formatting commands for C++ that come in another library altogether. Some are clunkier, but others are more streamlined in their application. Knowing both, you can decide which you like best.

This other library is called iomanip and is used to manipulate the input/output system in various ways — formatting it.

Having #included iomanip, we can use substitutions that insert into the stream with the normal output operator (<<). For instance, we could set the precision of further decimal outputs with the setprecision manipulator:

```
cout << setprecision(2);
```

There are also setw for setting the width, setfill for setting the filler character, and setiosflags that acts like setf. However, setiosflags doesn't have the alternative form with the mask-out and might end up setting both fixed and scientific at once, for instance.

Further, there is a minor issue with setw that bears mentioning. We saw before that the precision function takes and returns a streamsize-typed value. This is true of width, as well. But setw takes a raw int instead. This is fine if you are providing the parameter to setw literally, but if using a variable of type streamsize, the compiler might have an issue since the typical bit-size of streamsize is a bit larger than that of int.[30] You could fix it with a static_cast or just use width instead. It isn't as fluid as setw, I'll admit. But it is type-safe.

What's the problem with setfill and setprecision? Nothing. Their names are just longer. No big deal.

But don't forget that to use any of these manipulators, you need to #include iomanip at the top of the source code file!

2.6.7 And Back to iostream

What? There's more in iostream still?! Yep. It's the library that keeps on giving![31]

2.6.7.1 Manipulating Line Endings

You may have found online people using a manipulator called endl to end their lines on couts like so:

```
cout << "stuff on the line" << endl;
```

This manipulator is a combination of inserting a newline and an operation called flush which makes cout display immediately. Why wouldn't it always display immediately? Normally cout waits to display until it has about 2000 characters — a full standard text screen of data. This makes the display faster given the discrepancy between the CPU's gigahertz speeds and the screen's hertz speeds. We call this holding area where the 2000ish characters wait the buffer.

In addition to a full buffer, cout will also display at two other occasions without being flushed. It will display its current content when cin is trying to read an input. This is because cin tells cout of impending input so that any waiting prompt for the user can be displayed first. Otherwise they wouldn't know why the program had paused!

The other situation in which cout displays without being flushed is when the program ends — at main's return. Otherwise the user wouldn't get to see all those last comments and results you'd printed!

So is endl worth the extra typing? (After all, sticking a simple \n into your literal string is much simpler. Even if you add a '\n' after a variable output, it is the same number of keystrokes.) Some say

[30] Sorry for the bit pun. Couldn't help it!

[31] Seriously, we've just scratched the surface of many of these libraries. There's so much more under the hood! Check out cppreference.com sometime if you don't believe me.

yes. But others point out that constantly forcing a display will avoid the benefits of the buffer and make the CPU wait for more screen updates slowing down the overall program. Your mileage may vary, but it isn't to my taste and I won't be using it further here.

2.6.7.2 Other Manipulators

But endl isn't the only manipulator in iostream! There are quite a few more that mimic the flags you can set with setf. We'll just mention those for the floating-point flags we've used so far and a couple of others you might find immediately useful.

2.6.7.2.1 Floating-Point Display Manipulators

To make the decimal fixed after the ones position, you can use the fixed manipulator like so:

```
cout << fixed;
```

Why, you may ask, have we not done this in the first place instead of using the clunkier setf method? Well, we wanted to remind you of the scope resolution operator (::) and show its usage in this context. It also facilitated our discussion of preserving other programmers settings. The manipulator doesn't return the previous settings, you see. And, of course, we never want to pass up the opportunity to learn something new and useful! *smile*

The other manipulator, unsurprisingly, is scientific and is used to set up display of floating-point numbers in scientific notation. Again, this manipulator sends back no information about prior flag settings — just makes the adjustment requested.

Don't worry about the floatfield mask issue we mentioned previously, though. These manipulators take care of that issue automatically.

2.6.7.2.2 Justification of Display Fields

In addition to floating-point displays, we can use manipulators to affect the justification of a display in a certain width. Note that so far any width effects we've produced have right-justified the data within that field. That is, the data was pushed to the right of the width and padding was added to the left. The opposite can also be achieved. We can push the data to the left and have padding added to the right with left-justification. To do so, merely insert the left manipulator:

```
cout << left;
```

To put it back to right-justification for a later width setting, just use the right manipulator in a similar manner.

(For the curious, there are flag constants for justification that can be used with setf as well. These are ios_base::left and ios_base::right respectively. And remember, you are in right-justification mode by default!)

For those thinking carefully here, you might wonder where is centering? Sadly, that is not a justification setting. We'll discuss centering a little later when we learn more about strings and the string class data type.

2.6.7.3 Reporting Errors to the User

Our last new item from iostream is the error stream cerr. It displays to the screen just like cout, but instead of being buffered it displays everything you send it right away. After all, we only use it to display errors — serious problems the user should know about immediately!

It will also come in handy during our debugging efforts later in the book so don't forget about it just because it is seldom used in daily code!

2.7 Wrap Up

In summation, we've covered a **LOT** of information in this chapter! We learned how to make a basic C++ program with variables, constants, output, input, and simple calculations. We've even covered many standard library features that can make our programs more powerful, helpful, and beautiful!

I hope this chapter end finds you well and not struggling. If you have any troubles, please see your instructor or a qualified tutor for help! Don't just search the Internet. People are helpful there, but often too helpful. They'll teach you things you aren't prepared for and even give bad advice at times. If you must search, make sure you corroborate any advice with several sources and don't just trust the first blog or other posting you find on a subject.

Part II
Flow Control

Chapter 3

Decision Making

It often happens that we need to make a decision in the flow of a program. Such a decision might make some code run only under certain conditions or even to repeat itself under certain conditions. The former kind of decision is called branching and the latter looping. We'll explore both of these kinds of decision making in this chapter.

As we are making decisions here, we'll also explore the `bool` data type in more detail. We'll also make sure you know bad practices and how to avoid them!

3.1 Branching

There are many ways to branch code in C++. We'll start with an application that needs the simplest form: the `if` statement.

3.1.1 if Statements

Let's motivate our decision making by thinking about our poor user entering a dollar sign before price or pay information. They don't generally want to do that, do they? Sure, we could put a '\$' in the prompt for the amount, but then our program would be tied to the American economic system. We need to think about internationalization in our programs almost every day!

To make our program work whether the user enters their monetary unit or not, we'll have to make a decision as to whether they've typed it in or not. This needs not only an if statement as mentioned above, but also a cin function to tell us what is about to be read. This function is named cleverly peek. We call it with parentheses but nothing inside — like the one form of ignore we saw before.

```
cin.peek()
```

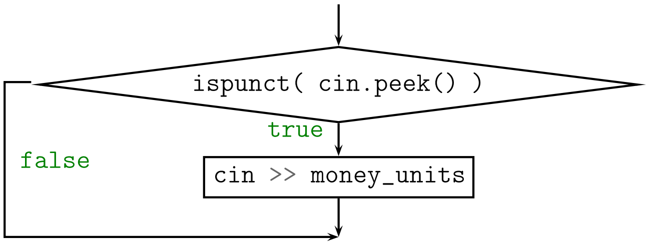
I didn't put a semi-colon on that example because we don't generally call it on a statement alone. Usually we take its returned value and use it in some way. In our case, we'll be using it to see if there is a monetary unit in the input:

```
double price;
char money_units;

if ( ispunct(cin.peek()) )
{
    cin >> money_units;
}
cin >> price;
```

Note that we've used the ispunct function from cctype to determine if the peeked value is any kind of punctuation. All the monetary symbols I know of are classified as punctuation in their locales.[1]

The if functions by testing the bool condition between its parentheses and, if this condition is true, it executes the statements between its curly braces. If the condition is false, the statements in the curly braces are skipped and the program just continues after the close curly as normal.[2] It's flowchart looks like this:



The diamond in a flowchart stands for a decision. Our condition can be true or false so the arrows coming out of the diamond are labeled with these so we know which way to flow when the condition has each value.

Now the run of such a fragment will result in a user being allowed to enter either:

```
Prompt for price:  $34.95
```

Or just:

[1] A fancy word for location — actually used in some programming tasks!

[2] Technically there need not be curly braces if only one statement is inside the if. But good style and good maintenance habits demand that we put the curly braces on in almost all situations. There will be an exception, however. Stay tuned!

```
Prompt for price:  34.95
```

Nice! But what if the user puts anything — even a space — before their input? Then we won't be peeking at the right value! We'll be seeing this space instead! This isn't good. It would cause the if condition to fail and then the program might try to read a $ or the like as the price!

It wouldn't bother a simple extractor (>>), because it always skips leading whitespace. But it does bother peek which respects all forms of char. How do we fix this? We need another helper. It is a manipulator from iostream called ws. It removes a contiguous sequence of whitespace from the current input position — stopping as soon as some non-spacing character is encountered. We'll use it like so:

```
double price;
char money_units;

cin >> ws;
if ( ispunct(cin.peek()) )
{
    cin >> money_units;
}
cin >> price;
```

By removing the whitespace before we peek, we avoid the potential for there to be a miscommunication between our peek result and the actions of the following >> operation. After all, the following >> operation would have removed all the leading whitespace, too, right? Therefore, we should do the same before peeking.

But, there is one more problem: what if the user's local custom is to put the monetary unit after the amount instead of before? (Yes, that's a thing. Get out more!) Well, we can handle that with a slight change and a second if:

```
double price;
char money_units{};

cin >> ws;
if ( ispunct(cin.peek()) )
{
    cin >> money_units;
}
cin >> price;
if ( money_units == '\0' && ispunct(cin.peek()) )
{
    cin >> money_units;
}
```

Okay, that's a little more than a second if. But let's take it one piece at a time. First, the empty curly braces on money_units. This will initialize this variable to its default value. The default for the char data type is the null character ('\0', remember?). Using the empty-brace syntax saves us having to type the escaped char literal ourselves.

Next we've used the operator == to test a variable's value. We're particularly interested on whether the money_units variable is still unentered by the user. If so, we might be dealing with a trailing units custom! Why is it two equal signs to test a variable's value instead of just one like we'd do in math? Well, the compiler has trouble telling the difference between using a single equal to assign a new value vs. using it to test a value. So, the designers of C (our ancestor language, you'll recall) made the test version two equal signs. This can be hard to remember, so watch out for a compiler warning that says something like 'use of assignment in condition, did you mean to use == instead?'. The message won't be exactly those words — the wording changes from one compiler to another and sometimes between versions of a compiler. And your compiler won't necessarily be configured to warn for this by default. If you'd like to force a message, you can turn it into an error situation by switching the order of your test:

```
'\0' == money_units
```

Then if you accidentally use a single equal, there will be an error as you cannot assign a new value to a literal!

Lastly, we've used the double-ampersand (&&) to combine two `bool` values with the logical operation AND.[3] The left `bool` comes from our equality test and the right one comes from the `ispunct` result. The rule for combining two `bool` values with a logical AND is:

Left	Right	AND Result
`true`	`true`	`true`
`true`	`false`	`false`
`false`	`true`	`false`
`false`	`false`	`false`

Thus our overall condition will only be `true` when both the user did not enter a previous unit and the upcoming input is punctuation and therefore probably a monetary symbol.

With all that in place, we can see that our input will now work for lots of inputs:

```
$34.95
␣$34.95
34.95
34.95$
```

(Here the ␣ mark is how we show spaces in examples.)

What's that you say? The user tried to enter a space before a trailing monetary unit and the program crashed on a later input? Oh no! I was afraid of this. We'll need a new helper: a loop!

3.2 Looping

When code needs to be repeated some number of times, we can use one of many repetition or looping structures/statements. Our first one will be the `while` loop. This loop will allow us to repeat a section of code as many times as necessary to complete the task at hand — from zero to infinity! (Okay, if the loop runs forever, the user is bound to break out of it with the little X in the corner or a Control+C combo.[4])

What we need here is a loop that will allow the user to enter spacing but still stop at the newline that ends all inputs. If we used `ws`, it would remove all the whitespace — including the newline! (If it removes the newline, the program will hang and wait for more input, you see...)

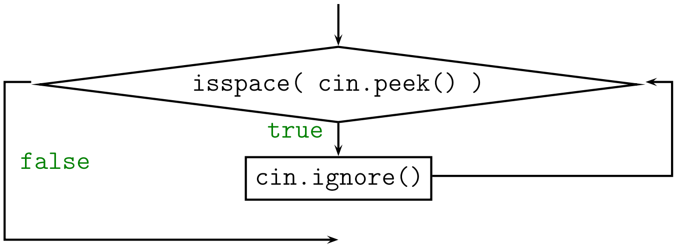
So, let's examine what `ws` does and try to modify that to our needs. The code for `ws` might very well look something like this:

[3] Technically we could have used the keyword `and`, but we chose the && syntax because it is more prevalent in existing code and you need to know it and recognize it. But knowing both forms is not bad for you, either.

[4] That's how you stop a program running amok in the terminal, btw. Hold the Control key while hitting the C key.

```
while ( isspace(cin.peek()) )
{
    cin.ignore();
}
```

The while loop here is a little misleading since it has no visible indication that anything is being repeated. It's flowchart looks like this:



Note how the last looped statement always flows back to the top to make the decision all over again. Only when the condition turns false do we continue on with the remainder of the program.

So how do we change this loop to make it respect the newline as the end of the input instead of just another space? Well, let's think of when we want the loop to stop. This is often a convenient place to start as all humans tend to dwell on when things will end: ”When is class over?”, ”When is our next break?”, etc.

We want to stop the loop when we see either a newline or any non-space character. But the loop condition needs to be not a 'stop' condition but a 'keep going' condition. Luckily these two are logical opposites. And we can code a logical opposite with the ! operator, remember?

So, let's code this up!

```
while ( ! ( cin.peek() == '\n' || !isspace(cin.peek()) ) )
{
    cin.ignore();
}
```

Wow! That's a mess. We had to put extra parentheses around our stop condition so we could take the opposite of the whole thing at once. And we had to put a logical NOT (!) on the isspace test as well. But what's that thing with the two vertical bars? That's the operator for logical OR. We said we wanted either the newline or the non-space to stop us, right? So I coded it as logical OR.

The or in human language and the logical or are slightly different, however. In fact, they have separate names. Normal spoken/written or is called exclusive or and logical OR is called inclusive or. The reason is clear from its defining table:

Left	Right	OR Result
true	true	true
true	false	true
false	true	true
false	false	false

Not only is logical or true when one or the other of its operands[5] is true but it is also true when **both** of its operands are true! Since we don't expect this in normal language, we call our version exclusive or — exclusively one or the other. And the logical version is called inclusive or because it includes the possibility that both operands could be true at once.

[5] This is the general term for the thing being operated on by an operator. We had addends and divisors for specific math operations, for instance. But we haven't named all of them for all the operators in existence. So we have this general term to cover the rest.

Is this a problem? No. Here, if we are a newline, we will be a space so the right side of the || will be false. And if we are not a space, we'll definitely not be a newline, so the left side of the || will be false. Since the first line of the logical OR defining table can't happen, it won't bother our loop at all.

3.2.1 DeMorgan's Laws

But the ugly ! and extra parentheses are getting on my nerves. Let's explore DeMorgan's Laws to find a way to get rid of them. Basically what DeMorgan proposed was that if we want to logically negate a logical AND, we need to negate both operands and change the AND to an OR. Similarly for the reverse — negating an OR negates the operands and changes the OR to an AND:

```
!(X && Y) == (!X || !Y)
!(X || Y) == (!X && !Y)
```

To see this is correct, we can use a truth table like those used to introduce AND and OR in the first place:

Left	Right	OR Result	OR Negation	NOT Left	NOT Right	AND of NOTs
true	true	true	false	false	false	false
true	false	true	false	false	true	false
false	true	true	false	true	false	false
false	false	false	true	true	true	true

Note how the fourth and last columns are the same. This shows that DeMorgan's law for negating an OR is valid. The table to show the negation of AND is correct would look similar.

How do we apply this? It would look like so:

```
while ( cin.peek() != '\n' && isspace(cin.peek()) )
{
    cin.ignore();
}
```

Note that I not only took the ! off the isspace to negate it, but also changed the == to its opposite: !=. This is a shortened form, clearly, of ! and ==. There was a decision in the C language to make all operators two characters at most, so it was smooshed.

This is very difficult to read, I understand, but it will execute more quickly than the previous coding and that's beneficial to our user. If it helps, an && operation with a ! involved is basically the logical equivalent of the English word 'but'. So we could read the condition as: ”we aren't looking at a newline, but it is some kind of space”.

3.2.2 Back to the Problem

So, does this help us and if so, how? Well, we have to put it in the right place, but it will help! Let's put it here:

```
double price;
char money_units{};

cin >> ws;
if ( ispunct(cin.peek()) )
{
    cin >> money_units;
}
```

```
cin >> price;
while ( cin.peek() != '\n' && isspace(cin.peek()) )
{
    cin.ignore();
}
if ( money_units == '\0' && ispunct(cin.peek()) )
{
    cin >> money_units;
}
```

Since it is acting like a `ws` but doesn't remove newlines, it is perfect to precede our `ispunct` `peek` that follows the user's `price` input. After all, the user may not enter a monetary symbol at this position (see the first three test cases above). If they didn't, there would be just the newline — possibly with spacing before it — after the `price` was read. Let's check our tests again:

```
$34.95
␣$34.95
␣34.95
34.95
34.95␣
34.95$
34.95␣$
␣34.95␣$
␣34.95$
```

Now we pass many more test that a typical user might enter! And we've learned so much about looping! It's a win-win!

3.3 More About bool

We've used `bool` quite a bit, but there is more to learn.

3.3.1 DeMorgan's Laws and Efficiency

Why will the DeMorgan's version of the newline-respecting loop above execute more quickly? Well, let's examine the initial coding and test it in a case with a few different inputs coming in from the user:

Input Peeked	A	B		C	
	`cin.peek() == '\n'`	`isspace(cin.peek())`	`!B`	`A || !B`	`!C`
$	false	false	true	true	false
\n	true	true	false	true	false
␣	false	true	false	false	true

As you can see, to evaluate each input requires five operations. This is quite a lot if the loop has to repeat many times. But, for the version taken through DeMorgan's process we have:

Input Peeked	A	B	
	`cin.peek() != '\n'`	`isspace(cin.peek())`	`A && B`
$	true	false	false
\n	false	true	false
␣	true	true	true

Now the tests each take three operations! Quite the savings as the loop repeats over and over.

3.3.1.1 Implicit Optimization

In fact, it is better than that in general. The compiler has made it so that when an AND's left side evaluates to false, the right side isn't even looked at! See the defining table above and how when the left side is false, the answer is false automatically — the right side seems to have no effect on this.

Similarly, when an OR's left side evaluates to true, the right side can be skipped. In this situation (again, see the defining table), the result is always true no matter what the right side's value is.

This built-in optimizing behavior is called short-circuiting because it cuts off the 'circuit' of calculations early on in the process — no need to evaluate the right side or the logic operation itself. The C++ standard requires this effect so you can depend on it from all C++ compilers. You don't have to do anything to get it, either!

3.3.2 Generating bool Values

So far we've seen that the classification functions from cctype make bool values and the operators ==, !=, !, ||, and && all make bool values. Are there any other ways to generate a true or false?[6]

There are four more operators that result in bool values. These are the inequality operators and mimic as closely as possible in plain text the familiar math operators:

C++ Operator	Math Operator
<	$<$
<=	$\leqslant$
>	$>$
>=	$\geqslant$

These can be used on almost any built-in data type: integers, floating-point, and characters. The bool data type is notably missing because there is no logical ordering between true and false.[7]

3.3.3 Logical Opposites

To help you apply DeMorgan's Laws to these new operators more easily, I've prepared the following chart. At first glance it might seem incomplete. But note that the rows are reversible.

Operator	Opposite
<	>=
>	<=
==	!=

The reason that < is not the opposite of >, of course, is that we'd be leaving a whole in our number line![8] The equal-to case is just as important as the less or greater cases. It must be included somewhere!

3.3.4 Equality and Floating-Point

Note that == doesn't play well with all data types. With floating-point data, the compiler is apt to complain that such comparison is imprecise.[9] This is true for most situations and so the compiler is right to admonish us. For instance, a 10 might code up inside the computer as 9.99999999999999 or as 10.0000000000001. These two are not quite equal even though they are effectively to normal precision.

To deal with this, we change the desired == test to an inequality test like <= (or change != to >). The test is formed like so:

[6] Other than to static_cast some other type. . .

[7] The computer will allow these comparisons and merely converts the bools to integers with implicit typecasting called coercion.

[8] If we were comparing numbers, of course. . .

[9] Mine even calls it 'unsafe'!

Desired Test	Coded Test
`A == B`	`abs(A - B) <= 1e-6`
`A != B`	`abs(A - B) > 1e-6`

Here we take the absolute value of the difference between the two values we meant to test. This makes sure we ignore whether the left is larger or the right was larger in the subtraction. Then, we test if this absolute difference is below or above a certain cutoff or epsilon. Here I've used 10^{-6}, but you can change this depending on your application. For instance, if you are dealing with measurements from a wooden ruler, you could easily change this cutoff to 10^{-2} as the precision isn't that good. But if you are dealing with a nice laser-based tool for measurement, you can change the cutoff to 10^{-9} or so. The value 10^{-6} is just a good go-to value if you aren't sure what kind of precision you are dealing with.

3.3.5 Equality and bool

Well, if `==` and `!=` don't work well with floating-point types, are they okay for the other types? They work fine for integers and characters. They'll even work for `bool`. But it isn't natural to code them against `bool` values. Let's explore that for a minute.

If you had a variable or expression that was already `true` or `false`, and then asked the computer if they were `==` to the literal value `true` only to produce a new `bool` value that is equivalent to the original — isn't that redundant? Indeed! It takes effectively three evaluations to find the answer! We could find out in just one by listing the variable or expression alone in an `if` or `while` head (the line with the condition is called the head of a branch or loop). Look at the following table:

Variable Value	Is It `true`?
`true`	`true`
`false`	`false`

So coding `bool_var == true` is the same as evaluating `bool_var` in the first place. (Evaluating a variable is just finding out the variable's current value.)

What if we want to know if the variable is `false`? Should we code `bool_var == false`? Again, look at the table:

Variable Value	Is It `false`?
`true`	`false`
`false`	`true`

Well, the answer seems to be the opposite of the original variable's value. Couldn't we just apply the `!` operation to it? That's two evaluation instead of three — still an improvement.

The only reason to test two `bool` values against one another is if they are neither one literals. But even then it is considered odd. An `&&` will tell you they are both `true`, after all. And to find out they are both false, you can `&&` their negations (see the last line of the logical OR chart above and apply DeMorgan's Laws).

3.4 Debugging with cerr

At this point you may be facing problems with some of your codes. Perhaps a branch you were sure should have executed was skipped or maybe a loop didn't run enough times. Maybe the loop is running too many times — maybe even forever!

To fix the forever problem, hit the Control key along with the C key. This will "close out" the errant program. This helps avoid having to close your terminal or whole IDE when a loop gets stuck.

For debugging this situation and the others mentioned, we can use the help of `cerr`. Remember that `cerr` is just like `cout` except without a buffer. So everything you send to it is immediately placed on the screen. This is very important in debugging! If you were to print debugging information with `cout`, after all, it would wait for the buffer to empty before being seen. This could delay it indefinitely in a particularly troubled situation. Without a buffer, cerr doesn't have this issue.

So, when a branch won't execute or a loop goes too short or too long, use a cerr statement in front of it to print the control variable(s) for that decision structure's condition. This will let you in on the value(s) held at that moment in the code before the condition is tested and then you will know why that condition went off or didn't as the case may be.

The only other potential problem you might face is if a loop is running too many times. In this situation, you may need to put a ignore on cin after the cerr you've placed in the loop to print its control variable(s). This will pause the program on at worst the second time through the loop and every iteration thereafter. Just remember to hit Enter/return to move on with the debug printing and the next loop iteration.[10]

3.5 More Branching

Is there more to branching than the if we've seen thus far? Of course! There are several variations on the if and we'll look at two more right now!

3.5.1 Adding an else Clause

The if is fine when you have an action to perform or skip. But what about when you have two alternative actions to decide between? We could put two ifs back-to-back:

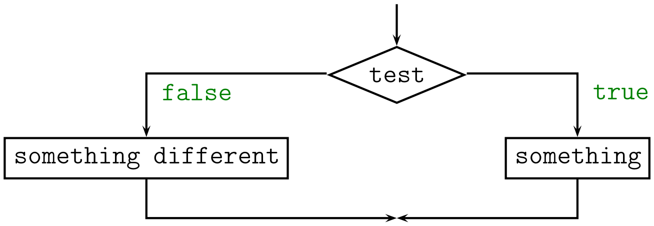
```
if ( test )
{
    // do something
}
if ( ! test )
{
    // do something different
}
```

But this makes the test twice — and once with a negation! Along comes the else clause. An else is an optional part to any if that leads to an alternate block of code.[11] This can be done like so:

```
if ( test )
{
    // do something
}
else
{
    // do something different
}
```

Note that you need not even list the alternative condition as it is always the opposite of the one that got us into the if. The flowchart for this construct looks like so:

[10] I'm assuming a full-on streamsize-maximum kind of ignore here — not just a one character removal.
[11] Code inside a pair of curly braces is called a block of code.



So what's an example of where this might be used? Let's say that we had a student's score on an exam and they were having trouble telling if they had passed or not. We'll print that message. There are two alternatives: pass or fail. So we need two branches just as with an if/else. The code might look something like this:

```
double score, max_possible;

cout << "Enter your score:  ";
cin >> score;
cout << "Enter maximum points possible on the exam:  ";
cin >> max_possible;

score = score / max_possible;

if ( score >= .7 )
{
    cout << "\nCongratulations!  You've passed!\n";
}
else
{
    cout << "\nI'm so sorry...  You've failed.\n";
}
```

Here we've got the test of whether they've passed paired with a congratulatory message and the alternative branch with an apologetic failure message.

Note on terminology: both the if and the else are called branches. But the entire thing taken as a whole is called a branching structure which many shorten to just 'branch' in practice. This is a common cause of concern and/or confusion amongst new programmers. Don't be scared off! Do some branching today!

3.5.2 Multiple Alternatives

What if the student above wanted a more fine-grained idea of their performance on the exam? Then we have five typical alternatives instead of two! How can we handle this!

One way would be nesting. Nesting is putting one thing inside each other like those Russian dolls that are so entertaining. Here, we'll put an if/else inside the else of another one. Let's start small with just three branches:

```
if ( test1 )
{
    // do A
}
else
{
    if ( test2 )
    {
        // do B
    }
    else
    {
        // do C
    }
}
```

This works fine. When `test1` fails (evaluates `false`), we enter its `else` branch and evaluate `test2`. This will choose, then, either branch `B` or branch `C`. And if `test1` succeeds (evaluates to `true`), we enter its own branch and execute whatever `A` entails. Thus only one of `A`, `B`, or `C` will execute at any one pass through this branching structure.

But it is rather bulky, isn't it? Let's pare it down, shall we?

Noticing that the only thing inside the outer `else` is the inner `if`/`else` structure — no lines fore or aft of it — we find that we can remove the 'excess' curly braces due to the above-mentioned rule on required bracing. This leaves only whitespace between the outer `else` and the inner `if` and lots of now seemingly excessive indention on the inner branching structure as a whole:

```
if ( test1 )
{
    // do A
}
else

    if ( test2 )
    {
        // do B
    }
    else
    {
        // do C
    }

```

Cutting down on the excess space, we arrive at:

```
if ( test1 )
{
    // do A
}
else if ( test2 )
{
    // do B
}
else
{
    // do C
}
```

This variation is called an `else-if` structure by many. It is also known by the archaic name of cascading or cascaded if.[12] So, how do we use this beast to tame the grade problem? It would look like this (assuming we'd already read and processed the user's score as before):

```
if ( score >= .9 )
{
    cout << "\nYou've got an A!\n";
}
else if ( score >= .8 )
{
    cout << "\nYou've earned a B.\n";
}
else if ( score >= .7 )
{
    cout << "\nYou've earned a C.\n";
}
else if ( score >= .6 )
{
    cout << "\nYou've gotten a D.\n";
}
else
{
    cout << "\nYou've gotten an F.\n";
}
```

Here we've got five branches — one for each grade letter. Only one branch can execute on any single pass through this code. The conditions are mutually exclusive and only one can be true after the prior ones have failed.

3.6 More Looping

There's no denying the while loop is excellent. And we'll explore more applications of it soon. But there are other looping structures and we want to study one more of those as well as what not to do as we go forward in this vein in this section.

3.6.1 Applications for while Loops

Let's look at a few applications suited to while loops.

3.6.1.1 The Yes/No Loop

The idea of the yes/no loop is quite simple: repeat a task until the user wishes to quit. Let's say you wanted to take the time-of-day calculating program and make it repeat as long as the user desired.

It might look something like this:

```
#include <iostream>
#include <ctime>
#include <limits>

using namespace std;

constexpr streamsize INF_FLAG{numeric_limits<streamsize>::max()};
```

[12] My teacher tried to tell me it looked like a waterfall. I couldn't find it, but the name sticks in my head now. *shrug*

```
constexpr short sec_per_min  = 60,
                min_per_hour = 60,
                sec_per_hour = sec_per_min * min_per_hour,
                hrs_per_day  = 24;
constexpr long  sec_per_day  = static_cast<long>(sec_per_hour)
                               * hrs_per_day;

int main()
{
    char yes_no;

    cout.fill('0');   // set filler to 0 digits

    cout << "\n\t\tWelcome to the Time-of-Day Program!\n\n";

    cout << "Would you like to know the current time?  ";
    cin >> yes_no;
    cin.ignore(INF_FLAG, '\n');
    while ( toupper(yes_no) != 'N' )
    {
        long sec_today = time(nullptr) % sec_per_day;

        short hour         = static_cast<short>(sec_today / sec_per_hour),
              sec_not_hour = static_cast<short>(sec_today % sec_per_hour),
              min          = sec_not_hour / sec_per_min,
              sec          = sec_not_hour % sec_per_min;

        cout << "The time is now " << hour << ':';
        cout.width(2);
        cout << min << ':';
        cout.width(2);
        cout << sec << ".\n";

        cout << "Would you like to check the time again?  ";
        cin >> yes_no;
        cin.ignore(INF_FLAG, '\n');
    }

    cout << "\nThank you for telling time with the TDP today!\n"
            "\nCome again!\n";

    return 0;
}
```

All of this is stuff we know, but we've put it together in a new way, so let's discuss it some.

I've moved the time constants outside the main along with the input constant for ignore's infinity flag. I've also added welcome and goodbye messages to the program. These reside outside the yes/no loop as they should not be repeated. Just because the entire program should be repeated, doesn't mean every line! The return 0 would be especially detrimental to the idea of a loop, for instance. It would stop the loop and the program as a whole!

But what of the loop itself? We prompt with a question of whether they want to do our task at all. Then we read a char and throw out the rest of the input line — just in case they entered a word instead of just y or n. Some people balk at my variable name: yes_no. They say, "Which is it? Don't you know?" Of course I don't! Not when I'm coding the program. I know it should be one or the other, but I don't know which it truly is!

Then we test the user's input in the while head. We use toupper to fold its case into just uppercase for our testing ease. If we hadn't, we'd have to test that it was lower or upper N to stop the program. It would have also been another DeMorgan's exercise — and who wants that, right?

Further, I'm checking for the negative response to have not happened on purpose. It helps to internationalize the program. Of all the languages I've studied, the negative response — in Romanized form — starts with an 'n'. The positive responses are all over the place, though:

Language	Negative Response	Positive Response
English	no	yes
French	non	oui/si
German	nein/nix	ja
Russian	nyet	da
Spanish	nada	si

So checking for just the N case let's many people answer our question even if their brain slips into another language. (It happens when you are studying languages a lot.)

The skeleton, for your convenience, looks like this for a typical application:

```
#include <iostream>
#include <limits>

using namespace std;

constexpr streamsize INF_FLAG{numeric_limits<streamsize>::max()};

int main()
{
    char yes_no;

    cout << "\n\t\tWelcome to the _________ Program!\n\n";

    cout << "Would you like to ______?  ";
    cin >> yes_no;
    cin.ignore(INF_FLAG, '\n');
    while ( toupper(yes_no) != 'N' )
    {
        // stuff to repeat

        cout << "Would you like to ______ again?  ";
        cin >> yes_no;
        cin.ignore(INF_FLAG, '\n');
    }

    cout << "\nThank you for ______ with the __P today!\n"
            "\nCome again!\n";

    return 0;
}
```

Remember to keep the welcome and goodbye messages outside the loop. Also make sure the initialization and update questions have the same sense. That is, a positive answer means the same thing — to keep going — on both questions.

3.6.1.2 Checking for Input Failure

Returning to a topic we talked about ages ago,[13] we'll tackle what happens when a user types something not numeric at a numeric prompt. That is, we ask for a number and they enter a letter or symbol or the like.

As we said before, this sets cin into a fail state. It won't do any more work for the rest of the program. Well, it won't do anything but a handful of functions. And it is exactly these functions can help us get it back to work! The first is the fail function. This function takes no parameters and reports with a bool that cin is currently in a failed state or not. We could use it like so:

```
short deer_in_park;

cout << "How many deer are in the park today?  ";
cin >> deer_in_park;
if ( cin.fail() )
{
    // do something!  cin is not working!
}
```

Now we need a function to fix cin and make it work again. This function is called clear. It also takes no parameters. But it doesn't have a result to store or use, either. We call it like this:

```
short deer_in_park;

cout << "How many deer are in the park today?  ";
cin >> deer_in_park;
if ( cin.fail() )
{
    cin.clear();
    // but input is still corrupt!
}
```

As indicated by the comment, however, a new attempt to read the variable deer_in_park would be thwarted by the exact same problem input the user had put the first time! The clear just makes cin forget there was a problem. It leaves the offending char in the buffer to be read again if that's what we wanted. Instead, we need to remove it so we can let the user enter the right thing. We'll use ignore for this:

```
short deer_in_park;

cout << "How many deer are in the park today?  ";
cin >> deer_in_park;
if ( cin.fail() )
{
    cin.clear();
    cin.ignore(numeric_limits<streamsize>::max(), '\n');
    cerr << "\nThat was not a number, please read more carefully!\n\n";
    cout << "How many deer are in the park today?  ";
```

[13] Or should it be pages ago? Either way. . .

```
    cin >> deer_in_park;
}
```

Here I've used the streamsize-max version of ignore to make sure they didn't enter more than a single invalid char. This is pessimistic, of course, but what's a little extra cleaning up between friends, right?

I do have one problem with the code as is, however. It only gives the user one second chance to get the information to us correctly. We should change this to a while loop to allow them many chances to fix their issue.

```
short deer_in_park;

cout << "How many deer are in the park today?  ";
cin >> deer_in_park;
while ( cin.fail() )
{
    cin.clear();
    cin.ignore(numeric_limits<streamsize>::max(), '\n');
    cerr << "\nThat was not a number, please read more carefully!\n\n";
    cout << "How many deer are in the park today?  ";
    cin >> deer_in_park;
}
```

Much better!

BTW, this loop is an excellent example of a particular kind of loop: the priming loop. The name comes from any number of physical processes which require the same basic action to get started as they require to keep going. For instance, when you need to pump water from a well, you have to have some water to wet the mechanism and form a seal before it will actually draw water up from the well. A classic engine needs a little fuel in it so it can fire off and draw more fuel from the tank. And, in olden times, we used to have to put down some rough paint to cover up the old color and make the nice paint adhere better to the wall before we could apply the nice paint. All of these actions are/were called priming the task and the verb "to prime" is/was used in general around them.

Since our fail protection loop uses the input of a numeric variable to both get going and continue around, it is also a priming loop.

3.6.1.3 Input Validation

So now that we can read numbers, how do we know that they are good for our purposes? That is, what if our formulas or application have to be over a certain domain?[14] We again turn to the while loop as we just did for fail detection and clearance. This time, though, we will simply focus on domain issues:

```
short deer_in_park;

cout << "How many deer are in the park today?  ";
cin >> deer_in_park;
while ( deer_in_park < 0 )
{
    cerr << "\nThat was not a valid number, please count more carefully!\n\n";
    cout << "How many deer are in the park today?  ";
```

[14] Remember that in mathematics, the domain is the set of values that a function is valid over. Usually this is given as one or more intervals on the number line — a subset of the real numbers, for instance.

```
    cin >> deer_in_park;
}
```

Here we trap them in the loop until they enter a non-negative value for the deer population. Zero is considered valid by design.

The while condition is the opposite of what we want to keep and this is tricky for many beginning programmers. They want to code the acceptance condition, but that would keep users in the loop when they got it right!

3.6.1.3.1 Input Validation with Failure Detection

What if we needed to make sure that a valid value was really a number — all at once? Then our friend || comes to the rescue!

```
short deer_in_park;

cout << "How many deer are in the park today?  ";
cin >> deer_in_park;
while ( cin.fail() || deer_in_park < 0 )
{
    cin.clear();
    cin.ignore(numeric_limits<streamsize>::max(), '\n');
    cerr << "\nThat was not a valid number, please be more careful!\n\n";
    cout << "How many deer are in the park today?  ";
    cin >> deer_in_park;
}
```

Please notice the subtle differences in the error messages of the last three examples. Due to the changing conditions that each reports, I felt it necessary to reword the message to more accurately reflect what the problem was and how the user should respond. Always take care to have a clear and thoughtful interface with the user.

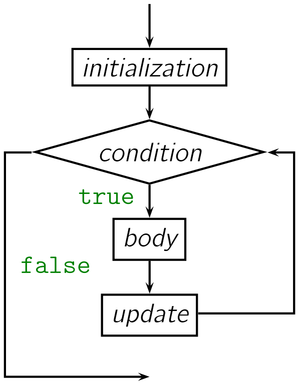
3.6.2 The for Loop

The while loop has proven itself a valiant helper in decision making during a program — a powerful tool for repetition. However, in some situations, we can use a more appropriate loop. There are three more loop constructs in C++. Let's explore one of them now.

The for loop is used when we know ahead of time the number of times to repeat a block of code. This number of times may be literal or just be able to be calculated ahead of time. It might even still be at the whim of the user to some extent.

Before we explore this a bit, let's look at the structure and flowchart of a typical for loop. Here they are:

```
for ( initialization; condition; update )
{
    body
}
```



The flowchart below should look familiar. It is a slightly enhanced version of the one for a while loop! The flow of a for loop is exactly the same. What's changed is actually as simple as collecting all the loop support parts into the head of the loop together. Support parts?

Yes — the parts that control the repetition: initialization, condition, and update.

Let's break it down. The so-called *body* of any loop is the sequence of tasks that need to be repeated — the guts and soul of the loop, really — its very reason to be! The other three listed parts are just the supporting cast to this effort. The *initialization* initializes or sets up the variable(s) used in the loop to make sure they are ready for a test. The *condition* tests the involved variable(s) to make sure it is time to repeat the *body* action(s). And the *update* changes the loop variable(s) to make sure the *condition* doesn't keep repeating the loop forever.

All loops have these four parts, by the way, the `for` loop just makes them more explicit by placing the three control parts in the head of the loop.

Maybe seeing it in action will help. Look at this loop, for example:

```
short deer_in_park;

cout << "How many deer are in the park today?  ";
cin >> deer_in_park;                                       // initialization
while ( cin.fail() )                                       // condition
{
    cin.clear();
    cin.ignore(numeric_limits<streamsize>::max(), '\n');
    cerr << "\nThat was not a number, please read more carefully!\n\n";
    cout << "How many deer are in the park today?  ";
    cin >> deer_in_park;                                   // update
}
```

In the above loop example, I've labeled the support parts with comments. The body is everything else inside the loop. The declaration of the input variable and the initial `cout` are incidental to the process.

The loop variable here is `cin`. When it inputs the `deer_in_park`, it has a chance to `fail`. When it does, we must `clear` and `ignore` and like to print a message about it to the user. These are the body. Then we read from `cin` again to give the user another chance. We loop back around and repeat if necessary.

3.6.2.1 Summation-Style Loops

Another simple loop style is best done with a `for` loop. That's the summation-style loop. These reflect the process of a finite summation from mathematics:

$$\sum_{i=0}^{n} x_i$$

Here, i advances through the $n+1$ values 0, 1, 2, ... n and a value from the sequence define by the x_is is added to a running total. It serves the same purpose as:

$$x_0 + x_1 + x_2 + \cdots + x_n$$

It just saves on space and — once you're used to it — cognitive bandwidth. We can perform this action in a simple `for` loop:

```
cout << "Please enter your six values:  ";
short n = 5;
double sum = 0.0, x;
```

```
4 for ( short i = 0; i <= n; i = i + 1 )
5 {
6     cin >> x;
7     sum = sum + x;
8 }
9 cout << "\nYour sum is " << sum << ".\n";
```

Here, we ask the user to enter the sequence of values to be added and read one per loop repetition. The support code makes sure the loop runs exactly six times. We can find this out by taking the value of `n` (5) and subtracting the starting value of `i` (0) and adding 1: $5 - 0 + 1 = 6$. Why is this? Let's follow along in a couple of different ways to see if one strikes your fancy.

First let's follow `x`, `sum`, and `i` on their journey:

x	sum	i	**Line Number**
–	0.0	–	3
–	0.0	0	4a
–	0.0	0	4b
2.0	0.0	0	6
2.0	2.0	0	7
2.0	2.0	1	4c
2.0	2.0	1	4b
1.0	2.0	1	6
1.0	3.0	1	7
1.0	3.0	2	4c

The a, b, and c on the fourth line labels are indicating the initialization, condition, and update respectively. This trace tells us that the `i` variable is initialized (4a) once and then the repetition begins by testing this value (4b). When it is `true` that `i` is less than or equal to `n`, we enter the loop body. Here we gather a new `x` value at line 6 (I've used 2 for the first value rather arbitrarily) and add it to the `sum` at line 7. After that 1 is added to `i` in the update (4c) and we return to the condition to see if another loop is appropriate.

Note that we are using a lot of vertical space with this tracking. And we still aren't done! We usually compact it to reflect each repetition of the loop together like this:

x	sum	i
–	0.0	0
2.0	2.0	1
1.0	3.0	2
4.0	7.0	3
3.0	10.0	4
2.0	12.0	5
3.0	15.0	6

This is a little harder to read, but tells us what we need to know about the number of repetitions. As you can see, `i` takes on the values 0 through 6 and when it becomes 6, the condition stops us going around again. This makes six times the `x` and `sum` variables got changed by the body.

But why is it plus one? Oh, right, well we can look at this a few ways from a mathematical standpoint also. The number of repetitions is equal to the number of values `i` takes on while the loop condition is `true`. This was when it was 0, 1, 2, 3, 4, and 5. Counting we immediately see this is also 6 items and so the problem turns into one of counting how many values are in an integer range.

So how many values are in the integer range $[a..b]$?[15] There are a, $a+1$, $a+2$, $\ldots$, $b-1$, and b.

[15] Here the .. indicates an integer or discrete interval rather than a continuous one you might have used when solving inequalities in algebra. Those use commas to separate the end-points.

This is harder to see. An attempt to compute it is, of course, $b - a$. But this turns out to be short by one. (Note our [0..5] example above.)

It is short because we've removed a entirely by the subtraction. We must therefore add it back in with the $+1$. (If we added a again, we'd just be back at b as it would add in not just a but all those integers that came before it down to 0!) This is akin to the pages problem. If the teacher tells you to read pages 95-100 tonight, you immediately think it is 5 pages. But you actually need to read page 95, too so it is really 6 pages. You have to add 1 to get the 95 — and only the 95 — back into the count.

So, in general there would be $b - a + 1$ iterations of a loop initialized to start at a and set to end when the loop control exceeded b. But is that general enough?

3.6.2.1.1 Generalizing to the Max

So what about a loop like this one:

```
for ( short i = b; i <= e; i = i + s )
{
    // do stuff
}
```

This loop will run $\left\lceil \frac{e-b+1}{s} \right\rceil$ times. Don't forget the notational symbols for the `ceil` function we learned earlier in section 2.6.2.3.

But a loop condition doesn't have to be "or equal to", of course, it can be strict as well. Does this change things? Yes, but only a little: $\left\lceil \frac{e-b}{s} \right\rceil$.

And a loop doesn't have to go up, either. After all, if we can add, couldn't we subtract? Sure! How does this change the original formula? Well, if the down-loop was like this:

```
for ( short i = b; i >= e; i = i - s )
{
    // do stuff
}
```

We would get a formula like this: $\left\lceil \frac{b-e+1}{s} \right\rceil$. And a strict comparison would take out the $+1$ as before.

Wow! This gets pretty complicated, eh? Not really. We can merge these four formulas into a single one if we adjust our thinking slightly. Let's use this formulation for the loop:

```
for ( short i = b; i C e; i = i + s )
{
    // do stuff
}
```

And let's keep in mind that `s` can be positive or negative — but not zero or we'd have an infinite loop! Also, note the placeholder `C` for the comparison. Let's make a helper variable o set to either 1 or 0 depending on if `C` is inclusive or strict respectively. Now there need be only one formula:

$$\left\lceil \frac{|e-b|+o}{|s|} \right\rceil$$

This covers all additive or subtractive loop situations. (We didn't mention the multiplicative ones, but that's a story for another time. . .)

3.6.2.2 Where'd My Variable Go?

Some of you might explore afield, of course, and find that the variable from our `for` loop disappears after the loop is done. It is only available inside the loop and in its head.[16] We could have kept using the variable had we declared it before the loop:

```
short i;
for ( i = b; i <= e; i = i + s )
{
    // do stuff
}
// can still use i
```

But this isn't the norm. Most programmers in this modern era will declare the control variable in the `for` loop head.

Note well, this cannot be done for a `while` loop! This feature of declaring the control variable in the head of the loop is only for `for` loops.

3.6.2.3 But That Update Was Icky

The update of `i = i + 1` was a little bulky for that space, don't you think? Well, you aren't alone! Also, it turns out that we update control variables by 1 a LOT. So, they made some extra operators that shorten that update area quite a bit. They are collectively known as shorthand operators. There are 8 of them we might find useful in the near future. They are:

Original Code	Shorthand
`v = v + 1`	`v += 1`
`v = v - 1`	`v -= 1`
`v = v * 1`	`v *= 1`
`v = v / 1`	`v /= 1`
`v = v % 1`	`v %= 1`
`v = v + 1`	`v++`
`v = v + 1`	`++v`
`v = v - 1`	`v--`
`v = v - 1`	`--v`

The first five are known as compound assignment operators and the last four are the increment and decrement operators.

There is one more thing about two of those last four, though. I've technically lied to say they are just the same as original code. The two I'm talking about are the ones with `++` and `--` after the variable name. And if they are by themselves on a statement or in the update area of a `for` loop, they are identical to that original code. But if you mix them with other code (an odd thing to do, but it happens sometimes), they have a slightly different meaning.

You see, all of the operators — even assignments — result in something that can be used further if need be. For instance, we can use the result of a multiply in a following addition: `a + b * c`. And we can further use the result of that addition in an assignment: `d = a + b * c`. And this goes for all operators in C++.

What of assignment's result? Isn't that the end of that statement? Not necessarily. We can end an assignment with a semi-colon, but we can also follow it up with another assignment: `a = b = c = 0`. In this expression we've assigned `c` to be 0, `b` to take on `c`'s value, and `a` to take on `b`'s value.[17] This

[16] We call the top line of a decision structure its head because the block below it is called the body. Anatomy 101, right?

[17] There's actually a little more to it than even that, but we'll discuss that another time.

form of multi-assignment is often done to initialize multiple variables to the same value at once to save typing.

So what does this have to do with v++ and v--? Well, instead of just updating the variable to be one more or less, they also have a result. The result of the prefix counterparts (++v and --v) is the new value of v. But for the postfix versions the value is the old value of the variable! This can cause trouble if it is not well understood. Make sure to comment on such use if you do it or even just see it uncommented in code in the wild.

Anyway, the above loop could have been coded as either:

```
for ( short i = 0; i <= n; i++ )
{
    cin >> x;
    sum = sum + x;
}
```

or:

```
for ( short i = 0; i <= n; ++i )
{
    cin >> x;
    sum = sum + x;
}
```

Seen in context, you may now realize why C++ has the name it does: it is incrementally better than its ancestor language C. *chuckle*

3.7 Nesting

Some flow situations call for more than a simple branch or loop. Sometimes we have to, for instance, branch within a loop or vice-versa. These situations, where one decision structure is placed inside another, are called nesting. This can lead to very interesting flow control and is well worth its own section of investigation!

3.7.1 What Can Go in What

It turns out this query is quite simple: anything can go in anything! You can put a branch inside another branch, a loop inside another loop, branches inside loops, or loops inside branches.

And the nesting isn't limited to one layer, either. You can nest as many levels deep as is necessary to capture the logic of the problem at hand.

3.7.2 Examples

Let's explore some examples of this concept to get a better feel for its complexity and utility.

3.7.2.1 Menus

A menu is a great example that shows off nesting a branch inside a loop. The branch is fairly obvious: decide what option the user chose from the menu and do something accordingly. But what loop is there? Well, consider a menu in your favorite program. After you choose something from the menu — File, perhaps — does the menu itself disappear or is it still visible at the top of the screen? It's still there! That means we need to loop around and display the menu again so the user has another chance to choose an option. This should continue until they choose to quit the program. Here's a sample menu:

```
char choice;
bool done;
done = false;
while ( ! done )
{
    cout << "\t\tMain Menu\n\n"
            "1) do Junk\n"
            "2) do Stuff\n"
            "3) Quit\n\n"
            "Choice:  ";
    cin >> choice;
    cin.ignore(numeric_limits<streamsize>::max(), '\n');

    choice = static_cast<char>(toupper(choice));
    if ( choice == '1' || choice == 'J' )
    {
        cout << "\nOption 1 -- JUNK -- Chosen!\n\n";
    }
    else if ( choice == '2' || choice == 'S' )
    {
        cout << "\nOption 2 -- STUFF -- Chosen!\n\n";
    }
    else if ( choice == '3' || choice == 'Q' || choice == 'X' )
    {
        done = true;
    }
    else
    {
        cout << "\n\aInvalid choice '" << choice << "'!!!\n\n"
                "Please try to read more carefully next time...\n\n";
    }
}
```

We see several standard features here. First of all, there is a bool variable to control the loop. This simplifies our thinking process to not think about what the user has to choose from the menu in order to quit. When they choose to quit and the proper branch is executed, we will change the value of the bool variable to stop the loop on its next test of the condition.

The next thing to notice is that we use a char variable to read the user's choice from the menu. This allows the user to enter either the number of the menu item or its significant letter. We try to design a menu with a unique significant letter in each item. This allows those who don't identify well with numbers to remember mnemonically which item they want. We've even included an ignore to make it so the user can type the significant word instead of just the letter if they are so disposed.

This also mimics the way the user can choose an application action multiple ways in a graphical interface — also known as an event-driven application. There, the user may choose via the menu in any of several ways or by using a key-combination sequence like Control+S to save a file.

Be careful when designing your menus to not only make the significant letters be unique within the menu, but also to limit the menu to 9 choices. This will let you number them with just the digits 1-9 which can all be represented as char. If you try to make a tenth item, it would either have to be two digits long — no longer a char — or it would have to be represented by 0. Normal people don't do well with items numbered 0...

Then, to make our tests easier in the if, we force to uppercase all inputs. This won't affect the number choices but will make it so we don't have to test both the upper and lower case forms of the significant letters.

Next we have our if branches to test which item was chosen. We have many == tests and we've used || to combine them so that more than one input can lead to each option's code.

We note along the way that there is an extra || on the quit option branch. This is a standard alternative quit mnemonic for exit. It was included to make our program easier for users moving back and forth between different applications all day.

Finally, we see an else branch. It's purpose is to catch any user inputs we didn't foresee. This would mean the user had selected something invalid for our menu. This doesn't happen in graphical user interfaces, but can easily in a console-based menu.

3.7.2.1.1 Asynchronous vs. Synchronous

As they are initially, menus are asynchronous in nature. That is, the user can choose any option at any time and doesn't have to choose them in any specified order.

Some situations, however, call for synchronicity. We need to have a selection, for instance, before we can copy or cut it. Normally such unavailable options (copy/cut before the selection is made) are grayed out in the menu of a graphical interface. We can't quite do that, but we can print an indicative message to the user that lets them know that an option is currently unavailable and make it so that if they choose it anyway, nothing but a message happens.

The bool Tool

bool is ideal for this situation because it can remember that something has or has not happened in its true/false nature. In fact, any time you need to remember later in a code that something happened earlier, think bool!

To enforce such prerequisites, we can use a helper bool — one in addition to the one controlling our loop. We'll set it to allow only the first in the sequence of actions to run at first. Then, once that option has been chosen, we'll change its value to allow the user to choose the second action in the sequence. Let's look:

```
char choice;
bool done, junk_done;

junk_done = false;
done = false;
while ( ! done )
{
    cout << "\t\tMain Menu\n\n"
            "1) do Junk\n"
            "2) do Stuff";
    if ( ! junk_done )
    {
        cout << " [not currently available]";
    }
    cout << "\n"
            "3) Quit\n\n"
            "Choice:  ";
    cin >> choice;
    cin.ignore(numeric_limits<streamsize>::max(), '\n');

    choice = static_cast<char>(toupper(choice));
    if ( choice == '1' || choice == 'J' )
    {
        cout << "\nOption 1 -- JUNK -- Chosen!\n\n";
        junk_done = true;
    }
    else if ( choice == '2' || choice == 'S' )
    {
        if ( ! junk_done )
        {
            cout << "\nPlease choose option 1 (junk) first...\n\n";
        }
        else
        {
            cout << "\nOption 2 -- STUFF -- Chosen!\n\n";
            //junk_done = false;
        }
    }
    else if ( choice == '3' || choice == 'Q' || choice == 'X' )
    {
        done = true;
    }
    else
    {
        cout << "\n\aInvalid choice '" << choice << "'!!!\n\n"
                "Please try to read more carefully next time...\n\n";
    }
}
```

We see now that the new bool junk_done is set to false initially because the user hasn't chosen the *Junk* option yet. Then, when the *Junk* option is chosen, we change junk_done to true to record that happening.

We also see that the menu is broken in pieces by a new if. This branch prints a message when the *Junk* branch has not been done. The message appears on the screen just after the 2) Stuff option. It says that this option is currently unavailable — our version of graying it out.

If we play out two scenarios for the *Stuff* option, we see how this all comes to a head. First, let's assume that the user chooses *Stuff* right away. junk_done will still be false and the if inside the *Stuff* branch will fire (execute) because !false is true. This will print our message that the user must choose *Junk* first and leave the branch to return to the top of the while loop.

Coming back in, let's say the user now chooses *Junk*. This changes junk_done to true. Next the user chooses *Stuff* a second time. This time, junk_done is true so !true is false and the else executes. So this time we actually perform the *Stuff* code!

What's the commented-out change of junk_done in the *Stuff* branch's else branch? That is in case you want two options to toggle off one another. Such that, once done, *Stuff* can't be done again until the user returns and does more *Junk*.

3.7.2.1.2 Sub-Menus

What about when you select one menu and it pops out another menu? Oh, sub-menus? We can do that in the console, too!

All we need do is place a whole loop/branch combo for a sub-menu into the branch for a regular menu item. This is bulky, but it works just fine. (In a little while we'll learn how to fix the bulk issue.)

What does it look like? It can look something like this:

```
char choice, sub_choice;
bool done, leaving;
done = false;
while ( ! done )
{
    cout << "\t\tMain Menu\n\n"
            "1) do Junk\n"
            "2) do Stuff\n"
            "3) Quit\n\n"
            "Choice:  ";
    cin >> choice;
    cin.ignore(numeric_limits<streamsize>::max(), '\n');

    choice = static_cast<char>(toupper(choice));
    if ( choice == '1' || choice == 'J' )
    {
        cout << "\nOption 1 -- JUNK -- Chosen!\n\n";
    }
    else if ( choice == '2' || choice == 'S' )
    {
        leaving = false;
        while ( ! leaving )
        {
            cout << "\t\tStuff Menu\n\n"
                    "1) do Dude\n"
                    "2) do Sweet\n"
                    "3) Return to Main Menu\n\n"
                    "Choice:  ";
            cin >> sub_choice;
            cin.ignore(numeric_limits<streamsize>::max(), '\n');

            sub_choice = static_cast<char>(toupper(sub_choice));
            if ( sub_choice == '1' || sub_choice == 'D' )
            {
                cout << "\nOption 1 -- DUDE -- Chosen!\n\n";
            }
            else if ( choice == '2' || choice == 'S' )
            {
                cout << "\nOption 2 -- SWEET -- Chosen!\n\n";
            }
            else if ( choice == '3' || choice == 'R' || choice == 'M' )
            {
                leaving = true;
            }
            else
            {
                cout << "\n\aInvalid choice '" << sub_choice << "'!!!\n\n"
                        "Please try to read more carefully next time...\n\n";
            }
        }
    }
    else if ( choice == '3' || choice == 'Q' || choice == 'X' )
    {
        done = true;
    }
    else
    {
        cout << "\n\aInvalid choice '" << choice << "'!!!\n\n"
                "Please try to read more carefully next time...\n\n";
    }
}
```

This sub-menu will repeat in place until the user chooses to return to the main menu. If you'd prefer the sub-menu to let the user have one shot and then return them immediately to the main menu, just take the `while` loop off. The only consequence is that the return branch of the menu's `if` structure will be empty and that's a bit odd and off-putting to a programmer. But we'll live. If you prefer, you could change its condition with DeMorgan's Laws and cut straight to the invalid message. That would keep you from having an empty branch and would give you practice at a much-used skill. *smile*

3.7.2.1.3 Options Menus

A special kind of sub-menu is one for setting up configuration options. Yeah, sometimes it is good to allow the user to tweak certain aspects of the program during the run. This ability to tweak an aspect is called configuration of the aspect. We use menu options to let the user choose which aspect to configure, so we call them configuration options.

Let's assume for a minute that the program has an integer option like the length of an output line or something. This integer will actually be a `streamsize`, of course, as it represents number of characters to display on an output stream — the size of that stream output. How might this look in the configuration sub-menu? We could do it like this:

```
    Configuration Menu
1) set Line Length
2) Return to Main Menu

Choice:
```

And when they choose the *Line* option, we read in their chosen length and move on with the program.

Or, we could make it a little more friendly and report the current setting:

```
    Configuration Menu
1) set Line Length [75]
2) Return to Main Menu

Choice:
```

This let's them know if they need to change it or not at a glance. But what if they accidentally select 1 instead of 2 anyway? Let's make sure we keep the input buffer clean with proper use of `ignore` and its parameter `streamsize`'s `max`. Then, when we go to read the line length, we can code it like so:

```
cout << "\nPlease enter new line length or <Enter> to accept "
     << line_length << ":  ";
cout.flush();
```

```
if ( cin.peek() != '\n' )
{
    cin >> line_length;
}
cin.ignore(numeric_limits<streamsize>::max(), '\n');
```

Recall that `flush` is a function for forcing `cout` to display right away. Recall also that `peek` doesn't cause `cout` to display a waiting prompt like most input does. Thus we need this to check for a newline at the beginning of the input. (If we'd used `ws` it would have thrown out the newline and hung the program waiting for more whitespace!)

Alternatively we could use the manipulator form for the `flush` function:

```
cout << "\nPlease enter new line length or <Enter> to accept "
     << line_length << ":  " << flush;
if ( cin.peek() != '\n' )
{
    cin >> line_length;
}
cin.ignore(numeric_limits<streamsize>::max(), '\n');
```

This manipulator is found conveniently in `iostream` with `cout` itself — no extra library required.

3.7.2.2 Validation Revisited

Before, we were rather pessimistic in our reaction to bad data by ignoring the entire input line instead of just the single `char` that caused the read to `fail`. We also didn't explore putting input validation together with domain checking. Let's tackle those things now.

3.7.2.2.1 Combining Validation and Domain Checking

This turns out to not be that hard, but it requires nesting.

We could take a rather extreme approach and use our validation loop for every `cin` of our domain loop:

```
short deer_in_park;

cout << "How many deer are in the park today?  ";
cin >> deer_in_park;
while ( cin.fail() )
{
    cin.clear();
    cin.ignore(numeric_limits<streamsize>::max(), '\n');
    cerr << "\nThat was not a number, please read more carefully!\n\n";
    cout << "How many deer are in the park today?  ";
    cin >> deer_in_park;
}
while ( deer_in_park < 0 )
{
    cerr << "\nThat was not a valid number, please count more "
            "carefully!\n\n";
    cout << "How many deer are in the park today?  ";
    cin >> deer_in_park;
    while ( cin.fail() )
    {
        cin.clear();
        cin.ignore(numeric_limits<streamsize>::max(), '\n');
        cerr << "\nThat was not a number, please read more carefully!\n\n";
        cout << "How many deer are in the park today?  ";
        cin >> deer_in_park;
    }
}
```

Although this will work, it isn't very pretty. Kinda bulky, in fact. Another approach is to nest an `if` inside a `while` loop with a modified condition:

```
short deer_in_park;

cout << "How many deer are in the park today?  ";
cin >> deer_in_park;
while ( cin.fail() || deer_in_park < 0 )
{
    if ( cin.fail() )
    {
        cin.clear();
        cin.ignore(numeric_limits<streamsize>::max(), '\n');
        cerr << "\nThat was not a number, please read more "
                "carefully!\n\n";
    }
    else
    {
        cerr << "\nThat was not a valid number, please count more "
                "carefully!\n\n";
    }
    cout << "How many deer are in the park today?  ";
    cin >> deer_in_park;
}
```

You can see that we used `||` to combine the conditions from the previous loops and so either of them will keep the user trapped in this loop. The other trick is that we've used a nested `if` to print just the right error message — and take any other necessary actions — for the issue that got us to keep going. This all gets the job done with much less code and is considered more elegant.

The only thing to take particular note of here is that we test for `cin` failing before we test the domain of the value. This is important to not waste time asking about the domain when the value isn't even a value yet! This kind of easy optimization should be done even at this early point in your programming career. Always think about the repercussions of your code on the user's day.

3.7.2.2.2 Really Nice fail Checking

But we are still assuming the whole line was useless if a single `char` causes failure on a number. Let's make our validation loop more friendly:

```
short number;
cout << "Enter value:  ";
```

```
cin >> number;
while ( cin.fail() )
{
    cin.clear();
    if ( cin.peek() != '\n' )  // doesn't seem possible,
    {                          // but see else below
        // x\n
        // x     \n
        cin.ignore();
        while ( cin.peek() != '\n' &&
                isspace( cin.peek() ) )
        {
            cin.ignore();
        }
        if ( cin.peek() == '\n' )
        {
            cout << "\nInvalid numeric format!!!\a\n"
                    "\nTry again:  ";
        }
        // x     9...\n   (clears but no message -- yet)
    }
    else     // numeric input is 'consumed' even if it
    {        // could not be stored
        cout << "\nNumber magnitude too large!!!\a\n"
                "\nTry again:  ";
    }
    cin >> number;
}
cin.ignore(numeric_limits<streamsize>::max(), '\n');
```

Here we start by clearing the failure so we can find out what char caused the problem. Strangely, if it was a number but just couldn't fit in our variable, the input is removed from the buffer, but the newline from the user's Enter/return is still left behind. Thus the peek check for a newline right off after we clear cin.

If we don't find a newline, things are more complicated. Here we have to throw out the offending char with a plain ignore and then look for empty space after that. Why should we look for the extra spaces? What if the user typed some 'stuff' and then later the number we were looking for? It could happen! We should be careful. Since, however, there might not be a number after the spacing, we use our while version of ws that respects the end of line mark (newline) and stops accordingly. Only if we get to the newline after that loop do we print a message.

You should try this out and see how much friendlier it can be. Try some values like these:

```
32768
-32769
flower
$32
```

3.7.2.3 2D Printing

Another place to use nesting is printing things that are two-dimensional like an addition table. This will involve at least a pair of nested loops — most likely for loops.

The idea here is to produce a table like so:

```
 + | 1 2 3 4
---+---------
 1 | 2 3 4 5
 2 | 3 4 5 6
 3 | 4 5 6 7
 4 | 5 6 7 8
```

We'll start by printing the top two lines which don't repeat themselves:

```
cout << " + |";
for ( short row = 1; row <= 4; ++row )
{
    cout << setw(2) << row;
}
cout << "\n---+" << setfill('-') << setw(2 * 4) << "-" << '\n'
     << setfill(' ');
```

Here I've used `setw` because I had literal widths and those are plain `int` anyway.

Next comes the rows of the table. These involve two values running in a particular pattern:

```
row| col
---+---------
 1 | 1 2 3 4
 2 | 1 2 3 4
 3 | 1 2 3 4
 4 | 1 2 3 4
```

The `col` value needs to run through all its values for each value taken on by `row`. To accomplish this, we'll have to nest one `for` loop inside another:

```
for ( short row = 1; row <= 4; ++row )
{
    cout << setw(2) << row << " |";
    for ( short col = 1; col <= 4; ++col )
    {
        cout << setw(2) << row + col;
    }
    cout << '\n';
}
```

Now you can see that the `col` loop will run through all of its values before `row` increments to its next value.

If we wanted to augment this to allow the user to enter the upper bound on the table size, we'd need to account for sums of at least two digits (like 5+5), we would need to make a `streamsize` type value and use `cout`'s `width` function instead of using `setw`. This is because the width of each sum would be variable depending on the limit the user entered.

We could handle this in two ways. One would be a simple `if` that would be limited to our industriousness to type in possibilities. But it would also be limited by the screen's size, so it wouldn't be too horrible. The other is to learn a new tool to calculate the number of digits it takes to print a number on screen. Doing so we could figure out algorithmically how wide to make each column in the table.

The branch approach might look like this:

```
if ( size+size < 10 )
{
    col_width = 2;
}
else
{
    col_width = 3;
}
// no need for any more due to screen size limitations
```

Here size can be a short and is the user-entered table bound. col_width, on the other hand, is our streamsize variable for the width of each column. Simple enough. This would precede the for loop that prints the top row of the table as we need it to space those column headings, too.

The new tool approach uses everyone's favorite math function: logarithms! Due to the work of Claude Shannon in the field of information theory (which he kinda founded), we know that the number of [base-ten] digits in a number is:

$$\lfloor \log_{10} x \rfloor + 1$$

Don't forget the notational symbols for the floor function we learned earlier in section 2.6.2.3. (This calls for the floor of the logarithm to the base in which we want to represent the number, btw. If you want to find out how many binary bits it takes to represent x, just change the logarithm base to 2.)

Trying it out to satisfy our curiosity, let's try 10: $log_{10}10 = 1$ and the floor of 1 is 1. Adding 1 we get 2 and it takes 2 base-10 digits to represent 10! If we try something smaller than 10, we get a logarithm that isn't quite 1 — a fraction between 0 and 1. The floor of this would be 0 and adding 1 would get us 1.

```
       x: 1 2  3  4  5  6  7  8  9  10 11 12 ... 99 100 101 ...
log10(x): 0 0. 0. 0. 0. 0. 0. 0. 0. 1  1. 1. ... 1. 2   2.  ...
floor( ): 0 0  0  0  0  0  0  0  0  1  1  1  ... 1  2   2   ...
      +1: 1 1  1  1  1  1  1  1  1  2  2  2  ... 2  3   3   ...
```

The only exception to this rule is 0 which takes 1 digit but whose logarithm we can't take. (And if you ever need to find this out for a negative value, just take its absolute value before the log and add 1 more for the negative sign which takes up a little more space.)

So, in code, this would look simply like so:

```
// number of digits +1 for the spacer
col_width = static_cast<streamsize>(floor(log10(size + size)) + 1) + 1;
```

The typecast is needed to make the double from floor fit properly into the right type to store in our column width variable. Again, this would go right above our display of the top row of the table.

Altogether it might look like this:

```
short size;
streamsize col_width;
cout << "What is the bound on your addition table?  ";
cin >> size;
while ( cin.fail() || size <= 0 || size > 80 / 3 - 2 )
{
    cout << "\nThat ";
    if ( cin.fail() )
    {
        cin.clear();
        cin.ignore(numeric_limits<streamsize>::max(), '\n');
        cerr << "wasn't a number";
    }
    else if ( size <= 0 )
    {
        cerr << "value was too small";
    }
    else
    {
        cerr << "value was too large to fit on screen";
    }
    cout << "!\n\nWhat is the bound on your addition table?  ";
    cin >> size;
}
// number of digits +1 for the spacer
col_width = static_cast<streamsize>(floor(log10(size + size)) + 1) + 1;
cout.width(col_width+2);  // +2 for the space and bar
cout << " + |";
for ( short row = 1; row <= size; ++row )
{
    cout.width(col_width);
    cout << row;
}
cout << '\n';
cout.fill('-');
cout.width(col_width+2);  // +2 for extra - and
cout << "---+";
cout.width(col_width*size+1);  // +1 for an extra - at the end
cout << "-" << '\n';
cout.fill(' ');
for ( short row = 1; row <= size; ++row )
{
    cout.width(col_width);
    cout << row << " |";
    for ( short col = 1; col <= size; ++col )
    {
        cout.width(col_width);
        cout << row + col;
    }
    cout << '\n';
}
```

Here I've added validation on the table size and made the bar of dashes match the size of the numeric parts of the table. I could have just checked `== 0` if I had made `size` `unsigned short` instead, but I was feeling too lazy to type `unsigned` at the time. *smile*

3.7.3 What NOT To Do

Once you start learning to nest branches inside repetition structures, various forces will conspire to bring you over to the dark side! You must not listen to them! Hold strong!

The things they would have you do involve three C++ commands that we inherited from our C ancestors and persist to this day in our midst. We shun them, but still they linger in the shadows and alleyways.

They are goto, continue, and break. These commands will take your program to new depths of insanity! They cause control of the code to evaporate into nothingness and leave strong programmers whimpering at their desks.

No, seriously. They do cause brain rot and should never be used. We even planned against these constructs since the '60s when Edsger Dijkstra wrote his seminal paper ”Goto Statement Considered Harmful”.

How can we get rid of them?! Never use goto — EVER!!!

As to continue and break, it just needs a little tender-loving care.

3.7.3.1 Removing a continue

For instance, were you to have this kind of code in your program:

```
while ( C )
{
    A
    if ( C2 )
    {
        continue;
    }
    B
}
```

It would normally cause the program to skip the B code when the condition C2 was true. The program would instead jump straight to the C test at the top of the while. To avoid this non-sense, just change the code to this:

```
while ( C )
{
    A
    if ( ! C2 )
    {
        B
    }
}
```

Now the code does the same exact thing, but there is no crazy jumping around!

3.7.3.2 Removing a break

Similarly, if you find this in your code:

```
for ( I; C; U )
{
    A
    if ( C2 )
    {
        break;
    }
    B
}
```

We would, under the condition C2 being true, stop the for loop prematurely. Instead of this craziness, we can change it up to:

```
I
t = false;                  // t is a temporary bool variable
while ( C && ! t )
{
    A
    t = C2;
    if ( ! t )
    {
        B
        U
    }
}
```

This has the same effect as the for loop with a break, but without the jumping around nonsense.

3.8 Standard Libraries II

Two of the most popular styles of programming are procedural and object-oriented programming. Procedural programming style focuses on procedures (actions) involved in a process and then applies those to data. Object-oriented programming style focuses on the data necessary to describe a process and then looks at actions that data may be involved with.

The programming we've done so far has been core to both of these styles. But we'll now be studying more about the overlaying styles. We'll look into OOP first in this section — but just a surface run. Then we'll dive headlong into procedural programming in the next chapter (4). And finally we'll come back to OOP for a deep dive in the chapter after that (5).

3.8.1 OOPs

C++ implements OOP style by allowing the programmer to define their own data types. Such types are called classes and their definition describes both the 'physical' and 'behavioral' aspects of the data.

C++ chose the term 'class' for such data types to bring to mind classification systems such as the one used in the biological sciences to classify animals into domains, kingdoms, phyla,[18] etc. The individuals described by such a class are termed the objects or instances of that classification. (You can also use the old terms 'type' instead of class and 'variable' instead of object if you like.)

We've already looked at two classes for our console interaction: those for input and output streams. In particular we've looked at the individual objects cin and cout (and a little at cerr). We've come to find that there are various syntaxes involved in using class objects that are different from typical procedural programming syntaxes: dots were a major factor, you may recall.

Well, now we're ready to tackle a new class: the string class. This will take us to a whole new level of OOP usage.

[18] That really is the plural of phylum!

3.8.2 The string class

The string class from the string library has facilities for constructing string objects, output of string objects, input of space-separated string values, assignment of string objects, concatenation of strings, subscripting a string object to access a particular character, and comparing two string objects to one another lexicographically (sort-of alphabetically). As well as utility functionality such as determining the length/size of a string object, determining if a string object is empty or not, inserting a new character sequence into a string object, erasing character sequences from a string object, and replacing one character sequence within a string object with another.

In addition to these things, the string class also provides enhanced subscripting, assignment, and input facilities, comparison facilities, and searching facilities.

3.8.2.1 Declaration (aka Construction)

Let's start at the beginning with creating a string object in the first place. We normally declare a variable and strings are no different. We just sometimes call declaring a string constructing the string or even defining the string instead.

The syntax can be very simple:

```
string s;
```

This declares a new string-typed object (variable) named s. It is by default empty (aka ""). This is a little different than with our built-in types which, of course, had no value by default. But strings know about values from the get-go and clean up their memory area to have a nice place to store your information when it gets here.

Or we could initialize the object with a value:

```
string t = "Hello ";
```

This declares t as a string-type object and initializes it to the value "Hello " (note the space).

We can even construct constants of the string type:

```
constexpr string PROGRAM_TITLE = "string mangler";
```

And, you can initialize one string object to be an exact copy of another object:

```
string u = t;
```

In fact, this and the initialization from a literal string have another syntax that you can use, too:

```
string t("Hello ");
string u(t);
```

Some people like this syntax rather than the = syntax to remind them that initialization is not assignment (more on that later...). But it brings to mind for me one more construction possibility — one that requires the parentheses syntax:

```
constexpr string BORDER(70, '*');
```

Here BORDER is constructed as a sequence of 70 stars (aka asterisks). The parentheses are required, of course, because if we used = syntax, the definition would read:

```
constexpr string BORDER = 70, '*';
```

And now we are trying to initialize BORDER (a string object) with the integer 70 and then move on to declare a second string constant whose name is '*' — highly illegal!

Of course, we can also use the curly-brace syntax of initialization for initializing string objects as well. But for that last form, we really need parentheses. If we used curly-brace syntax like this:

```
constexpr string BORDER{70, '*'};
```

We'd end up with a string containing an F and a star! Why? The compiler takes this to mean a string containing the following list of chars and 70 is the ASCII code for a capital F. (The coercion is automatic and silent. Kinda annoying if not deadly!)

3.8.2.2 Displaying strings

Displaying strings is as easy as with the builtin types. We just need to insert it using the insertion operator (<<):

```
cout << t << s << "!\n";
```

or to display our border between program segments on-screen, we could do:

```
cout << '\t' << BORDER << '\n';
```

As you can see, it mixes with other types of data just fine.

3.8.2.3 Assigning strings

During the run of the program, you can change one string to look like another string value with the assignment operator — just like you've done for built-in types:

```
s = "new stuff";
```

or:

```
s = t;
```

or even:

```
s = "";
```

That last one would empty the string out like it was just now default constructed.

3.8.2.4 Inputting strings

Alternatively, you can extract the user's information from an input stream into a string variable to give it a value. Just use the extraction operator (>>) that we use with the builtin types:

```
cin >> s;
```

The one thing to remember is that extraction is a whitespace hater and will not ever store spacing into even a `string`! It will separate the user's input into 'words' at space boundaries and bring them in one at a time into your `string` object(s).

3.8.2.5 Concatenating strings

You may be wondering, how do we read space-containing data like addresses into a `string` if extraction stops at them and skips leading ones? Before we get into that, we need to talk concatenation.[19]

To attach `strings` one to the end of the other, we use the + operator. For instance:

```
s = "Jason";
cout << t + s << "!\n";
```

would print `"Hello Jason"` followed by an exclamation mark and newline. This doesn't harm the two concatends![20] It merely makes the concatenated result. So this:

```
t + s;
```

will compile but has no visible effect...

To store the result rather than print it, you could, of course use the assignment operator:

```
u = t + s;
```

(*snaps fingers* Now I lost that spare copy of `"Hello "` I was saving...*shrug*) And now I can still print it:

```
cout << u << "!\n";
```

or I could use concatenation some other way(s)...

3.8.2.6 Reading strings Containing Spaces

Let's start by reading all the words on a line and printing them back out to the user:

```
string word;

cout << "Enter your sentence/phrase:  ";
cin >> word;

while ( cin.peek() != '\n' )
{
    cout << word << '\n';
    cin >> word;
}
cin.ignore();           // throw out newline
cout << word << '\n';   // prints the last word -- loop stopped because
                        // this word was trailed by a newline...
```

[19] Concatenation is a fancy word for "attach one thing to the end of another". Programmers love this word and use it all the time. Get used to it.

[20] Just kidding. That's not a real word. But it sounds good — like addends in addition, right?

It doesn't work perfectly. The user has to hit Enter/return right after their last word, for instance. But it is good enough for us to work with and build from. Also note that we print as we read because we can only store one at a time in a single variable like this.

This hurts some student's heads as they think we are printing as the user is typing. This is not the case. We are printing as we are reading and our reading doesn't start until the user has hit Enter/return and are done typing.

```
string word, line;

cout << "Enter your sentence/phrase:  ";
cin >> word;
line = "";
while ( cin.peek() != '\n' )
{
    line += word + ' ';
    cin >> word;
}
cin.ignore();                // throw out newline
line = line + word + '\n';   // stores the last word -- loop stopped because
                             // this word was trailed by a newline...

cout << "You entered:\n\t" << line << '\n';
```

Here we have added a second string named line which accumulates — summation style — the words and spaces between them. (The reason this isn't a for loop like our earlier summation examples is that this one isn't bounded by a known limit. We have no idea ahead of time how many words the user will enter on a line. Since it is an unknown number of repetitions, we use a while loop instead.

3.8.2.6.1 A Deeper Look at Concatenation

I'd like to also have a word with you about that concatenation of a space character. In the spirit of full disclosure, however, I feel I should tell you that concatenation is pretty free form within certain limitations.

Let s be a string object, c a char, and L a literal string. Then you can concatenate:

```
    s1 + s2
    s  + c
    c  + s
    s  + L
    L  + s
```

But you cannot concatenate just chars or string literals:

```
    L1 + L2
    L  + c
    c  + L
    c1 + c2
```

To accomplish these needs a helper string object:

```
string t = L1;
   .... t + L2 ....
```

or:

```
string t(L);
.... t + c ....
```

for instance.

Others of these are trickier:

```
string t = c;
```

will fail miserably... This is because, there is not a constructor that takes a single `char` and makes it a `string`! Odd, but true...

Thus we must fall back on our old friend:

```
string u1(n, c);
```

where `n` is a non-negative integer and `c` is a `char`. We'll just use 1 for the integer so we only get one copy:

```
string t(1, c);
.... t + L ....
```

or:

```
string t(1, c1);
.... t + c2 ....
```

And now they work fine.

Now that the first pair of concatends is fixed up to have a `string` object, the result will of course be a `string` object and we can continue using it to concatenate any `string`, literal string, or `char` that we need or want!

That is, we could have done that long ago display of `"Hello Jason!\n"` as all concatenation:

```
cout << t + s + "!\n";
```

This evaluates as:

```
cout << ((t + s) + "!\n");
```

Or there's even:

```
cout << t + s + '!' + '\n';
```

which evaluates as:

```
cout << (((t + s) + '!') + '\n');
```

Not that you necessarily would, but you could!

But, to make things a little easier than stopping and making all these helper variables — how would you name them, anyway?! — we'll learn a new trick called anonymous construction! This looks like you are declaring a variable (object) but you just don't give it a name!

```
string(1, c) + L
string(L) + c
```

Note how we have just the syntax for declaring our temporary variable but without its name. Thus both `string(1, c)` and `string(L)` will be anonymous objects — memory space without a name.

We could even use this to make those borders mentioned above without having to come up with a name or making it a constant or such:

```
cout << '\t' << string(70, '*') << '\n';
```

3.8.2.6.2 Back to Our Goal

But back to our goal of reading the user's input as a single `string`. Another technique for gathering the user's multi-word data that will respect their relative spacing — the amount (and even type of) space that they place between words, before words, even after words![21]

This technique uses the function `getline`. It comes from the `string` library and gathers an entire line of input text into a `string` from an input stream. In its simplest form, we could just call it like so:

```
getline(cin, s);
```

This reads everything from the given input stream (`cin`) and up to a newline (`'\n'`) and stores it into the given `string` object (`s`).

The newline is then removed from the stream, but not stored into the `string`.

To End, But Where

By default, `getline` ends its reading at a `'\n'` character. That is the way `cin` signals that the user hit Enter/return, after all. But in some situations, we've found that we like to end at some other character. So `getline` can be called with three parameters: the input stream from which to get data, the `string` in which to store the data, and a `char` at whose input we stop — a terminator of the input, if you will. As with the `'\n'`, this terminating `char` is removed from the input stream but not stored in the `string`.

Why didn't they code it as `cin.getline`? Suffice it to say that both objects were deemed focal to the process — both `cin` and `s` here — and so neither could be chosen as the object to 'dot' for the call. Therefore they were both sent in as input to the function: where to retrieve the `char`s from and where to store them to, respectively.

But this has a weird issue. If you perform the `getline` call 'immediately'[22] following an extraction (>>), `getline` will return right away and give you an empty `string`. It won't even cause the program to pause and let your user read the prompt, much less type anything in reply. That's because extraction always leaves behind a straggling newline — from when the user hit Enter/return earlier. That whitespace just signaled the end of the translation to >> and was left behind in the input buffer to await later pointing and laughing as it attempted to be input.[23]

[21] I know it doesn't seem like much to respect their spacing that much, but it can be important in certain applications/situations.

[22] Immediately in the eyes of `cin`. Nothing else has happened to `cin` in the intervening statements of the program, that is.

[23] Just kidding again. No one is in the buffer area pointing and laughing. But it feels like it sometimes when we make a mistake, doesn't it?

But then you used getline which tries to respect all spacing except it uses the newline specially as a signal to stop storing characters into the specified string. When it sees the newline as its first input char, it says, "Hey, I'm done!" and sends you back an empty string as proof of its hard work.

Ideally the programmer responsible for the extraction could be goaded into cleaning up after themselves, but this may not be feasible. So often we must clean up the stray newline ourselves.

A proactive approach to this would involve peeking for a newline and throwing it out before our getline attempt. But a sudden peek isn't always respectful of prompt displaying issues, as we well know. And we can't just throw out the whitespace with ws as we did before because we are trying to respect the user's relative spacing — the whole reason for this getline nonsense! So we'll have to tell cout to display any waiting prompt ourselves with a flush:

```
cout << "Enter your sentence/phrase:  ";
cout.flush();             // or cout << flush;
if ( cin.peek() == '\n' )
{
    cin.ignore();
}
getline(cin, line);

cout << "You entered:\n\t'" << line << "'\n";
```

We could instead take a more reactive stance:

```
getline(cin, t);
while ( t.empty() )   // t.length() == 0
{
    getline(cin, t);
}
```

Not always the best tool, but useful in some situations. Mainly it gives us the chance to learn the empty and length functions from the string class. empty reports whether the calling string has no characters right now — a true/false result, of course.

length, on the other hand, returns the number of characters currently in a string object. Which is better? That's for you and your programming team to decide in the moment!

3.8.2.7 Return to Menus

When last we left menus, we had implemented synchronicity and sub-menus. Now let's take another look at the synchronicity code to see what the string class can do for us there.

Our last stab at synchronicity had this as the menu display:

```
cout << "\t\tMain Menu\n\n"
        "1) do Junk\n"
        "2) do Stuff";
if ( ! junk_done )
{
    cout << " [not currently available]";
}
cout << "\n"
        "3) Quit\n\n"
        "Choice:  ";
```

That's all well and good, but why break up a perfectly good `cout` like that? If we install a new `string` variable for that currently available message, we can streamline things a bit:

```
string junk_message = " [not currently available]";

cout << "\t\tMain Menu\n\n"
        "1) do Junk\n"
        "2) do Stuff" << junk_message << "\n"
        "3) Quit\n\n"
        "Choice:  ";
```

then just a tweak in the *Junk* branch:

```
if ( choice == '1' || choice == 'J' )
{
    cout << "\nOption 1 -- JUNK -- Chosen!\n\n";
    junk_done = true;
    junk_message = "";
}
```

If you are toggling instead of just synchronizing, we need to tweak the *Stuff* branch, too:

```
else if ( choice == '2' || choice == 'S' )
{
    if ( ! junk_done )
    {
        cout << "\nPlease choose option 1 (junk) first...\n\n";
    }
    else
    {
        cout << "\nOption 2 -- STUFF -- Chosen!\n\n";
        //junk_done = false;
        //junk_message = " [not currently available]";
    }
}
```

But, as we implement that last change, we get an idea! The `junk_done` variable is kinda redundant now. With `junk_message` in tow, we don't need the second variable. We could just check whether or not `junk_message` was empty instead of whether `junk_done` was `true` or `false` everywhere:

```
if ( choice == '1' || choice == 'J' )
{
    cout << "\nOption 1 -- JUNK -- Chosen!\n\n";
    junk_message = "";
}
else if ( choice == '2' || choice == 'S' )
{
    if ( ! junk_message.empty() )
    {
        cout << "\nPlease choose option 1 (junk) first...\n\n";
    }
    else
    {
        cout << "\nOption 2 -- STUFF -- Chosen!\n\n";
        //junk_message = " [not currently available]";
    }
}
```

This simplifies our life and our code.

3.8.2.8 string Comparison

To compare two string objects we can use all the normal comparison operators:

```
    ==   !=   >   >=   <   <=
```

All of these work and give ASCII-betical results as per typical dictionary order. (Just like they did with char, but with more oomph!)

That is, "Apple" < "apple" would result in true.

You can even compare a string object to a literal string or to a single char. I'm not sure why you'd do the latter, but you could... And you can still compare two char values to one another. But you cannot compare two literal string values to one another. That apple thing above was just a 'for example' — not real code. To perform such a test, you would have to store one of the two in a string object (named or anonymous):

```
string t{"Apple"};
if ( t < "apple" )
{
    cout << "This always executes...\n";
}
```

You also can't compare chars to literal strings. But why would you want to do that, anyway? (You can still use a helper object or an anonymous object to make these comparisons work correctly, but they'll still be in ASCII-betical order.)

But there is one other way to compare strings: the compare function. The compare function is like all six comparison operations combined into one action! In order to do this, it needs more than a mere bool result, of course, so it uses a small integer (we'll consider it a short). The way to use this result is summarized by the following table:

I want to know if...	So I code...
s1 < s2	s1.compare(s2) < 0
s1 == s2	s1.compare(s2) == 0
s1 > s2	s1.compare(s2) > 0

You can even do multi-combinations, of course, like if you wanted to know that the first string was less than or equal to the second (s1 <= s2), you could code to check .compare()'s result against 0 with <= like this:

```
if ( s1.compare(s2) <= 0 )
{
}
```

for instance.

So this combining of the comparison results into a single integer makes the compare function more efficient than having to call on any pair, trio, or even more of the comparison operators. For example, to completely piece out the relationship that two `string` values have, we'd need to code:

```
if ( s1 < s2 )
{
    cout << "first is lexicographically before second\n";
}
else if ( s1 > s2 )
{
    cout << "first is lexicographically after second\n";
}
else  // s1 == s2 by necessity
{
    cout << "first is lexicographically the same as second\n";
}
```

This must on occasion evaluate two comparisons to properly orient the `string`s. (Note that this is the pattern we often follow because it works with almost any kind of data and helps with those tricky floating point data to avoid having an == test involved.)

With the `.compare()` function, I can instead code:

```
short comp_res;

comp_res = s1.compare(s2);
if ( comp_res < 0 )      // s1 < s2
{
    cout << "first is lexicographically before second\n";
}
else if ( comp_res > 0 ) // s1 > s2
{
    cout << "first is lexicographically after second\n";
}
else  // s1 == s2 by necessity
{
    cout << "first is lexicographically the same as second\n";
}
```

And thus only have to compare the `string`s themselves once but still have all the relevant information for my processing needs.

Is this efficiency necessary? Yes, because, as you'll see shortly, comparing `string`s can be terribly expensive.

But it is still ASCII-betical in nature. *sigh*

So we should probably fix that, hunh? Let's look at how compare or any of the comparison operators do their job and see if we can come up with a way to fix it.

Efficiency is, as usual, our goal. So what do these comparison functions do? Well, they line the strings up next to one another and look at them char-by-char until a difference is encountered and then use the way those two differing chars compare to decide the string pairs' result. For instance:

```
Apple
apple
^
difference!  and 'A' is < 'a', so "Apple" must be < "apple"
```

or:

```
apple
application
^||||
 ^|||
  ^||
   ^|
    ^
    difference!  and 'e' is < 'i', so "apple" must be < "application"
```

or even:

```
apple
apple
^|||||
 ^||||
  ^|||
   ^||
    ^|
     ^
     ran out of string!  always ==, so "apple" must be == to "apple"
```

Etc. To do such a thing, we'll need several helpers. One of them we already know: `while`. We'll need it to walk from `char` to `char` along the aligned `string`s. But we'll also need to be able to do other things: tell what an individual `char` from a `string` is, tell how many `char`s are currently in a `string`, and . . . well, the other one will become more obvious in context.

So let's think about this step by step. We need to walk from one position of the `string`s to the next until we either reach the end or find a difference in the characters at the two aligned positions. We could use our `while` loop to do this if we just knew those other pieces: what are positions within the `string`, how many of them are there, and how do we specify which one we want to look at?

The positions within the string turn out to be offsets or distances from the beginning of the `string`. That is, instead of numbering the relative positions of characters within the sequence starting at 1 like many folks would, computer scientists number them starting with 0. Here is a diagram of how a comp-sci looks at a `string` in memory:

```
+===+===+===+===+===+
| a | p | p | l | e |
+===+===+===+===+===+
  0   1   2   3   4
```

The mathematicians would use subscripts to distinguish the individual characters from one another — s_0, s_1, s_2, s_3, s_4 — but we can't really do that in a plain text environment, can we? So, instead, our ancestor C programmers decided that the logical alternative to subscripts was to place the relative position inside a pair of square brackets. Let's say that the `string` `"apple"` was named `s`, then we could access the `'e'` by:

```
s[4]
```

This is still called subscripting despite the syntax. But it is also known as indexing. 4 is the subscript or index, s is the subscripted or indexed variable/object, and 'e' would be the result.

But if we are going to programmatically walk a while loop from one position to the next, we need to have a variable to control this walk: a loop control variable (LCV)! And if we need a variable, we need to first know its data type — for declaration purposes.

The most appropriate data type would obviously be some kind of unsigned integer. It doesn't need negatives, after all since we start at 0. And it doesn't need decimals as each position is a discrete jump from the previous and next.

But which one? We have 4 unsigned integer types: unsigned short, unsigned int, unsigned long, and unsigned long long. Well, the string class has taken over that decision for us, thankfully. They have made a decision based on the specific characteristics of the system you are compiling to that will be appropriate for any string that system can hold.

To keep you from further worry about the issue of which unsigned integer type was chosen, they even gave it a platform-independent name: size_type. (This is analogous to how the ctime library's time function named its resulting type time_t so you didn't have to worry what type was big enough to hold the seconds since the epoch.)

They named it size_type because any type capable of holding the size of the string would also be capable of holding all the positions leading up to that value. But there is a slight complication. They put this alias for the underlying unsigned integer inside the string class. This means our notation for using this data type name is going to be string::size_type. Isn't that horrible looking?

(This could have been worse, if you have to access this data type name without a using directive in effect, it is truly: std::string::size_type. After all, the string class that size_type is inside of is itself inside of the namespace std..!)

Similar to how we made our own constants before to ease use of the ios_base:: flag constants for stream formatting, we can use one of two other facilities to rename the string's size_type alias to our own type name: typedef or using aliasing. typedef makes a type definition. In our case, we are simply defining our type name to represent the same thing as a previously known type — renaming another type. So, for instance, you might do:

```
typedef string::size_type StrSz;
```

or:

```
typedef string::size_type StrPos;
```

Or both! (It depends on if you want to focus on this type being for sizes of string objects or for positions within them.)

Notice that if you cover the typedef keyword itself, the rest will look just like a variable declaration. (Assuming you can make yourself realize string::size_type is the name of a type. And that is no small task for many students!)

Place this statement between your using directive and your main's head to rename the string's size_type alias to some name you can more easily remember/type. (Like constants, typedef's can be placed globally — outside the main function. We never do this with variables and we'll talk about the problems that lead us to that decision in Chapter 4.)

The using alias is similar to this[24] It makes an alias for the given type but its syntax is inside out

[24] But different from a using directive!

from the typedef:

```
using StrSz = string::size_type;
using StrPos = string::size_type;
```

But we need one more piece of info before we can form our first approximation to the string comparison loop: how long is the string? Without this, we cannot stop before accessing outside the string — a memory access violation waiting to happen![25]

Turns out, the string class provides both the .size() and .length() functions toward this purpose.[26] In fact, they both return the exact same result. There are two of them to make each programmer feel comfortable in their word choice for their situation. (Weird, no?)

So, without further ado, I present you the ”walk through a string” loop:

```
string s;
string::size_type c{0};
while ( c != s.length() )
{
    // use s[c] somehow
    ++c;
}
```

At the end of this loop, c will be equal to s.length() (or s.size() if you prefer). And at each position we can use c to subscript the string toward some end.

Now let's put in the parallel/aligned string:

```
string s, t;
string::size_type c{0};
while ( c != s.length() &&
        c != t.length() )
{
    // use s[c] and t[c] somehow
    ++c;
}
```

Note that DeMorgan's laws make us use an && here to combine our boundary tests. We want the loop to end when either string's boundary has been breached (c == ?.length()) and so that's:

```
c == s.length() || c == t.length()
```

But we want the loop to continue in the opposite situation and so the == tests must be negated and the || must transform to an && as well!

Now, to find the difference between our aligned characters — s[c] and t[c] — we just need to compare them. We'll initially record this comparison in a bool variable that indicates ”all previously inspected parallel character pairs were equal”. But that's quite the mouthful. It also could be seen — opposingly — to represent that ”a difference in the parallel character pairs has been encountered”. This is easily shortened to diff_found like so:

[25] This kind of violation is often known as a segmentation fault because you go outside your segment of memory.
[26] We actually saw .length() already, but .size() works, too.

```
string s, t;
string::size_type c{0};
bool diff_found{false};
while ( c != s.length() &&
        c != t.length() &&
        ! diff_found )
{
    if ( s[c] != t[c] )
    {
        diff_found = true;
    }
    ++c;
}
```

Here diff_found is initially false because we haven't inspected any character pairs and so there have been no differences spotted as yet. We && it to the boundary tests to make it of equal importance in stopping our loop as soon as a difference is spotted. But it is negated because we want to keep going when we haven't yet seen a difference — the pairs of characters are still all equal.

The nested if takes care of resetting the value of diff_found. But, truly, it isn't necessary. We could equally well have coded it like so:

```
string s, t;
string::size_type c{0};
bool diff_found{false};
while ( c != s.length() &&
        c != t.length() &&
        ! diff_found )
{
    diff_found = ( s[c] != t[c] );
    ++c;
}
```

It does, after all, store the truth value of the != comparison in diff_found — we had just cut out the else branch that would have stored false because it was already false from initialization.

But now we can more easily see that this is really an integral part of the loop control rather than internal to the loop body. After all, we want to stop as soon as we find the difference? So, we should probably code more directly:

```
string s, t;
string::size_type c{0};
while ( c != s.length() &&
        c != t.length() &&
        s[c] == t[c] )
{
    ++c;
}
```

Note how the != became an == since we had been negating diff_found before... (This is a safe combination because the &&s will short-circuit to false and stop the while loop before subscripting with an illegal position. If we'd commuted the && clauses, we might accidentally walk outside the strings' boundaries before checking this possibility! Remember the segmentation fault? It turns out that even small things like the order of tests being &&'d together is important!)

Next we need to post-process this loop will make the final decision as to whether or not the two strings are <, ==, or > and in what order. But when the loop ends, we just know that we've found the end of a string or the position of difference — we don't know yet which it was that stopped the loop. (Also note that the second occasion was aided by the merging of the difference finding into the loop head to avoid the extra position bump! That is, we didn't update c again after the diff_found would have caused the loop to stop anyway.)

To recap our cases:

strings	lengths	End Position
apple	5	
vs		4
application	11	
apple	5	
vs		0
Apple	5	
apple	5	
vs		3
app	3	
apple	5	
vs		5
apple	5	

So we might end up inside the two strings or at the end of at least one of them. We should probably eliminate the possibility that we've overrun one of the strings first for safety (just like our && clauses did via the &&'s short-circuit evaluation).

But, what string did we run off of? Either the shorter one or both simultaneously, right? And if the both situation, aren't they both the 'shorter'? Well, sort-of. We might even make our while slightly more efficient with this idea (and it will certainly make our post-processing more efficient...):

```
string s, t;
string::size_type c, shorter_length{s.length()};
if ( t.length() < shorter_length )
{
    shorter_length = t.length();
}
c = 0;
while ( c != shorter_length && s[c] == t[c] )
{
    ++c;
}
```

A little pre-processing has cut our && clauses by a third! And the longer the common [equal] prefix these strings share, the bigger savings that will become!

Now to post-process. One way is:

```
if ( s.length() == t.length() ) // the two have the same length
{
    if ( c == s.length() )          // and we reached the end
    {
        comp_res = 0;                   // equal!
    }
    else if ( s[c] > t[c] )         // 1st is > here
    {
        comp_res = +1;                // greater!
    }
    else // s[c] must be < t[c]    // 1st is < here
    {
        comp_res = -1;                // less!
    }
}
else                            // strings were of different lengths
{
    if ( c == shorter_length )        // c reached end of one
    {
        if ( s.length() ==            // it was
             shorter_length )         //    the 1st
        {
            comp_res = -1;               // less!
        }
        else                          // 2nd must have been shorter
        {
            comp_res = +1;               // greater!
        }
    }
    else                           // c must have still been inside
    {
        if ( s[c] > t[c] )            // 1st is > here
        {
            comp_res = +1;               // greater!
        }
        else // s[c] must be < t[c]   // 1st is < here
        {
            comp_res = -1;               // less!
        }
    }
}
```

Wow! That was quite the mouthful! A further analysis of these situations leads to a simpler branching structure:

```
if ( c == shorter_length )  // reached end of at least one string
{
    if ( s.length() == t.length() )  // they were same!
    {
        comp_res = 0;
    }
    else if ( s.length() == shorter_length )  // 1st is shorter
    {
        comp_res = -1;
    }
    else                                        // 1st is longer
    {
        comp_res = +1;
    }
}
else  // stuck in the middle of both strings -- can't be equal
{
    if ( s[c] > t[c] )
    {
        comp_res = +1;                         // 1st is greater
    }
    else
    {
        comp_res = -1;                         // 1st is less
    }
}
```

But we've just rewritten `.compare()`! What about getting rid of the ASCII-betical nightmare that is case sensitivity? That part is actually quite easy. Just wrap all character comparisons in a call to either `toupper` or `tolower` as suits your tastes:

```
while ( c != shorter_length &&
        tolower(s[c]) == tolower(t[c]) )
// ...
if ( tolower(s[c]) > tolower(t[c]) )  // fold case
```

Now the upper or lower caseness of the `string`'s `char`s is ignored during comparison. As the comment says, we fold the case into a single possibility so we don't worry with either.

3.8.2.9 Centering strings

When printing a program title thus far, we've been just tabbing over a bit to make it look 'nice'. But if we could center that title, it would look even better! The only thing stopping us now is figuring out the width of the user's window. By standard agreement, all terminal windows start out as 80 characters wide. The modern user is apt to grab the edges of this window and drag it wider, but we cannot account for that kind of shenanigans. So we'll stick with an 80 character line for our title centering.

The code for this is quite simple. Let's say the program's title is stored in a variable called `prog_title`. Then we can do the following:

```
string welcome = "Welcome to the " + prog_title + " Program!";
cout << string((80 - welcome.length()) / 2, ' ') << welcome << "\n\n";
```

This is as centered as it can get. Even though when the program's title has an odd length this will leave a space one shorter to the left side of the message than to the right. This is, of course, because of the integer truncation of the division. But you can't print half a space, anyway!

3.8.2.10 Find & Replace

One of the most popular activities to perform with a sequence of characters — a `string` — is to search for sub-sequences within it or to replace one sub-sequence with another. To help with these tasks, the `string` `class` provides a multitude of functions to suit almost every possible situation!

Function	Description
`.find(s)`	find sub-string `s` within the calling `string`
`.rfind(s)`	find forward-facing sub-string `s` within the calling `string` searching backward from the rear of the `string`

These can be used, for example, to find any `"the"` sequence in a line of text:

```
//                  1         2         3         4         5
//        0123456789012345678901234567890123456789012345678901
string a{"The fox jumped quickly over the lithe brown feather."};
string::size_type front, rear;

front = a.find("the");   // is set to 28
rear = a.rfind("the");   // is set to 47
```

Note that the front occurrence of "the" is not at position 0. The search is case sensitive as is normal in the computer.

Also note that the rear occurrence of "the" is not also 28. This is because it is not a word matcher but a substring matcher — any matching sequence makes it work.

And what happens when the sub-sequence is NOT found? Well, a value greater than or equal to the length of the calling string is returned. It's name is string::npos. It stands for "your search was Not found at any POSition". You can check for this by testing the returned value's equality with this constant or simply see if the returned value is strictly less than the calling string's length:

```
if ( str.find(s) < str.length() )   // or:  str.find(s) != string::npos
{
    // okay to use this position
}
else
{
    // couldn't find s in str!
}
```

But what would you want to use the *find()[27] result for? Probably to focus on that sub-sequence of characters for further operations like:

Function	Description
.find(s, p)	as find(s) beginning at position p
.rfind(s, p)	as rfind(s) beginning from position p
.insert(p, s)	insert s in front of p
.erase(p, n)	erase n characters starting at p
.erase(p)	erase all characters from p to the end of the string
.replace(p, n, s)	replace n characters at p with s

(Or maybe for subscripting?)

In all of these, p represents the position in front of which to insert, at which to begin erasing, or at which to replace. To find this position, you'll typically have to *find() it first:

```
if ( str.find(s) < str.length() )   // or:  str.find(s) != string::npos
{
    str.replace(str.find(s), s.length(), "new string");
}
else
{
    cout << "Couldn't find '" << s << "' in string!\a\n";
}
```

[27] The * notation here is used to represent some sequence of characters like the r for rfind or even the empty sequence like found in front of find itself!

But we could easily make this more efficient — cutting out the multiple calls to find — if we used our friend size_type to store the result in a variable:

```
string::size_type pos = str.find(s);
if ( pos < str.length() )   // or:  pos != string::npos
{
    str.replace(pos, s.length(), "new string");
}
else
{
    cout << "Couldn't find '" << s << "' in string!\a\n";
}
```

But who wants to replace just the first occurrence? Most people want to find and replace all copies of the text in the original string, don't you think? To accomplish this, we'd need to put the process in a loop like so:

```
pos = str.find(s);
while ( pos < str.length() )    // or: pos != string::npos
{
    str.replace(pos, s.length(), "new string");
    pos = str.find(s, pos);
}
```

Here we've looked for the text to replace and, when found, replaced it with the new text and searched for the next occurrence of the text to replace. This continues until the text to replace is not found.

A typical use of this loop might be to replace all tabs in an input string with single spaces:

```
pos = entered.find('\t');
while ( pos < entered.length() )    // or: pos != string::npos
{
    entered.replace(pos, 1, " ");          // contrast ' ' !
    pos = entered.find('\t', pos);
}
```

Note that, even thought the replacement text is a single space, we must enclose it in double quotes. The replace function doesn't accept single chars.

Of course, having replaced all tabs with single spaces, we may have created a double-space scenario in the string. Or, they may have just double-tapped the Spacebar by habit or accident. Then we should replace these doubled spaces with single spaces, too:

```
pos = entered.find("  ");
while ( pos < entered.length() )    // or: pos != string::npos
{
    entered.replace(pos, 2, " ");
    pos = entered.find("  ", pos);
}
```

Be careful if you type this into your program to use, make sure the first and last literal strings have two spaces inside and the middle one only has one space. If you aren't careful, this could cause a truly explosive situation!

What do I mean by 'explosive'? I mean an infinite replacement! Let's say that you were replacing all the "The" sub-strings with "There" for some reason. This is quite dangerous. Let's explore:

```
    The first ...
    There first ...
    Therere first ...
    Thererere first ...
    Therererere first ...
```

This will go on and on and on until the user runs out of memory! The reason is that we are searching for the next occurrence of "The" from the same spot we found it before. Earlier we would replace the original sub-string with something that looked different. This time, however, the replacement text contains a copy of the search text. This causes the infinite replacement pattern seen above.

To avoid this possibility, we can use an offset. We'll let the search start a little further along the second and later times we look for the search text at least when the search text is a sub-string of the replacement text:

```
if ( replace_with.find(search_for) < replace_with.length() )
{
    cerr << "Warning!  You've got a potentially infinite "
            "replacement!  Adjusting...\n";
    offset = replace_with.length();
    // minimum offset is replace_with.rfind(search_for) + 1, but
    // this is easier and even safer.
}
else
{
    offset = 0;
}
```

Now when the text to search_for is detected as a sub-string within the text we want to replace_with, we set an offset (which is of data type string::size_type, of course) to move the next search a little further over than just where we found the last occurrence of search_for. Otherwise, we leave the position alone with a 0 offset:

```
pos = entered.find(search_for);
while ( pos < entered.length() )
{
    entered.replace(pos, search_for.length(), replace_with);
    pos = entered.find(search_for, pos + offset);
}
```

Why the 0 for offset when it is safe? Why not always scoot over further? Well, if we always moved further over, we'd mess up that double-space to single-space substitution, for instance. To work most effectively, that needs to be an in-place search so that it will reduce triple space sequences and four-space sequences and so on to single spaces. (Think it over...it's quite the eye-opener!)

Why do I keep using a less-than test against the length of the string I searched within instead of the earlier suggested npos test? It tends to hurt not just beginners' heads but even seasoned programmers' heads to have that double negative test there. (You know, 'not equal to the non-position'.) However, as I think we've seen, it is more effective in typing for many situations — it is quite a bit shorter than replace_with.length() for instance.

Finally, what about that comment in the offset branch about using rfind? That is true. We could have used that version of the offset initialization, but using simply the length is both easier to type and safer. Whether it meets every user's needs is a matter of study and/or opinion. I leave that decision to you and your teacher.

Another way to have tackled this particular infinite replacement would have been to watch for word boundaries during the search. This at first thought seems ridiculous, but it isn't really that hard as it turns out.

We can take one of two approaches. One is limiting to particular scenarios and the other is more easily extensible.

The first approach — the limiting one — is to use the cctype classification functions to watch for word boundaries. We can do this like so:

```
pos = entered.find(search_for);
while ( pos < entered.length() )
{
    if ( ( pos == 0 || ! isalpha(entered[pos - 1]) ) // at beginning of word
         &&                                            // and
         ( pos+search_for.length() >= entered.length() ||  // at end of
           ! isalpha(entered[pos+search_for.length()]) ) ) // word
    {
        entered.replace(pos, search_for.length(), replace_with);
        offset = 0;
    }
    else
    {
        offset = search_for.length();
    }
    pos = entered.find(search_for, pos + offset);
}
```

Here we've checked that the char before our match was not a letter or there was no char before our match to confirm that we were at the beginning of a word. We've also made sure that either our match ended the string or what followed wasn't a letter to confirm that we were at the end of a word. Only if both these conditions were true have we replaced the target text with the new text. Further, if we replace the text, we can offset the next search not at all. But if we don't match, we must offset to avoid finding the same non-word match again.

What of the second approach? For that one, we get to specify exactly what constitutes a non-word character. This might seem more limited at first than just using everything that isn't a letter on the system, but it is more configurable to different applications' needs and so is the more general approach. We start with our lists or non-word characters:

```
const string punct = ";:.,><}{][)(*&^%$#@!~`\"\\/|?+=-_'";
const string space = " \t\n\r\v\f\b\a";
const string word_seps = space + punct;
```

I've named the group of all non-word characters word_seps because they are the characters that separate words. I have taken a couple of liberties with the list of spacing characters, I suppose. I don't know that most people would qualify the 'alarm' escape as a space ('\a'). And backspace ('\b') might not be a space, either, in many people's views. But the rest are definitely whitespace — space, tab, newline, carriage return, vertical tab, and form feed.

So what do we do with these? We check the before and after characters of our match against them:

```
pos = entered.find(search_for);
while ( pos < entered.length() )
{
    if ( ( pos == 0 ||                                 // at beginning of word
           word_seps.find(entered[pos - 1]) != string::npos )
         &&                                             // and
         ( pos+search_for.length() >= entered.length() ||  // at end of word
           word_seps.find(entered[pos + search_for.length()]) != string::npos) )
    {
        entered.replace(pos, search_for.length(), replace_with);
        offset = 0;
    }
    else
    {
        offset = search_for.length();
    }
    pos = entered.find(search_for, pos + offset);
}
```

See how we use the find function to check that we did find the characters before or after in the word_seps. (Remember, *not* the non-position is really a position.)

3.8.2.11 Processing One Word at a Time

Now let's say the user has entered a line of text and they need something done to each word in the line. Maybe it's something fun like Pig-Latin translation. Maybe it's something weird like reversing each word's letters. Maybe it's something more serious. Who knows! But they need it done and they need it done now!

The problem with the first of the above methods of finding words is that it just checked for letters vs. non-letters. The second method brought punctuation into consideration and limited us to punctuation for a particular application. For instance, dashes can be part of words — like a hyphenated word — or not — like the one just before this phrase. Single quotes (apostrophes) are the worst! Some people use them to surround something for emphasis and they are also used for contractions and possessives. If the user decides to edit all occurrences of "don" in their text because it is too archaic, we'd match the first part of "don't" as well.

This can get worse if we are dealing with programmers who regularly consider an underscore as part of a word not to be messed with — variable and constant names? What can we do? We could add special negative checks to the word matching condition above. But it is already quite complicated. Maybe another way would prove sleeker?

There are other find functions that come with the string class. These take a string value that is used as a set or list of characters to either allow or disallow in the match. They then search through the calling string for the first or last match. The first match is pretty clear, but 'last' match? Those versions search from the end of the string toward the front so the thing they find are later in the string — the 'last' match in the string from a front-oriented view. Here's a handy chart:

Function	Description
.find_first_of(s)	finds the position of the first character from s to match in the calling string
.find_first_of(s, p)	as find_first_of(s) but beginning from position p
.find_last_of(s)	finds the position of the last character from s to match in the calling string — looking from the end
.find_last_of(s, p)	as find_last_of(s) but starting from position p — still moving toward the front
.find_first_not_of(s)	finds the position of the first character matching none of those in s
.find_first_not_of(s, p)	as find_first_not_of(s) but beginning from position p
.find_last_not_of(s)	finds the position of the last character from the calling string to match none of those in s — looking from the end of the calling string
.find_last_not_of(s, p)	as find_last_not_of(s) but starting from position p — still moving toward the front

Wow! That's a lot of functions! At least their names are fairly easy to remember. And each has a version with a position specified instead of the default.

Let's look at a simple example or two first, and then we'll get back to our words in a line problem. . .

Let's suppose the we have this situation:

```
//                  1         2         3         4         5
//        0123456789012345678901234567890123456789012345678901
string a{"The fox jumped quickly over the lithe brown feather."};
string::size_type front, rear;

front = a.find_first_of("aeiouyAEIOUY");     // is set to 2 -- the 'e'
rear = a.find_first_not_of("aeiouyAEIOUY");  // is set to 49 -- another 'e'!
```

On the other hand, if we used the starting position parameter, we could find:

```
front = a.find_first_of("aeiouyAEIOUY", 3);     // is set to 5 -- the 'o'
rear = a.find_first_not_of("aeiouyAEIOUY", 48);  // is set to 46 -- an 'a'
```

Note that in both sets of searches, I found only lowercase letters even though I allowed for uppercase to be found as well. This is due to the fact that only the first match is found and it just doesn't worry about any other possibilities.

But how do we use these functions to find words in a line of text? That's fairly clever, really. We start with finding the first thing that isn't a word separator:

```
string::size_type beg;
string line;

beg = line.find_first_not_of(word_seps);
```

Here we're reusing the word_seps constant from above and adding new variables for the user's line of text and the beginning of the word. We would have read the line in with getline earlier, of course. Some of you are wondering why we are searching instead of just using the first character of the line — position 0 — as the beginning of the word. Well, what if the user has put spacing ahead of the first word like in the beginning of a paragraph? Maybe the user's line starts with some kind of punctuation like a quotation might. One never knows what the user is capable of! Be cautious. . .

Now that we have the position of the first word, we should find its end. That can be done by reversing our search:

```
string::size_type end;

end = line.find_first_of(word_seps, beg);
```

Starting from the first letter of the first word, we look for the first character that is a word separator. That signifies the end of the word so we store it in our `end` variable. Keep in mind, though, that `beg` is in the word and `end` is just after the word's content. This will become important shortly.

Now we have two options. We can pull out a copy of the word and then store it in a variable or we can directly store the word into the variable without the intermediate copy. Said that way, it seems obvious which to choose. But let's look at both tools just in case we want to use one more than another in other coding situations. The latter approach uses the `assign` function:

```
string word;

word.assign(line, beg, end - beg);
```

This takes characters directly out of the `line` variable and stores them immediately in the `word` variable. The copying starts at the position `beg` and runs for `end-beg` characters. This difference is exactly the length of the word we found. Why not `+1` like we've done in the past with discrete subtraction (random modulo base, page counting, etc.)? Well, this is just subtraction because the `end` is not inclusive but exclusive. Since one end of the range is not included, we don't have to add one to put it back in after the difference is taken.

The other way to take out a word is to use the `substr` function like so:

```
string word;

word = line.substr(beg, end - beg);
```

This function copies the `end-beg` characters starting at `beg` from the `line` and returns them as a new `string`. Then we take that `string` and store it in our `word` variable. This is a little slower and takes twice as much memory as the `assign` variation.

Now we'll add a little protection and get this code:

```
beg = line.find_first_not_of(word_seps);
end = line.find_first_of(word_seps, beg);
while ( end < line.length() )          // entire word inside string
{
    word.assign(line, beg, end - beg);
    cout << "Word:  '" << word << "'.\n";
    beg = line.find_first_not_of(word_seps, end);
    end = line.find_first_of(word_seps, beg);
}
// this if catches a word abutting the end of the string
if ( beg < line.length() )
{
    word.assign(line, beg, line.length() - beg);
    cout << "Word:  '" << word << "'.\n";
}
```

This should find all the words in the line and report them out to the user. If we'd wanted to, we could have done anything else to them in place of or in addition to the cout to print them back out.

3.8.2.11.1 Taking Care of the Apostrophes

Wait! What about those apostrophes? Oh, yeah. Let's see. We were worried about things like "don't" and "Fred's" and the like, right? Okay. Let's double-check our constants:

```
const string punct = ";:.,><}{][)(*&^%$#@!~`\"\\/|?+=-_'";
const string space = " \t\n\r\v\f\b\a";
const string word_seps = space + punct;
const string::size_type MAYBE_SEP = word_seps.length() - 1;
```

Note that the apostrophe (single quote) is the last thing in the punctuation string. And there's a new constant! It says that anything at or after this position is not necessarily a separator except under special circumstances.

Let's see how we might use this constant in checking for contractions and possessives. We need to check every time we find an end for a word that it really isn't a single quote for special circumstances. It might play out like this:

```
beg = line.find_first_not_of(word_seps);
end = line.find_first_of(word_seps, beg);
while ( end < line.length() )
{
    while ( end < line.length() &&                    // not off the end yet
            word_seps.find(line[end]) >= MAYBE_SEP && // a possible separator
            end+1 < line.length() &&                  // it has a follower
            word_seps.find(line[end + 1]) == string::npos )// followed by word
    {                                                    // bits
        end = line.find_first_of(word_seps, end + 1);    // move over and try
    }                                                    // again
    word.assign(line, beg, end - beg);
    cout << "Word:  '" << word << "'.\n";
    beg = line.find_first_not_of(word_seps, end);
    end = line.find_first_of(word_seps, beg);
}
if ( beg < line.length() )
{
    word.assign(line, beg, line.length() - beg);
    cout << "Word:  '" << word << "'.\n";
}
```

Note how we repeat our end search from just past where we last found the end. If we started at beg or even at end, we'd find the same separator again. So we start the next search a little further down.

Why a loop, though? Well, I'm from the south. We talk very slowly down there, as you may have heard. But we make up for it by having multiple contractions at a time: couldn't've, I'd've, etc. This needs a loop to work appropriately all over the country. *grin*

The only thing I'd add would be to watch for my personal pet rule: possessives of words ending in S don't have to have an extra S after the apostrophe. It isn't too hard and I think it would make a fine exercise for the interested reader.[28]

[28] I've always wanted to write a book just so I could say that!

3.8.2.11.2 Handling the Inter-word Gaps

But if you are trying to transform them somehow and report them back out changed but still in context as they were before, you'd need to keep track of what was before each word and possibly after the last word. Let's call this stuff the 'gap' information. It could be spaces or punctuation. Some of it is between words and some is before/after words. So I think gap is the best name we're gonna get. How would things need to change in our code? Let's see:

```
string gap;

beg = line.find_first_not_of(word_seps);
if ( beg != 0 ) // there is 'gap' in front of us!
{
    // gap is the stuff before the first word
    gap.assign(line, 0, beg - 0);
    cout << "Leading gap:  '" << gap << "'\n";
}
end = line.find_first_of(word_seps, beg);
while ( end < line.length() )          // entire word inside string
{
    word.assign(line, beg, end - beg);
    cout << "Word:  '" << word << "'\n";
    beg = line.find_first_not_of(word_seps, end);
    gap.assign(line, end, beg - end);
    cout << "     Then gap:  '" << gap << "'\n";
    end = line.find_first_of(word_seps, beg);
}
// this if catches a word abutting the end of the string
// -- i.e. no trailing 'gap'
if ( beg < line.length() )
{
    word.assign(line, beg, line.length() - beg);
    cout << "Word:  '" << word << "'\n";
}
```

Here gap is a new string that will hold those non-word characters from the user's original line of input. The first copy needs to be protected because a 0-length gap would look odd begin displayed. Maybe we could have taken this approach instead, though:

```
// gap might be the stuff before the first word
gap.assign(line, 0, beg - 0);
if ( ! gap.empty() ) // there is 'gap' in front of us!
{
    cout << "Leading gap:  '" << gap << "'\n";
}
```

This might make some people on your team happier since it doesn't make it look like the branch was the assign function's fault. But others would point out that the first version keeps us from calling for a rather silly 0-length assignment in the first place. So maybe that one with a better comment.

3.8.2.12 Making string Content Name-case

One thing that we often need to do to user data is change its case more permanently than we did in the comparison section above (section 3.8.2.8). This is because the user will often type things into the program in either all lowercase or all uppercase depending on when they last hit CapsLock. Many don't even pay attention to what they are doing!

So in a report, we might want a business name or user's name to look nice and in a proper way.[29] To do this, we'll have to visit every character and make sure it is the right case for its position in the string. We made this kind of loop in the comparison section, actually, but it was designed to stop when a mismatch in the parallel strings was found. This time we'll be processing all the characters in the string, so we'll use a for loop instead of a while loop:

```
string first;
// read in the user's first name
for ( string::size_type c{0}; c != first.length(); ++c )
{
    // do something with first[c]
}
```

Of course we want to change the elements of the string into proper case where that comment is. But we have three choices. One is to try to change the first character to uppercase before the loop and shorten the loop. The second is to change the first character to uppercase after the loop — reprocessing it from the lowercase the loop made it. And the third is to make the decision as to whether we are doing the first character inside the loop. Each can be found in 'wild' code out in industry, but prevalence doesn't make it right.

Let's start with the last one:

```
string first;
for ( string::size_type c{0}; c != first.length(); ++c )
{
    if ( c == 0 )
    {
        first[c] = static_cast<char>(toupper(first[c]));
    }
    else
    {
        first[c] = static_cast<char>(tolower(first[c]));
    }
}
```

Remember that the static_cast is needed here because these transformation functions from cctype return an ASCII code as an integer instead of an actual char.

While this works fine, it has to decide on every character of their name if that really is the first character or not. After the first loop, this is a wasteful test taking up the user's precious time. That's why it might be better to do the uppercasing either before or after the loop rather than inside it.

Let's try doing the uppercase transformation before the loop:

```
string first;
if ( ! first.empty() )
{
    first[0] = static_cast<char>(toupper(first[0]));
    for ( string::size_type c{1}; c != first.length(); ++c )
    {
```

[29] By proper here, I mean cased like a name: first letter uppercase and rest lowercase. This is sometimes called name-casing the string. Some applications call for this, others demand it not be done. Which situation are you in this time?

```
        first[c] = static_cast<char>(tolower(first[c]));
    }
}
```

Here we have to protect the initial subscript of the first character with a not `empty` test. Before all the subscripting was inside the loop and its condition protected them. Now the [0] is before the loop and needs separate protection. I included the loop inside the `if` because it now assumes at least one character in the `string` — which the `if` condition guarantees. If we put it outside the branch, it wouldn't be so protected and a zero-length `string` would cause trouble in the loop!

How? Well, let's test it. `c` would start at 1 which would be unequal to the `length` of 0 so the loop would start. Then we'd access the 0 position and the program would crash — hopefully.[30] We could change the test to < instead of !=, but the latter is more popular these days and we don't wanna look weird in front of the other programmers, do we? *smile*

Finally, let's explore the uppercasing being after the loop:

```
string first;
for ( string::size_type c{0}; c != first.length(); ++c )
{
    first[c] = static_cast<char>(tolower(first[c]));
}
first[0] = static_cast<char>(toupper(first[0]));
```

Here we have to reprocess the first character of the user's name once — processing it overall twice. But the syntax is clean and the speed is quite nice. This is my personal preference amongst the three variations.

What if they want to do this with a whole name instead of just their first (or maybe last) name? We'd just have to put this loop and assignment of the uppercase front letter inside the word extracting loop above:

```
beg = line.find_first_not_of(word_seps);
end = line.find_first_of(word_seps, beg);
while ( end < line.length() )
{
    word.assign(line, beg, end - beg);
    for ( string::size_type c{0}; c != word.length(); ++c )
    {
        word[c] = static_cast<char>(tolower(word[c]));
    }
    word[0] = static_cast<char>(toupper(word[0]));
    cout << "Word:  '" << word << "'.\n";
    beg = line.find_word_not_of(word_seps, end);
    end = line.find_word_of(word_seps, beg);
}
if ( beg < line.length() )
{
    word.assign(line, beg, line.length() - beg);
    for ( string::size_type c{0}; c != word.length(); ++c )
    {
        word[c] = static_cast<char>(tolower(word[c]));
```

[30] More on that in the next section!

```
    }
    word[0] = static_cast<char>(toupper(word[0]));
    cout << "Word:  '" << word << "'.\n";
}
```

Now we've at least done something with those words we were extracting. *smile*

3.8.3 Processing exceptions

So far, we've been accessing individual `char`s in a `string` with the subscript (or indexing) operator (`[]`). This either gives you the value at a particular position within a `string` or fails. Sometimes the failure to find that character crashes the program, sometimes it does not. We've been very careful with `if`s and loop conditions to *NOT* go off the end of the `string`.

If we wanted more of a guarantee of a crash when we did make a mistake, however, we could use a different access method known as `at`. It is called with the dot (`.`) operator and the `string` on the left of that. Then you put the position you want inside the parentheses of your call to the `at` function:

```
s.at(0)
```

This would access the first character of the `string` `s` or crash the program if `s` were `empty`.

Why would we want to guarantee a crash? It could help in the debugging phase of program development to find a problem more quickly if we guarantee a crash instead of just hope we've done due diligence with our bounds checking. Then it is just up to our thorough testing to find all those crashes.

So how does `at` guarantee the crash? Well, it will cause what is known as an exception in the program when it detects our access for a `string` is out of bounds. In fact, that's the name of the exception it causes: `out_of_bounds`. When an exception is never dealt with in a program, it causes the program to crash and a strange and cryptic message is printed for the user which they will hopefully report verbatim to the coding team for debugging purposes.

Is that it? We just let it crash? Well, you don't have to. You can deal with the exception since you know now that it can happen sometimes. Let's talk vocabulary for a minute, though.

When a function causes an exception, we say it `throw`s an exception. To handle an exception you know might happen, you should `try` to `catch` it. (After all, you can't `catch` what you aren't expecting, right? It'll just bounce off your head causing damage if it were hard enough.)

The code for this might look something like this:

```
string word;
bool repeat{true};
while ( repeat )
{
    repeat = false;    // we're gonna succeed this time!
    cout << "Please enter your word:  ";
    cin >> word;
    cin.ignore(numeric_limits<streamsize>::max(), '\n');

    try
    {
        cout << "\nThe 6th letter of your word is:  '"
             << word.at(5) << "'.\n";
        // You can put several functions in here that
        // all cause the same exception and just catch
        // once at the end of the code if you want.
    }
    catch ( out_of_range )  // (out_of_range ex)
    {
        repeat = true;
        cout << "\nYour word isn't long enough for our purposes.\n"
                "Please try again with a longer word...\n";
        //cout << "\nSpecifically, the function said:  " << ex.what()
        //     << '\n';

        // this is a bad idea:
        //cout << "\nThe first letter of your word is:  '"
        //     << word.at(0) << "'.\n";
        // it might cause another exception from this catch block!
        // then we can't catch it and it'll crash the program...
    }
    /*
     * if your try has functions that cause multiple
     * types of exceptions, you can have multiple catch
     * blocks to follow it and catch each type
     * separately
     */
}
```

As you can see, we use at to possibly generate an out_of_range exception inside a try block. This is then caught in a catch block following the try. Had the catch named the exception it caught, we could have used the what function on it to report that same cryptic message to the user that a crash would have printed. But I like our message better — it is more appropriate to our application, after all.

Also note that you can list as many catch blocks as you need to process all possible exceptions that might be thrown at you from the try block. What other functions could throw an exception? Let's take a quick glance at cppreference.com and see:

Function	Might throw
at	out_of_range
assign	length_error
insert	length_error, out_of_range
erase	out_of_range
replace	length_error, out_of_range
substr	out_of_range

So many of the string class functions we've learned can throw exceptions at us. Should we try to catch them all? Some would say yes, others no. Exception handling is a bulky bit of code. But it would probably be well worth it to do so rather than face the crash report later. After all, it isn't like at that we could just replace with well-bounded subscript operations.

Also note the comment at the end of the example catch block. As indicated, if you perform some action in a catch block that might throw another exception — even the same kind as is being caught by this block — you won't be able to catch it here and it will crash the program if no code after you tried to catch it. So be extra careful what you do in a catching situation.

3.9 Even More Branching

While the if and its kin are quite general and powerful at branching to make decisions, we often desire more elegance or optimization in our code. Therefore, we have two other forms of branching to study here: switches and the ?: operator.

3.9.1 The switch Branch

The switch branching structure can make certain decision situations execute more efficiently and so is a good idea to learn. It also helps with debugging and maintenance of code. Let's look at what those certain situations are and how these benefits manifest.

3.9.1.1 Rules for switches

A switch can be used in place of a cascaded if under certain conditions. These conditions are:

- all tests are for equality (==)
- all tests involve a common expression (CE)
- all tests involve — other than the CE — constant or literal expressions
- all constant/literal expressions are unique (this is checked by the compiler and reported during compilation)
- all values involved are of a 1D discrete type (char or integer and technically bool)

In addition, the equality conditions in the if form can be combined with logical or (||) and there can be a else clause at the end.

First let's look at the general structure of a switch. It starts with the switch keyword and uses two other keywords: case and default. Let's take a look:

```
switch ( CE )
{
    case value:
    {
        // code for this value
    } break;
    case value: case value:
    {
        // code for these values
    } break;
    case value:
    case value:
    {
        // code for these values
    } break;
    default:
    {
        // code for any other values
    } break;
}
```

The common expression (CE) is listed once at the top of the switch inside parentheses. Then we have a mandatory pair of curly braces around all the values the CE is expected to take on. Each value is preceded by the keyword case and followed by a colon. Then we list the statements associated with this value or values and follow them with a break statement. This break is a necessary part of the switch's cases and not to be avoided like when we talked of it with respect to loops earlier.

The default block is there to match any value of the CE not listed in a particular case value — emulating an else at the end of a cascaded if. It is often listed last, but doesn't have to be. Some programmers like to list it first to make sure all extra cases are handled.

Another myth about the default is that it alone needs no break. This is untrue. The rule is that the *last* block in the switch doesn't need a break. If we list the default block first, it will need a break to function correctly. We generally, of course, put a break on all blocks for consistency and just to make sure nothing bad happens. After all, you never know when, during maintenance, another programmer will add a set of case values to the end of the switch to handle something new and forget to add a break to your old last block! Then your block will continue to execute through their block each time it is chosen!

Finally, we emulate a || combo by listing out multiple cases one right after another. This can be done on separate lines or across a single line as you like. The reason this works is that the computer looks for the first case value that matches the value of the CE and just executes code until a break occurs. The case values themselves don't have any executable content and so are blithely ignored in this.

I've put curly braces on the statements for each case set, but these are only necessary if we want to declare new variables inside the block. Also, some people would put the break inside the curly braces instead of afterwards like I have here. I put it outside because I feel it is more a part of the switch structure than of that case's code.

There are also a variety of ways to indent switches. Some folks will indent as I've done here. Others will not indent the case keywords inside the mandatory curly braces. This violates our earlier style rule to always indent inside a pair of curly braces, but since these braces are mandatory, some people feel it is acceptable to not indent here. Also, some people will indent the break when not using the curly braces on the case blocks — technically making them not blocks at all but just lists of statements. Others will keep the break at the same level as the case keywords. We all seem to agree to indent the statements inside the case whether we use curly braces on it or not.

Here are a few examples of such style variations:

```
switch ( CE )
{
    case value:
        // code for this value
    break;
    case value: case value:
    {
        // code for these values
    } break;
    case value:
    case value:
        // code for these values
    break;
    default:
```

```
switch ( CE )
{
case value:
    // code for this value
break;
case value: case value:
{
    // code for these values
} break;
case value:
case value:
    // code for these values
break;
default:
```

```
switch ( CE )
{
    case value:
        // code for this value
        break;
    case value: case value:
        // code for these values
        break;
    case value:
    case value:
        // code for these values
        break;
    default:
        // code for any other values
        break;
}
```

```
switch ( CE )
{
case value:
    // code for this value
    break;
case value: case value:
    // code for these values
    break;
case value:
case value:
    // code for these values
    break;
default:
    // code for any other values
    break;
}
```

```
switch ( CE )
{
    case value:
    {
        // code for this value
        break;
    }
    case value: case value:
    {
        // code for these values
        break;
    }
    case value:
    case value:
    {
        // code for these values
        break;
    }
    default:
    {
        // code for any other values
        break;
    }
}
```

```
switch ( CE )
{
case value:
{
    // code for this value
    break;
}
case value: case value:
{
    // code for these values
    break;
}
case value:
case value:
{
    // code for these values
    break;
}
default:
{
    // code for any other values
    break;
}
}
```

And, of course, our original example can be done without the 'extra' indention.

As to the benefits, the matching of equal values can be done much more efficiently than the general tests that the if structure supports and so a switch will run more quickly than an equivalent cascaded if. Also, since the compiler checks that all case values are unique, we get a little help when you slip or have a copy/paste incident causing duplicate values to be checked in separate branches.[31] And finally, if used with an enumeration, the compiler warns if any of the constants are not listed in a case indicating a missed situation.

This sounds like a lot of stuff, but it turns out to happen a lot! Let's look at a few examples to get the hang of it.

3.9.1.1.1 Menus Revisited

One of the primary places to use a switch is when processing the user's response to a menu. Look back at our basic menu example:

```
choice = static_cast<char>(toupper(choice));
if ( choice == '1' || choice == 'J' )
{
    cout << "\nOption 1 -- JUNK -- Chosen!\n\n";
}
else if ( choice == '2' || choice == 'S' )
```

[31] Yes, each case block is considered a branch within the switch structure. This is akin to how the whole cascaded if was a branching structure and each if/else-if/else was a branch within it.

```
{
    cout << "\nOption 2 -- STUFF -- Chosen!\n\n";
}
else if ( choice == '3' || choice == 'Q' || choice == 'X' )
{
    done = true;
}
else
{
    cout << "\n\aInvalid choice '" << choice << "'!!!\n\n"
            "Please try to read more carefully next time...\n\n";
}
```

Here we have a 1D discrete type (char) being compared entirely with equality (==) and having or combinations (||) and an else at the end. Oh, and all the literal values are unique and being compared to a common expression (choice that was uppercased). Perfect! This will make an excellent switch.

```
switch ( toupper(choice) )
{
    case '1': case 'J':
    {
        cout << "\n\tChoice 1 -- JUNK -- chosen!\n\n";
    } break;
    case '2': case 'S':
    {
        cout << "\n\tChoice 2 -- STUFF -- chosen!\n\n";
    } break;
    case '3': case 'Q': case 'X':
    {
        done = true;
    } break;
    default:
    {
        cout << "\n\aInvalid choice '" << choice << "'!!!\n\n"
                "Please try to read more carefully next time...\n\n";
    } break;
}
```

Note how the toupper is placed now inside the switch head. It also doesn't need the static_cast any longer since we aren't storing it into a char variable. Add to that convenience the improved speed of the switch and the unique value checking, we have a winning combination: menus and switches!

3.9.1.1.2 Suffixes for Numbers

Putting a suffix on a number is a common application need. You don't always want to use cardinals "number 1", "number 2", etc. Sometimes you want to use the ordinals: "1st", "2nd", etc. How to map a number to its suffix? Let's explore the pattern to make sure we know what we want first:

```
  0th   1st   2nd   3rd   4th   5th   6th   7th   8th   9th
 10th  11th  12th  13th  14th  15th  16th  17th  18th  19th
 20th  21st  22nd  23rd  24th  25th  26th  27th  28th  29th
 30th  31st  32nd  33rd  34th  35th  ...
 .
```

```
   .
   .
  100th 101st 102nd 103rd 104th 105th  ...
  110th 111th 112th 113th 114th 115th  ...
  120th 121st 122nd 123rd 124th 125th  ...
   .
   .
   .
```

Notice how the 1, 2, and 3 slots are all special except during the tens. Those are always "th" just like everyone else. So, we might think to code this as:

```
switch ( number )
{
    case 1:
        suffix = "st";
        break;
    case 2:
        suffix = "nd";
        break;
    case 3:
        suffix = "rd";
        break;
    default:
        suffix = "th";
        break;
}
```

Here number is an integer and suffix is a string.

But this doesn't take account of the twenties, thirties, hundreds, etc. We'd end up saying, for instance, 101th instead of 101st! We need to focus on what makes them similar and it seems to be the ones digit. To extract just the ones digit, we use modulo, of course:

```
switch ( number % 10 )
{
    case 1:
        suffix = "st";
        break;
    case 2:
        suffix = "nd";
        break;
    case 3:
        suffix = "rd";
        break;
    default:
        suffix = "th";
        break;
}
```

But now the teens are showing up special as well: 11st! The simple fix is to add cases to the switch for the exceptions. But they have the same ones digits. It is their tens digits that differ! I guess we'll have to nest the switch in an if to handle the exceptions:

```
if ( number / 10 % 10 == 1 )
{
    suffix = "th";
}
else
{
    switch ( number % 10 )
    {
        case 1:
            suffix = "st";
            break;
        case 2:
            suffix = "nd";
            break;
        case 3:
            suffix = "rd";
            break;
        default:
            suffix = "th";
            break;
    }
}
```

Here we've used both integer division and modulo to extract just the tens digit. I could have also done `number % 100 / 10`, but this code seemed slimmer and is going to be slightly more efficient if the compiler is paying attention. This is because the division by 10 is later followed by a modulo by 10. The CPU calculates these two values together at once. So the smart compiler will just hold onto that second value from the first calculation and not recalculate it all over again.

Still, it is lucky for us that all the teens are `"th"` and not just those three!

But this is a bit overkill. Since the exceptions are like all other numbers, we can use an initialization to handle both their situation and the `default`:

```
suffix = "th";
if ( number / 10 % 10 != 1 )
{
    switch ( number % 10 )
    {
        case 1:
            suffix = "st";
            break;
        case 2:
            suffix = "nd";
            break;
        case 3:
            suffix = "rd";
            break;
    }
}
cout << "You are " << number << suffix << " in line.\n";
```

This does cause us to reset the `suffix` variable for the special `case`s, but it saves our time coding and is usually considered worth it.

3.9.1.1.3 Randomized Messages

Sometimes the patter from your program grows stale to a long-term user. It would be nice if your responses were more varied. We can emulate that with randomization.

We'll select a few messages for a given part of the program and then pick one randomly:

```
switch ( rand() % 4 )
{
    case 0:
        mesg = "That would be ";
        break;
    case 1:
        mesg = "I believe that is ";
        break;
    case 2:
        mesg = "When I was your age, that was ";
        break;
    case 3:
        mesg = "\a&*^*&()&&%$ ";
        break;
}
cout << mesg << answer << ".\n";
```

Sorry, I ran out of steam on that last one. But you get the idea. *smile* (answer would be whatever answer we'd calculated to tell them in this program.)

3.9.1.2 Fallthrough

As we said earlier, a break is necessary to stop the switch from executing after a case block. If we don't have one, we will execute into the next block and so on until a break is found or we run off the end of the switch.

Sometimes this can be done on purpose for good effect. For instance, if two branches need to perform nearly identical actions, but one has a little more work before their overlap and nothing extra afterwards. This is a little tricky to think about until you see it in action, so let's look at a couple of examples.

3.9.1.2.1 Days Until Now

Let's say we had the day, month, and year of a date and need to know how many days had elapsed in that year up to and including that date. We could do it like this:

```
days_so_far = day;
for ( short m = month - 1; m >= 1; --m )
{
    switch ( m )
    {
        case 11:
            days_so_far += 30;
            break;
        case 10:
            days_so_far += 31;
            break;
        case  9:
            days_so_far += 30;
            break;
        case  8:
            days_so_far += 31;
            break;
        case  7:
            days_so_far += 31;
            break;
        case  6:
            days_so_far += 30;
            break;
        case  5:
            days_so_far += 31;
            break;
        case  4:
            days_so_far += 30;
            break;
        case  3:
            days_so_far += 31;
            break;
        case  2:
            days_so_far += 28;
            break;
        case  1:
            days_so_far += 31;
            break;
        // no 0 case since it wouldn't do anything anyway
        // (month-1==0 means we're in January so no whole
        // months have passed...)
    }
}
// check for leap year and after February
```

Here, all the variables not declared in the fragment are `short` integers. We start the month loop at the `month-1` because the initial setting of `days_so_far` to `day` handles the days that have passed during this current month. Two notes:

- The `+=` updates may throw out warnings on some compiler setups. This is because the C++ standard says `short` computations can be coerced into `int` instead and then this result will be 'too big' to fit back into the `short` variable.

- We could have used our enumerations for `MonthNums` and `MonthDays` from section 2.3.3.2, but we just used the literals for expedience. This isn't the best choice, but is sometimes done when a deadline approaches. In the next revision we'll make sure to clean this up.

- We didn't need to line up the month numbers like that, but some programmers think it looks pretty. Just thought I'd give it a showing for their sake. Maybe you'll like it, too.

You may wonder why I looped backwards through the months. That was just for fun. We could have looped from 1 to the `month - 1` instead and it would have worked just the same — addition is commutative, after all. The same applies to the order of the `case`s within the `switch` — they could have been ordered ascending instead of descending and all would have worked just fine.

But, this uses all the `break`s and doesn't demonstrate our point. Here is a version that takes advantage of the fall-thru principle:

```
days_so_far = day;
switch ( month - 1 )               // previous month; this month is done
{
    case November:
        days_so_far = static_cast<short>(days_so_far + Nov_days);
        [[fallthrough]];
    case October:
        days_so_far = static_cast<short>(days_so_far + Oct_days);
        [[fallthrough]];
    case September:
        days_so_far = static_cast<short>(days_so_far + Sep_days);
        [[fallthrough]];
    case August:
        days_so_far = static_cast<short>(days_so_far + Aug_days);
        [[fallthrough]];
    case July:
        days_so_far = static_cast<short>(days_so_far + Jul_days);
        [[fallthrough]];
    case June:
        days_so_far = static_cast<short>(days_so_far + Jun_days);
        [[fallthrough]];
    case May:
        days_so_far = static_cast<short>(days_so_far + May_days);
        [[fallthrough]];
    case April:
        days_so_far = static_cast<short>(days_so_far + Apr_days);
        [[fallthrough]];
    case March:
        days_so_far = static_cast<short>(days_so_far + Mar_days);
        [[fallthrough]];
    case February:
        days_so_far = static_cast<short>(days_so_far + Feb_days);
        [[fallthrough]];
    case January:
        days_so_far = static_cast<short>(days_so_far + Jan_days);
        [[fallthrough]];
    // no 0 case since it wouldn't do anything anyway
    // (month - 1 == 0 means we're in January so no whole
    // months have passed...)
}
// check for leap year and after February
```

In this revision we've used the enumerations and put in the suggested static_casts. We've also changed out our breaks for a new notation: [[fallthrough]]. This indicates to both the compiler and other programmers that we are purposefully leaving out the break and letting the code fall through to the next case block. This isn't absolutely necessary, but it is a good idea.

How does this work without a loop? Well, when the prior month — the one that is complete — is found in a case value, we add on its days. Then we fall-thru to the month that preceded it and so on until we reach the end of the switch. This continues to add on the days of all those preceding months down through January. Since it automatically adds up all the prior months, we don't need to loop.

Clearly, here, the order of the case values is extremely important! If you decide to change them up to ascending you will get the number of days left in the year plus the previous month's days and the current month's days and less the days left this month. It'd be a mess!

Lastly, the `[[fallthrough]]` mark on the `January` branch didn't really need to be there, but I put it there for consistency and completeness.

3.9.1.2.2 Roman Numbers

Another thing that comes up from time to time (writing about the Super Bowl, enumerating list items, printing a fancy clock, pages for the before text material in a book, etc.) is converting normal Arabic numbers into Roman form. This can be done in a pretty simple way and can be more elegantly done using a `switch` with fall through.

The basic idea is to convert each digit of the Arabic number separately and then concatenate them together in a `string` of Roman digits. We'll not go into the theory here, but you can read all about it on the web at any number of fine websites.

Let's take just the ones digit for an example. The pattern for the ones digit conversion looks like this:

Roman Ones

	V
I	VI
II	VII
III	VIII
IV	IX

Here the different colors represent different parts of the pattern. First off, 4 and 9 are different from everyone else and so should get their own branches to deal with it. Second, all numbers from 5-8 start with a V. We should take advantage of this commonality to optimize our time spent coding. Finally, all numbers 1-3 and 6-8 end in some number of Is. For 1-3, this number of Is is exactly the number itself. For 6-8, it is 1-3 Is. There are two ways to map 6-8 to 1-3 — subtraction or modulo.

Let's look at what we've got so far:

```
switch ( ones )
{
    case 4: case 9:
        roman += 'I';
        if ( ones == 4 )
        {
            roman += 'V';
        }
        else
        {
            roman += 'X';
        }
        break;
    case 1: case 2: case 3:
        for ( short i = 1; i <= ones; ++i )
        {
            roman += 'I';
        }
        break;
    case 5: case 6: case 7: case 8:
        roman += 'V';
        for ( short i = 1; i <= ones % 5; ++i )
        {
            roman += 'I';
        }
        break;
    // no need for a 0 --- nothing added for this situation
}
```

First, why are 4 and 9 combined? I thought they had nothing in common? Well, upon further inspection, they both started with an I — kinda like all the 5-8 start with a V, so I thought I'd take advantage of that fact to simplify the structure a little.

Second, we is 5 okay running through that `for` loop? That's because `5 % 5` is 0 and the `for` loop just doesn't run then. (1 is immediately not <= 0.)

Also, we now see the overlapping branches! So let's combine them:

```
switch ( ones )
{
    case 4: case 9:
        roman += 'I';
        if ( ones == 4 )
        {
            roman += 'V';
        }
        else
        {
            roman += 'X';
        }
        break;
    case 5: case 6: case 7: case 8:
        roman += 'V';
        [[fallthrough]];
    case 1: case 2: case 3:
        for ( short i = 1; i <= ones % 5; ++i )
        {
            roman += 'I';
        }
        break;
    // no need for a 0 --- nothing added for this situation
}
```

Here we do the `for` loop only once and use modulo to cap it so that we can use the same loop without worry. We could have also either modded by 5 or subtracted 5 before falling through and left the loop bound at `ones`. That would make it slightly more efficient. Maybe on the next version.

Roman Tens, Hundreds, & Thousands

	L		D	
X	LX	C	DC	M
XX	LXX	CC	DCC	MM
XXX	LXXX	CCC	DCCC	MMM
XL	XC	CD	CM	

And what about the other digits? Well, I'll leave those to you, but suffice it to say that the pattern — for us — stops at 3999 because the Roman symbol for 5000 is V̄ and we can't exactly do that on the console in a `string`, now can we? But, as far as it goes, the other digits are very similar to the ones digit. In fact, we could just copy/paste the ones code and change the digit variable and the letters we are appending to the string. The only one that's at all different is the thousands and it is just shorter. Since we'll limit the number to 3999, we just won't execute the 4-9 code for that digit. But more on that later...

Actually, upon further reflection, we could probably use a generic digit variable instead of four specific ones and put the above switch in a while or for loop. This avoids the problems with copying and pasting and gives us a chance to use strings and switches even more! After all, we'd need to change which string we were taking digits out of each round. We just need to make sure we are concatenating the digits in the right order to the overall Roman string. I'll leave this mostly to you, but I'll give you this hint. You'll need a switch like this one to change the string in use each time around the loop:

```
switch ( divisor )
{
    case 1000:
        current_place = THOUSANDS;
        break;
    case 100:
        current_place = HUNDREDS;
        break;
    case 10:
        current_place = TENS;
        break;
    case 1:
        current_place = ONES;
        break;
}
```

Good luck and have fun!

3.9.2 The ?: Operator

I've been looking at the 4/9 code above and it's irritating me. That's a big if-else for such a tiny assignment choice. Maybe we can do better. Let's look at a new operator that makes small decisions. It has many names: selection, conditional, ternary, to name a few. But many just call it the ?: operator[32] because those are its symbolic components. They are not, however, typed in side-by-side like the insertion, extraction, comparison, and shorthand math operators with multiple symbols. They are separated by a value to act on.

But perhaps I'm getting ahead of myself. Let's look at a typical if-else that we'd be able to replace with this new operator:

```
if ( condition )
{
    action value1;
}
else
{
    action value2;
}
```

Here we have a common action taking place in both branches and a different value being acted upon in each branch. It is all controlled by the value of the condition — true or false. true leads to value1 being used in the action and false uses value2 in the action.

[32] Read as "question-colon operator".

To replace this structure with a ternary operation, we'd code this:

```
action ( condition ? value1 : value2 );
```

Now that you see the ?: operator in action, you hopefully get what I was saying before: the ? part and the : part are separate from one another. They are separated by the value to use when the test is true. Then the : is followed by the value to use when the test is false. The test itself resides before the ? symbol.

The parentheses around the ?: may not be needed depending on the action and the context of the ternary's placement.

Are there any other limitations? Yes. The two values must be of *exactly* the same data type. You can't even mix things as similar as double and short together, typically.

3.9.2.1 Two-way Examples

Let's look at some examples to get the feel of it. . .

3.9.2.1.1 Roman Numbers Revisited

So we were concerned with the if structure in the 4/9 case of our Roman number switch above (section 3.9.1.2.2):

```
if ( ones == 4 )
{
    roman += 'V';
}
else
{
    roman += 'X';
}
```

To transform this into a selection operation, we just pull out the roman += action and make the decision of 'V' versus 'X' based on whether the ones digit is a 4 or not:

```
roman += ones == 4 ? 'V' : 'X';
```

This would take us as the programmers ridiculously less time to type and takes the compiler just as much time to compile — maybe less? — and runs just as efficiently on the user's end. It's a win-win-win!

Note that here the parentheses are not needed as the == and the += take place with the right precedence order. That is, == always evaluates before +=.

But I took the advice above and used a string to store the letters so my code looked like this:

```
if ( ones == 4 )
{
    roman += current_place[1];
}
else
{
    roman += current_place[2];
}
```

Am I out of luck? No! Of course not. You can do this one in two different ways. The first is to do like we did above and put the subscript operation inside the conditional operation:

```
roman += ones == 4 ? current_place[1] : current_place[2];
```

Or, you could just decide, based on the ones value, to use either a 1 or a 2 within the subscript:

```
roman += current_place[ones == 4 ? 1 : 2];
```

Saving even more keystrokes and time! Try to ferret out the minimum change you can do in such situations. Once you get used to the transformation, you'll do it automatically instead of after the fact.

But just to help, let's review a few more. . .

3.9.2.1.2 Plural Agreement

Making a noun plural or singular depending on a variable's value is, to some of us, pretty important. It shows that the programmer took at least a modicum of time making their interface nice and clean of typos. I can't even remember all the times I've yelled at an app for printing the likes of "1 file(s) downloaded." or "1 files downloaded.". At least the former programmer made it possibly work. The second one didn't even concern themselves with it.

So how hard is this task? Not hard at all! In fact, let's go straight to the ternary form:

```
cout << count << " " << ( count == 1 ? "file" : "files" )
     << " downloaded.\n";
```

Here the parentheses are needed as == and << would not get along properly. It would try to print the value of count again and then compare cout to 1 in the ?: operation!

Note: In English, 0 is plural, too. Weird, no?

3.9.2.1.3 Gross Pay

What about gross pay with overtime involved? Yep, that's a ?:, too:

```
gross = hours > 40 ? 1.5 * rate * hours : rate * hours;
```

or, trimming it up with a little factoring — but maybe confusing folks:

```
gross = rate * ( hours > 40 ? 1.5 * hours : hours );
```

It might confuse others because it looks like the 1.5 is just times the user's hours worked instead of the rate as well. Also, the parentheses are needed here again due to the tug-of-war between * and > over the hours variable.

3.9.2.1.4 A Counterexample

Note that many students of programming feel that this is a perfect tool to assign a bool variable a value:

```
bool_var = condition ? true : false;
```

But this is just wasteful! That condition will be true when the bool_var is set to true and false when the bool_var is set to false. Why add an extra layer of processing? Just store the condition result in the bool_var:

```
bool_var = condition;
```

What if you want to store the opposite? Then apply DeMorgan's Laws to the condition (if necessary):

```
bool_var = ! condition;
```

3.9.2.2 Ternary with Identity

The ternary operator normally requires two values to act upon. This can be emulated under certain circumstances.

For instance, let's revisit plural agreement for a second. It can be coded more effectively than a whole `if-else` as a single `if` and no `else`:

```
cout << count << " file";
if ( count != 1 )
{
    cout << 's';
}
cout << " downloaded.\n";
```

Can we now not use a `?:` to shrink this code? We can, but it takes some tricky thinking. First, we consider what is in the missing `else` branch. Were we to code it up, we would display nothing there:

```
cout << count << " file";
if ( count != 1 )
{
    cout << 's';
}
else
{
    cout << "";
}
cout << " downloaded.\n";
```

Now it is two-way, but the types of the values are different. Luckily it is easy enough to change a `char` literal to a `string` literal:

```
cout << count << " file";
if ( count != 1 )
{
    cout << "s";
}
else
{
    cout << "";
}
cout << " downloaded.\n";
```

You thought I was going to do the anonymous construction trick, didn't you? Well, always take the shortest path to your goal!

So now we can make this a ternary operation again:

```
cout << count << " file" << ( count != 1 ? "s" : "" )
     << " downloaded.\n";
```

We can do the same thing with the gross pay calculation from above:

```
gross = rate * hours * ( hours > 40 ? 1.5 : 1.0 );
```

Here we are multiplying by 1 in the alternative branch.

What both of these examples have in common is that the value in the alternative branch is the identity value for the action in question. You've used identities before but maybe not called them that. An identity is a value that, when acted on usually with another value gives a result that is the other value unchanged. For instance, 0 plus any other number is that other number. 1 times any other number is the other number.

Here we have that the empty `string` printed on any output stream — like `cout` here — doesn't change the stream at all. Are there any other identities for C++ actions? So glad you asked! Here's a handy table of them:

Action	Identity
`+`	`0, ""`
`*`	`1`
`||`	`false`
`&&`	`true`
`<<`	`""`
`pow`	`1`
`=`	the variable on the left

Note that addition and concatenation share an operator and so I've put their identities on the same row together.

Any operator with an identity value can be turned into a selection operation this way.

3.9.2.3 Greater Than Two-Way Selection

That's right, as the title of this section implies, you can actually do not just two-way and one-way (with identity!) branches in `?:` form, you can even make three-way and higher branches into conditional operations.

How can this be so? Well, recall our original exploration of the cascaded `if` (section 3.5.2). The `if` in an `else-if` was originally nested inside a plain `else`. Therefore, we can use nesting with operators to achieve three-way or higher `?:` operations.

Let's take this as a simple example:

```
cout << "The value " << number << " is ";
if ( number > 0 )
{
    cout << "positive";
}
else if ( number < 0 )
{
    cout << "negative";
}
else // number == 0 necessarily
{
    cout << "zero";
}
cout << ".\n";
```

We can achieve this by realizing the nesting, again:

```
cout << "The value " << number << " is ";
if ( number > 0 )
{
    cout << "positive";
}
else
{
    if ( number < 0 )
    {
        cout << "negative";
    }
    else // number == 0 necessarily
    {
        cout << "zero";
    }
}
cout << ".\n";
```

And so it becomes:

```
cout << "The value " << number << " is "
     << ( number > 0 ? "positive" : ( number < 0 ? "negative" : "zero" ) )
     << ".\n";
```

The first thing to note is that the ?: operation has the third lowest precedence of any C++ operator. That means that it doesn't conflict with anything except two very slow operations.[33] This means for us that we don't need the inner parentheses on this:

```
cout << "The value " << number << " is "
     << ( number > 0 ? "positive" : number < 0 ? "negative" : "zero" )
     << ".\n";
```

While that helps, this is still rather cumbersome and is getting rather long for a display in a presentation or the like. We'd like to wrap it, but how? There are several popular styles and people come up with variations all the time:

```
cout << "The value " << number << " is "
     << ( number > 0 ? "positive"
                     : number < 0 ? "negative"
                                  : "zero" )
     << ".\n";
```

```
cout << "The value " << number << " is "
     << ( number > 0 ? "positive" :
          number < 0 ? "negative" : "zero" )
     << ".\n";
```

[33] There's actually more to it than this, but a full discussion of precedence and such is beyond the scope of this work.

```
cout << "The value " << number << " is "
     << ( number > 0 ? "positive" :
          number < 0 ? "negative" :
                       "zero" )
     << ".\n";
```

```
cout << "The value " << number << " is "
     << ( number > 0 ? "positive"
        : number < 0 ? "negative"
                     : "zero" )
     << ".\n";
```

And that's just to show a few of them! Play around with style variations from program to program but just not within the same program. Do a whole program in a single set of styles. If you don't like the effect, try it differently the next program.

One last thing: make sure you don't nest ternary operations too deeply as they become unseemly and unwieldy rather quickly. I'd recommend no more than 5 levels deep (that's the equivalent of 3 else-ifs in a cascaded structure).

3.9.3 Factoring a Branching Structure

Factoring is an important skill and has many facets. We'll start with factoring a branching structure to make it less redundant. We can see a good example of this just above in the number sign report fragment:

```
cout << "The value " << number << " is ";
if ( number > 0 )
{
    cout << "positive";
}
else if ( number < 0 )
{
    cout << "negative";
}
else // number == 0 necessarily
{
    cout << "zero";
}
cout << ".\n";
```

This code is well-factored already. What would be a redundant version of it? Why should it be this way instead? Here, let's look:

```
if ( number > 0 )
{
    cout << "The value " << number << " is "
         << "positive" << ".\n";
}
else if ( number < 0 )
{
    cout << "The value " << number << " is "
         << "negative" << ".\n";
}
else // number == 0 necessarily
{
    cout << "The value " << number << " is "
         << "zero" << ".\n";
}
```

Note that the common prefix on the output is repeated in every branch and even the period and newline are repeated in every branch. Since these appear at the beginning of every branch and the end of every branch respectively, we can factor them in that direction. This resembles doing factorization in algebra and thus the name:

$$a \cdot x \cdot b + a \cdot y \cdot b = a \cdot (x + y) \cdot b$$

Sadly, we don't have an inverse operation like distributivity. But that would just make the code bulkier so. . .

3.9.4 Summing Up Branching

So, with so many choices, how do you decide which branching structure to use in a particular situation? For ease of programming small situations, the `?:` is awesome! Just make sure you can really use it: same action and same type of value in all branches.

For those situations that you can use them, the `switch` can't be beat. But it does have quite a few restrictions on its use so be sure they are all covered in your problem. (The list is in section 3.9.1.)

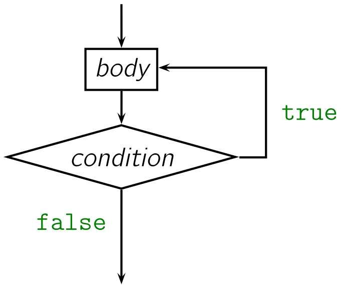
And an `if`? It is applicable at any time. Even if a `switch` or `?:` would be better, the `if` will work. But try to use the other two when they are appropriate as it will help in so many ways!

3.10 Even More Looping

Believe it or not, there are also two more loop forms. These are `do` loops and range-based `for` loops. Let's look at each with examples.

3.10.1 do Loops

Let's begin with a flowchart. The `do` loop executes as shown in the flowchart below.



The body is executed at least once as it happens before the condition is ever tested. Once tested, the condition may send the loop back through the body or out and on with the next bit of the program — one never knows!

The odd and somewhat worrisome thing about the `do` loop is that we don't know without analyzing a particular piece of code where the initialization and update phases of the loop are! The initialization might be before the `do` loop starts or inside the body. If it is inside the body, it might be the same line as the update. (A priming loop for sure! See section 3.6.1.2 for more on this.)

One particularly good place to use a `do` loop is for menus. After all, you have to print the menu, read the user's choice, and decide if it was to quit at least once, right? So here, then, is our final version of the menu example:

```
char choice;
bool done{false};
do
{
    cout << "\t\tMain Menu\n\n"
            "1) do Junk\n"
            "2) do Stuff\n"
            "3) Quit\n\n"
            "Choice:  ";
    cin >> choice;
    cin.ignore(numeric_limits<streamsize>::max(), '\n');

    switch ( toupper(choice) )
    {
        case '1': case 'J':
        {
            cout << "\n\tChoice 1 -- JUNK -- chosen!\n\n";
        } break;
        case '2': case 'S':
        {
            cout << "\n\tChoice 2 -- STUFF -- chosen!\n\n";
        } break;
        case '3': case 'Q': case 'X':
        {
            done = true;
        } break;
        default:
        {
            cout << "\n\aInvalid choice '" << choice << "'!!!\n\n"
                    "Please try to read more carefully next time...\n\n";
        } break;
    }
} while ( ! done );
```

This will make the karma police stay off your back for forcing all those while loops to go around once earlier. *chuckle*

3.10.2 Range-based for Loops

Range-based for loops at first seem of limited usefulness as they currently only apply to strings. But later (chapter 6) we'll see another type of container of lots of things that we can use them with as well.

So what is a range-based for loop and why is it like a standard for loop? Well, it looks like this:

```
for ( char ch : str )
{
    // do something with ch
}
```

In this form, the loop will automatically visit every character in the string str and copy it into the char ch. Then you can do anything you want inside the loop with ch and it won't affect the characters in str at all.

We could use it, for instance, to print an all uppercase version of a string for a title on a table or chart we're printing. Let's take a look:

```
cout << string((80 - str.length()) / 2,' ');
for ( char ch : str )
{
    cout << static_cast<char>(toupper(ch));
}
```

```
cout << '\n';
```

I went ahead and centered it on the line to make it look even nicer.

The two caveats with these loops are that you no longer know the position of the character within the `string` and you can't do anything even slightly different with any single value within the `string`.

Don't worry, these loops will become more useful in later chapters as we learn more tools and places to apply them.

3.10.3 Summing Up Looping

Let's start with range-based `for` loops. They are only used to walk through the elements in a `string` and perform the exact same action to each element. Next come standard `for` loops as they are the easiest to identify: use them when you know exactly how many times to repeat. `do` loops are next easiest: when you don't know how many times to repeat, but you know it must happen at least once. Lastly, there are `while` loops which can handle any repetition situation.

3.11 Wrap Up

So we have lots of ways to control decision making in a program. These range from the powerful and general `if` and `while` down to the very specific `?:` and range-based `for`. Using the right combination of tools for the job is the art of it.

Chapter 4

Functions

So far all of our programs have been housed in the main function and if we wanted to reuse code from a prior program, we would have to copy/paste it and make modifications so it would work in the new program — variable names, etc. This is a tedious and error-prone enterprise at best. And it leads to some hideously long main programs!

In this chapter we'll look into a flow control tool that can package up a set of code for easy reuse without constant editing and even cross-program reuse. This tool is the function.

4.1 When? Who? Where? Why? What? How?

In this section we'll answer these questions as they pertain to functions.

4.1.1 Why Functions?

Easy code reuse is our primary concern when writing functions. But it has other benefits, too. The primary one is known as encapsulation. This means basically that the code that's in the function stays in the function — no one need know what goes on in there!

This is also called the Black Box Principle — like those devices in airplanes. Again, it elicits the image of an opaque box that can't be seen into. But trick this box out with an input hopper and an output spigot like a typical function machine and you have something special![1]

In addition to these monumental benefits, there are other smaller ones:

- Functions allow for easier testing and debugging.
- Functions ease the update and maintenance phases of program development.
- Functions can clarify the code that calls on them as well as simplify it greatly.
- Functions play an intricate role in the Top-Down Design process (aka Stepwise Refinement).

4.1.2 How Do We Use Functions?

There are three parts to a function that make it complete/whole:

- Declaration or Prototype of a Function
  The declaration tells what information the function needs to begin its work. This information is known as the formal arguments.

  ```
  head ;
  ```

  The head has a return type telling what type of information the function's work will result in, a name by which the function will be called later on, and a parenthesized argument list. (The term argument is more often used than function parameter or input.) In the argument list, each argument is specified by at least a data type, but ideally also a name specifying this argument's role in helping the function solve the problem it attempts to solve. If the function needs more than one argument, they are comma separated much like the names of multiple variables in a variable declaration. Only here each argument needs its own type — even if it is the same as the previous argument's type.
  Some examples:

  ```
  double pow(double base, double exponent);
  double sqrt(double x);
  int rand(void);
  void srand(unsigned seed);
  void ignore(void);
  void ignore(streamsize max_count, char terminator);
  ```

  The semi-colon ends the prototype/declaration just like any C++ statement. This declaration tells the compiler and the prospective caller[2] what the function needs to begin its work, what its purpose is, and what value — if any — it will return to the caller.
  Note that the last three examples return nothing. They have a `void` return type to signify this. The `void` in `rand`'s parentheses isn't necessary, but makes it more clear that the function doesn't need any information to start its work.

- Call to a Function
  The call to a function makes it execute. The call also provides the actual arguments the caller wants the function to work with on this particular call. Another call may provide different arguments. For instance, we might call `pow` with `radius` and 2 one time — for an area of a circle — and with `side` and 3 another — for a volume of a cube. `pow` will call these `base` and `exponent` each time

[1] Note that that was two links side-by-side.
[2] Programmer calling/using the function.

regardless of what variable or value was passed in actual fact. That's why the arguments listed in the declaration are known as the formal arguments and those in the call the actual arguments — they are necessarily different. (The term 'formal' may be thought of as the name your parents' friends call you by when you are at a gathering. The actual argument is less formal and may not be stored under a particular name at all.)
Some examples:

```
d1 = pow(x1 - x2, 2.0);
cout << sqrt(d1 + d2);
srand( time(nullptr) );
r = rand() % 6 + 1;
if ( isspace( cin.peek() ) )
{
}
cout << str.length();
cin.ignore();
cin.ignore(numeric_limits<streamsize>::max(), '\n');
getline(cin, str);
getline(cin, str, '~');
```

As we see, some actual arguments are variables, some are constants, and some are literals. Some are even calculation (expression) results! Among those are the results of other function calls! There are even empty argument lists when the function has a void or empty formal argument list. Further, the result of the function is being used as appropriate or usual. If a use out of the ordinary were attempted, the compiler would stop that, too.

- Definition of a Function

```
int main ( void )   // head -- must match prototype exactly!
{                   // \
    // stuff        //  |
                    //  +--- body
    return 0;       //  |
}                   // /
```

A function definition places the head of the function — exactly like in the prototype — atop a function body. As usual, a body here is a pair of curly braces with a list of semi-colon terminated statements inside. The last statement in a function should be its return statement. The expression on this statement with the keyword return should match in type to the return type listed in the function's head. If the return type is void, an empty expression can be used:

```
return;
```

4.1.3 Where Do Functions Go in the Source Code?

Well, we actually need to know where each of the functions' three parts go relative to the rest of the source code. Here's the general plan:

```
        #include <libs>
        using directive
    P R O T O T Y P E S
        main definition ----------\___C A L L S
    D E F I N I T I O N S --------/
```

A function may be called from any **other** function definition. That is, a function should not be called from inside its own definition nor from any of the definitions of functions it calls. When this happens we call it recursion — because the function will re-occur. It is a tricky process to control and we will learn more about it in a following course.

4.1.4 When Should We Code Functions?

There are several occasions that call for writing a function. These include:

- Code is more cluttery than meaningful.
  If the code you've found looks like a right mess and you feel it should be gutted and thrown asunder even though it is working just fine, perhaps you should hide it in a function. The function provides a black box, after all, and we wouldn't be able to see any of this mess from the call site.

- Code seems general or reusable.
  You'll get better at this one as you program more and more. For now look to the tips in this book and those of your teacher as to what is going to be reusable in the future.

- Code is obviously repeated with only data values, variables, constants, subexpressions, etc. being changed from place to place (the code and types are the same).
  This one is tricky at first, but you get better as you practice, obviously. If you were good at those 'spot the differences' puzzles in children's magazines and restaurant place-mats, you'll probably be good at this. An automated difference checking tool can be a lifesaver here. Ask your teacher if there are any installed at your school.

- The more data values you can 'parameterize', the more likely you'll be able to reuse your function. Here, parameterize means to make into a parameter. We don't have a verb 'argumentize', oddly enough. And it is true, the more of those differences you identified in the last bullet you can make into function arguments, the more likely it is that someone can call your function in their situation.

4.1.5 Who Are These Function People?

There are actually many people involved in a typical program development cycle. There are programmers — each of whom write their own parts of the program. There are testers who run the resulting program against sets of test data meant to make sure it works under normal and extreme conditions. There are optimizers who take proven code and tweak it to make it run just a little faster here and there.

But here in an introductory programming class, you are likely to play all of these roles yourself. Well, except optimizer. We don't do too much of that at this level. You've probably already been an application programmer and tester/end user, for instance.

Now, with the advent of functions in your programs, you'll play two more roles: function caller and function programmer. That's right: the programmer who writes the function isn't necessarily the one who calls it, too. Remember that programming is a team effort! After the initial design meeting(s), programmers go their separate ways and work on their individual parts. They'll call their own functions in a test harness or driver, but not in the actual application code. That'll be done by someone else who was writing that part of the program.

Just try to keep all of it clear in your head. When you are writing the function, you know all about it — its ins and outs and how it does its job. When you are calling the function, you only know the ins and outs and an idea of its purpose. You don't know how it does its job. And you shouldn't! Do you want to know how pow does its job? It works for positive, negative, zero, and decimal exponents, after all. That's a broad job description. It is quite complicated and not for the faint-hearted. I'm pretty sure it involves calculus! So keep to yourself and just call the darn function.

4.1.6 How Do We Design Functions?

There are two main ways to design functions. We can start from scratch or we can use existing code we think is either repeated or reusable. Let's start from scratch first and we'll look at taking out existing code into a function in a later section (4.5.1).

Let's design a function for rounding a value to a nearest multiple. Let's further assume that our C++ library implementation is missing the round function — just to make it interesting. One of the basic formulas, you'll remember, is:

```
floor(value / multiple + 0.5) * multiple
```

See section 2.6.2.3 for more on how this works.

How do we build a function for this code? We analyze it for the basic ins and outs first. Here we find:

```
  floor(value / multiple + 0.5) * multiple
// known  IN       IN      known     IN
```

That is, we know the floor function and it is always adding 0.5. However, the value and multiple need to be supplied by the caller for us to work with — what value do they want to round and to what nearest multiple do they wish to round.

Next we decide on the types of the inputs. Here they will both be double in case they want to round to the nearest 0.25 or the like. Thus our argument list is:

```
(double value, double multiple)
```

I just put the value first because it seemed the most important input. The order doesn't really matter as long as the caller knows what each argument means with respect to the task being solved.

Now we decide the return type. Looking through our formula and our argument types, we see that floor always returns a double and that times a double would still be a double. Thus our return type will be double.

Next we pick a name for the function. We want this to be as clear as possible without going overboard. There will also be comments on the prototype to help explain the function's purpose. Here I'm going to go with round_nearest. This is better than just plain round without being too wordy. Our function head now looks like this:

```
double round_nearest(double value, double multiple)
```

Next we put the task code into a function body and decide if there are any local variables necessary to help our work. Here we have just the formula to calculate so no helper variables are needed. This gives us:

```
{
    return floor(value / multiple + .5) * multiple;
}
```

Now we put the head on top of the function body below the main and copy it to the top of the source file — between the using and the start of main — and add a semi-colon.

Finally to the comments. Remember that the prototype comments should discuss at least purpose, inputs, and output. Comments at the definition can expand on this minimum with discussion of code details.

Thus our prototype looks like this:

```
// Function to round caller's value to the nearest integer
// (or whatever multiple they desire -- see multiple
// argument).
//
// NOTE:  The rounding position doesn't have to be a power
//        of 10, it can be a multiple of 5 or 2.5 or 0.43 or
//        ~anything~!
double round_nearest(double value, double multiple);
```

And our definition can look like this:

```
// Function to round caller's value to the nearest integer
// (or whatever multiple they desire -- see multiple
// argument).
//
// NOTE:  The rounding position doesn't have to be a power
//        of 10, it can be a multiple of 5 or 2.5 or 0.43 or
//        ~anything~!
//
// multiple is used to pre-scale the value to the ones
// position.  Then a .5 is added to translate (shift) the
// floor function over to align with the traditional round
// to the nearest function.  Finally the multiple argument
// is used again to post-scale the result from the ones
// position to the original.
double round_nearest(double value, double multiple)
{
    return floor(value / multiple + 0.5) * multiple;
}
```

What about calls? Well, that's up to the caller. *grin* We're just talking design and implementation here.

But, just to sum up, the process is:

1) Have the code ready that performs the task.

2) Decide what needs to be given by the caller and what is already known.

3) Determine the return type of the function.

4) Pick a name for the function.

5) Put the task code into a function body and plan out any local variables.

6) Copy the head to both the body and the top of the code with a semi-colon.

7) Add comments to both the prototype and the definition.

4.2 Examples

Before we move on to more details, let's analyze a few more functions to see how it looks.

4.2.1 Input an Uppercase Letter

One repeated task we've done is read in a char and change it to uppercase. We also always ignore the rest of the input line if any. This task code would look like this:

```
char get_up_char(void)
{
    char ch;
    cin >> ch;
    cin.ignore(numeric_limits<streamsize>::max(), '\n');
    return static_cast<char>(toupper(ch));
}
```

Here we've got the base code which needs to read in a char. Many beginning programmers will assume this is also an input to the function and list it as an argument. But it isn't something that is known to the caller beforehand! It is really part of our result. Thus we make it a local variable.

The only other detail here is the static_cast. This is just to suppress possible warnings from persnickety compilers about the integer ASCII code versus the return type issue. (Darn those old C programmers!)

I'll leave the comments to the interested reader... *grin*

4.2.2 Removing Leading Whitespace

Our next function is more clutter than meaning as well as being a reusable tool. It is the while loop we used to remove whitespace that might precede a newline This function looks like so:

```
char peek_ahead(void)
{
    while ( cin.peek() != '\n' &&
            isspace( cin.peek() ) )
    {
        cin.ignore();
    }
    return static_cast<char>(cin.peek());
}
```

Once again, no char is necessary as input. We use peek to look at each upcoming char from the input. I decided to return the last peeked character as a bonus result. Otherwise we would have had a void return type and had a different name like remove_space_to_newline or some such monstrosity.

4.2.3 Printing a Program Title

This function will center a program's title along with welcome text:

```
void welcome(string prog_title)
{
    const string welcome = "Welcome to the " + prog_title + " Program!";
    cout << string((80 - welcome.length()) / 2, ' ') << welcome << "\n\n";
    return;
}
```

Here I've made a local `string` variable to help center the message.

We also finally have a `void` return to demonstrate that not all tasks result in a value. Here our task is to display on the screen. We call such results side effects because they happen to the side of the normal flow of information between the caller and the function.

4.2.4 Centering Helpers

Speaking of centering, that `80-.../2` stuff gets old, doesn't it? Let's make a helper function to make that padding for us:

```
string center_left_pad(string to_center, streamsize width, char pad)
{
    return string((width - to_center.length()) / 2, pad);
}
```

Here we need no local helper variable. We do need several pieces of information to make this function as reusable as possible. We take in the `width` in which we are centering and the `char` to pad with. Taking these extra parameters helps the caller configure each call to their current needs.

We could have alternatively queried the `width` and `fill` of `cout`, but this way we can use it even when the caller intends to display on a graphical interface or to a file or network connection. Always plan for the future!

Then the `welcome` function above could call it like this:

```
void welcome(string prog_title)
{
    prog_title = "Welcome to the " + prog_title + " Program!";
    cout << center_left_pad(prog_title, 80, ' ') << prog_title
         << "\n\n";
    return;
}
```

In addition to using the `center_left_pad` function here, I've changed out the local variable for changing the parameter! This is not always the best decision, but it serves a purpose. It shows that changing the argument won't change the caller's original value (their actual argument). However, it uses the name `prog_title` in an unclear way and is likely not worth the slight speed and memory advantage.

But that's not all! We can also make a helper to 'calculate' that centering's right padding:

```
string center_right_pad(string to_center, streamsize width, char pad)
{
    string::size_type padding = width - to_center.length();
    return string(padding - padding / 2, pad);
}
```

The calculation is a bit different for the right side because of odd-length `string`s. We generally err to putting that extra padding character on the right side rather than the left. The integer division by 2 here will truncate the 0.5 and leave the left side with one space less on odd-length values of `to_center`. Then subtracting what `center_left_pad` calculated as the length of pad from the needed padding of both sides, we get our right-side padding.

When would the caller need this? Well, if they were centering a column of a table that wasn't at the end of the row, they'd need both front and rear padding on the data. Again, plan ahead and make the functions you provide as reusable as possible. Even if it means making an extra helper function to go alongside the original on you'd planned!

4.2.5 Displaying a Monetary Value

We are often called upon to format an output in a particular way, in fact. Here we are tasked with printing a monetary value to the user. To make things more interesting, this function must run across multiple applications such as inheritances and stock pricing/dividends as well as simple prices in a store. The implementation (aka definition) is:

```
void display_money(double amount, char symbol, bool in_front,
                   streamsize prec)
{
    streamsize old_prec = cout.precision(prec);
                                   // no scientific format on money
    ios_base::fmtflags old_flags = cout.setf(ios_base::fixed);
    cout.setf(ios_base::showpoint);  // ALWAYS show decimals -- even 0s
    if ( in_front )
    {
        cout << symbol;
    }
    cout << amount;
    if ( ! in_front )
    {
        cout << symbol;
    }
    cout.flags(old_flags);
    cout.precision(old_prec);
    return;
}
```

We need helper variables to preserve the caller's format settings so we don't mess up anything in the surrounding program. For more on this need, see section 2.6.5.2.1.

I've also factored out the middle on my branching structure. Many programmers would code the above as:

```
if ( in_front )
{
    cout << symbol << amount;
}
else
{
    cout << amount << symbol;
}
```

But I noticed that amount was printed at the end of the first branch and the beginning of the second branch and just took that middle part out to always happen in between the other printing that was specifically before or after it. This isn't for everyone, so use it to your own taste.

In addition to this definition, we are going to provide a pair of helper constants for that bool argument. After all, having raw true or false values in a call is not only tacky, but confusing! (These values have no intrinsic meaning to the average human — programmer or not. Avoiding raw bools is commonplace in good design.)

```
const bool SYMBOL_IN_FRONT = true,
           SYMBOL_IN_BACK  = false;
```

The caller can now use these in their calls like so:

```
display_money(x, '$', SYMBOL_IN_FRONT, 2);
cout << '\n';

cout << "The cantaloupes we received in today cost ";
display_money(x, 'L', SYMBOL_IN_BACK, 2);    // can't do 'pounds' symbol
cout << ".\n";
```

Contrast this with:

```
display_money(x, '$', true, 2);
cout << '\n';

cout << "The cantaloupes we received in today cost ";
display_money(x, 'L', false, 2);    // can't do 'pounds' symbol
cout << ".\n";
```

And you can see the improvement the constant names make.

4.3 Scope

I've mentioned something in passing that I think deserves a little more attention. In particular, what's that 'local' I used when describing the variables inside a function rather than their arguments? Well, that's all about the scope (aka visibility) of identifiers. There are four kinds of scope that interest us at this time: global, namespace, local, and block.

Global scope is everything not inside a function. This includes the function itself and any constants you've made outside a function. It can also include any typedefs or using aliases you've made for clarity and ease of use. All of these things are readily shared by all functions in a program with no harm done.

However, global variables are a strict no-no! The trouble is that you might place your variable globally and another programmer on the team decides they want a variable with that same name. Many programmers will use a variable and then scroll up to declare it later in their coding process. If the programmer forgets to declare their variable locally, they will use your global. Then, when it is your code's turn again — maybe you called their function as part of yours, your variable has been altered and could possibly damage your results.

So, long and short: **NO** global variables!

Identifiers inside a namespace are local to that area unless a using directive pulls them out of it. We will look at using our own namespaces later, perhaps.

Local scope means the identifier is inside a function and no one outside that function can see or use it. Even if they have a variable, for instance, with that same name, it is theirs and not yours. They don't conflict or overlap in any way.

Block scope makes an identifier only visible in the closest pair of curly braces. We sometimes, for example, declare a helper variable inside a loop that does something complicated but we don't need that result after the loop is over — not even this round of the loop! (That's right, a block scope identifier is destroyed at the close curly of the block and recreated should the open curly ever be passed again in the execution.)

4.4 Arguments

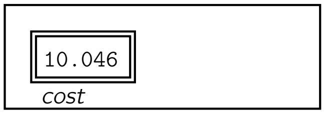
This section focuses on arguments. We talk about the passing mechanism: how do those actual arguments become formal arguments. We'll see the usual mechanism for this and a new one with a little more oomph. We'll also discuss two tools for making functions more configurable and easy to use for your caller.

4.4.1 Passing Arguments

First we should explore how arguments we've used so far are dealt with in the computer's memory during a call. Then we'll talk about alternatives.

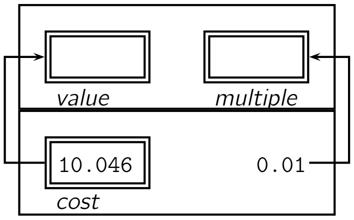
4.4.1.1 Normal Argument Passing

The following image is a picture of how a function looks inside the memory of the computer. It exists on a place called the function call stack or execution stack. It's box is called its frame or record on this stack. Within the frame, I've drawn a variable we are about to use in our call to the round_nearest function. Here is the call, just for reference:

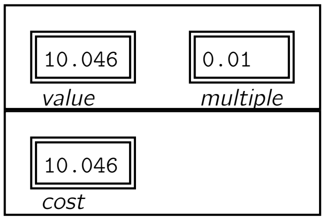


```
cout << round_nearest(cost, 0.01);
```

Now, as we call the round_nearest function, its memory area is carved out on the function call stack. It holds memory for the function's arguments and would also hold space for any local variables had there been any. They are stacked with the called function atop the caller — the function that called the new one, thus it being called the function call 'stack'. The values from the call — that of cost and the literal value 0.01 — are transferred to the function's value and multiple arguments, respectively. Function arguments are always lined up from left to right in this fashion. The compiler doesn't understand the semantics (aka meaning) of each argument — just their data types. As long as those match, it is happy. This is indicated by arrows in the diagram:



At this point, the called function (round_nearest) and the caller exist simultaneously on the stack. But only the function on top is executing on the CPU. The caller has been suspended and is waiting for the called function to return. The stack looks like this as the called function runs:



As the function finishes its computation, the value 10.05 is returned to us and we pass it on to the << operator on cout. This operation, in its turn, takes the same memory — possibly a little more or less — on the function call stack that round_nearest took up. In the end, after the insertion operation returns, the stack looks just like it did to begin with — our cost variable is even still 10.046 — unchanged from its original value by all of the function's work.

Because the function simply receives a copy of the actual argument's value in its own memory area, we call this mechanism of passing arguments the value mechanism. And, in a like vein, we call the arguments themselves value arguments. Some people like the more formal-sounding "pass-by-value arguments", but that's too rich for my blood!

One more quick thing, I said above that the arguments must match in type — but it is a little more involved than that. This is because for simple value arguments, the compiler will allow coercion to occur. So, for instance, we could pass an integer to the `double` arguments of `round_nearest`. Worse, we could pass a `char` or a `bool`!

4.4.1.2 Another Way

Let's say, on the other hand, that a function's arguments weren't value arguments. Let's say they had the right somehow to change our actual argument(s) in place in our memory. This mechanism would allow them to refer directly to our memory locations and both read and write them. We therefore call this mechanism of passing arguments the reference mechanism and the arguments so passed reference arguments.[3]

Why would we want to do this? Well, it can afford us the chance to write a function that produces more than a single result in a single call. With the `return` mechanism, you see, there can be only one result or no results per call. With the reference mechanism of passing arguments, we give the called function write access to our memory. If this involves more than a single argument, the function can write multiple answers to our memory during its execution. This will give us multiple answers upon the function's `return`!

What might this look like on the function call stack? Well, we'd start out just like before with just us on the stack. But let's use a different us and a different function where reference arguments might make sense. We don't need to change the caller's `value` on `round_nearest` — we're `return`ing the answer — and certainly not their `multiple`.

Here is a new function to be called:

```
void swap(char & a, char & b)
{
    char c{a};
    a = b;
    b = c;
    return;
}
```

This function purports to swap the content of the two variables to which it is allowed to refer. We can tell it is referring to its arguments by the ampersand (&) after each argument's type. Note how until now our arguments had no such syntax — just a type and an argument name.

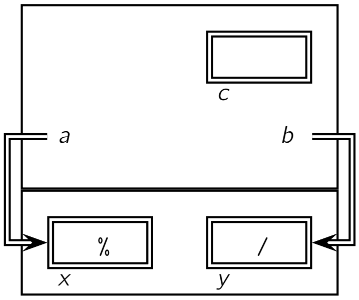
It does this by copying the first variable's value into a local variable named `c`. Then the first variable's memory space is overwritten with the value from the second variable. Finally the second variable's memory is overwritten with the value from the local variable — which used to be that of the first variable!

This trio of assignments is your first real algorithm. Although we've used this term earlier to just denote any sequence of steps to solve a problem, it is also used to signify a solution to a language and platform independent neutral situation. Here, swapping the contents of two memory locations needs a third and three assignments. In future chapters we will study many classic algorithms like this. This becomes especially important in chapter 6 on the `vector` and `array` `class`es for container storage.

We set up a call to this function like so:

```
char x = '%', y = '/';     // sample initial values
swap(x, y);
// now x will be '/' and y will be '%'
```

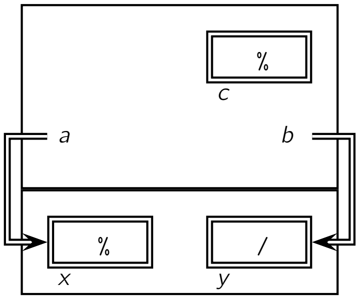
[3] Yes, or "pass-by-reference arguments". *bleah*



As the function is called, the actual and formal arguments are once again aligned from left to right. Thus a will be set to refer to x and b will refer to y. The compiler sets up these references as well as the local variable c.

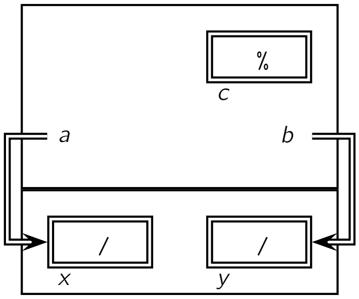
Note that the references take up no new memory — they just exist to refer to the original memory space of the caller's actual arguments. This might look like so on the stack:

When doing the alignment, by the way, the compiler takes special care to match the argument types exactly. This is because the formal arguments are meant to refer to a memory location of an exact size/layout. If this doesn't match the actual argument exactly, disaster could strike!



As the swap function executes, the value of a is accessed to store in c. But as we know, a has no memory of its own. So it must refer down-stack to the memory in its caller — our memory. It was linked to x at the call time, so that is whose value it finds and the value '%' is put in its local variable c:

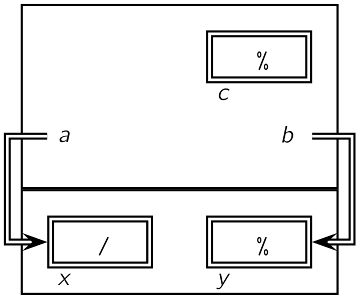
Note how the reference arrows are still there in the diagram unlike the transfer of data arrows from the value arguments in the round_nearest example. Those arrows were just for the call itself and then went away. The reference arrows exist throughout the execution of the called function.



In the next statement, the swap function takes the value of b and puts it in a. Again, b has no memory of its own and so it grabs the value from its referenced memory — that of our y variable. This is then stored in a. But it has no memory and so the value is actually dropped into the referenced memory — our variable x. In memory, things now look something like this:

Awesome! We've got our first result stored in x. And this result will stay there even after the swap function finishes executing and returns control of the CPU to us.

But it does appear that we now have two copies of the '/' character instead of one of those and a '%'. How can this get fixed?



Looking to the next statement, the swap function copies the value of c into b. Since b has no memory, it simply refers to the memory in our stack frame for y. Thus the '%' from c drops into our variable y like so:

Now we have both values back — and they have switched memory locations! The references worked!

But we aren't quite done. The swap function is still running. It next returns. It's frame and the references with it all disappear from the call stack. Even though its return type was void and its return statement empty, we have two answers afterwards!

4.4.1.3 More Details of References

Is that all? Is there anything more to this reference thing? Certainly!

The reference mechanism not only takes up less space than the value mechanism — it doesn't need to make copies into local space, after all. It also executes faster than the value mechanism because of this lack of copying. So we could use the reference mechanism to speed up our function calls. But that wouldn't be safe for our variables which might be accidentally or maliciously changed. It also wouldn't work if we wanted an actual argument that was a literal or constant — the former has no memory to refer to and you can't change the latter!

Not to worry! We can fix all those things with a simple keyword. Some of you may have guessed it already: const. This can be used to mark the formal argument to not change during the function's execution. When combined with the reference mechanism on an argument, it makes the argument as fast as a reference but as safe and flexible for the actual argument as if using the value mechanism.

Where might this come in handy? Should we use it on all our arguments that don't need to change? No, we should use it only on arguments of a larger size than the builtin types. Builtin types all run at the standard speed of the computer (whatever gigahertz your CPU is rated at). But larger types — like strings — take more time to transfer and set up in new space. This makes them ideal candidates for passing by constant reference. For instance, we could alter our previous welcome function to use this for its string argument:

```
void welcome(const string & prog_title);
```

I'm only showing the prototype here because the definition only need change its head. The body doesn't have to change at all just because its argument is moving in slightly differently. That is, we still use const& arguments just like we used value arguments except that we can't change them. (This does mean that the variation where we changed the prog_title argument instead of making a local welcome string can't work now. But we didn't like it that well, anyway.)

In fact, this leads to a general rule for passing any class object: pass by reference to change the original or by const& to avoid both changes and copying the object.

4.4.1.3.1 Caveat to the Reference Rule

Let's look at our code to print an uppercase version of a string again. Making a string uppercase is actually a fairly reusable tool. Maybe we should make it into a function. Let's start with the code we want to make into a function. We could uppercase a string like this:

```
for ( string::size_type i = 0; i != s.length(); ++i )
{
    s[i] = static_cast<char>(toupper(s[i]));
}
```

We couldn't use a range-based for loop because each pass makes a copy of the char at the next location, right? Yes, that was the case before. But now that we know about references, it's time to see them all over the place! Or at least here. With some reference syntax we can make our range-based for loop able to change the elements of the string as it goes by:

```
for ( char & c : s )
{
    c = static_cast<char>(toupper(c));
}
```

Much more compact! Still need the typecast to avoid the integer to char conversion warning, but really nice on the for head!

Now to put this into a function. We have a few design choices as to the string argument. We clearly need this string s to come into the function. c is local to the loop and so local to our function as well. But the string argument can be value, reference, or even const&.[4]

[4] BTW, some people even code the type for a constant reference argument as type const & instead of const type &.

So, which do we use? Let's look at each and its repercussions and then decide.

Value Version: This would look like so:

```
string toupper(string s)
{
    for ( char & c : s )
    {
        c = static_cast<char>(toupper(c));
    }
    return s;
}
```

This one makes a copy of the caller's `string` into our own memory space and calls it `s`. We then walk through each character in `s` by reference and change it to its uppercase form. This changes the copy of the caller's `string` — not the caller's actual `string`. Then we `return` a copy of our now uppercase `s` to the caller for them to print or store as they see fit.

That's a lot of copying and if the caller stores the result back into their original `string`, we've wasted all that memory and time. The only way this makes sense is to store the result somewhere else or to insert the result directly to `cout`.

Reference Version: The code for this version looks like this:

```
void toupper(string & s)
{
    for ( char & c : s )
    {
        c = static_cast<char>(toupper(c));
    }
    return;
}
```

Note that we not only changed the argument to a reference but also changed the `return` type to `void`. This is because we don't need to send back a copy of something we've already stored directly in the caller's memory.

This variant works well when the caller wants to change their original `string` but not so well when they just want to print the result. Then they have to do something like this:

```
toupper(s);
cout << s;
```

Not a simple one-liner. Also, looking at this call, we realize it isn't like the `cctype` `toupper` function which `return`s its result. That might be a little off-putting to callers.

Constant Reference Version: This variation looks like this:

```
string toupper(const string & str)
{
    string s{str};
```

This helps them visually see that the reference is unchanging. Others don't like reversing the `const` from their usual place for it. I leave it up to you and your teacher to decide.

```
    for ( char & c : s )
    {
        c = static_cast<char>(toupper(c));
    }
    return s;
}
```

This one takes the constant reference as formal argument str and then immediately copies it into local variable s. This is because some string must change to all uppercase and it can't be str as it was marked const.

Since we can't change the actual argument this time, we also have to send back our result via the return mechanism again.

Making a local copy like this is a little time/space consuming and reminds me of the copy made by the first version. In fact, it takes the same amount of overhead to do the value version of this function as it does this one. And here we thought const& was a panacea to solve our large object passing woes!

Conclusion: The results seem clear: value wins. Even though we are creating a changed form of the string, we don't necessarily want to change it in place. Also, the function not performing like the char version is a little odd.

The rule needs amending, clearly:

- pass by reference to change the object in-place
- pass by constant reference to view the object but not change it
- pass by value to change the object but not the original

This last one doesn't happen a whole lot, but it is well worth having in there — just in case.

4.4.1.3.2 Function Design Again

Having written a nice toupper function for strings, we turn immediately to the name-casing situation from earlier. We had settled on this variation (now with a range-based for):

```
for ( char & c : s )
{
    c = static_cast<char>(tolower(c));
}
s[0] = static_cast<char>(toupper(s[0]));
```

Note that we can use a reference on the element catcher in the range-based for loop. This allows us to make changes to the element as we go past it! Very handy. . .

As a function (and learning from our uppercase experience) it would look like this:

```
string name_case(string s)
{
    for ( char & c : s )
    {
        c = static_cast<char>(tolower(c));
    }
    s[0] = static_cast<char>(toupper(s[0]));
    return s;
}
```

But that for looks an awful lot like the uppercasing function we made before. In fact, it could be a reasonable addition to our growing 'family' of functions here. (A family of functions is a group of functions that share common characteristics and are, in fact, often tightly related. We don't always need all of them in a particular application, but they often go together and we let the compiler strip them out if unused.)

It could be realized thusly:

```
string tolower(string s)
{
    for ( char & c : s )
    {
        c = static_cast<char>(tolower(c));
    }
    return s;
}
```

And then we can rewrite name_case to use it:

```
string name_case(string s)
{
    s = tolower(s);
    s[0] = static_cast<char>(toupper(s[0]));
    return s;
}
```

Nice! Now if we need to manage the case of some strings, we'll know to come looking for this function family. (And in a few sections — 4.5.2 — we'll make it even easier to reuse them!)

4.4.2 Function Overloading

Overloaded functions are [two or more] functions which:

- have the same name

but

- either have a different number of arguments
- or have different types of arguments.

The compiler can tell which function is being called by merely counting the number of actual arguments or noting that the actual arguments' types match one function better than another. The match is done in our usual left-to-right manner.

The actual job these functions perform would ideally be similar if not the same. Otherwise, they'd probably have had different names, right?

Note that the return type is **NOT** involved in this process at all!!! Since the returned value can simply be thrown away — not used — the compiler cannot be assured that it can match return types during a call. So it doesn't even try.

4.4.2.1 Revisiting Rounding

Let's take a look back at our round_nearest function with an eye toward overloading. If we were to overload with it, we might choose this approach:

```
// Function to round caller's value to the nearest integer
// (or whatever multiple they desire -- see multiple
// argument).
//
double round_nearest(double value, double multiple);

// As above, but assume one's place...
double round_nearest(double value);
```

We've just declared the functions so far, but already you can see the point to the separation. With the second function the caller doesn't have to type , 1.0 all the time. The definitions are similar but subtly different:

```
// Function to round caller's value to the nearest integer
// (or whatever multiple they desire -- see place argument).
//
// NOTE:  The place doesn't have to be a multiple of 10, it
//        can be a multiple of 5 or 2.5 or 0.43 or
//        ~anything~!
double round_nearest(double value, double multiple)
{
    return round_nearest(value / multiple) * multiple;
}

// As above, but assume one's place...
double round_nearest(double value)
{
    return round(value);
}
```

Note how the one's place overload doesn't have to scale the value — just round it. It might seem silly to write our own function to just call the cmath function round, but it makes this a package deal — the caller doesn't have to remember the cmath function at all — just our round_nearest suite.

There is another advantage to this setup, as well. The general overload only focuses on the scaling aspect rather than the rounding part. It passes that off to the one's place version. This often earns the smaller function the designation of a 'helper' function since it serves to help the more work intensive function do its job.

Instead, we could have coded the functions like so (comments removed to help you focus):

```
double round_nearest(double value, double multiple)
{
    return round(value / multiple) * multiple;
}

double round_nearest(double value)
{
    return round_nearest(value, 1.0);
}
```

This saves work for the one's place function by simply calling the more general overload with the value the caller left off. But it puts more work on the general overload.

However, the earlier separation not only helped with the workload balance, but also separated the two aspects of a general rounding scheme: rounding to the one's place and scaling to the one's place. This helps us debug the functions by helping us isolate possible problems with good test cases.

4.4.2.1.1 Testing Functions Adequately

Speaking of testing, how might we adequately test these two functions before putting them into a program? Test them first? Yes, of course! Why put them into production before they are known to work? We often write separate programs to test a function or two [or three or. . .]. Such a program might look like this:

```
int main(void)
{
    char yes_no;
    double x, mult, ans, ans1;
    cout << "Test round_nearest() function?  ";
    cin >> yes_no;
    cin.ignore(numeric_limits<streamsize>::max(), '\n');
    while (toupper(yes_no) != 'N')
    {
        cout << "\nEnter number to round:  ";
        cin >> x;
        cout << "Enter multiple to round to:  ";
        cin >> mult;

        ans = round_nearest(x, mult);
        ans1 = round_nearest(x);

        cout << "\nRounding " << x << " to the nearest "
                "multiple of " << mult << " I got " << ans
             << ", is that okay?\n";
        cout << "\n(BTW, I took the liberty of rounding to "
                "just the one's place and got " << ans1
             << ", neat, hunh?)\n\n";

        cout << "Test round_nearest() function again?  ";
        cin >> yes_no;
        cin.ignore(numeric_limits<streamsize>::max(), '\n');
    }
    return 0;
}
```

Here we ask a user — the tester[5] — to enter both potential arguments to the function pair. We then call the function with both argument patterns and record the answers. Finally we print the results out and nudge them to mark that down in their log with a phony question we don't intend to read a response to. Their log could be a spreadsheet with a list of test cases and expected answers where they

[5] We are the tester here, but this will often be a person from the dedicated testing department who runs tests for all projects in the company.

log a checkmark vs. an actual — erroneous — result and any other details they feel appropriate to help the coder fix the issue.

We wrap the whole thing in a yes/no loop to ease the tester's job. If we had more disparate functions to test — perhaps we'd also written a whole family of rounding functions like `round_up` and `round_down` with the scaling features of our `round_nearest`, we would probably provide a menu to the tester instead of just a yes/no loop.

4.4.2.2 Too Similar Types in Overloading

What if we have used different types in our overloading, but they are similar in that they have a common type that looks like them all? Let's use a random number suite to test this idea:

```
// randomly generate a value within the inclusively bounded
// range specified
short  rand_range(short min,  short max);
long   rand_range(long min,   long max);
double rand_range(double min, double max);
```

Here we've got a family of `rand_range` functions with very similar argument types: two integer types and a floating-point type. These being all numeric types are similar to the raw literal type `int` and although calls like these:

```
cout << rand_range(4.2, 42.0) << '\n'; // okay, calls double version
cout << rand_range(4L, 42L) << '\n';   // okay, calls long version
```

will succeed with ease, a simple one like this:

```
cout << rand_range(4, 42) << '\n'; // eek!  should the 'int' be made
                                   // into short, long, or double
                                   // for compatibility?!?
```

fails miserably!

The compiler gets confused between the three overloads because `int` can easily — in just one step each — convert to `short`, `long`, or `double`. Since the coercions are so equitable, the compiler reports an ambiguity between the overloads. These warnings are often confusing to new programmers, so don't fret! They become more readable as you see them more and learn to read the compiler's messages.

How can we fix this issue? We can take one of two approaches: typecasting or helper constants. These look like so:

```
// to fix this, use a typecast:
cout << rand_range(static_cast<short>(4), 42) // maybe don't need two casts
     << '\n';                                 // okay, calls short version

// or use helper constants (or variables):
const short m = 4, M = 42;
cout << rand_range(m, M) << '\n';       // okay, calls short version
```

Here the programmer has cleverly named their minimum value `m` in contrast to the maximum value of `M`. This is bad practice in general, but might work okay in a one-off situation.

But now the compiler knows clearly which function to call either way we fix the situation. Note that we only had to cast one of the arguments to `short` for the compiler to figure out which call to make.

This made one of the arguments an exact match and therefore it only had to coerce one argument and that made it happy enough to not call out "ambiguity!"

Before we shift gears slightly, though, let's refresh our memory of how these functions might look:

```
short rand_range(long min, long max)
{
    return min + rand() % (max - min + 1);
}

short rand_range(short min, short max)
{
    return static_cast<short>(min + rand() % (max - min + 1));
}

double rand_range(double min, double max)
{
    return min + rand() % RAND_MAX / (RAND_MAX - 1.0) * (max - min);
}
```

4.4.2.3 Reference Types and Overloads

One type-based overload that won't ever cause ambiguity is when overloading based on a reference type. Let's look at some functions for inputting values from the user:

```
double read_numeric(double & num, const string & prompt, const string & errmsg);
long   read_numeric(long & num,   const string & prompt, const string & errmsg);
short  read_numeric(short & num,  const string & prompt, const string & errmsg);
```

These functions will use a `fail` loop to protect the input from non-numeric-ness. They don't concern themselves with domain validation so we can keep the argument list down to essentials. That can be added with another layer of function like `read_range` or something similar.

Note that the compiler can't get confused on a reference argument because the types for references can't be coerced. This isn't so with the constant reference arguments for the messages, of course, but is for the plain references for the answers. Why is the answer being both `return`ed and referenced? This gives the caller the option of holding a backup or 'undo' value with ease. In case the user changes the input during a menu choice or the like and wants to go back to the original without retyping it.[6]

Again, for completeness and review, let's look at these function's definitions:

```
double read_numeric(double & num, const string & prompt, const string & errmsg)
{
    cout << prompt;
    cin >> num;
    while ( cin.fail() )
    {
        cin.clear();
        cin.ignore(numeric_limits<streamsize>::max(), '\n');
        cerr << errmsg;
        cout << prompt;
        cin >> num;
```

[6] Perhaps not of great use here with simple numbers, but a good technique to keep in mind for larger data like image files or the like!

```
    }
    return num;
}

long read_numeric(long & num, const string & prompt, const string & errmsg)
{
    cout << prompt;
    cin >> num;
    while ( cin.fail() )
    {
        cin.clear();
        cin.ignore(numeric_limits<streamsize>::max(), '\n');
        cerr << errmsg;
        cout << prompt;
        cin >> num;
    }
    return num;
}

short read_numeric(short & num, const string & prompt, const string & errmsg)
{
    cout << prompt;
    cin >> num;
    while ( cin.fail() )
    {
        cin.clear();
        cin.ignore(numeric_limits<streamsize>::max(), '\n');
        cerr << errmsg;
        cout << prompt;
        cin >> num;
    }
    return num;
}
```

We note the redundancy here in the codes. Why do we even need three separate functions? Because the actual action of reading the different types is changing. In fact, it is realized by cin's extraction operator (>>) via overloading of a special sort. We may discuss this in a later course or later in this course as time permits.

But never fear! We will learn a technique soon (section 4.6.1) for removing this redundancy.

4.4.2.4 To Sum Up

That was a lot, perhaps, to deal with, so here is a summary of the benefits of overloading:

- only having to come up with one name when we don't need different names (that is, when the concept is the same)
- flexibility for the caller who can call with just the information they have and need worked with in their situation
- sometimes allowing for more efficient code for special cases (as with the round to the one's place above)
- although we have to write multiple functions, they can often rely on one another to do part of each other's work

- even when they do rely on one another, it helps to debug based on observed logic errors and which function's code must have been responsible for that issue

Care should be taken when overloading based solely on types to make sure that — for all but non-const references — you don't fall into coercion traps (aka an ambiguity). If you do, a simple typecast can [typically] get you back out with no problems.

4.4.3 Default Arguments

Another argument tool is defaulting arguments to special values. This is akin to how many dialog boxes have defaults selected for various bits and bobs in them. Consider the simple interface a printing dialog gives you by pre-selecting a printer, number of copies, and even the Ok button is the default when you just hit Enter/return. Each of these values has a default status that serves well in most cases. But, at your discretion, each can be changed to a value that serves you better right now.

This is similar to how defaulted argument values can aid your function's caller. A default value for an argument to a function can provide a flexibility or convenience for your function's caller. If there are certain parameters which are often the same but may sometimes need to change, you can leave them as parameters but set a default value for them. This way, should your caller need to provide special values, they can; but typically they can simply allow the values to default.

That is, defaulted argument values allow the caller of a function to provide their value for an argument or leave it off to use a default value instead.

4.4.3.1 Rounding Yet Again

One example of using a default argument could be our rounding suite from earlier. Instead of overloading two functions, just provide a default argument for the one missing from the helper overload:

```
double round_nearest(double value, double multiple = 1.0);
```

The compiler then knows that this function can be called as:

```
round_nearest(x)   // defaults to rounding to the
                   //  nearest whole number
```

or still as any of:

```
round_nearest(pay, .01)   // round to nearest penny

round_nearest(calc, .0001)  // round to nearest ten-thousandth

round_nearest(minutes, 15)  // round to nearest quarter hour
```

It just depends on the caller's needs...

We should take a careful look at this function's definition, however, as things might not be as you first expect them to be:

```
double round_nearest(double value, double multiple /* = 1.0 */ )
{
    return round(value / multiple) * multiple;
}
```

Here we do **not** specify the default value on the parameter! We are, in fact, forbidden to do so by the standard. What would happen, for instance, if we were to accidentally change the default in the prototype but not the definition? The compiler wouldn't know which default to use, now would it? Sure, a compile-time error isn't a big deal to us by now, but surely we could avoid it by just having the default listed only once by rule.

Can you place the default on just the definition instead? No. How, after all, would the compiler know what value to substitute for a missing parameter if it were not listed until the definition — all the way after main? It must be defined on the first head of the function the compiler sees — the prototype here.

This might seem less efficient, of course, because we are now sometimes multiplying and dividing by 1.0. Indeed, it is less efficient. But it saved the programmer countless seconds of typing and debugging to have just the one function instead of two.

4.4.3.2 Caveats & Guidelines

There are some issues with defaulting arguments, however, that bear discussion:

- Default argument values can only be listed on a single function head; you may not repeat them on the other function head! Therefore, we typically place default values in the prototype as it is the first head the compiler will see (and also the only head the caller is likely to see before the function is called).
- Arguments with default values must be at the end of the argument list.
- Caller may leave actual arguments off the end to indicate that they wish to use the defaulted value for those arguments. I.E. they cannot skip a defaulted argument and fill in a later one:

  ```
  f( arg_value, arg_value, , default_replacement)
  ```

- It would be best if you could place those arguments whose [defaulted] values are least likely to be changed further back in the list. If you are unsure, perhaps a poll of your coworkers could help? But if you cannot make such a determination, that's okay.
- Non-const reference arguments are almost never defaulted since we make no global variables to which they can refer. (We do have cin, cout, etc. which are global, but we don't know their actual types yet. . . that's for later. *smile*)
- Constant reference arguments may be given a default value. This can be used to good effect especially with string parameters meant for prompts, error messages, or the like:

  ```
  f( ..., const string & prompt = "" )
  ```

4.5 Tools for Better Functions

This section deals with tools and techniques to help you make functions more reusable and more efficient as well as making the processes of creating and testing functions a little easier.

4.5.1 [Re]Factoring

Let's talk again about factoring. You may recall our discussion of factoring a branch from section 3.9.3. It simply means removing redundant code to a common place. Here we'll take the redundant or excessive code and place it into a function for re-usability.

4.5.1.1 Budgeting Woes

Let's say we had code to take in the user's budget amount and remember their monetary unit and its placement for a later report. It might look like this:

```
char pre_unit, post_unit;
bool unit_in_front, unit_entered;
double budget;
const char default_unit = '$';
const bool default_unit_side = true;  // in front/left of the money

unit_entered = false;
cout << "\nEnter last year's budget:  ";
cin >> ws;
if ( ispunct(cin.peek()) )
{
    cin >> pre_unit;
    unit_in_front = true;
    unit_entered = true;
}
cin >> budget;
if ( peek_ahead() != '\n' )
{
    cin >> post_unit;
    cin.ignore(numeric_limits<streamsize>::max(), '\n');
    unit_in_front = false;
    unit_entered = true;
}
if ( ! unit_entered )
{
    unit_in_front = default_unit_side;
    if ( unit_in_front )
    {
        pre_unit = default_unit;
    }
    else
    {
        post_unit = default_unit;
    }
}
```

While there are certainly other tweaks we could do to this code, it will work fine to gather the user's input and assign a unit for the money in the report as well as whether that unit should be displayed before or after the money's value.

It has already been factored somewhat in that we are reusing our old friend `peek_ahead` to remove a run of whitespace that might precede an end of input.

However, this pre- and post- `peek`ing is tedious and cumbersome. With proper factoring and a little clever initialization, we can avoid quite a bit of it.

Let's design two functions for pre- and post- `peek`ing respectively:

```
// read an optional char from after an [numeric] input;
// default is returned if no value is entered by user.
char get_opt_post(char def_value)
{
    if ( peek_ahead() != '\n' )
    {
        cin >> def_value;
    }
    return def_value;
}

// read an optional char from in front of a numeric input;
// default is returned if no value is entered by user.
char get_opt_pre(char def_value)
{
    cin >> ws;
    if ( ! is_numeric( static_cast<char>( cin.peek() ) ) )
    {
        cin >> def_value;
    }
    return def_value;
}
```

While the `ispunct` test was good for our typical situation, we decided here to branch out to allow for letters as well. To do so, we allow the non-optional item to be anything that can start a number. This was factored out to a helper function as well:

```
// true if ch is a digit, ., -, or +.  false otherwise.
//
// basically anything that can ~start~ a numeric value.
bool is_numeric(char ch)
{
    return isdigit( ch ) // IS a digit
           || ch == '.'  // OR a decimal point
           || ch == '-'  // OR a 'negative' sign
           || ch == '+'; // OR a 'positive' sign
}
```

Quibble over the name all you want, this is a helpful function in many situations.

So, to use the optional notation helper functions, we could do this in our prior program:

```
char pre_unit, post_unit;
double budget;
const char default_unit = '$';
const bool default_unit_side = true;  // in front/left of the money

cout << "\nEnter last year's budget:  ";
pre_unit = get_opt_pre('\0');
cin >> budget;
post_unit = get_opt_post('\0');
if ( pre_unit == '\0' && post_unit == '\0' )
{
    bool unit_in_front = default_unit_side;
    if ( unit_in_front )
    {
        pre_unit = default_unit;
    }
    else
    {
        post_unit = default_unit;
    }
}
```

That's about half as much code. Only setting the default for the local region is bulky now.

Does any of this help at report time? Well, our initial report might have looked like this:

```
cout << "\nYour new budget will be ";
if ( unit_in_front )
{
    cout << pre_unit;
}
cout << new_budget;
if ( ! unit_in_front )
{
    cout << post_unit;
}
cout << " next year.\n";
```

Now our report can be as simple as:

```
cout << "\nYour new budget will be " << pre_unit << new_budget
     << post_unit << " next year.\n";
```

Here we've taken advantage of the fact that null characters don't normally display on screen at all to remove the ifs. If that isn't the case for your target system, then you'd need to protect their display by testing that they weren't null on either side:

```
cout << "\nYour new budget will be ";
if ( pre_unit != '\0' )
{
    cout << pre_unit;
}
cout << new_budget;
if ( post_unit != '\0' )
{
    cout << post_unit;
}
cout << " next year.\n";
```

While not an improvement in lines of code, it does save a variable in the long run!

4.5.1.2 Reading Coordinates

Another situation that lends itself to the above optional notation helpers is input of coordinates in the plane. Many people will leave off standard notation like the parentheses around the coordinates or even the comma that separates them! To help with this, we can code this:

```
t = get_opt_pre('\0');
if ( t == '\0' )
{
    cout << "missing (";
}
else if ( t != '(' )
{
    cout << "invalid:  need (";
}
cin >> x1;
t = get_opt_pre('\0');
if ( t == '\0' )
{
    cout << "missing ,";
}
else if ( t != ',' )
{
    cout << "invalid:  need ,";
}
cin >> y1;
t = get_opt_post('\0');
if ( t == '\0' )
{
    cout << "missing )";
}
else if ( t != ')' )
{
    cout << "invalid:  need )";
}
```

Here `t` is a `char` and `x1` and `y1` are both `double`s. The messages need work, but the intent is clear and we've used our helper functions. However, it still seems redundant, doesn't it? Both of the sections for the x and y coordinates seem similar. Let's make sure. In the olden days, we would have had to print both fragments and hold the papers up to the light. If they were clear, we had redundancy and any fuzzy parts were differences we could use for parameters or results from the new function.

Nowadays, however, we have automated difference checkers. They go through two files[7] and tell what they have in common and what is different. Some are more terse and hard to read but make automated adjustments easier. The ones we are interested in, though, are going to make it visibly clear what is the same versus different. Such graphical or at least nice text mode apps are so common they have a dedicated page or two at Wikipedia.

Which one you end up using will depend on your system, what's even installed there, and your preferences once you've tried some of them. But here are my recommendations for free tools:

[7] You'll have to make a separate file for each fragment of code you feel is redundant to run it through the checker.

Mac: On macos, the going standard is FileMerge. This comes with your XCode install. You can use it from the command prompt/Terminal app with the moniker 'opendiff'.

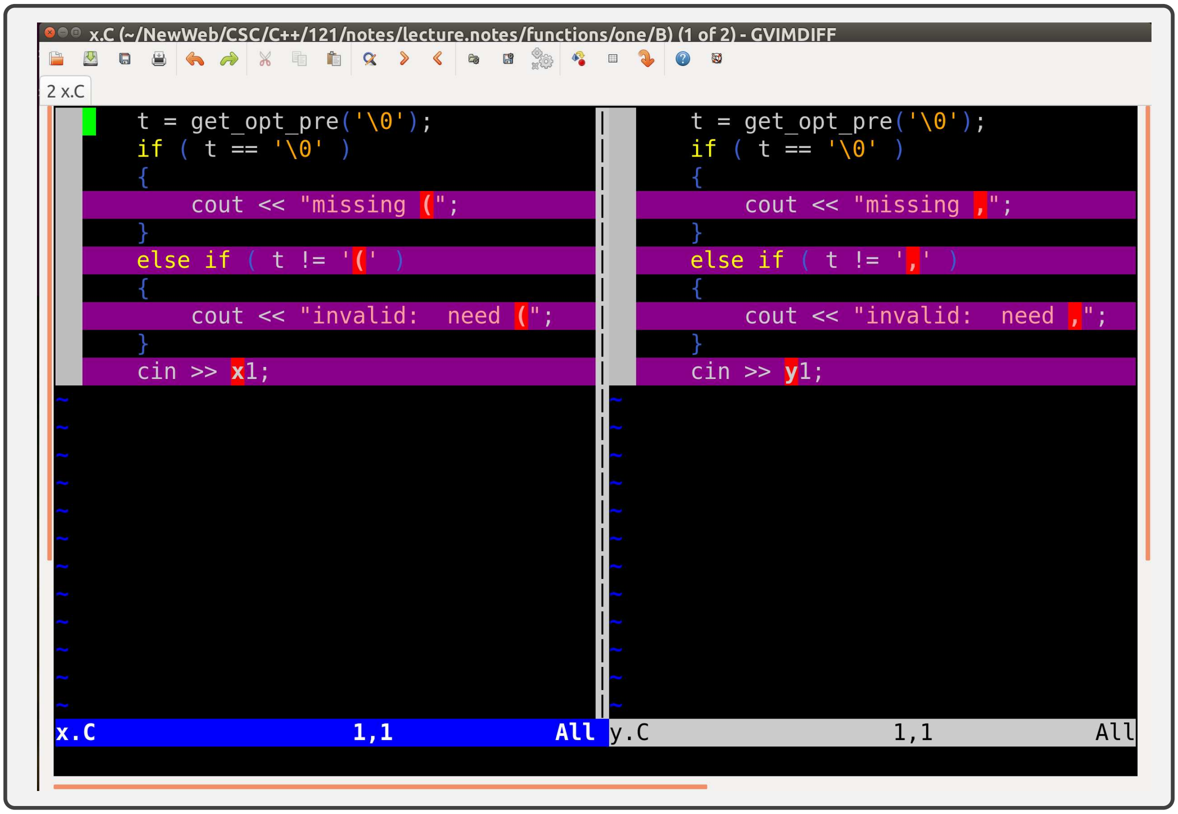
Windows: On Windows, the going standard is, I believe, WinMerge. (Visual Studio used to come with one, but it has been discontinued.)

Linux: There are many competing standards here, but vimdiff is good for editing the compared files. I like xxdiff for just looking for differences but it is experiencing problems on newer Ubuntu systems right now. Also, kdiff3 is a popular option and comes with the KDE desktop. It has the interesting feature that it works with up to three files at once — hence the name.

If you are interested in using the same software on several systems, there are many options, but I think Meld is pretty popular.

For more on these tools and the many other options, check out the Comparison of file comparison tools page at Wikipedia.[8] For more on how the tools work and their history, see the File comparison page instead.

Anyway, if we run the above fragments through one of these checkers, we can see the differences displayed like so:



x.C (~/NewWeb/CSC/C++/121/notes/lecture.notes/functions/one/B) (1 of 2) - GVIMDIFF

2 x.C

```
    t = get_opt_pre('\0');
    if ( t == '\0' )
    {
        cout << "missing (";
    }
    else if ( t != '(' )
    {
        cout << "invalid:  need (";
    }
    cin >> x1;
```

```
    t = get_opt_pre('\0');
    if ( t == '\0' )
    {
        cout << "missing ,";
    }
    else if ( t != ',' )
    {
        cout << "invalid:  need ,";
    }
    cin >> y1;
```

```
x.C                 1,1          All   y.C                 1,1          All
```

This shot is from the GUI version of vimdiff and we can clearly see from the highlighting what is duplicated — no highlighting — and what is changed — purple for lines and red for differences within the lines.[9] From this information, we can see that the character that should by default precede the coordinate is different, the variable that we read into is different, and that the messages are mostly the same, but do contain some changes. Thus, we might merge these two into the following function:

[8] Don't you just love the double use of 'comparison' in the page title? *chuckle*

[9] Yes, colors are configurable in most of these difference checkers.

```
// retrieve a numeric value from keyboard stream with an optional
// preceding character -- the leader/header; perhaps a unit or notational
// symbol?  the leader is extracted and checked against a desired symbol.
// if the leader is not the desired symbol, an invalidity message is printed.
// if the leader is not present at all, a message is printed indicating that
// this symbol was missing altogether.  the caller can customize these
// messages with the last two arguments.
double get_numb_with_lead(char desired_leader, const string & missing_msg,
                          const string & invalid_msg)
{
    double number;
    char t = get_opt_pre('\0');
    if ( t == '\0' )
    {
        cout << missing_msg;
    }
    else if ( t != desired_leader )
    {
        cout << invalid_msg;
    }
    cin >> number;
    return number;
}
```

Note that the default character and messages have become parameters to the function and the coordinate being read is now the return value of the function.[10]

We might now call this function like so:

```
x1 = get_numb_with_lead('(',
                        "Open parenthesis missing!\n",
                        "Point must have a parenthesis before x-coordinate.\n");
y1 = get_numb_with_lead(',',
                        "Comma missing!\n",
                        "Point must have a comma between coordinates.\n");
```

Now it doesn't hurt to have nice messages. The code is immensely stripped down and we have our new re-usable function! The post-peeking code remains the same, but I'll leave its extraction into a function to your exercise. *smile*

This factoring of already factored code, btw, is often termed re-factoring. Basically the same thing, but with now multiple layers of function goodness.

4.5.2 Separate Compilation

We've gotten some really useful functions now, it would be nice to stop copying and pasting them between programs, wouldn't it? Wouldn't it be nice to just #include them like we do a library? Well, let's do it!

Yep, we can build our own libraries. This process is known as separate compilation because we put the C++ code in a separate file. Also the compiler will actually translate the two separate C++ files (the application and the library) into binary separately and then merge them together (this is called linking).

[10] After I named this function I realized the horror that was the idea of "getting numb with lead". We definitely need a new name, but I leave that to you.

4.5.2.1 A Tale of Two Files

So, how do we make a library? First thing to know: a library actually comes as two files. No, not the application and library. The library itself is two files: the interface and the implementation. We generally only see the interface file as it is the one we #include in our application. The implementation files for the standard C++ libraries are already in binary form and the compiler picks those up for linking automatically.

The interface file typically contains the prototypes of all functions available in the library. Consisting mostly of function heads, the interface file's name is typically ended with '.h' — (heads, get it?). In addition to function heads, though, the library's interface file can also contain constant definitions and typedefinitions.

The implementation file is much simpler. It begins by #include'ing its associated interface file and then defines those functions which were prototyped in the interface. Since it actually contains C++ codes for these functions — not just their heads — we usually end the file name in a '.cpp' just like a normal C++ file. But it will **never** have a main because it is just a collection of functions.

Since these two files are part of the same library, they share a base name. That name, like all of our identifiers, should reflect the content of the library — what is its purpose — what is the purpose of its constituent functions. Name your library after the common theme of the functions it supplies. You do have a common theme, don't you?

4.5.2.2 The Interface File

Due to the fact that interface files are #included so many times, it became necessary early on to compensate for the possibility of accidentally including them multiple times — in the worst cases in a circular fashion! This leads to potential warnings or even errors about duplicate symbols/definitions. Worse, for circular inclusions, we might either crash the compiler or cause an infinite loop in the compiler. (Which depends on how the compiler was written.)

The solution to both multiple as well as circular inclusion turned out to be a set of #if pre-processor codes. Together these lines are known as multiple-inclusion protection, circular-inclusion protection, or just the inclusion guard.

```
#if ! defined( UNIQUE_SYMBOL_FOR_THIS_LIBRARY_INTERFACE )
#define UNIQUE_SYMBOL_FOR_THIS_LIBRARY_INTERFACE

   // prototype, const, typedef, etc.

#endif
```

How this works is that we check whether a symbol/identifier unique to our library has been defined or not. If not, we start by defining it and then proceed to the prototypes and such that make up the library's offering. If it was defined before, then we've already been here and we skip to the #endif at the end of the file — avoiding all those duplicate warnings and errors.

This is quite tedious to type and making a unique symbol for each library doesn't really have to make it such a long name. Thus we came up with the slightly simpler:

```
#ifndef LIB_NAME_H_INC
#define LIB_NAME_H_INC

   // prototype, const, typedef, etc.

#endif
```

Here the pre-processor directive *#ifndef* asks all at once IF a symbol has Not been DEFined. This serves two nice purposes together: making the two lines line up nicely to make it easier to ensure the symbol is the same on both lines and making it easy to copy/paste the test line down to create the definition line. I do it in about 12 keystrokes in my editor (Vim) — no mouse intervention necessary.

4.5.2.2.1 Modern But Is It Better?

Modern compilers almost all implement a directive known as a `pragma`. One of these `pragmas` is:

```
#pragma once
```

This is touted as a one-line replacement for the above trio of pre-processor directives. But it is known to have edge cases where it does not work — the compiler gets confused and includes the file twice or not at all instead of once.

There are, therefore, programmers who put both the `pragma` AND the `ifndef` structure in their `.h` files! I recommend just the `ifndef` structure for now and keep an eye on compiler `pragma` effectiveness to decide when it becomes a truly universal tool.

4.5.2.3 Using a Library

To use a library in another piece of code you must, of course, *#include* the library’s interface file in the other code file:

```
#include "lib_name.h"
```

Why the quotes instead of angle brackets? Why list the ’.h’? One thing at a time. For the first question, it is because the angle brackets we’ve used on standard libraries mean that the compiler can find them in the standard installation directories/folders. The double-quotes, on the other hand, mean the compiler can find the file in question in the current directory.[11]

For the second question, it is because we saved our file with the `.h` extension, right? The standard libraries for C++ have interface files without extensions. This was done to help folks distinguish the libraries for C++ from those of our ancestor language C which did all end in the `.h` extension. And don’t forget that we also changed all our inherited library names to start with an [extra] ’c’ to indicate that heritage.

But just as critical as *#include*’ing the header/interface is to compile the other code file along with the library’s implementation file! This is done in different ways on different systems. On my Linux machine, I list the names of all the C++ files (not the `.h` files) on the command line of the compiler execution. When I’m on XCode on my Mac, I make sure all the files — `.cpp` and `.h` — are listed in the project. In Visual Studio Code — on either box — I make sure all my files are in the same folder together. And, if I’m not mistaken, Visual Studio has you list the `.h` and `.cpp` files in separate folders labeled ”Header Files” and ”Source Files” respectively.

Once these things are done, the compiler knows what `.cpp` files to compile separately and then link together to form the binary/executable. That last thing is done by the part of the compiler known as the linker, of course. It brings together the separate object files[12] from each compiled `.cpp` file as well as some system-specific binary code and links them together to form the final executable.

[11] All right, if the compiler doesn’t find a double-quoted file in the current directory/folder, it will then fall back on the standard install locations. But making all of them double-quotes would slow down a typical compile tremendously so we always use the right delimiters for the task at hand.

[12] This has absolutely nothing to do with the object-oriented programming we’ll study in chapter 5. It has to do with an old name for binary code.

4.5.2.4 A Driver

A particularly important thing to do for any new library is to make a driver program to test all of its functions. Recall that a driver tests a function like a test driver tests new automobiles on the manufacturer's private track. We write a driver program whose entire purpose is testing to test all the functions provided in a library.

What does this look like? It is typically lots of looping. It can be as simple as a series of do or while loops surrounding each function's testing. Or it could be as interesting as a menu-driven app where each function is an option and the user gets to choose what function to test and in what order. This last is usually the best way to go, in fact, but some new programmers will balk at such a task thinking it is too big of a design.

What could these loops look like? Well, they need to gather the inputs the user wants to test the function with, then call the function, and then report the results if any from the function. After that, they can use a simple yes/no mechanism to let the user decide if further testing is in order. Perhaps something like this:

```
do
{
     // read test inputs
     cout << "What value to round?  ";
     cin >> num;
     cout << "What place to round to?  ";
     cin >> place;
     // call function
     result = round_nearest(num, place);
     // print result(s)
     cout << "\nRounding " << num << " to the nearest " << place
          << " is " << result << ".\n";
     cout << "\nTest again?  ";
     cin >> ans;
} while ( toupper(ans) != 'N' );
```

Of course, this will force the user to test each function at least once. So if you don't want to do that, you could use a while loop:

```
cout << "Test function ___?  ";
cin >> ans;
while ( toupper(ans) != 'N' )
{
     // read test inputs
     cout << "What value to round?  ";
     cin >> num;
     cout << "What place to round to?  ";
     cin >> place;
     // call function
     result = round_nearest(num, place);
     // print result(s)
     cout << "\nRounding " << num << " to the nearest " << place
          << " is " << result << ".\n";
     cout << "\nTest again?  ";
     cin >> ans;
}
```

If going for the menu-driven driver, you can use the do loop version inside the branch that matches that function's option selection. (Remember, no more than 9 functions per menu level so use sub-menus to group together similar functions if necessary!)

4.5.2.5 Examples

Let's look at a few examples to help you get your feet wet. Let's put the optional notation crew in library form to start. The interface file would look like this:

input.h

```
#ifndef INPUT_H_INC
#define INPUT_H_INC

#include <string>

// read an optional char from after a [numeric] input;
// default is returned if no value is entered by user.
char get_opt_post(char def_value);

// read an optional char from in front of a numeric input;
// default is returned if no value is entered by user.
char get_opt_pre(char def_value);

// Function to peek ahead on the keyboard stream until a
// non-space or a newline is encountered.  All leading
// spacing (except newlines) will be ignored.  The next
// character in the keyboard stream at the end of this
// process is returned -- but NOT extracted!!!
char peek_ahead(void);

// retrieve a numeric value from keyboard stream with an
// optional preceding character -- the leader/header;
// perhaps a unit or notational symbol?  the leader is
// extracted and checked against a desired symbol.
// if the leader is not the desired symbol, an invalidity
// message is printed.
// if the leader is not present at all, a message is printed
// indicating that this symbol was missing altogether.  the
// caller can customize these messages with the last two
// arguments.
double get_numb_with_lead(char desired_leader, const std::string & missing_msg,
                          const std::string & invalid_msg);

#endif
```

Notice that the `get_numb_with_lead` function takes `string` arguments and so needed the `string` library `#include`d. Also, we didn't do a `using` directive but rather placed the `std::` syntax on each occurrence of `string`. This is important because it keeps the lookup of names for the programmer `#include`'ing our library clean. That is, they don't have to use the standard `namespace` unless they want to. If we had the `using` directive in the `.h` file and they `#include`d it, they would be forced to keep using that `namespace`, too![13]

[13] We might see a couple of tricks later as to how to avoid this issue, but for arguments and `return` types, this rule will always be in effect.

The implementation for this library is pretty straight-forward:

input.cpp

```cpp
#include "input.h"

#include <iostream>
#include <cctype>
#include <string>
#include "classify.h"

using namespace std;

// read an optional char from after a [numeric] input;
// default is returned if no value is entered by user.
char get_opt_post(char def_value)
{
    if ( peek_ahead() != '\n' )
    {
        cin >> def_value;
    }
    return def_value;
}

// read an optional char from in front of a numeric input;
// default is returned if no value is entered by user.
char get_opt_pre(char def_value)
{
    cin >> ws;
    if ( ! is_numeric( static_cast<char>(cin.peek()) ) )
    {
        cin >> def_value;
    }
    return def_value;
}

// retrieve a numeric value from keyboard stream with an
// optional preceding character -- the leader/header;
// perhaps a unit or notational symbol?  the leader is
// extracted and checked against a desired symbol.
// if the leader is not the desired symbol, an invalidity
// message is printed.
// if the leader is not present at all, a message is printed
// indicating that this symbol was missing altogether.  the
// caller can customize these messages with the last two
// arguments.
double get_numb_with_lead(char desired_leader, const string & missing_msg,
                          const string & invalid_msg)
{
    double number;
    char t = get_opt_pre('\0');
    if ( t == '\0' )
    {
        cout << missing_msg;
    }
    else if ( t != desired_leader )
    {
        cout << invalid_msg;
    }
    cin >> number;
    return number;
}

// Function to peek ahead on the keyboard stream until a
// non-space or a newline is encountered.  All leading
// spacing (except newlines) will be ignored.  The next
// character in the keyboard stream at the end of this
// process is returned -- but NOT extracted!!!
char peek_ahead(void)
{
    while ( cin.peek() != '\n' &&      // and yet   (but)
            isspace( cin.peek() ) )
    {
        cin.ignore();
    }
    return static_cast<char>(cin.peek());
}
```

Note that this file does use the standard namespace. This is safe because it is never #included but compiled separately instead.

Also note that we #include our own header first before all other libraries. This is traditional style and helps everyone know what library we are working on.

And we also brought in more libraries than just string to help out. These are listed here and not in the .h because they are only used here! Why #include string again? It is good style (and a hard-to-break habit) to always bring in the libraries you are using in a particular file.

But notice also that is_numeric isn't there but there is another library of our design brought in called classify. Let's look at its interface:

classify.h

```
#ifndef CLASSIFY_H_INC
#define CLASSIFY_H_INC

// true if ch is a digit, ., -, or +.  false otherwise.
bool is_numeric(char ch);

#endif
```

And its implementation:

classify.cpp

```
#include "classify.h"

#include <cctype>
```

```
using namespace std;

// true if ch is a digit, ., -, or +.  false otherwise.
//
// basically anything that can ~start~ a numeric value.
bool is_numeric(char ch)
{
    return isdigit( ch ) // IS a digit
           || ch == '.'  // OR a decimal point
           || ch == '-'  // OR a 'negative' sign
           || ch == '+'; // OR a 'positive' sign
}
```

Here the interface needed no libraries to help but the implementation did need one.

Note how one library depended on another. If this had happened at the interface file level, we could have had a circular inclusion without those inclusion guard directives!

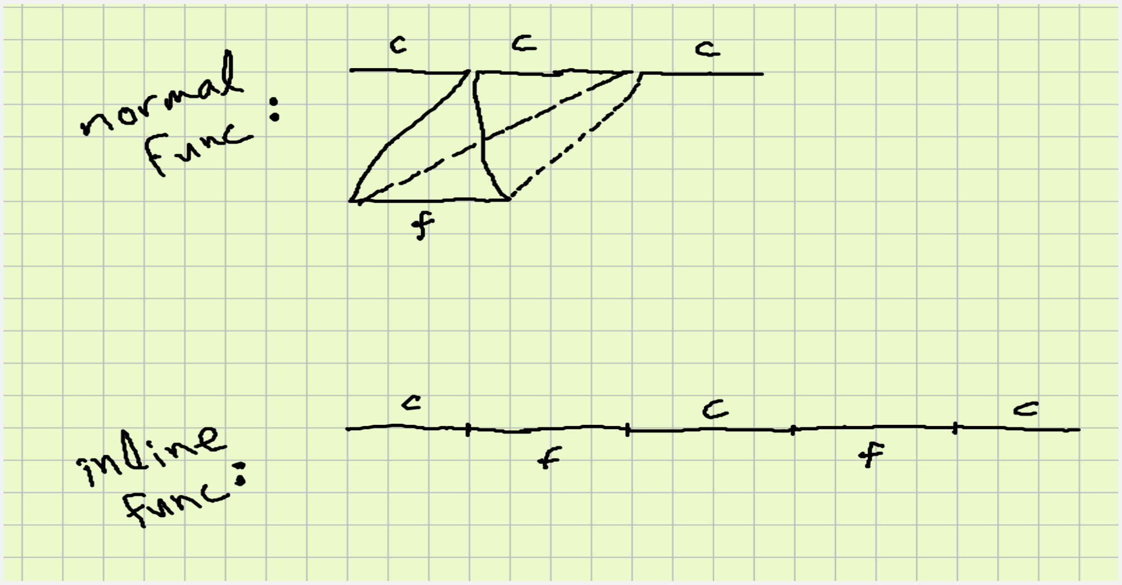
Finally, be aware that the application that wants to `peek_ahead` can just `#include` the `input` library. They need the `classify` library handy, but need not `#include` it. Both library's implementations must be compiled together with the application's `.cpp` file, of course.

4.5.3 inline Functions

When calling a function, it takes time to set up the function's activation record on the function call stack, link references, copy values, and generally get things going. Then, when the function returns, the value is copied from the `return` line to the return area of memory, the activation record is wiped, the `return` value is given to the caller, and finally that is wiped as well. If only the computer didn't have to do that work every time we called a function. If only we could make it more efficient somehow. . .

Here comes the `inline` keyword to the rescue!

The whole idea of making a function `inline` is to increase its run speed. The actual binary code for the function is spliced into the calling function's code at the site where a call would normally be generated. Effectively, it makes the first diagram look like the second:



Here 'c' stands for a calling function and 'f' the called function. Also here, the length of horizontal lines are proportional to the number of bytes in the binary representation of the code and the lengths of all lines — horizontal, arced, and dashed — are proportional to the running times of the code. This kind of diagram is truly creatable even if this is just a fictional one.

Note how the regular function transfers control to a separate area of memory where that function's instructions in binary form are kept. Then, after executing there for a bit, the function's `return` sends things back to the caller just after it had left.

But in the `inline` function call, there is no separate area for the function's code. It's code is spliced in line with the calling function's code — hence the name. This speeds execution immensely as we don't have to do all that setup and tear down for the function call itself — just run the function's code!

This is a great deal of work for the compiler, of course. And it can slow down compilation for the programmers. But any gains we make in run time for the prospective user lead to reputation points for the company/project and increased sales of same.

Also note that to `inline` a function is merely a suggestion to the compiler which makes the final decision of whether to make the function `inline` or not. The compiler knows, after all, more about the hardware and software situation on the target platform than we do and can make a more informed decision as to whether the `inline` is a good idea or not.

Why might it not be a good idea? Is it the slowdown in compilation time? No! It's the increase in binary size. Note in the diagram that not only did the `inline` function not have a separate memory area, but its code also repeated each time it was called. This extra code takes up more space than a traditional function which just sits in one place in memory and is referred to over and over. But, still, the time gains for the user are paramount and so we try!

4.5.3.1 Examples

Let's see how to `inline` functions with a final visit to the rounding suite:

```
/*
 * INLINE overloaded helper for 'efficiency':
 */
// As below, but assume one's place...
inline double round_nearest(double value)
{
    return round(value);
}

// Function to round caller's value to the nearest integer
// (or whatever multiple they desire -- see multiple
// argument).
//
// NOTE:  The place doesn't have to be a multiple of 10, it
//        can be a multiple of 5 or 2.5 or 0.43 or
//        ~anything~!
inline double round_nearest(double value, double multiple)
{
    return round_nearest(value / multiple) * multiple;
}
```

Note that we've put the `inline` keyword in front of the `return` type on the definition. Further, we are **just** defining these functions — not prototyping them. This means that these definitions must be before the calls so that the compiler has their heads for call-site verification as usual. But, even more, the compiler needs the definitions so that it can check the binary size of the code against its heuristic for deciding whether to actually `inline` the functions or not.

So we cannot prototype inline functions. Instead, we place their definitions with any other functions' prototypes.

4.5.3.2 inlining Library Functions

This rule applies also in a library interface file where normally real code is forbidden. That begs the question: Do we have to write all these inline functions with heavy use of std:: syntax, then? After all, we can't do a using directive in a .h file, remember?

The answer is no. We can place a using directive inside a function definition instead of at global scope in the header file. This only affects the lookup of names inside that single function, then. Does this save us on things like string parameters or returns, too? Again, the answer is no. We still have to use the std:: syntax in the function's head because the using directive only affects name lookup **inside** the function.

Finally, there is an extreme situation where all of a library's functions end up inline. In this situation we would have an implementation file that just had a single *#include* in it and nothing else! Is this necessary? No. We can eschew the implementation file in such cases and have a library of just a single .h file. (Keep in mind that such circumstances are rare!)

4.5.3.3 inline and Default Arguments

And how does this affect defaulted arguments? Well, those have to be on the first head of the function the compiler sees and an inline function has only one head. So this time we must put the defaulted values on the function definition instead of the prototype because we don't have one!

4.5.3.4 inlining With Style

There was actually mass confusion at the onset of the inline function concept. The code figures in the C++ standard wrote the functions across instead of up-and-down as normal. What? Yep. Instead of a nice top-to-bottom layout, they typeset them left-to-right:

```
inline double round_nearest(double value) { return round(value); }
```

This was done to save space in the printed document. Print? They printed documents back then? Yes. Not everything was a PDF on a tablet or phone. In 1998 when the first standard hit, things were still being printed regularly. It was estimated that putting the code samples mostly sideways saved a couple hundred pages off an already enormous document.

But the sideways examples led many to feel that inline'ing was all about doing code in one line. Simply not true as our diagram above proves! But the myth persisted and can still be found on websites today!

The only time to use such bad style is to save space for a presentation or publication. If you need more code on the slide or page, then you can do the sideways style to get it there and then explain it in the text or your discussion. Otherwise, please use top-to-bottom style as always!

4.5.3.5 Rules and Suggestions

So, when should we suggest to the compiler to inline a function? There are no hard-and-fast rules for this, sadly. But some general guidelines follow:

- The code is generally simple and short.
- The function is 10 or fewer statements. (Not lines — statements. Also, a for loop head counts as three statements since it collects three statements worth of work together like that. Also count statements inside decision structures like branches and loops.)

- There are five or fewer branches. (Not branching structures, simple branches like an if or an else or a case.)

As to counting statements, some things don't count as statements. These include not only curly braces which take up a line and comments, but things like the break in a switch's case or a return with no actual calculation or a variable declaration without initialization.

This might sound difficult to come by, but take a look at some of our functions. Many of them fit this bill! (Also note that the 10 statements is just a guideline — not a rule. You might make an exception for a function that has 11 or even 12 statements, but 13 should be right out!)

4.5.4 Help Debugging

When debugging, it is convenient to check for common idiotic argument values via a simple mechanism rather than a large if-else structure. Such a mechanism is given in the assert/NDEBUG macros. (Macros are like either a function or a constant depending on how they are used. assert here is a function-style macro and NDEBUG is a constant-style one.) The assert macro can be found in the old C library now known as cassert.

For instance, if your function needed two of its parameters to both be positive and their sum to be at least 5, you could code up the following asserts:

```
assert(x > 0);
assert(y > 0);
assert(x + y >= 5);
```

If any of these fail to be true, the text of the test is printed along with the line number in the code and source file name in a message indicating its failure — and the program is halted at this point.

When you are done debugging and ready to ship the product to users, just make sure to do:

```
#define NDEBUG
```

or use a compilation-wide definition[14] of NDEBUG to shut off all of your asserts at once. You never want them to fire off during a regular run by a user — quite embarrassing!

The main problem with assert is that you have no idea what call to the function caused the problem. You know which function died from the actual text printed, but which of so many calls to that function caused the arguments to be so far off?!

That's when you begin debugging with cerr. With judiciously placed cerr outputs, you can decide exactly how far your program got before the crash. Depending on your circumstances, for instance, here you might put a call to cerr before each call to the suspect function. Label which call is which in a string literal, of course. (Don't forget that the use of cerr provides extra utility in that it can print the values of variables, constants, and expressions along with the text.)

4.6 Advanced Techniques

In this section we'll see two more advanced techniques to make code reuse easier and make having multiple results from a function a little more natural.

4.6.1 template Function Basics

Although we've only had one or two examples that merit this tool so far, when we get to containers later (chapter 6) we'll be swimming in chances to use it. The first example was our swap function (section

[14] How to do this varies from system to system. Please ask your teacher how to do it at your school or in your environment.

4.4.1.2). We only used it once, but imagine if we had to swap more than just the one type of data in a program. We would end up with the same code over and over and just the types of the data would change:

```
void swap(double & a, double & b)
{
    double c{a};
    a = b;
    b = c;
    return;
}
```

```
void swap(char & a, char & b)
{
    char c{a};
    a = b;
    b = c;
    return;
}
```

```
void swap(long & a, long & b)
{
    long c{a};
    a = b;
    b = c;
    return;
}
```

```
void swap(short & a, short & b)
{
    short c{a};
    a = b;
    b = c;
    return;
}
```

```
void swap(bool & a, bool & b)
{
    bool c{a};
    a = b;
    b = c;
    return;
}
```

```
void swap(string & a, string & b)
{
    string c{a};
    a = b;
    b = c;
    return;
}
```

Here we see but a few examples of this notion. All of the code remains the same — only the types of data being acted on change.

Our other example was the `read_numeric` overload from section 4.4.2.3. We saw definitively that the code was identical and just the types were changing.

In fact, it is this kind of overloading that will give us the main chance to use this tool.

Okay, okay! So what is this tool? Well, it is the `template` mechanism. As the name implies, we will define a function as a template or pattern for the compiler to follow in creating binary functions during the compilation process. It won't be exactly code that can compile directly, but just a guide for the compiler to follow in creating such code to then translate to binary.

Let's take a look at it with the `swap` function from above:

```
template < typename SwapT >
    inline
void swap( SwapT & a, SwapT & b )   // both arguments exactly same type
{
    SwapT c;                        // default initialization
    c = a;                          // \
    a = b;                          //  |-- assignment with self
    b = c;                          // /
    return;
}
```

(Note, I made a slight change in how the local variable c gets its value to make a point. Just bear with me. . .)

We've used two new keywords: `template` and `typename`. The first says to the compiler that the following 'item' in the code is actually a pattern to follow rather than normal code. In our case, the next item is a whole function definition.[15]

[15] In the next volume we will see how to make `template`s out of other things as well.

Then the typename keyword is used inside angle brackets to introduce a placeholder for the name of a type in the pattern. Since this type is being swapped around by our algorithm,[16] I've called it SwapT. (The full name SwapType would have also been acceptable, but I felt it was a bit long so I shortened it in the spirit of time_t and wchar_t.)

Alternatives

Another keyword that is sometimes used to introduce the pattern's type name is class, but this has an alternative use to introduce a real data type as we'll find more about in chapter 5 and so I avoid it here. Likewise, some people are ultra-lazy and use single letters to name their pattern types like T or U. This is a bad policy as it is an identifier and should have meaning just like the names of variables or functions or type aliases.

I've also made some notes off to the side about what each line of code represents in terms of the pattern type and its usage. The head of the template function demands that both parameters be of the exact same type because they are both pure references.

The declaration of the helper variable uses a default construction when SwapT is string. It won't technically do anything when SwapT is a built-in type, but we are planning for the general case here.

Finally, the compiler notices that the type must be assignable with itself — a short to a short, for instance. While this seems a non-requirement, it is listed and can oddly be thwarted by advanced code.

After the compiler has read through this template function's code, it has a new listing for a potential function named swap. But that isn't its official name. Internally, the compiler sees it as swap<SwapT>. This keeps us in mind that it isn't a regular function but a plan for a function.

Alright, then, how do we call it? There are two ways. We can tell the compiler explicitly what the SwapT should be or we can let it figure it out for itself — via type deduction. An explicit call uses angle brackets after the function name filled in with the desired type to use in the algorithm:

```
short x, y;
swap<short>(x, y);
```

We just list the explicit type once because the template only called for one typename. This confuses some students who think they need to list it twice — once for each parameter.

This way is okay, but not needed since the template type is used in our function's argument list. When that happens, we can allow the compiler to deduce the necessary type from context:

```
short x, y;
swap(x, y);
```

Here the compiler uses its knowledge that x and y are short integers to fill out the template's pattern.

However it gets a prospective type for SwapT, the compiler next checks its requirements list:

- Are the two argument's types the same?
- Is there a default constructor for this type?
- Is the type self-assignable?

It might seem that the middle one will thwart our efforts with the short example we've used here, but it turns out that the built-in types have default constructors that they just hardly ever use! So it works in our favor here.

[16] Recall that an algorithm is a plan to solve a problem. So it is basically a generic way to talk about a C++ function.

Since all of these things are true for both attempts to use the swap template, both would instantiate[17] a new binary function named swap<short>. In future attempts to call swap with short arguments, the compiler will use this instantiation directly instead of going through the requirements checklist all over again.

Now that we've covered the basics — yes, that's all there is to them, we can look at the read_numeric example, too:

```
template < typename ReadT >
   inline
ReadT read_numeric(ReadT & num, const string & prompt, const string & errmsg)
{
    cout << prompt;
    cin >> num;
    while ( cin.fail() )
    {
        cin.clear();
        cin.ignore(numeric_limits<streamsize>::max(), '\n');
        cerr << errmsg;
        cout << prompt;
        cin >> num;
    }
    return num;
}
```

Here we see another implication of our newfound template knowledge: numeric_limits is a template of some sort, too! In fact, it is a templated class with instantiations holding appropriate constants for all the built-in data types' properties.

4.6.2 tuple returns

When we want a function to have multiple results in a single call, we've learned to use the reference mechanism like so:

```
inline void read_point(double & x, double & y)
{
    x = get_numb_with_lead('(',
                           "Open parenthesis missing!\n",
                           "Point must have a parenthesis before "
                               "x-coordinate.\n");
    y = get_numb_with_lead(',',
                           "Comma missing!\n",
                           "Point must have a comma between "
                               "coordinates.\n");
    get_trailing_opt(')',
                     "Close parenthesis missing!\n",
                     "Point must have a parenthesis after "
                         "y-coordinate.\n");
    return;
}
```

Here we've used the optional notation functions defined in section 4.5.1.2. That last line is my call to your exploration of placing that post-peeking code from the relevant example.

[17] This is our actual verb for making a binary output of a template. Maybe we got tired of 'compile'? *shrug*

But there is a newer way to handle such results. We can use a special packaging tool to return multiple values at once! This tool comes in two flavors. One is specialized for two returns like our example here and the other is generalized for two or more at a time.

4.6.2.1 A pair of Results

The first is the pair and comes from the library utility. It can be used like this:

```
inline pair<double,double> read_point(void)
{
    double x, y;
    x = get_numb_with_lead('(',
                           "Open parenthesis missing!\n",
                           "Point must have a parenthesis before "
                               "x-coordinate.\n");
    y = get_numb_with_lead(',',
                           "Comma missing!\n",
                           "Point must have a comma between "
                               "coordinates.\n");
    get_trailing_opt(')',
                     "Close parenthesis missing!\n",
                     "Point must have a parenthesis after "
                         "y-coordinate.\n");
    return {x, y};
}
```

This should work on any C++17 compliant compiler (or newer). If your compiler is a little older — C++11-ish, you might need to use this as your return line:

```
return make_pair(x, y);
```

This more explicitly creates the pair being expected.

How does the caller get those results? There are many ways:

```
pair<double,double> p1;
cout << "\nPlease enter your first point:  ";
p1 = read_point();
cout << "\nI read (" << p1.first << ", " << p1.second << ").\n";
```

This has the unfortunate side-effect of using the names first and second for the x and y parts respectively. Also, we have to use the dot syntax to get them from inside the pair variable.

To avoid both of these things, we can use a structured binding in C++17 and above:

```
cout << "\nPlease enter your first point:  ";
auto [x1, y1] = read_point();
cout << "\nI read (" << x1 << ", " << y1 << ").\n";
```

This gives nice names to the contents of the returned pair and removes the need for those annoying dots. This idea is called a structured binding because it binds parts of the structure (or group of values) returned to the variables we want. (See section 5.4.3 for more on structures. Its discussion requires reading much of the earlier part of that chapter!)

But what's that auto doing there? That is deducing the type of the result from read_point for us. Otherwise, we'd have to retype pair<double,double> all over again. Can we use this with other functions? Certainly, but we've had fairly simple situations so far and haven't needed it. It makes sense here and might make sense on future situations, too. I'll mention them when we get there.

There is also a third way, but it is almost worse than the first:

```
pair<double,double> p1;
cout << "\nPlease enter your first point:  ";
p1 = read_point();
cout << "\nI read (" << get<0>(p1) << ", " << get<1>(p1) << ").\n";
```

This uses the get template to pick out the components of the pair numerically. They are numbered from 0 just as are positions in a string.

4.6.2.2 More Than Two Results

What's the other way to return multiple answers, you say? It is called a tuple and is found in the tuple library. It gets its name from the names we give groups of things: couple, triple, quadruple, etc. Most end in 'uple' and we just put a 't' on the front because 'uple' sounded weird alone. *chuckle*

Here is the code for that and a 3D point reader:

```
inline tuple<double,double,double> read_3D_point(void)
{
    double x, y, z;
    x = get_numb_with_lead('(',
                           "Open parenthesis missing!\n",
                           "Point must have a parenthesis before "
                               "x-coordinate.\n");
    y = get_numb_with_lead(',',
                           "Comma missing!\n",
                           "Point must have a comma between "
                               "coordinates.\n");
    z = get_numb_with_lead(',',
                           "Comma missing!\n",
                           "Point must have a comma between "
                               "coordinates.\n");
    get_trailing_opt(')',
                     "Close parenthesis missing!\n",
                     "Point must have a parenthesis after "
                         "z-coordinate.\n");
    return {x, y, z};
}
```

As before, the return statement might need to be modified on older compilers (C++11/14):

```
return make_tuple(x, y, z);
```

This is a more explicit way to generate a tuple, but also more bulky.

Again, calls can be made with a tuple holder variable and the get mechanism:

```
tuple<double,double,double> p1;
cout << "\nPlease enter your first point:  ";
p1 = read_3D_point();
cout << "\nI read (" << get<0>(p1) << ", " << get<1>(p1)
     << ", " << get<2>(p1) << ").\n";
```

There isn't a way to use the dot to access the elements because the tuple can have any number of elements and they couldn't name them all, now could they?

But you can also use structured binding with tuples:

```
cout << "\nPlease enter your first point:  ";
auto [x1, y1, z1] = read_3D_point();
cout << "\nI read (" << x1 << ", " << y1 << ", " << z1 << ").\n";
```

This is very convenient, of course.

4.6.2.3 Anti-Examples

Of course, not all multiple result functions should use this mechanism. For instance, our swap should stay using references. If we didn't, we'd make copies of the incoming results and then copy the swapped values back to the caller in a pair and all this copying costs time and space! Using the reference makes more sense as it saves all that time and space for other pursuits.

Some would even prefer not to use it on the read_point scenario as it precludes us from being able to overload the 3D version. Did you notice that? Since the return type isn't used in overload deduction — only the argument list — we had to have a separate name for that one.

So when should it be used? That's a good question, but it'll have to wait until quite a bit later for a really good answer. That's at least a course away, sadly.

4.7 Warnings: What Not to Do

There are two things you do **NOT** want to do with functions, at least for now:

- use more than a single return statement in a single function
- use recursion — even unintentionally

4.7.1 Multiple returns

To avoid the first one, we can use a helper variable and store the result for the function in it and later use a single return to send it back. So, instead of coding:

```
inline string suffix(long number)
{
    if ( number / 10 % 10 != 1 )
    {
        switch ( number % 10 )
        {
            case 1:
                return "st";
                break;
            case 2:
                return "nd";
                break;
            case 3:
                return "rd";
                break;
        }
    }
    return "th";
}
```

We would instead use this:

```
inline string suffix(long number)
{
    string suff{"th"};
    if ( number / 10 % 10 != 1 )
    {
        switch ( number % 10 )
        {
            case 1:
                suff = "st";
                break;
            case 2:
                suff = "nd";
                break;
            case 3:
                suff = "rd";
                break;
        }
    }
    return suff;
}
```

Having multiple `return`s isn't such a bad practice for this small function, but it can get hairy when having errors `return` early from a function. In these situations, you might even `return` from a branch that catches the error condition and then move on with the rest of the function. This leads to confusion for both the caller and the maintainer. Always use proper `else` structure to branch around code that is to not be executed. Always stop a loop gracefully with its condition instead of just `return`ing from the middle of it via some `if`.

4.7.2 Unintentional Recursion

We've already said that recursion is too hard to do without much more practice with functions. But it is possible to form a recursion by accident! These situations often can be turned into a proper loop.

This person, for instance, has a simple menu defined by separate functions like so:

```
char get_choice(void)
{
    char choice;
    cout << "\t\tMain Menu\n\n"
            "1) do Junk\n"
            "2) do Stuff\n"
            "3) Quit\n\n"
            "Choice:  ";
    cin >> choice;
    cin.ignore(numeric_limits<streamsize>::max(), '\n');
    process_choice(choice);
    return choice;
}

bool process_choice(char choice)
{
    bool quitting{false};
    switch ( toupper(choice) )
    {
        case '1': case 'J':
        {
            cout << "\n\tChoice 1 -- JUNK -- chosen!\n\n";
        } break;
        case '2': case 'S':
        {
            cout << "\n\tChoice 2 -- STUFF -- chosen!\n\n";
        } break;
        case '3': case 'Q': case 'X':
        {
            quitting = true;
        } break;
        default:
        {
            cout << "\n\aInvalid choice '" << choice << "'!!!\n\n"
                    "Please try to read more carefully next time...\n\n";
        } break;
    }
    if ( ! quitting )
    {
        choice = get_choice();
    }
    return quitting;
}
```

Note how each function calls the other to form a loop-like effect. This is horrible coding and should be avoided at all costs! (Well, just never do it, okay?)

This could be redone much more simply and avoid potential stack overflow[18], we could by using a nice do loop:

```
inline char get_choice(void)
{
    char choice;
    cout << "\t\tMain Menu\n\n"
            "1) do Junk\n"
            "2) do Stuff\n"
            "3) Quit\n\n"
```

[18] Stack overflow is when too many functions are called and the finite sized function call stack becomes full and calls one more time.

```
            "Choice:  ";
    cin >> choice;
    cin.ignore(numeric_limits<streamsize>::max(), '\n');
    return choice;
}

inline bool process_choice(char choice)
{
    bool quitting{false};
    switch ( toupper(choice) )
    {
        case '1': case 'J':
        {
            cout << "\n\tChoice 1 -- JUNK -- chosen!\n\n";
        } break;
        case '2': case 'S':
        {
            cout << "\n\tChoice 2 -- STUFF -- chosen!\n\n";
        } break;
        case '3': case 'Q': case 'X':
        {
            quitting = true;
        } break;
        default:
        {
            cout << "\n\aInvalid choice '" << choice << "'!!!\n\n"
                    "Please try to read more carefully next time...\n\n";
        } break;
    }
    return quitting;
}

inline void menu(void)
{
    char choice;
    bool done;
    do
    {
        choice = get_choice();
        done = process_choice(choice);
    } while (!done);
    return;
}
```

We even get to `inline` the functions now to speed up the processing a bit.

4.8 Wrap Up

In this chapter, we've studied the mechanism of code reuse called functions. We've gone from simple functions that printed messages and did simple calculations for us to large functions that made decisions and looped. We even ended up with functions that sent back multiple answers in a few different ways!

Now go out there and practice your function writing skills. I'll see you back here soon for the next part on aggregating data.

Part III
Data Aggregation

Chapter 5

Classes

C++ uses the keyword `class` to implement the concept of object-oriented programming (OOP). It is also the way we introduce a new data type into a C++ program. Both of these relate to the general idea of an abstract data type (ADT).

5.1 Basics

An abstract data type combines descriptions of both the data that make up a type of thing and the actions/behaviors that the data can be involved with. An ADT is the generic way to do OOP, in fact. And, when done by an OOP practitioner, it will focus heavily and firstly on the data itself and only flesh out the actions part as an afterthought. An OOP designer will often give their ADT a flavor of anthropomorphism or personification. They like to give the data a personal feel by making it more life-like and animated.

5.1.1 OOP in C++

Once an ADT is translated into a specific language (like via a C++ `class`), it has been not just designed but implemented. Such an implementation can be used in a program to make the realization of a solution more clear or more natural.

5.1.1.1 Examples

Look at the stream concept that Bjarne used to design the input/output system for C++. Certainly `cin` and `cout` are objects and we've focused heavily on them and some on the actions that they are involved with like extraction (>>) and insertion (<<) and the like.

I've also heard tales of these objects as boat captains on electric streams that run between the CPU and the keyboard and screen. Extraction is actually, then, a group of sailors working in the cargo bay — each specialized in a different type of data — translating sequences of keystrokes into usable data. Likewise, insertion is played by sailors specialized in translating data into sequences of characters suitable for display.

Although the `string` `class` could also use a bit of personification, it generally involves beads dangling from an actual string. This leaves the operations we perform a bit difficult to describe in this metaphor.

5.1.1.2 The class Concept

As mentioned, C++ realizes the OOP paradigm via its `class` mechanism. A `class` describes the members of a group (a class of things; like in classification) in as general of terms as possible/desirable.

It will describe to the compiler the physical attributes which make this `class`' members distinct from other objects in the world as well as the behaviors in which this `class`' members can participate. The physical attributes are represented by data — member variables. And the behaviors are represented by member functions — also known as methods.

C++ then treats this description as a data type — just like its built-in types. (Recall the `string` `class` from the standard libraries and that `cin`/`cout`/et al are objects of `class`es defined in the standard libraries.) A `class` you create will be treated no differently than those from the standard libraries — it will be a data type of which you can declare variables (aka objects or instances of the `class` type). You'll also be able to assign those variables to one another — changing their values to be alike. And, of course, you'll be able to both pass and `return` information to/from functions.

When you define `class` functions (aka methods — functions specifically for the `class`), they can be called with respect to a particular object using the dot (`.`) operator just as you've done with `cin`/`cout`/`string` objects you've declared.

You can also refer to parts of the `class` generally rather than with respect to a particular object using the scope resolution (`::`) operator. We've seen this several times with `string` member types and constants and with constants from the `class` `ios_base` used in output formatting.

5.1.2 Learning by Doing

A *fairly* simple example we should all be familiar with is the concept of a die. You know, like you roll for a turn of a game. Mostly you are familiar with 6-sided dice, but there are other sizes and shapes — both smaller and larger. There are even ones whose pip-counts[1] aren't contiguous but jump by 2s or 3s. Sometimes we even want to adjust the values by a consistent amount up or down. Also having the die be virtual like this makes it easier to make weird shapes like percentile or 3-sided that aren't easily done with a physical die. For more on dice, please see this immensely helpful article[2] at Wikipedia.

For instance, if you were interested in a die that only ever came up 2, 4, or 6, we could start with a 3-sided die and multiply every value it came up with by 2. If, on the other hand, we were interested in a die that only ever came up with 1, 3, or 5, we could start out the same way as before and then subtract 1 from every result.

Perhaps my favorite (so far) is a die that comes up -1/3, 0, or 1/3. This can be created with the 3-sided die multiplied by 1/3 and then having 2/3 subtracted from each value.

In general, this will be what mathematicians call an arithmetic sequence or arithmetic progression. If you haven't covered such before in a prior math course, you might wish to review the idea at PurpleMath before going on.

[1] A pip is a divot on the surface of a die — singular form of dice.
[2] Be sure to look at the side article on dice notation! Really fun reading!

In the terms of arithmetic sequences, the adjustment added to every result is the initial value of the sequence and the scaling factor is the common difference between elements of the sequence. The number of sides is just the length of a finite sequence.

As a C++ class, this might look something like the following:

```
class Die
{
    long sides;         // number of sides on the die
    double scale,       // multiplier for pips
           adjustment;  // added to pips after scaling
public:
    void print(void);   // display on screen (aka print, output, write, etc.)
    bool read(void);    // input from keyboard; true returned upon successful
                        //     input (aka input, entry, etc.)

    double min(void);   // provide statistics about this die's values
    double avg(void);   // upon being rolled; values may change due to
    double max(void);   // alteration of the member variables (say by
                        // the object being re-read)

    bool can_be_smaller(Die d);       // compare two Die objects
    bool can_be_larger(Die d);        // to see their potential
    bool is_typically_smaller(Die d); // and typical relationship
    bool is_typically_larger(Die d);  // to one another on the
    bool is_same(const Die & d);       // number line

    double roll(void);   // provide a typical roll of this Die

    // make altered versions of the original Die object
    Die scale_by(double applied_scale);
    Die translate(double applied_adjustment);
    Die change_shape(long new_sides);
};
```

This definition makes the Die type known to the compiler in all its details of member data description and behavioral function declaration. Once known, the compiler lets us use it just like a built-in type as mentioned above.

Let's explore this particular class bit-by-bit.

5.1.2.1 The Data Inside

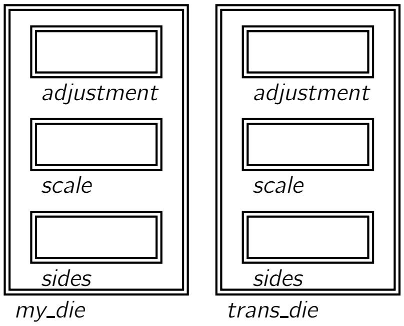
Because the Die class began with:

```
class Die
{
    long sides;
    double scale, adjustment;
    // ...
```

we know that all Die-type objects will be composed of a long integer and two doubles. These members can be discussed generally as Die::sides, Die::scale, etc., but will more usefully be discussed as object.sides, object.scale, etc. — with respect to a particular object of type Die.

We can declare such objects like so:

```
Die my_die, trans_die;
```



These objects might look in memory like the diagram at right. Here we see that each object gets its own set of the member variables — separate from those of any other object. Also note that the individual boxes are *not* to scale — they are just representing that each member variable has a separate memory location from the other two.

(In future, we might label such a diagram with member variable types as well as names to help in a discussion.)

5.1.2.2 private vs public

Now let's look at the next few lines in context with the first few we already studied:

```
class Die
{
    long sides;
    double scale, adjustment;
public:
    void print(void);   // display on screen (aka print, output, write, etc.)
    bool read(void);    // input from keyboard; true returned upon successful
                        //     input (aka input, entry, etc.)
    // ...
```

Here we have our first two behaviors: input and output. But, before we explore those explicitly, let's look more closely at that new keyword before them: `public`. What's that all about?

It says to us two things:

- the following members of the `class` are available to all the program to use
- those before this were different somehow

So, if the `print` and `read` functions of `Die` are accessible from anywhere in the program — both inside and outside the `class`, then what about `Die::scale` and such? They must be more restricted in access allowance, right? How restricted? They can only be accessed by the `class` members themselves. In this case, the member functions: `Die::print`, `Die::read`, etc.

This other access mode[3] is known as `private` — but we didn't use that keyword to introduce those members. Why? Well, I'm being economical with my keystrokes again. In C++ all `class` members are `private` by default. Had I wanted to, I could have coded it like this:

```
class Die
{
private:
    long sides;
    double scale, adjustment;
public:
    void print(void);   // display on screen (aka print, output, write, etc.)
    bool read(void);    // input from keyboard; true returned upon successful
                        //     input (aka input, entry, etc.)
```

[3] Fancy terminology, right?

```
    // ...
```

But this seemed wasteful to me. Other programmers would have gone to the other extreme and placed the private parts — the member variables here — at the bottom of the class definition like so:

```
class Die
{
public:
    void print(void);   // display on screen (aka print, output, write, etc.)
    bool read(void);    // input from keyboard; true returned upon successful
                        //     input (aka input, entry, etc.)
    // ...
    // make altered versions of the original Die object
    Die scale_by(double applied_scale);
    Die translate(double applied_adjustment);
    Die change_shape(long new_sides);
private:
    long sides;
    double scale, adjustment;
};
```

This is helpful in the process of data hiding, but just makes us type the keyword private and its colon. Is it worth it? The curious programmer can still see them by scrolling down a bit further. I don't think it is worth it in general. But I leave it to you and your teacher to decide on your style. I'm going to keep doing it my way, however — consider yourself warned.

These two access modes, though, are enforced by the compiler during the compilation process. Whenever it sees a use of a public class member, all is fine. But when it sees a use of a private member, it had better be inside the class! If it isn't, the compilation produces a lovely error message about that member being private in this context — or something like that. As usual, the exact message will depend on your compiler and the proclivities of those who programmed it.

Why do we make the member variables private and the member methods public? Well, it makes sense that the functions are public so that the class' behaviors can be used in the program at large. Making the variables private makes sure they don't get changed poorly by some programmer not involved with the class design team. Only programmers working on the code of the Die class should be messing with its data members! Anyone else might change the sides of some poor Die object to 0 or -3 or the like! We want to make sure this can't happen and so we make these variables private so the compiler enforces this decision.

Is it always this way? In designing a class, all variables should be private always. This protects them from accidental or even malicious change by other programmers on the project. We'll talk more about this issue when we get to mutator functions (section 5.2.2).

However, there may be occasions where a method or two might be private, too. It depends on if their functionality should be automatic or at will. If those functions should always fire off under the right conditions, then we make them private and make sure the proper other functions call them under those conditions. If they should be usable at will, we make them public and so any other part of the program can use them whenever they want.

5.1.2.3 Input and Output

Now back to the Die::print and Die::read functions themselves.

First, let's see how they can be used. It is just like a method call against a string object or a console stream. We need an object and a dot:

```
Die my_die;

my_die.print();
```

Here we declare the object my_die and call the Die::print function with respect to it. But if we call to print the object right after declaring it, what will be displayed? Probably some kind of garbage! Let's read the value in first:

```
cout << "Please enter your die specifics:  ";
my_die.read();
```

Note that the read function of the Die class doesn't itself prompt the user but depends on its caller to do so ahead of time. This keeps the function generic and reusable across different situations within the same program or even across many programs.

But, since this function returns a bool telling of success or failure, we should probably use that in some way.[4] Perhaps in a while loop:

```
cout << "Please enter your die specifics:  ";
while ( ! my_die.read() )
{
    // do something about the problem...
    cout << "Please enter your die specifics correctly:  ";
}
```

Some programmers balk at this style — having the method call inside the while head like that. But it is considered very object-oriented as the object is focal in the loop condition and it is the object's own success or failure at reading that we are looping based because of.

If it really grates on your nerves, you can always add a helper variable:

```
cout << "Please enter your die specifics:  ";
success = my_die.read();
while ( ! success )
{
    // do something about the problem...
    cout << "Please enter your die specifics correctly:  ";
    success = my_die.read();
}
```

Here success is a bool variable to hold the truth value returned by the read function when called with respect to my_die. Is this extra variable worth all the trouble? Probably not...

But once we've read in the my_die object, we can report its value to the user properly:

```
cout << "\nI read that you want to roll ";
my_die.print();
cout << ", is that right?  ";
```

Note that the Die::print function will not print any labels or the like to keep its use general enough to be called from anywhere in any program.

[4] What could cause a Die object to fail to read? We'll discuss that in section 5.1.2.5 on method definitions.

5.1.2.3.1 Confusing Points

There are two main issues students of programming have with these functions:

- what exactly does this `read` function do?
- where are the objects they work with?

As to the first, the comments clearly tell us that the `read` function's purpose is to gather the data content of a `Die` object from the user at the keyboard. This means it will be using `cin` to extract that data somehow. And the moniker `read` kind of lends itself to this task/situation.

Many students, however, use a different name for this method like `input` and get confused. They think of the caller of the function sending input to the object and this leads them to put in parameters for those values and then they store those values into the member variables. But that breaks data hiding altogether as well as the privacy barrier we put up around our member variables!

5.1.2.3.2 Calling Objects Revisited

We've seen the concept of a calling object before with `cin`, `cout`, and `string` objects. But let's make it more explicit here. When we used `my_die` above to call the `Die::print` and `Die::read` functions, it was acting as the calling object for each. That means that it was the object with respect to which each was acting. The compiler ties that function and object together for the duration of this call. As soon as the function ends, the tie is severed and things go back to normal. The tie is a reference-like mechanism, too, so any changes done inside the function to member variables will stick to the calling object like glue!

This is why the `Die::print` and `Die::read` functions seem to not have objects to work with in their declarations: all `class` methods must have a calling object. This reduces their necessary object arguments by one. We might have expected `Die::print`, for instance, to take a `const Die &` or `Die::read` to take a `Die &`. But, since they were tied to calling objects, these arguments were unnecessary.

5.1.2.3.3 Always Present

Any `class`, btw, should have basic input and output functions like the `Die` `class`' `print` and `read` methods. They shouldn't prompt or label or even print excess whitespace like newlines. They are meant to be as reusable as possible, so we keep all of that stuff back in the application — not in the `class`.

Remember that the `class` defines a data type to be treated like any built-in type. What would happen if every `short` integer prompted for itself like: `Please enter value between -32768 and 32767:`? That would be lunacy! It would be impossible to use it in almost any reasonable interface.

5.1.2.4 Helper or Type-Specific Functions

On the other hand, there are functions in any `class` that are special to just that `class` because they wouldn't make sense on any other data type. In our case, we have three basic kinds of helper methods that are specific to the `Die` type: calculations, comparisons, and transformations.[5]

5.1.2.4.1 Calculation

There are four calculation functions. Three calculate statistics and one actually rolls the `Die` — a random calculation:

```
class Die
{
```

[5] Some programmers use the term helper only for functions that are `private` and used only by the `class` itself. We do not take that definition in this book.

```
public:
    // ...
    double min(void);   // provide statistics about this die's values
    double avg(void);   // upon being rolled; values may change due to
    double max(void);   // alteration of the member variables (say by
                        // the object being re-read)

    double roll(void);   // provide a typical roll of this Die
    // ...
};
```

As before, an object seems to be missing, but that information will be supplied by the calling object:

```
cout << "\nYou might roll " << my_die.roll() << " sometime.\n";
```

Since these calculation functions all return double values, they can be used in a chained cout like this.

5.1.2.4.2 Comparison

There are also a few comparison functions. These are meant to tell the caller which of two Die objects is larger, smaller, etc. In this vein, they return bool values just like >, <, etc. This allows them to be used in an if or while head.

```
class Die
{
public:
    // ...
    bool can_be_smaller(Die d);       // compare two Die objects
    bool can_be_larger(Die d);        // to see their potential
    bool is_typically_smaller(Die d); // and typical relationship
    bool is_typically_larger(Die d);  // to one another on the
    bool is_same(const Die & d);       // number line
    // ...
};
```

Due to the way the range of values for two Die objects can overlap on the number line, we mostly have them saying things of a qualified nature — can_be_* and is_typically_* — instead of the more definite names you might expect. The only definitive name is is_same. This is because you can tell pretty precisely when two Die objects have the same content in them.

But, as we said, here we have two Die objects involved: the calling object and an argument object. The argument object is either a copy of the caller's object given as the actual argument or a constant reference to it. The why of this will have to wait until we make our class methods more efficient and robust in section 5.3.

5.1.2.4.3 Transformation

Finally, there are three transformation methods that make new objects that are like the calling object except in a specific way and are thus transformations of it.

```
class Die
{
```

```
public:
    // ...
    // make altered versions of the original Die object
    Die scale_by(double applied_scale);
    Die translate(double applied_adjustment);
    Die change_shape(long new_sides);
};
```

These take built-in-type arguments to change the indicated member variable of the calling object during the creation of a new object for `return`. They are made that way to behave more like standard arithmetic operations like addition and multiplication. Some programmers would try to alter the calling object instead of making a new object to represent the transformation. Which is better? Let's see. . .

5.1.2.4.4 Arithmetic vs Normal

The way our `class` is set up now, we have arithmetic-style transformation methods. These can be used to get new versions of an original `Die` object that behave slightly differently. We could use them to do something like this, for instance:

```
Die my_die, trans_die;

// fill in my_die from a read()

trans_die = my_die.translate(4.2);  // scoot new die 4.2 right from my_die
```

This gives us a new transformation but lets us keep the original `Die` object as well.

What if we didn't need to keep the original? Then we could have coded this:

```
Die my_die;

// fill in my_die from a read()

my_die = my_die.translate(4.2);  // scoot die 4.2 right from original
```

No harm done. Don't even need the extra variable.

But what about a non-arithmetic implementation? Well, these would look like so:

```
class Die
{
public:
    // ...
    // alter the original Die object as indicated
    void scale_by(double applied_scale);
    void translate(double applied_adjustment);
    void change_shape(long new_sides);
};
```

To change the original `Die`, we'd just:

```
Die my_die;

```

```
// fill in my_die from a read()

my_die.translate(4.2);  // scoot die 4.2 right from original
```

To keep the original the same, however, we would need to do this:

```
Die my_die, trans_die;

// fill in my_die from a read()

trans_die = my_die;
trans_die.translate(4.2);  // scoot new die 4.2 right from my_die
```

That's a little extra work. Of course the non-arithmetic champions say that the arithmetic version with the `my_code=` was almost as long.

I suppose your mileage may vary. I'm fond of the arithmetic style and it works so nicely for so many situations that are too advanced to detail at this time.

5.1.2.5 Defining class Functions

So far we've defined the `Die` `class` and called many of its methods in sample code fragments. But how are these methods defined themselves? They were only prototyped in the `class` definition, after all. They are typically defined separately from the `class` definition and their look-and-feel depends mightily on how many objects are involved in their execution. Let's start out simply with one object — the calling object.

5.1.2.5.1 One Object

Well, they are defined typically outside the `class` in a fashion like this:

```
void Die::print(void)
{
    cout << 'd' << sides << '~' << scale << (adjustment >= 0.0 ? "+" : "")
         << adjustment;
    return;
}
```

Here we see three things. First we see that the `Die` `class` is used to scope the `print` function just as we do in general discussions of the method. This reminds the compiler that this definition is for the same `print` function that was prototyped as part of the `Die` `class` earlier. Without this scoping, the compiler would think this was a new `print` function that just looked oddly similar to that other function from the `Die` `class`. Also, it would cause this next thing to go awry.

The second thing is that we are printing more than just the member variables! I thought we were supposed to print no labeling?! Well, there are standards to uphold here. The information for a `Die` is always printed with a `'d'` before the number of sides.[6] Then, our `class` has two more parts. We need them to separate from one another or they'll become unintelligible. We chose some special notation instead of spacing them out. This keeps them all tied together in one space-separated block and yet the data items are separated for later reading. I used a tilde in front of the `scale` and either a plus sign or a minus sign in front of the `adjustment`. (The minus sign is implicit in a negative `adjustment` value.)

[6] See the above-linked Wikipedia article near the bottom for more on dice notation.

The third thing, then, is that the Die::sides, Die::scale, and Die::adjustment are used without their scope resolution attached as well as without an object dotted. This is because the compiler remembers it is in the Die scope from the function scoping and it will automatically dot the calling object for us without us needing to remember. This is in fact a good thing because the calling object — albeit tied to this function during a call — doesn't exist by name here in this function. So we'd have a hard time dotting it even if we tried!

Calling Objects Again

So remember, when using the member variables directly inside a member function, they are going to be those of the calling object — not anyone else's. This is because of the tie between a called method and its current calling object. That tie is severed when the method returns and a new one created when a new call is made.

So, when we call:

```
my_die.print();
```

the sides, scale, and adjustment will be those of the my_die object from the calling function. And when we do:

```
trans_die.print();
```

the sides, scale, and adjustment will be those of the trans_die object from the calling function.

Unfortunately, all of this happens invisibly and magically to the new programmer's perspective and is quite exasperating at times. Don't fret, you'll get the hang of it!

What about other functions? Well, the read function is similar to the print function:

```
bool Die::read(void)
{
    char t; //, t2, t3;
                      // t2            t3
    cin >> t >> sides >> t >> scale >> t >> adjustment;
    if ( t == '-' )   // t3
    {
        adjustment = -adjustment;
    }
    return !cin.fail(); // && t == 'd' && t2 == '~' && (t3 == '+' || t3 == '-');
}
```

We see here primary code and plans for a future version as well. The current code reads a char variable before each member variable to represent the needed/necessary notation the print function displayed just to separate the members in the output stream. After those are read, the last char value from before the adjustment is checked to see if it was a minus sign and, if so, the adjustment is negated from its current (necessarily positive) value. (I suppose the user could have entered a -0, but that would be rather strange, right?)

Finally, we return a bool indicating cin's lack of failure. This is all we've coded for now, but note there is much more attached in the future plan in the comment!

The future plans include two more char variables so we can remember all of the entered notation separately. We change the minus sign check to use the third of these chars as well.

Finally, we add a lot of logical and code to the return statement. These anded clauses check the notation's accuracy as well as the original failure check for cin. This will be a major change in how this function operates moving the caller from being able to use the old-style:

```
cout << "What are the parameters of your die (dS~M+A)?  ";
my_die.read();
while ( cin.fail() )
{
    cin.clear();
    cin.ignore(numeric_limits<streamsize>::max(), '\n');
    cerr << "\n\aInvalid parameter set!  Please try again!\n";
    cout << "What are the parameters of your die (dS~M+A)?  ";
    my_die.read();
}
```

to making them use a new and more OOP-style:

```
cout << "What are the parameters of your die (dS~M+A)?  ";
while ( ! my_die.read() )
{
    cin.clear();
    cin.ignore(numeric_limits<streamsize>::max(), '\n');
    cerr << "\n\aInvalid parameter set!  Please try again!\n";
    cout << "What are the parameters of your die (dS~M+A)?  ";
}
```

This is because we're changing the very meaning of failure for this `class`' input method. Thus the caller can no longer rely on just `cin`'s opinion of how the input went. (Which is good, since that view depends entirely on the numbers and cares not a whit for the notation that was supposed to separate them!)

The calculation functions are much like input and output, so I'll forgo them for now. But the comparison functions are a bit different, so let's explore them in more detail.

5.1.2.5.2 Two Objects

When we compare two objects, we only have to name one of them — the argument object. Let's explore `is_same` first, for instance:

```
bool Die::is_same(const Die & d)
{
    return sides == d.sides &&
           abs(scale - d.scale) <= 1e-6 &&
           abs(adjustment - d.adjustment) <= 1e-6;
}
```

Here we have a mix of member variable access patterns. We've got the raw member variable names for accessing the calling object's data and the dotted ones for accessing the argument object's data. Other than this, we see our familiar pattern for checking equality betwixt floating-point data since == is too fallible.

We can also look at, say, `can_be_smaller`:

```
bool Die::can_be_smaller(Die d)
{
    return min() < d.min();
}
```

Here we learn that not only can member variables be used undotted to mean those from the calling object, but we can also call member functions without a dot to denote calling them with respect to our same calling object! And we still have a call to that function with respect to the argument object, as well.

The other three comparisons are similar to `can_be_smaller`, so we'll not dwell on them here.

5.1.2.5.3 A New Object

Finally, let's look at the transformation methods that create new objects for their `return`ed results. Let's start with `change_shape`:

```
Die Die::change_shape(long new_sides)
{
    Die cd;
    cd.scale = scale;
    cd.adjustment = adjustment;
    cd.sides = new_sides;            // what if this is 0 or negative?!
    return cd;
}
```

Here we see that a new object is created locally inside the method and its member variables are set accordingly to either the same values as those of the calling object or to the new value given in the argument list. Then, at the end of the function, we `return` the local object which is copied back to the caller for them to do with as they see fit.

We do have a note to ourselves about an issue we might face in the future: what if the caller sends in a 0 or negative number of sides? Should we just store it or deny their attempt? This is fodder for our next section on usability for the `class` (5.2).

The other two are similar, but slightly different. Let's just look at `translate` for an idea:

```
Die Die::translate(double applied_adjustment)
{
    Die td;
    td.sides = sides;
    td.scale = scale;
    td.adjustment = adjustment + applied_adjustment;
    return td;
}
```

Here the calling object's value is added to by the argument value instead of just being changed to it. But the pattern is otherwise unchanged from above.

5.1.2.6 The Whole Picture

First, how do these things normally go together? Well, let's consider that in the past we've placed type definitions — like `typedef`s and `using` aliases — as well as function prototypes at the beginning of the code. (Well, after the library includes but before the main.) This means, of course, that the `class` definition which is both of these things — a type definition and a lot of prototypes for methods — should go above the main.

And, before, we placed (non-`inline`) function definitions after the main. Thus, we'll put our `class` method definitions after the main as well.

Normally I'd now bring you a whole program listing to show how all of this comes together. But it is a tremendous bit of code this time: nearly 250 lines! So this time, I'm going to link it to you on a website instead. Here, then, is phase one of our Die class exploration.[7]

5.2 Making It More Usable

The class design from phase one was perfectly fine, but not as flexible as we'd like it to be. There could be situations which would prevent its easy re-use. For instance, the programmer might want to create an object to represent a particular Die to roll — maybe the special '-1/3, 0, 1/3' I spoke of. The current class depends on the end-user for input of Die parameters — not the programmer.

Or the programmer might want to alter the parameters of a Die before display, but the only way to create a Die having different parameters would be to 'read' it from the end-user or to calculate a scaling or translation of a Die originally entered by the end-user.

Or the programmer might want to send the parameters of a Die to a graphical display device — drawing the Die to be part of some kind of visual rolling. We don't have any such capability — and I certainly don't plan to put it in. We need some way for our system to be semi-transparent to the private data. (We still don't want to allow arbitrary change, but anyone should be able to look at our privates, right?)

Or the programmer might want to take mouse clicks from a GUI and convert them to Die parameters. (Sliders or drop-down selections or whatever. . .) Again, I'm not going to code for that. So we need a way for the programmer to store reasonable values within a class object.

To make our classes more re-usable, we will want to make the following additions:

- accessors
- constructors
- mutators

In addition to the programmer using the class gaining flexibility, we as class implementers will be able to make our code much more efficiently using these facilities. (Well, not the accessors, those are only for the out-of-class programmers.)

5.2.1 Accessors

For sake of convenience, let's tackle the accessors first. Accessors allow the programmer to access the data in their objects by returning to them a copy of the member variable values. This is necessitated by the whole private/public access specifier thing. However, it also has the advantage of allowing programmers to do things with our class type that we haven't anticipated — or simply don't want to deal with.

For instance, if the programmer wants to visualize a Die on a graphics display, they can access the member parameters with our accessors and send them on to the graphics routines for displaying the Die's proper face values. We don't have to deal with graphics cards, device drivers, OS kernel modules,[8] window coordinate systems, clipping, scaling, etc.

5.2.1.1 The Plan

We'll add the following functions to our Die class:

[7] Since this file ends in .cpp, some browsers — Firefox™— won't open it directly. But you'll have to save it first and then open it in your compiling environment to play with it, anyway.

[8] There is a part of any OS called the kernel. Seriously!

```
long get_sides(void);
double get_scale(void);
double get_adjustment(void);
```

These are named `get_` and member variable to indicate that the programmer who owns the `class` object is trying to get a copy of the member variable's value.[9] This is only a typical pattern and not a hard-and-fast rule, but it does make them easier to spot.

Each takes no arguments and `return`s the same type as the member variable they are designed to retrieve.

5.2.1.2 The Implementation

The accessors are defined like so:

```
long Die::get_sides(void)
{
    return sides;
}

double Die::get_scale(void)
{
    return scale;
}

double Die::get_adjustment(void)
{
    return adjustment;
}
```

That's it. They pretty much all look like this. Some programming environments will even give you a menu selection or hotkey that creates accessors for a `class`' member variables for you!

5.2.1.3 Putting It to Use

Again, accessors are rarely used from inside the `class` — only by programmers outside the `class`. We can get to the `private` data inside the `class` whenever we want, after all.

We'll find a use-case for those using our `class` shortly, however, so stay tuned!

5.2.2 Mutators

Mutators mutate the values in a `class` object to those the programmer who declared the object desires — maybe. Mutators are the very heart of the `private`/`public` access specifier thing. Without them, it would all break down into chaos and mayhem pretty quickly! These special functions provide a centralized place to code error checking, data validity assurance testing, and/or anything else we want to enforce as far as the data goes.

Some even say it makes a programmer think twice before making a change to the data because they must call a function rather than use a simple assignment statement. Maybe they'll think better of it or simply think more clearly about what value they want to change it to.

Although a `Die`'s `adjustment` doesn't really have much in the way of error checks or validity tests we can perform, the `sides` and `scale` can. Other simple `class` types do, too:

[9] Interestingly, this seems to be a pattern back-borrowed from Java — a C++ derivative language.

- you wouldn't want the denominator of a rational number class to become zero (0)
- you wouldn't want to have a time-of-day be set to 29:-41
- you wouldn't want to have an alarm set for -28945 seconds in the future

When a mutator cannot make a change validly, it should just ignore the programmer making the attempted change or return to them a false value to indicate the change couldn't be made. (Then it is their problem for not checking the mutator return value, right?)

Even though Die adjustments don't allow for much in the way of error checks, they do represent another aspect of mutator use — simply changing the member data in ways we didn't anticipate (or don't want to deal with). If the programmer is dealing with a graphical interface and receives mouse clicks which represent somehow the parameters of a Die, they can store those into a Die object for holding or other uses (like averaging or relative positioning on the number line or even scaling!).

5.2.2.1 The Plan

We'll add the following functions to our Die class:

```
bool set_sides(long new_sides);
bool set_scale(double new_scale);
bool set_adjustment(double new_adjustment);
```

These mutators are named set_[10] and member variable to indicate that the programmer who owns the class object is trying to set a new value into the member variable. This is only a typical pattern and not a hard-and-fast rule, but it does make them easier to spot.

They all take a single parameter that is the same type as the member they are designed to mutate. No need for any extras here. Just the facts, as they say...

They will all return a bool to tell the caller of their success or failure at their mission. It is, of course, up to the caller to check that result and act on it.

5.2.2.2 The Implementation

Our general mutator design looks something like this:

```
bool okay = false;
if ( /* argument data is valid */ )
{
    member = argument;
    okay = true;
}
return okay;
```

Of course, this is just pseudocode since the ”argument data is valid” bit isn't gonna compile.

For example, our sides member variable for the Die class can be set like so:

```
bool Die::set_sides(long new_sides)
{
    bool okay = false;
    if ( new_sides >= 1 )
    {
```

[10] For this reason, some people call mutators 'setters' instead.

```
        sides = new_sides;
        okay = true;
    }
    return okay;
}
```

Here we check that the new value for `sides` is reasonable before storing it in the member variable for real. If it isn't reasonable, we don't store it and `okay` stays `false` and we `return` that. But if it is a good value, we store it, change `okay` to `true`, and `return` that!

The adjustment has no bad values — except a few values for the `double` type itself (`inf`, `-inf`, `NaN`). These can be difficult to check for depending on your compiler and library version/implementation.

The infinities are tricky, but `NaN` has a CPU-based way to check. Only the `NaN` bit-pattern is unequal to itself!

```
// only NaN is not equal to itself
if ( new_adjustment == new_adjustment )
{
    adjustment = new_adjustment;
    okay = true;
}
```

But the compiler doesn't like to do `==` (or `!=`) on floating-point data.

To tell the compiler to shut up about this warning for this specific code, we could use a `pragma`:

```
#pragma SOMETHING HERE TO TURN DOUBLE== WARNING OFF
   if ( new_adjustment == new_adjustment )  // only NaN is not equal to itself
   {
       adjustment = new_adjustment;
       okay = true;
   }
#pragma SOMETHING HERE TO TURN DOUBLE== WARNING BACK ON
```

But the *SOMETHING* will vary from compiler to compiler and sometimes even from version to version of the same compiler. Sometimes it is hard to find even with Google™and knowing the basic form of what you are looking for!

Luckily, `cmath` has the functions `isinf` and `isnan` since C++11 that check for infinity and `NaN` values:

```
if ( ! isinf(new_adjustment) && ! isnan(new_adjustment) )
{
    adjustment = new_adjustment;
    okay = true;
}
```

So now we can use these to avoid problems in the setting of the `scale` and `adjustment` member variables of our `Die` `class`:

```
bool Die::set_adjustment(double new_adjustment)
{
    bool okay = false;
    if ( ! isinf(new_adjustment) &&   // new_adjustment isn't an infinity
         ! isnan(new_adjustment) )    // nor is it a NaN...
    {
        adjustment = new_adjustment;
        okay = true;
    }
    return okay;
}

bool Die::set_scale(double new_scale)
{
    bool okay = false;
    if ( abs(new_scale) > 1e-6 &&                    // new_scale != 0.0
         ! isinf(new_scale) && ! isnan(new_scale) )  // not bad, either
    {
        scale = new_scale;
        okay = true;
    }
    return okay;
}
```

But there are situations where we can't do error checking at all. It really isn't all that unusual, sadly. What kinds of situations are like this? Peoples names, for instance, cannot be validated. Are you seriously going to tell them that that isn't their name or that they've spelled it wrong or something? *phbbt*[11] Good luck!

So the base pattern in these situations is this:

```
member = argument;
return true;
```

We keep the overall `bool return` pattern for consistency of design and use — even though it isn't useful here. This keeps the caller from constantly wondering, "Is this the mutator that `return`s something or the one that doesn't?" They can just always check the result for `true/false` and not worry over it.

5.2.2.2.1 Extreme Cases

Some data types call for grouped mutation schemes rather than individual mutators for each data member. For instance look at a `Rational` number `class`. It has a `numerator` and a `denominator` linked in an implicit division relationship. There is also the common and programmatically sensible idea of keeping a fraction in 'lowest terms' — canceling out any common divisors between the `numerator` and `denominator`. Let's say that such a `Rational` number object is currently 4/15 and the programmer wants it to be 3/7 instead. If we coded separate `set_numer` and `set_denom` functions, they'd end up with 1/7 instead:

```
obj.set_numer(3);  // 3 cancels with 15 to form 1/5
obj.set_denom(7);  // no cancellation, now 1/7
```

In order to make this work out correctly, we really need to mutate these two members at the same time. Any change to one, after all, will change the other because of the cancellation effect:

[11] This is the onomatopoeia for the sound of 'blowing someone a raspberry'. It took me many hours of research. Think about it!

```
obj.set(3,7);    // okay, all set together:  3/7
```

If the caller doesn't need one of them to change specifically, they can always call like this:

```
obj.set(numer, obj.get_denom());
```

Here we just use the accessor to grab the current value of the denominator and pass it into the mutator call.

5.2.2.3 Putting It to Use

One place to use mutators is in the arithmetic methods. We can update scale_by, for instance, from:

```
Die Die::scale_by(double applied_scale)
{
    Die sd;
    sd.sides = sides;
    sd.adjustment = adjustment;
    sd.scale = scale * applied_scale;
    return sd;
}
```

to:

```
Die Die::scale_by(double applied_scale)
{
    Die sd;
    sd.set_sides(sides);
    sd.set_adjustment(adjustment);
    sd.set_scale(scale * applied_scale);
    return sd;
}
```

The first two aren't doing much since the sides and adjustment of the calling object have already been error-checked. But the third one will protect us from bad applied_scale values.

5.2.2.3.1 And Also Input

Note that the input function is the most egregious place to forget to mutate! If we don't trust data coming from another programmer in a mutator, we certainly don't trust data coming from a user! *grin*

Thus, our read function becomes:

```
bool Die::read(void)
{
    char d, s, a;
    long sds;
    double scl, adj;
    cin >> d >> sds >> s >> scl >> a >> adj;
    if ( a == '-' )
    {
        adj = -adj;
    }
    return !cin.fail() &&
           d == 'd' && s == '~' && (a == '+' || a == '-') &&
           set_sides(sds) && set_scale(scl) && set_adjustment(adj);
}
```

Here we've renamed our proposed char variables to alliterate with their member variables — except for d, of course. We've also added local variables to protect our member variables from direct overwriting in the case of bad data.

Finally, we logically and together not only the lack of failure on cin and the notation situation, but also whether or not the member variables mutate correctly!

5.2.2.3.2 Toward Software Engineering

The result of the above &&'d set_*[12] calls is that we'll have a partially valid object at times. We'd rather have a consistent object at all times — all valid or all invalid.

For instance, if sds, scl, and adj are all fine, the class is in a consistently valid state and we are happy. Even if sds fails to initialize — returning false, we'll end up in a consistently invalid state since neither of scl nor adj will even be attempted to mutate. But consistent is good so we are still happy.

But if adj were invalid, the class would be only 2/3 valid — making us unhappy. Worse yet, if scl is invalid we'll fail to even test adj and end up with a 2/3 invalid object. Of the 8 possible scenarios — each value can be validly or invalidly entered — we have only two consistent states and 6 inconsistent ones.

In fact, there will always be only two consistent states and $2^n - 2$ inconsistent ones where there are n member variables in the class. The only good situation for us is when there is only one member variable such that there are no inconsistent states! But this happens quite rarely so. . .

But having that consistency would mean a whole new layer of function abstraction — decoupling the validity tests and the setting of member variables' values[13] — and therefore lots of more complicated code. Maybe some other time. . .

5.2.3 Constructors

Constructors are special methods that let the programmer creating an object of our class type initialize it properly at declaration. Recall that string class objects can be initialized in several ways:

```
string s;    // no special initial value -- defaults to empty

string t("Bob"),        // initialize to literal string value
       v = "Bob, too",  // initialize to literal string value
       m{"Also Bob"};   // initialize to literal string value

string u(42, '&');   // initialize as 42 ampersands

string q = u,    // initialize as a copy of u
       r(v),     // initialize as a copy of v
       n{m};     // initialize as a copy of m
```

We can make our class objects behave in similar ways. Albeit appropriate to the class we are creating, of course. None of this string and char non-sense for a Die to be rolled. *grin*

[12] Recall the * here denotes matching potentially multiple characters so we are talking about all of the setters at once. And it is you doing the matching — not the compiler or your code.

[13] Decoupling simply means breaking the two apart. More on that concept in chapter 6

As you'll recall, we can even use a class name with a parentheses enclosed list of comma-separated initializers to create an anonymous object:

```
string()         // a default object
string("Bob")    // a string object containing Bob
string(42, '&')  // a string object repeating & 42 times
string(s)        // a string object exactly like the named object s
```

(All but the third can be curly-brace enclosed, too.)

The middle two forms are more prominent, of course. Why would you need to create an empty string on the fly? And why not just code "" instead? Similarly, if you already have a string, why not just use it instead of making a copy of it?

Constructors are automatically called by the compiler when an object is instantiated of our class.[14] Which constructor to call depends on what arguments are specified:[15]

```
no arguments                             ==> default constructor
a [const] reference to another class object ==> copy constructor
other arguments                          ==> appropriate constructor overload
```

So, as you can see, the types and number of arguments passed to a constructor determine what kind of constructor it is (default, copy, or otherwise). Constructors must be overloaded to distinguish from one another. Why? Because constructors must also be named after the class they construct. All string class constructors are named 'string', for instance. And all our Die class' constructors will be named 'Die'.

5.2.3.1 The Plan

We'll add the following code to our class definition:

```
Die(void);                       // default ctor
Die(const Die & d);              // copy ctor
Die(long new_sides);
Die(long new_sides, double new_scale, double new_adjustment);
Die(long new_sides, double new_alterer, bool is_scaling);
```

Quick note: the word 'constructor' is often abbreviated 'ctor' in comments and the like. I've even seen it in published articles in respected journals!

Constructors other than the default and copy don't have special names but just overload the call signature of a constructor for the class.

The last one will need help because constructing with either sides and scale or with sides and adjustment would both be long,double signatures. So we combine them with a helping bool argument to tell them apart.

bool argument? Well, since we can't know from just long,double which double member was intended to be set along with the sides, we need a bool or enumeration to tell them apart. Since there are only two to differentiate, we use a bool for simplicity. When it is true, we'll change the scale member and when it is false, we'll change the adjustment instead.

[14] That is, when a variable is declared of our type. Recall the 'instance of' vernacular we spoke of earlier

[15] That is, the constructor to call depends on the overload signature of the function since all constructors have to share the same name. More on that shortly.

This will be supplemented with a set of helper constants that the caller can use to specify their intent more clearly:

```
const bool CHANGE_SCALE = true,
           CHANGE_ADJUST = false;
```

These are placed above the class definition for ease. We'll discuss placing them inside the class in section 6.7 later.

5.2.3.2 The Implementation

Let's look at each type of constructor in turn.

5.2.3.2.1 Default Constructors

Here is our default constructor:

```
Die::Die(void)
{
    sides = 6;
    scale = 1.0;
    adjustment = 0.0;
}
```

It fills in the member variables when no data was provided by the declaring programmer.

Notice the lack of a return type[16] or statement! This is because all constructors automatically return the object being constructed.[17] Also, the object being constructed is considered our calling object, btw.

To maintain consistency and avoid mistakes, this behavior was deigned best provided automatically by the compiler rather than individual programmers. Programmers tend to get 'creative' and do things 'cleverly' when you least expect it...

5.2.3.2.2 Copy Constructors

The copy constructor is called whenever an object is created as a copy of another object that already exists:

```
Die     P;      // default constructed
. . .
Die     Q = P,  // Q is created as a copy of P
        R(P),   // R is created as a copy of P
        S{P};   // S is created as a copy of P
```

These three syntaxes are mere conveniences/options and work the same way — all call the same copy constructor. That copy constructor will look like this:

```
Die::Die(const Die & d)
{
    sides = d.sides;
    scale = d.scale;
```

[16] You may have noticed this during the plan above, even!

[17] Actually, a reference to this object is returned. But that isn't important for now...

```
    adjustment = d.adjustment;
}
```

We simply need to copy the member variables out of the argument object and store them into our calling object.

In fact, it is so easy that the compiler will do this for you if you forget. Why? This function is used so often that the compiler really needs this constructor to function!

Although such use is not often coded as above (it can happen, but isn't prevalent), there are four other places where the compiler needs to make copies of objects automatically for us: pass by value arguments, return [by value] results, a by-value walker in a range-based for loop, and catch by value exceptions.

That is, the returned value is created as a copy of the return expression's value. For instance, the function Die::scale_by will return its result by value and thus must make a copy of the local object (sd) to give back to the caller since sd will shortly be destroyed as the function ceases to exist!

Copy Assignment Operator

As standards advance, even mundane tasks become more difficult. Here we have evidence in the need to tell the compiler that, just because we wanted our own copy constructor doesn't mean we want to write our own operator for assignment (aka a copy assignment operator). These two will go hand-in-hand later in your studies, but we don't need the tediousness of operator= just yet. *smile*
Instead, we just tell the compiler that we want the default or compiler-supplied operator= with this syntax:

```
Die & operator=(const Die & d) = default;
```

Granted, it looks a bit odd right now, but next term, perhaps, you'll get it. No worries!

Similarly, the formal argument is created as a copy of the actual argument during a pass by value. This is done from time to time, but we usually try to pass class objects by constant reference instead of by value. This avoids the extra memory and the time taken to make the copy. Now that we have a name for all of that, we can say that it avoids calling the copy constructor!

When using a range-based for loop, the index value is typically copied from the string by value unless reference was stated to allow changes.

Finally, when catching an exception by value, the copy constructor is used to make a copy of the originally thrown exception for local use in the catch block.

So, if the compiler is going to make a copy constructor for us anyway, why are we writing one? I suppose you don't have to, but it is good practice. You see, one day you'll find a situation that requires you to write your own copy constructor because the one the compiler provides won't be good enough any more. It won't happen soon, but it will come.

In fact, coding a copy constructor now might just set you up for some extra warnings from the compiler. But we'll deal with those in the next section on efficiency and robustness...

5.2.3.2.3 Other Constructor Overloads

Both of the above constructors showed that trusted values don't have to be mutated. But now we start receiving values from outside the class — data that cannot be trusted. This data should always be mutated to ensure our safety![18]

[18] Why not just check it again here? We've already coded it once! The whole idea of functions is to avoid duplicating code, after all. Always keep that in mind — even now in the realm of classes.

```
Die::Die(long new_sides)
{
    sides = 6;                // ensures object has valid data even if
                              // new_sides is bad data
    set_sides(new_sides);
    scale = 1.0;
    adjustment = 0.0;
}
```

Here we set up default values — even for the `sides` member! — and call the mutator for the `new_sides` value check. The `sides` is defaulted, too, to make sure that it has a reasonable value even if the subsequent mutation fails.

```
Die::Die(long new_sides, double new_scale, double new_adjustment)
{
    sides = 6;
    scale = 1.0;
    adjustment = 0.0;
    set_sides(new_sides) &&
    set_scale(new_scale) &&
    set_adjustment(new_adjustment);
}
```

Similar to above, but here we've `&&`'d the mutators together just like we did in the `read` function above. Not particularly necessary, but not harmful, either.

```
Die::Die(long new_sides, double new_alterer, bool is_scaling)
{
    sides = 6;
    scale = 1.0;
    adjustment = 0.0;
    if ( set_sides(new_sides) )
    {
        if ( is_scaling )
        {
            set_scale(new_alterer);
        }
        else
        {
            set_adjustment(new_alterer);
        }
    }
}
```

Here's that weird one with the `bool` parameter. Let's look at this one in some detail. All members are set to defaults to start. Then, if the `sides` sets successfully, we try to set the other member that was requested to be altered. We do it in an `if` to keep that consistency we talked about with the `read` function.

Then, we check which of the `double` members was intended to be changed with another `if` based on the `is_scaling` argument. When it is `true` (aka `CHANGE_SCALE`), we change the `scale` to the `new_alterer` value. Otherwise, it must be `false` (aka `CHANGE_ADJUST`) and we change the `adjustment` instead.

5.2.3.2.4 Remember!

Keep in mind that constructors are special in that there can be no return statement since the object being constructed (the calling object) is automatically returned from the constructor by the compiler.

5.2.3.3 Putting It to Use

To use constructors, just make a variable! For instance:

```
Die my_die, trans_die;
```

would call the default constructor twice — once for each of my_die and trans_die.

Other constructions might look like these:

```
Die obj1{42},     // create a 42-sided die
    obj2{3,.5,2}, // create a 3-sided die with scale .5 and offset 2
    obj3{8,2,CHANGE_ADJUST};  // create an 8-sided die with default
                              // scale and adjustment 2
```

And so on. *smile* With the five constructors we've created, there are six different patterns! (That bool argument doubles down on the last one, you see. . .)

5.2.3.3.1 Improving Other Methods

But that's not all! We can also improve our arithmetic functions by using these constructors. When last we left them, scale_by, for instance, looked like this:

```
Die Die::scale_by(double applied_scale)
{
    Die sd;
    sd.set_sides(sides);
    sd.set_adjustment(adjustment);
    sd.set_scale(scale * applied_scale);
    return sd;
}
```

This can now be shortened to:

```
Die Die::scale_by(double applied_scale)
{
    Die sd(sides, scale * applied_scale, adjustment);
    return sd;
}
```

Pretty nice, eh? Here the constructor will call through to the mutator for us to protect against the possibility of a bad applied_scale value. Calling a function that already does the job you want to do is one of the prime tenets of coding with functions, after all. Never reinvent the wheel!

5.2.4 The Whole Picture

Again, this program comes to more lines of code than we want to repeat here (almost 500!), so I'm placing it on a website for you. Here is phase two of our Die class exploration.

5.3 Making It More Efficient and Robust

Let's next turn our attention to fine tuning the class mechanism to work in a speedier, leaner, more secure, and more elegant fashion is our next goal. Four steps are to be taken here:

- inline'ing class methods when appropriate
- use of member initialization lists for constructors
- proper declaration of const-ness for class methods
- RVO (or return value optimization)

5.3.1 inlining Methods

To inline a class method, simply define it instead of prototyping it within your class definition. The compiler takes the definition of a function within the class definition as an automatic suggestion to inline the function — because you wouldn't possibly clutter your class definition with a function definition if it weren't simple, short, etc. like an inline function should be! The inline keyword is **NOT** even required here. (*bounce*)

Here is an example of an inline function for our Die class:

```
class Die
{
    // ...
public:
    bool set_sides(long new_sides)
    {
        bool okay = false;
        if ( new_sides >= 1 )
        {
            sides = new_sides;
            okay = true;
        }
        return okay;
    }
    // ...
};
```

Note the lack of the inline keyword and the normal style of the definition. I particularly point out the latter because many people think that sideways style we discussed in section 4.5.3 on function inline'ing is still the right thing to do! *shakes head* Silly nonsense!

5.3.2 Member Initialization List vs In-class Initialization

Thus far you may have received warnings from your compiles. These warnings tell of a tool called a member initialization list that should be used to make your constructors more efficient. What's this all about and how can we get in on the act to avoid these horrible warnings?

It turns out that the compiler will — just for class member data — zero-out or default construct all member variables. It does this before the constructor body is run and then the constructor can just change the non-zero data to new values. But, this fact was not well-publicized and so not everyone took advantage of it.

Studies showed that it was slowing down C++ code vs similar codes in other languages. Since the committee is dedicated to making C++ efficient and competitive, they were about to change this ruling and stop the pre-zeroing behavior. But then those that were using this feature said, ”Wait! You can't remove that! We've got thousands of lines of code depending on that behavior and can't afford to go back and fix it all to a new standard.” Keeping things backward compatible is often a good goal in itself, so the committee thought long and hard and came up with the idea of the member initialization list for constructors.

When the compiler sees that you've used a member initialization list, it will eschew its pre-zeroing behavior and initialize the members as you've done in your list instead. Still before the body of the constructor is run!

To emphasize this timing, the member initialization list is placed on the constructor definition between the function's head and its body. Such placement before the body says, in effect, ”I'll be taken care of before the function itself runs.”

So what does it look like? It would look like this for our `Die` default constructor:

```
Die(void)
    : sides{6},
      scale{1.0},
      adjustment{0.0}
{
}
```

I'm assuming here that this function is `inline` but not showing that it was defined inside the `class` definition as that is space-consuming beyond reason.

So the syntax is a colon followed by a comma-separated list of parenthesized or curly-braced initializers[19] — one per member variable — in the original order of declaration. That last bit is important as, if you rearrange the initializers, the compiler is almost certain to produce a different warning from before about having to put the initializers back in the declaration order for you. Quite annoying!

Is the style of one initializer per line necessary? No. I could have just as easily coded it like so:

```
Die(void)
    : sides{6}, scale{1.0}, adjustment{0.0}
{
}
```

or even like this:

```
Die(void) : sides{6}, scale{1.0}, adjustment{0.0}
{
}
```

It all depends on your personal style and how much space you want to leave for side-comments.

Finally, note that the definition body is present but empty. This shows that all initialization was done in the member initialization list but that we *ARE* defining the constructor here and now. The member initialization list has to appear on a constructor definition — not on a prototype!

The empty body bothers some folks so they will go to the trouble to put an empty statement in it like so:

[19] Curly braces are in vogue, of course.

```
Die(void) : sides{6}, scale{1.0}, adjustment{0.0}
{
    ;
}
```

I'm not fond of this as it just wastes time, space, and concentration for the reader.

But the overall effect of this is to double the throughput of some constructor calls. Those that would have overwritten all the member variables from their defaulted/zeroed state are run at twice their original speed with this technique. Those that change only some of the member variables are still sped up but not as much as doubled. Only those that use all zeroed initialization won't get a speed boost from this technique. But that would require extra logic from the compiler that most aren't willing to enter so for now, we either use some form of initialization or suffer the horrid warnings about it. *shrug*

5.3.2.1 Hidden Gems

But you don't have to initialize every member all the time. What? Yep, you can also delegate the initialization to another constructor:

```
Die(long new_sides)
        : Die{}
    {
        set_sides(new_sides);  // always call the mutator to validate
                               // data coming from the outside!
    }
```

Here we've told the compiler that before the `Die(long)` constructor runs, we should do the same member initialization as on the default constructor. Again, this is called delegating or delegation of the initialization to another constructor or just delegating to another constructor.

5.3.2.2 An Alternative

Then, in C++11, it was allowed to initialize simple members of a `class` when they are declared. This can allow you to skip some member variables in the member initialization list if they always take the same initial values:

```
class Die
{
    long sides{6};
    double scale{1.0}, adjustment{0.0};
public:
    Die(void)
    {
    }
};
```

This will still compile without warnings but without a member initialization list as well.

As implied above, these two techniques can be mixed and matched to cover all of the member variables initializations.

5.3.2.3 The Most Vexing Parse

One other catch: if you have a member variable which is of a `class` type[20] (say a `string`):

[20] More on this idea later!

```
class ClassName
{
    // ...
    string str_memb;
    // ...
};
```

You *must* use just empty parentheses or curly braces in your member initialization list to default construct that member (i.e. to call that member's default constructor):

```
ClassName( /*...*/ )
    : /*...,*/
      str_memb() /*,
      ...*/
{
    // ...
}
```

(The ... are for parameters, other members initializations, and any body code. Those details are left out so we can focus on what we're dealing with here.)

Normally you default construct a `class` variable by having nothing at all:

```
// some function body
{
    string s;     // default constructs variable s
}
```

In fact, trying to default construct a normal variable with empty parentheses is a hidden error — called the most vexing parse[21] — it declares a function named like your intended variable taking no arguments and `return`ing the proposed type of your variable:

```
string t();  // declares a function t with no args and
             //   returning a string
```

So watch out for that!

This also affects the in-`class` initialization: you must use empty curly braces to default construct a member variable of another `class` type or you'll be declaring a function instead of a variable.

5.3.3 const-Correctness of Methods

Let's look back at a few of our functions from the last version of the `Die` `class`:

```
double Die::max(void)
{
    return static_cast<double>(sides) * scale + adjustment;
}

bool Die::can_be_larger(Die d)
{
```

[21] Parse means to read and interpret here. We talk of compilers parsing their input — your program.

```
    return max() > d.max();
}

bool Die::isSame(const Die & d)
{
    return sides == d.sides &&
           abs(scale - d.scale) <= 1e-6 &&
           abs(adjustment - d.adjustment) <= 1e-6;
}
```

Note how the Die argument to isSame is passed by const& as is our wont but the Die argument to can_be_larger is by value instead. We said we could/should pass class objects by value when we were going to change them inside the function, but we don't change it here. We just call for its maximum value. This shouldn't change it, and, looking at the max function itself, we see it doesn't change anything. So why can't we make the argument of can_be_larger const& like that for isSame?

Well, the compiler only checks for something remaining constant when we tell it to. If we don't tell it something is const, it assumes it will/can change. Here, the thing that might change is not exactly the argument to can_be_larger but the calling object of max. Since that calling object wasn't known to be const, we can't mark d from can_be_larger const&, either. The effect chains back.

So, how do we mark a calling object const? Well, there were a few alternatives:

```
double const Die::max const(const void) const
{
    return static_cast<double>(sides) * scale + adjustment;
}
```

As you can see, we could have placed it before the function name, after the function name but before the argument list, inside the argument list, or after the argument list. The trouble with before the function name is that this would have just modified the return type instead.[22] The same problem exists inside the argument list — it would have altered an argument instead of the calling object. That leaves between the function name and argument list or after the argument list. For whatever reason, the committee/Bjarne chose after the argument list. *shrug* Six of one, half-a-dozen of another, right?

So, to mark the calling object of max constant, we do this:

```
double Die::max(void) const
{
    return static_cast<double>(sides) * scale + adjustment;
}
```

That done, we can now mark the argument of can_be_larger as constant as well:

```
bool Die::can_be_larger(const Die & d)
{
    return max() > d.max();
}
```

This logic extends to the calling objects of min and avg as well.

Although that does make us happy, we can take it further. We should probably mark any calling object whose members aren't meant to change in the function as constant, shouldn't we?

[22] Recall that a const can go before or after a type!

Why not? It's a few extra keystrokes and it will make the compiler check for accidental changes in all the functions which will make for more solid code. This is a case of making our class more robust at the cost of a little extra typing — well worth it!

That means we should also mark the calling objects of can_be_larger, can_be_smaller, etc. all constant as well. Wow, this is a *LOT* of typing. Is it really worth it? **YES!** It definitely is. The extra checks from the compiler are a really nice safety net that we just can't pass up now that we know how to turn them on.

One more thing, though, before we go on. We don't usually say 'mark the calling object const' because it is just too long. Instead we say that we are 'marking the method const'. Which is a bit shorter. It still conveys that the caller of the method can't be changed and that's the main goal.

Placing the keyword const after the function's head (both prototype and definition) has several effects:

- We are promising not to alter the calling object's member variables during the call.
- Therefore the compiler will check up on us and verify that promise.
- Once the calling object is known to remain constant, the compiler may be able to make some efficiency adjustments to the binary codes. This depends on the specifics of the target platform, of course.
- If an object is const (somehow — maybe it was marked const& as an argument), the compiler will only allow the programmer to call methods which have been marked as const upon that object. This further improves the robustness of our class.
- Once the calling object is known to remain constant, you may be able to make some efficiency adjustments to the class in some way. See const& argument upgrade for can_be_larger, for instance.
- Finally, the const-ness of a method is considered in determining its overloadability. This effect only applies to methods of a class, of course, since non-class functions don't have calling objects to keep const...

(That last one is mostly trivia for now, but will come in quite handy in another semester or so!)

But, then, what should be const in general? If we are going to make this a regular thing, we need a rule of thumb to go by. Accessors, most behavioral functions, and printing should all be marked const.

That leaves mutators, constructors, and reading to *not* be const.

Just a final thought: the const-ness of a method, IMHO, should, but apparently doesn't, affect the ability to call a method against a temporary/anonymous object. So be careful!

5.3.4 return Value Optimization

Let's examine now the arithmetic functions like scale_by here:

```
Die Die::scale_by(double applied_scale)
{
    Die sd(sides, scale * applied_scale, adjustment);
    return sd;
}
```

This is fine, but it is doing a lot of work. On most systems, it will look like this:

- create local memory for sd
- fill local memory

- copy local memory to return area
- destroy local memory
- return area is used by caller
- destroy return area

We can cut this in half by enabling a compiler optimization known as the return value optimization or RVO for short. Enabling RVO is as easy as creating an anonymous object on the return line itself:

```
Die Die::scale_by(double applied_scale)
{
    return Die(sides, scale * applied_scale, adjustment);
}
```

With that in place, the compiler recognizes that the object is just being returned and can optimize the situation by constructing it directly in the return area:

- fill return area
- return area is used by caller
- destroy return area

That cut our workload in half! If we further inline the function, we'll increase the throughput even more!

This technique isn't just good for methods, either! It can be used anytime a class object is being returned from a function by value. In fact, it comes in handy so often, the committee recently mandated that compilers must seek out RVO opportunities. It used to be that compilers could ignore these!

In addition, the committee/compiler developers invented a new form of RVO called named-RVO. In named-RVO, the compiler finds an object declared in the local function scope that is minorly manipulated and then returned and moves it to the return area instead of local space on the function call stack. This requires no intervention by you, but is not yet mandatory.

5.3.5 The Whole Picture

Again, this program comes to more lines of code than we want to repeat here (almost 400!), so I'm placing it on a website for you. Here is phase three of our Die class exploration.

5.4 Interesting Usage and Knowledge

There are a couple of interesting things we can do with classes at this point. One is called composition and involves having one class with a member variable of a different class type. The other is method chaining. This is like operator chaining as discussed in section 2.3.5, but the syntax is a bit different.

There is also another way to group multiple variables together that describe real-world data. This is called a structure and differs slightly from a class. We'll discuss this in this section as well.

5.4.1 Method Chaining

Let's look at part of the main from earlier:

```
scaled_die = my_die.scale_by(scaling_factor);

both_adj_die = trans_die.scale_by(scaling_factor);
```

Here, the four objects are, of course, of the Die class type. trans_die was a translated version of my_die from earlier in the code.

If we hadn't already stored trans_die, though, we could have achieved this same result by:

```
both_adj_die = my_die.translate(offset).scale_by(scaling_factor);
```

or, if you prefer:

```
both_adj_die = my_die.translate(offset)
                     .scale_by(scaling_factor);
```

or even:

```
both_adj_die = (my_die.translate(offset)).scale_by(scaling_factor);
```

The extra parentheses are not necessary, but some people like them...

Again, this is a form of chaining — using the result of one function to then call another function. We've just applied the concept to class methods. It is all the rage and considered quite elegant coding style.

If I weren't sending the result to the stat_block function shortly, we could even continue like so:

```
my_die.translate(offset)
                        .scale_by(scaling_factor)
                                                 .print();
```

All spacing here is excessive and for elucidation only! (No one codes like that.)

As noted before, anonymous objects are **not** protected against such non-sense as this:

```
my_die . scale_by ( scaling_factor ) . read();
```

Here we accidentally called read instead of print ... easy enough to do. But luckily this is protected against by our const return type from scale_by. Unfortunately, it is not normal practice to put const on return types. So this won't help you when dealing with third-party code.[23]

So please try not to do such strange and idiotic things, okay? And try to always protect returned objects with const, too...

You think it'll be easy to avoid, eh? Look at this 'code':

```
Die().read();
```

That compiles cleanly and runs 'just fine'. It is totally worthless. After all, it constructs an anonymous object as an unscaled, untranslated 6-sider and then pauses the program to make the user type a set of die parameters — prompt-less — and stores that data in the same anonymous object and finally throws away not only the bool result of the read function but also the anonymous object itself!

And worst of all: there's no way to protect us from this one!

At least we should notice the prompt-less input pause. But debugging it is quite difficult as it is so out of place!

[23] Third-party here refers to code from another source like an open-source project you found on the web.

5.4.2 Composition

As mentioned above, composition is the process of having one class have one or more members of other class types such that the one class is composed of members of another class. The class composed of the others tends to rely on the original classes in many ways — calling many of its methods, for instance.

Let's look at a simple class that uses composition and see what it might entail:

```
class DiceGroup
{
    Die base_die;  // use library supplied class to compose
                   // new class
    long count, adjust;
    /*
     * In future versions, we might want to let the user
     * label their group of dice...
     */
    // string label;

public:
    // constructors
    DiceGroup(const Die & new_base, long new_count = 1, long new_adjust = 0)
        : DiceGroup{}
        { set_base(new_base); set_count(new_count); set_adjust(new_adjust); }
    DiceGroup(const DiceGroup & dg)
        : base_die{dg.base_die}, count{dg.count}, adjust{dg.adjust}
        { }
    DiceGroup(void)
        : base_die{}, count{0}, adjust{0}
        { }

    // input/output
        // NOTE:  our count and adjustment are optional, but
        //        the scale and adjustment of the base die
        //        are required!
    bool read(void)
    {
        bool read_okay;
        Die d;
        long c = 1, a = 0;
        char t;
        cin >> ws;
        if ( isdigit(cin.peek()) )
        {
            cin >> c;
        }
        read_okay = !cin.fail() &&
                    d.read();
        if ( peek_ahead() != '\n' )
        {
            cin >> t >> a;
            if ( t == '-' )
            {
                a = -a;
            }
            read_okay = read_okay && !cin.fail();
        }
        return read_okay &&
               set_count(c) && set_base(d) && set_adjust(a);
    }

    void print(void) const
    {
        if ( count != 1 )
        {
            cout << count;
        }
        base_die.print();
        if ( adjust != 0 )
        {
            if ( adjust > 0 )
            {
                cout << '+';
            }
            cout << adjust;
        }
        return;
    }

    // accessors
    const Die get_base(void) const { return base_die; }

    long get_count(void) const { return count; }

    long get_adjust(void) const { return adjust; }

    // meta- or pseudo- accessors -- accessing information derived from/about
    // our class' core data
    double get_max(void) const
    {
        return static_cast<double>(count) * base_die.max() +
               static_cast<double>(adjust);
    }

    double get_avg(void) const
    {
        return (get_max() + get_min()) / 2;
    }

    double get_min(void) const
    {
        return static_cast<double>(count) * base_die.min() +
               static_cast<double>(adjust);
    }

    // mutators
    bool set_base(const Die & new_base)
    {
        base_die = new_base;
        return true;
    }

    bool set_count(long new_count)
    {
        bool okay = false;
        if ( new_count >= 0 )
        {
            count = new_count;
            okay = true;
        }
        return okay;
    }

    bool set_adjust(long new_adjust)
    {
        adjust = new_adjust;
        return true;
    }

    // helpers, behaviors, etc.
    double roll(void) const
    {
        double sum = static_cast<double>(adjust);
        for ( long d{1}; d <= count; ++d )
        {
            sum += base_die.roll();
        }
        return sum;
    }

    const DiceGroup merge(const DiceGroup & dg) const
        { return base_die.isSame(dg.base_die)
               ? DiceGroup{base_die, count + dg.count, adjust + dg.adjust}
               : DiceGroup{}; }
};
```

That's a lot of code (not quite 150 lines), but it shows several aspects of composition and even class development in general that we might like to study.

5.4.2.1 Method Order in a class Definition

Notice that the constructors not only delegate to one another, but rely on the setters/mutators. However, not only do the constructors get defined before the mutators, but they are even defined to delegate to ones amongst themselves that are defined later in order!

This is normally a no-no, right? Calling a function before even seeing its prototype is verboten and just doesn't work. But when the compiler is working on a class definition, it assumes the called function will be a member function either prototyped or defined before the class definition is over. If that is not the case, then the compiler will raise one holy fit!

And this rearrangement allowance is only present when compiling a class definition. At no other time will the compiler just wait for a prototype or definition to appear before the end of a file or the like.

5.4.2.2 Reliance on the Composed class

Note also that many functions depend on those from the composed class. read and print, for instance, depend on those methods from the Die class. Likewise, the meta-accessors[24] get_min and get_max rely on their similar functions from the Die class. And even the setter and getter for the composed object rely on the Die class constructors and mutators to have kept the object in proper working order its entire life.

5.4.2.3 Error Handling in Arithmetic Functions

In the merge function, we see how we can handle error conditions during an arithmetic function. Merging, it would seem, is only handled by this class when the base-dice of the two groups are the same kind. If they differ, a default group is created which — looking back to the constructors — has 0 dice in it. It is left to the caller to check the count of the returned object to see if the operation succeeded.

5.4.2.4 Increased Opportunities for Method Chaining

We also note that there are Die and DiceGroup objects returned from some member functions. These afford the programmer using the DiceGroup class more opportunities for method chaining. They can use a Die method chained off of the DiceGroup's get_base function result, for instance.

5.4.2.4.1 To Cache or to Chain

Let's look at a consequence of the above two notes about composition/composed classes. In the driver for the DiceGroup class, we find this code:

```
cout << "\nNow enter a compatible set of dice for merging:  ";
while ( ! two.read() || one.merge(two).get_count() == 0 )
{
    cin.clear();
    cin.ignore(numeric_limits<streamsize>::max(), '\n');
    cerr << "\n\aInvalid set of dice!  Please try again!\n";
    cout << "\nNow enter a compatible set of dice for merging:  ";
}

// ...

cout << "\nWhen merged together, they'll look like this:\n";
stat_block(one.merge(two));
```

Here, the DiceGroup one has already been input and we need a second DiceGroup suitable for use in a merge operation. To make sure it works out, we put the error condition check in the head of the input loop. However, later, we re-merge the two groups to display some statistics about their combination. Although this shows chaining at play, it bothers some that we aren't caching the merge result and are calling for its calculation twice instead.

To cache the result would require one of two possibilities. Here is one:

```
cout << "\nNow enter a compatible set of dice for merging:  ";
merged = DiceGroup{};
```

[24] We sometimes call a helper/calculation function a meta-accessor if it relates information about our data members that involves very few calculations.

```
if ( two.read() )
{
    merged = one.merge(two);
}
while ( merged.get_count()==0 )
{
    cin.clear();
    cin.ignore(numeric_limits<streamsize>::max(), '\n');
    cerr << "\n\aInvalid set of dice!  Please try again!\n";
    cout << "\nNow enter a compatible set of dice for merging:  ";
    if ( two.read() )
    {
        merged = one.merge(two);
    }
}

// ...

cout << "\nWhen merged together, they'll look like this:\n";
stat_block(merged);
```

This does the job, but is bulky and clunky all at once. It uses an error initialization of the merged caching variable to avoid having to store the read result in an extra bool for loop testing. It does factor out this initialization to keep from having an else on both if branches, at least!

Is this the only way? Well, there is another, but it is even worse:

```
cout << "\nNow enter a compatible set of dice for merging:  ";
while ( ! two.read() || (merged = one.merge(two)).get_count() == 0 )
{
    cin.clear();
    cin.ignore(numeric_limits<streamsize>::max(), '\n');
    cerr << "\n\aInvalid set of dice!  Please try again!\n";
    cout << "\nNow enter a compatible set of dice for merging:  ";
}

// ...

cout << "\nWhen merged together, they'll look like this:\n";
stat_block(merged);
```

Here we nest an assignment statement into the left side of a dot operation to chain a method. This is one of the ugliest constructs we've ever seen or will ever see. It makes reading the code a whole new level of crap-tastic! Please never nest assignments . . . anywhere!

Suffice it to say, the uncached merge result is probably the lesser of evils here.

5.4.2.5 The Whole Picture

This program and its supporting libraries come to more lines of code than we want to repeat here (almost 500!), so I'm placing it on a website for you. Here is phase four of our Die class exploration.

Note how we put the Die class in a library for easy reuse! As the definition of a new type, its definition goes into the header or interface file. Any non-inline function go in the implementation file as usual.

5.4.3 Comparison with Structures

There is another way to aggregate multiple variables together that describe real-world data. This is called a structure and differs slightly from a class. It came to us directly from our ancestor language C and now has the format:

```
struct Die
{
    long sides;
    double scale, adjustment;
};
```

What's different about a struct is that it is by default public access. That totally breaks the idea of hiding an ADT's data and protecting it from dangerous changes. This was all left to the programmer's best judgement in C. Thank goodness Bjarne gave us the class with private access in C++!

Some sources will tell you that a struct cannot have methods or constructors, but this is just not so. A struct can have anything a class can have — even private members. It is just that the class is preferred for OO development and we leave the struct for a handful of support tasks. We'll see one of these later in section 6.7. In these support roles, the struct will keep its public nature but may still have constructors or other support methods.

5.5 Wrap Up

In this chapter we've learned just about all there is to know about basic object-oriented programming with the C++ class mechanism. (There are advanced concepts we've left out, but let's talk about them another semester.)

We covered data hiding with private and public access specifiers, the use of mutators and accessors to allowed controlled viewing and changing of private data members, and the use of constructors to fill in private data members correctly from the object's very beginning.

We then talked about making classes more efficient and robust with inlining, member initialization lists or in-class initialization, and const-correctness. And, finally, we talked about basic composition and method chaining.

We even took a moment to acknowledge the classic C struct and see its use in a C++ light.

Chapter 6

Containers

Sometimes it is useful to collect together multiple values before processing them all at once. This will help us attack certain new problems we'd have once found insurmountable as well as helping us design old programs in new and better ways.

One simple example is that of collecting multiple values about which we want statistical information: mean, median, etc. Some of that information we could collect as the data was entered — mean, minimum, maximum, for instance. We could just have a single variable to hold each piece of data entered and overwrite it as we looped around:

```
cin >> data;
while ( /* there's more data */ )
{
    // update statistics
    cin >> data;
}
```

But other things like the median (or any of the quartiles, percentiles, etc.) require us to have all the data available and examinable at once rather than just one piece of information at a time like we'd have done in chapter 3. (After all, the median is the middle element of a set of data. This requires us to not only see all the data at once rather than one piece at a time, but also to place the data into order somehow so that we know what the middle is!)

We've done such gathering of lots of data before with words/phrases — gathering sequences of `char`s — thanks to the `string` `class`. But if we wanted to hold multiple `double`'s, `short`'s, or other data, we were out of luck. We'd've had to have made multiple variables with stupid names like:

```
+=====+    +=====+    +=====+    +=====+    +=====+    +=====+
| 42  |    | 47  |    | 12  |    | 34  |    | 17  |    | 25  |
+=====+    +=====+    +=====+    +=====+    +=====+    +=====+
 age1       age2       age3       age4       age5       age6
```

This could easily become horrible to maintain and follow. We need something better. Some way to store multiple values in a single variable.

What's that? A `class` can store multiple variables? Well, yes, but it still requires us to name them all individually. We'd be right back to where we started.

Plus, the `class` imposes security mechanisms: `private` areas, accessors, and mutators. That would stink even worse for our purposes here!

What we need is a general container — something that can contain other pieces of data and allow us to get it back at will. Something we can name just the container and refer to the individual values by an index like we did `char`s in a `string` object.

6.1 A Tale of Two Containers

Luckily, C++ has just the thing. In fact, it has two of them tailored for slightly different use cases. The `array` `class` is used when we have an exact number of items to contain. Those items can be of any data type — except `void`.[1] This can be useful when we want to talk about things relating to days of the week, months of the year, letters in the alphabet, or even cards in a deck.

The other container C++ offers is tailored for those situations where we aren't sure ahead of time how many elements we'll need to store. Sometimes the user will introduce new data continually as the program runs, in fact. For these situations, we use the `vector` `class`. Whereas the `array` was a fixed size, the `vector` can grow to suit the user's demand for more memory.

6.2 Basics of Containers

We'll break this down by container type: fixed-size (`array`s) and growable (`vector`s). In each case we'll talk about declaring a container of that type, initializing it if need be, and using the elements in the container. And, for `vector`s, we'll also particularly look at reading the elements from the user. (That's not as tricky for `array`s since they are of a size fixed at compile time.)

The two sections may appear quite similar at first glance, but they differ in critical ways. Be sure to read them both carefully to learn the differences between how these two containers behave.

6.2.1 arrays

First, I'd like to point out an issue of nomenclature. There was a construct in our ancestor language — C — that is often termed a C-style array. This is not what our discussion is about here. These constructions use square brackets (`[]`) in their initial declaration as well as in their later use. In this book we'll be talking about a `class` also known as `array` that was added to the C++ standard in C++11. An object of the `array` `class` is different from the C-style array construct. That construct is bare-bones and has no helper functions or extra functionality. The `array` `class` adds many useful tools to this basic form and should be used in **all** modern designs!

Now on with the discussion of the `array` `class` and its usage.

[1] But who wants a bunch of nothing, anyway, right?

Declaring an array object first requires that we #include the array library. Then we can make one like so:

```
array<string, 7> lunches;
```

The angle-brackets are used to denote both the type of elements in the array (the base type) and the size of the array (how many elements does it hold). All of this together makes up the actual data type of the array. Then we name the variable — here we've called it lunches.

Here we've made an array to store the lunch foods for a user for a week. While this is somewhat obvious, we usually prefer using a constant for the size of the array:

```
constexpr size_t DAYS_IN_WEEK = 7;

array<string, DAYS_IN_WEEK> lunches;
```

What's that new data type: size_t? That is the type alias that the array class uses to describe the number of elements in itself. Unlike the size_type for the string class, this one is global instead of class-scoped.[2] The full reasoning for this might be covered in a subsequent semester. For now, we'll just let-it-go...

Also note that it is, again like string::size_type, an unsigned integer type of some sort. That can be important when it comes to calculation wrap-around, remember?

We can either initialize the array's contents right away or we can read it in from the user. If we initialize it, we use a curly-brace enclosed, comma separated list of values preceded by an equal sign:

```
constexpr size_t DAYS_IN_WEEK = 7;

array<string, DAYS_IN_WEEK> lunches = { "pizza", "pot stickers", "tacos",
                                        "pasta", "soup", "sandwich",
                                        "BBQ" };
```

Here our values are of the string type and so had to be in double quotes. Numeric types or bool would need no quotes, of course. And if the base type were char, they would need single quotes!

If you give too many initializing values, the compiler complains with a full-blown error. If you give too few, the compiler silently fills in the remaining slots with default values for the base type.

Truth-be-told, the equal sign isn't always needed. But there are times it becomes necessary and so we often err on the side of consistency in programming.

To use the elements from the array, we can, as with the string class, use either the subscript operator ([]) or the at method:

```
cout << lunches[3] << " & " << lunches.at(4) << ".\n";
```

As before, the [] operator will get the data or die. The at method, on the other hand, gets the data or kills your program with an exception if you fail to catch it! Not much difference, really, but you can protect yourself from the [] crash with simple bounds checks and you can protect yourself from the at's exception by trying to catch it.

Of course, we'd typically use a loop to go through the elements. This can be a regular for loop or a range-based for loop. Typically we want all the elements treated the same here, so I'll show a range-based for:

[2] It isn't even in a namespace like std!

```
for ( const string & lunch : lunches )
{
    cout << '\t' << lunch << '\n';
}
```

Here each lunch is tabbed over once on a separate line. Note the use of a const& on the index's type to make sure we don't accidentally change the elements as we go by. (The reference also makes it so we don't make a copy each time. Much more efficient.)

To do something different with each thing — or maybe treating the last one slightly differently than the rest — we would have to use a regular for loop:

```
for ( size_t i{0}; i + 1 < lunches.size(); ++i )
{
    cout << lunches[i] << ", ";
}
cout << "and " << lunches.back() << '\n';
```

Here we print a comma-separated list of the lunches.[3] The +1 on the condition makes it so that we stop when the next element is the last rather than the current element. Why not use a -1 on the size side? That's a story for later (section 6.2.2) when we discuss vectors. For now just consider it a matter of consistency.

It should come as no surprise that the array class has a size method. What may be surprising is that it doesn't also have a length method! The committee felt that the length idea was only good on strings for some reason. *shrug*

Finally we use the back method to grab the last element. This is much more effective than subscripting by the size minus one![4] There is also a front method, but accessing position 0 with [] or at is just as easy if not simpler so most rarely use it.

What if we want to read the meal names from the user? We can, again, use either for loop style. To read all the names, use a range-based for:

```
for ( string & lunch : lunches )
{
    getline(cin, lunch);
}
```

Here we use getline so the user's lunch name can have spaces inside. Make sure your prompt tells them to enter one lunch name per line! Also note the use of a pure reference on the index's type so we can change the element.

What about a standard for loop for input? That's not usually necessary with an array. But, if you wanted to, it could be done like so:

```
for ( size_t i{0}; i < lunches.size(); ++i )
{
    getline(cin, lunches[i]);
}
```

shrug Seems like more typing than it's worth...

[3] The Oxford comma lives!

[4] The string class has a back method, too. We just never had occasion to use it because the string is an odd container in that its elements are rarely separated like this. It is a whole thing rather than a collection of individuals — most of the time.

6.2.2 vectors

Declaring an vector first requires that we #include the vector library. Then we can make one like so:

```
vector<string> lunches;
```

The angle-brackets are used to denote both the type of elements in the vector (the base type). All of this together makes up the actual data type of the vector. Then we name the variable — here we've called it lunches.

Thus we've made an vector to store the lunch foods for a user for a week.

We can either initialize the vector's contents right away or we can read it in from the user. If we initialize it, we use a curly-brace enclosed, comma separated list of values preceded by an equal sign:

```
vector<string> lunches = { "pizza", "pot stickers", "tacos", "pasta",
                           "soup", "sandwich", "BBQ" };
```

Here our values are of the string type and so had to be in double quotes. Numeric types or bool would need no quotes, of course. And if the base type were char, they would need single quotes!

The number of initializers sets the size of the vector but it need not be permanent, as we'll soon see.

Truth-be-told, the equal sign isn't always needed. But there are times it becomes necessary and so we often err on the side of consistency in programming.

To use the elements from the vector, we can, as with the string class, use either the subscript operator ([]) or the at method:

```
cout << lunches[3] << " & " << lunches.at(4) << ".\n";
```

As before, the [] operator will get the data or die. The at method, on the other hand, gets the data or kills your program with an exception if you fail to catch it! Not much difference, really, but you can protect yourself from the [] crash with simple bounds checks and you can protect yourself from the at's exception by trying to catch it.

Of course, we'd typically use a loop to go through the elements. This can be a regular for loop or a range-based for loop. Typically we want all the elements treated the same here, so I'll show a range-based for:

```
for ( const string & lunch : lunches )
{
    cout << '\t' << lunch << '\n';
}
```

Here each lunch is tabbed over once on a separate line. Note the use of a const& on the index's type to make sure we don't accidentally change the elements as we go by. (The reference also makes it so we don't make a copy each time. Much more efficient.)

To do something different with each thing — or maybe treating the last one slightly differently than the rest — we would have to use a regular for loop:

```
for ( vector<string>::size_type i{0}; i + 1 < lunches.size(); ++i )
{
    cout << lunches[i] << ", ";
}
cout << "and " << lunches.back() << '\n';
```

Here we print a comma-separated list of the lunches.[5] The +1 on the condition makes it so that we stop when the next element is the last rather than the current element. Why not use a -1 on the size side? Just a moment, let's mention the size method itself, first.

It should come as no surprise that the vector class has a size method. What may be surprising is that it doesn't also have a length method! The committee felt that the length idea was only good on strings for some reason. *shrug*

So, about the +1 vs -1 for stopping one short of the end. Many would feel more comfortable if I'd placed the -1 on the right side like this:

```
i != lunches.size() - 1
```

But here we introduce the possibility of a negative value when the size reports 0. And negatives can't happen since we are in an unsigned type. So that's dangerous. (Remember the circle of doom?) So we don't do that. Instead, we put the +1 on the left side to stop ourselves with only positive values possible rather than 'negatives' creeping in.

One other thing, what about that size_type? That's just like the one from the string class except this one is chosen just for a vector of this base type. There is a footnote in the standard that implies strongly that all size_types of various containers should just be size_t, but it is not a guarantee. I always use the size_type of the given container to make sure it matches the system's needs/abilities.

Finally we use the back method to grab the last element. This is much more effective than subscripting by the size minus one! There is also a front method, but accessing position 0 with [] or at is just as easy if not simpler so most rarely use it.

One last thing, though, since the vector may not have anything stored in it yet, we should make sure the back element exists before using it like that. If we don't, it is considered undefined behavior. This might not seem that bad at first glance, but programmers hate it when something the computer does is not defined very explicitly in a standard way!

To protect ourselves, we can use the empty method like so:

```
if ( ! lunches.empty() )
{
    for ( vector<string>::size_type i{0}; i + 1 < lunches.size(); ++i )
    {
        cout << lunches[i] << ", ";
    }
    cout << "and " << lunches.back() << '\n';
}
```

This function returns true when the vector has nothing stored in it and false when there are any elements at all — even just a single one. Why not just put it around the back's cout? We might as well protect the for loop bound as well. This way if someone on the team changes our +1 to a -1, we'll still be safe.

What if we want to read the meal names from the user? Unless we are sure of the size of the vector, we shouldn't be reading into it directly with [] or at. If we are sure there are spaces, we could use either for loop style. To read all the names, use a range-based for:

[5] The Oxford comma lives!

```
for ( string & lunch : lunches )
{
    getline(cin, lunch);
}
```

Here we use `getline` so the user's lunch name can have spaces inside. Make sure your prompt tells them to enter one lunch name per line! Also note the use of a pure reference on the index's type so we can change the element.

What about a standard `for` loop for input? That's not usually necessary with an `vector`. But, if you wanted to, it could be done like so:

```
for ( vector<string>::size_type i{0}; i < lunches.size(); ++i )
{
    getline(cin, lunches[i]);
}
```

shrug Seems like more typing than it's worth. . .

But what if we are unsure about the current number of slots in the `vector`? Can `empty` help us again? It could, but it is easier to take a different approach. We'll just call `clear` before we begin our input loop. This method erases all elements from a `vector`.

Then how do we put things in an `empty` `vector`? We have to *push* them to the *back* of the `vector`!

```
string temp;
lunches.clear();
cin >> temp;
while ( /* there're more elements */ )
{
    lunches.push_back(temp);
    cin >> temp;
}
```

We just need to decide how to stop this loop. We can't use `failure` since `string`s never `fail` on an input stream. Let's use a flag value. How about the ever-popular `"quit"`? It's simple and never a lunch name. Sounds like a plan:

```
string temp;
lunches.clear();
cin >> temp;
while ( temp != "quit" )
{
    lunches.push_back(temp);
    cin >> temp;
}
```

What about running out of memory? Not a problem. If that happens, the whole computer will shut down. So we are safe as long as the user isn't doing a primitive, manual form of denial-of-service attack (DoS) on our program. But then, that could have happened with a single `string` input, too. We just didn't bother to mention it at the time.

As implied above, the `push_back` method makes the `vector` grow as needed to store the user's data. This is the typical mode of operations for a `vector`. If you are starting out initializing it to a set number of values, it should have probably been an array. Normally a vector is declared empty and then we push_back data onto it when needed.

6.3 Standard Methods & Helpers

Let's talk about some of the most useful methods and helper functions these two containers give us to work with. We'll also make sure you know what methods to avoid for each class.

6.3.1 arrays

For the array class, there are a few useful functions we might use from time to time:

Function	Notes
assignment	use the = operator to change a whole array's contents to be just like another
[]/at	access the specified element
front	access the first element
back	access the last element
size	tell the declared number of elements
get	like with tuple and pair, access element positions
fill	copies the value specified to all element slots
comparisons	use operators like == and > to compare whole arrays at once

6.3.1.1 Not So Useful

On an array, the empty function always returns false unless the declared size is 0, but who would do that?

There is also a method named max_size which purports to produce the largest number of elements an array of this base type can hold. It is theoretical at best. It always returns the maximum of the size_t data type that is used to report the size. This is of limited use as we can just use numeric_limits to get this information, too.

6.3.2 vectors

For the vector class, there are a few useful functions we might use from time to time:

Function	Notes
constructors	default, copy, and at least the initialization list version we used above
assignment	use the = operator to change a whole vector's contents to be just like another
[]/at	access the specified element
front	access the first element
back	access the last element
size	tell the declared number of elements
empty	this method is now useful and tells us if we contain any elements at all or not
push_back	copy a new element to the end of the vector

Continued on next page

(Continued)

Function	Notes
emplace_back	construct a new element into a new slot at the end of the vector; more efficient than push_back for contained class objects; see section 6.5 below
pop_back	remove the last element
clear	erase all elements
resize	change the size to something new; new elements are defaulted; if smaller, we've lost elements
capacity	interesting if not useful; reports the currently available storage of the vector which might be greater than size — the used portion of the vector's space
comparisons	use operators like == and > to compare whole vectors at once

In addition to the above noted constructors, there are two others:

```
vector<string> test(5);             // creates 5 default strings
vector<string> test(5, "Hello");    // creates 5 copies of "Hello"
```

The first is of limited use, but the second can be used sometimes in the guise of the array's fill method.

You could even encapsulate it into a helper function if you didn't know the size of your vector right away:

```
template <typename ElemT>
inline vector<ElemT> make_vec(typename vector<ElemT>::size_type size,
                              const ElemT & value = ElemT{})
{
    return vector<ElemT>(size, value);
}
```

Here we use the template mechanism to make this function more generally reusable. This entails two new usage elements, however. One is the use of typename not only in the template head to introduce the name of the fill-in-the-blank type for the template, but also to explain to the compiler that the size_type for the vector in question is really the name of a type. The compiler gets confused in this situation when the base type of the vector isn't an explicit type, you see.

The second usage of note is the defaulting of the value parameter to make it so you don't have to pass it when you don't want to. This uses the template typename in an anonymous construction pattern to make the right kind of default value for that type no matter what it ends up being.

Finally, just to finish our discussion, we anonymously construct the vector to return thus invoking RVO to make the function run more efficiently.

In the vein of these two extra constructors, there is another resize that takes an element to copy to any newly created slots instead of defaulting them.

As for capacity, it can be understood like this. Let's take the lunches vector from above and say they'd filled in 6 slots with meals:

```
[]:    0     1     2     3     4     5     ?     ?
    +================================================+
    | xxx | yyy | zzz | www | aaa | bbb |     |     |
    +================================================+
 .size() == 6
 .capacity() == 8
```

There's room for two more push_back's before we'll have to grow again. This is optimistic in that we hope not to have to talk to the OS any more than necessary. Each time we talk to the OS, after all, it makes us wait for all the other processes on the system to take their CPU turn before we regain control and can run some more! There can be hundreds of them! That could take milliseconds![6]

At the third push_back the vector would grow the capacity to 16 while only adding a single element to the size. Again, this holds off the inevitable OS discussion and our imminent delay.

6.3.2.1 Not So Useful

There is also a method named max_size which purports to produce the largest number of elements an vector of this base type can hold. It is theoretical at best. It always returns the maximum of the size_type data type that is used to report the size. This is of limited use as we can just use numeric_limits to get this information, too.

6.3.3 An Example Application

This example program came in at almost 200 lines of code, so I'm placing it on a website for you. Here is the heights averager program. Now, let's walk through it by line numbers.[7]

How? Well, you don't get line numbers in a web browser, but you can — almost always do, in fact — in a programmer's editor. So download the file and load it up in your favorite editor for programming. I'm using Xcode™ at the moment. But I wrote the original codes and this book in Vim. So any number of environments would work: Visual Studio™, VS Code™, Atom™, CodeBlocks™, CLion™, etc. Now just put that side-by-side with this book and follow along.

The first six lines are boilerplate at this point so we'll not discuss them further.

Then we start prototyping functions. On lines 8-13 we see the total function takes its vector argument by const& because it need not change it and we don't want to make a copy as usual with class objects. Its task is to add up the elements in the vector and return the sum.

After a long comment (15-30), we have a pair of helper constants for a bool argument to the input_all_nums function (31-35). The vector here is passed by pure reference to allow changes to its contents — we are inputting data into it, after all. There is also a const& string for prompting the user if desired. The alternative would be for the caller to prompt before the call. And finally the bool parameter which controls whether the input data are to be appended to the current vector contents or those old data are to be erased and new data stored. Hence the constants ADD_NEW_NUMS and ERASE_OLD_NUMS to go along with this argument.

Finally the display function (37-46) takes its const& vector to print on screen and also three string arguments — also by const&.[8] These strings are to be printed before the vector contents, after the vector contents, and between the vector elements respectively. They default to standard set notation from math.

[6] AKA an eternity.

[7] Remember that to see your own line numbers, download the file and load it into your favorite IDE/editor. Then place that side-by-side with this text so you have a nice flow.

[8] Remember that anything larger than a built-in type should probably be passed as const& for use and plain reference for changing. The full rules are in section 4.4.1.3.1.

Next (line 48) we start the main program. Here we declare a vector named heights to store all the heights the user wants averaged. We welcome the user. Then we call for the entry of the user's heights with a prompt and let the append argument default to erasing the original contents (of which we have none).

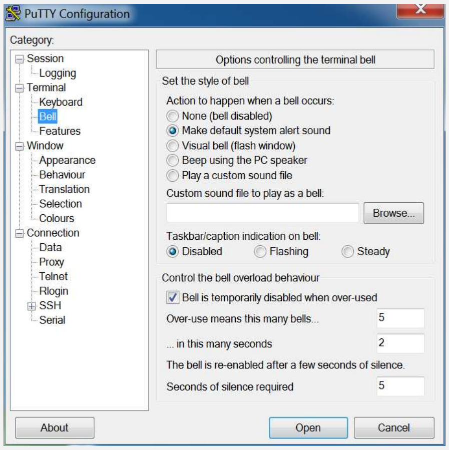
Upon return from that function, we print the number of heights read. We take a little trouble to agree the plurality of the noun "value" to the size of the vector. (Remember that 0 is considered plural in English.)

If the heights vector isn't empty (line 63), we display its contents with the default strings to augment the look of it. There are some commented-out alternative strings that I've used in the past to good effect. The first is the most obnoxious. It prints nothing before or after the numbers but prints not only a space between the numbers but also beeps each time! If there are many numbers this can give a person a headache. Be careful with such constructs in your UI (User Interface).

The next one is a notation I borrowed from the field of quantum mechanics. I can't recall what it is used for, but it was neat so I tried it. Looks nice.

Terminal Bells

If you are using a Unix terminal like Putty™, you might be able to turn off the bells from a '\a' or slow them down or the like. This is done in the Terminal subpanel called Bell. You can see in the picture that there are many options for the bells including changing its sound, turning it into a reverse-video flash (for the hearing impaired), temporarily disabling it if it is overused — like here, and flashing the taskbar instead of the whole window.



Options controlling the terminal bell

Set the style of bell

Action to happen when a bell occurs:

○ None (bell disabled)
● Make default system alert sound
○ Visual bell (flash window)
○ Beep using the PC speaker
○ Play a custom sound file

Custom sound file to play as a bell:

Taskbar/caption indication on bell:

● Disabled ○ Flashing ○ Steady

Control the bell overload behaviour

☑ Bell is temporarily disabled when over-used

Over-use means this many bells... 5

... in this many seconds 2

The bell is re-enabled after a few seconds of silence.

Seconds of silence required 5

The third one I'm particularly proud of. It puts the curly braces around the numbers as usual but prints each one of the numbers on a separate line indented one tab-stop from the curly braces. It looks really nice!

The last one is an homage to Porky Pig™ of classic cartoon fame.

Finally, on lines 73 and 74, we print the average of the user's heights by dividing the total by the size of the vector.

In the else (76-80) we print a message for a empty vector. Then we say goodbye and thank them for their time, essentially.

But that's not all, of course! We haven't defined the vector processing functions yet. They start on line 88 with the total function. Here we use a range-based for loop to add up all the elements of the vector onto a running sum that started at 0. When we are done, we return the sum to the caller.

On line 103 we start the display function. It prints its pre text first and then, if the vector isn't empty, it prints all but the last element (remember the +1 trick!) followed up by the between text. After the for loop, we print the back element from the vector. We end on line 121 by always printing the post text.

Finally, the input_all_nums function starts on line 125. We need a temporary double to read in the individual doubles from cin before we push them onto the back of the vector. If we knew how many elements the user had ahead of time, we could have resized the vector and just read the numbers directly into a subscripted position of the vector. But that isn't a normal circumstance. The user almost never knows how many values they have — they just have a sheet of numbers gathered from the field observation location. Sometimes these days that would be done on a tablet, but they aren't necessarily from a spreadsheet and so aren't likely counted in any way.

On 146 we check to see if the caller wanted us to append new numbers to their vector's current contents or to erase the old numbers from the vector and start fresh with these new entries. if they didn't want to append the new values, we clear out the old numbers first.

Then we print the prompt, if any, and try to read the first number (150-1). As long as this doesn't fail, we push the new number onto the back of the vector and try to read another number.

After the input loop (157-8) we clear cin's failure and ignore any non-numeric stuff left in the buffer.

Now that you understand how the program currently works, I encourage you to play with it and see how it might be changed. This is what I've hoped you've been doing all along with the other codes, but still, it doesn't hurt to state it more explicitly from time to time.

Also, don't forget to be writing small try-it-out programs to see how individual features work so you can get a feel for them in isolation before melding them with other features where they could interact and make things more difficult.

6.4 List Management

Many people want us to manage their lists of data for them. These could be lists of songs they like, contacts they want to remember, ingredients or steps for a recipe, groceries they need to buy, etc. We need to learn, therefore, to store lists of data in a vector and do basic list operations on said vector.

6.4.1 List Operations

List operations include displaying the list elements, inserting new elements into the list, removing old elements no longer needed or desired, searching for an element, and sorting the elements.

6.4.1.1 Display

When displaying the elements of the list, we have a few questions to be answered first:

- does the user want all items displayed?
- if not all, do they want a sub-range or subset of the elements?
- should the display be numbered?
- if numbered and not all elements, should the numbers be absolute or relative?

Since we are thinking the user might want a sub-range[9] printed, we make that function our workhorse:

```
const bool DISPLAY_NUMBERED = true,
           DISPLAY_PLAIN    = false;
void display(const vector<double> & vec,
             vector<double>::size_type begin,
             vector<double>::size_type end_before,
             bool numbered = DISPLAY_PLAIN);
```

[9] A range here is a contiguous run of positions and by sub-range we mean it might not start at the beginning or end at the last element.

In just the prototype we already see two things. One is about the numbering. We make that an integral part of the display from the ground up. The second is about the range boundaries themselves. The beginning is to be inclusive and the ending is to be exclusive. This is odd at first glance, but makes counting the number of entries a little faster to both code and execute. After all, if the bounds were doubly inclusive, we'd have to subtract and add one as we did with our random number generation (section 2.6.3). This way we merely subtract like we did with sub-strings (section 3.8.2.11).

But what if they do want to display the whole list? We make an inline overloaded helper like so:

```
    inline
void display(const vector<double> & vec,
             bool numbered = DISPLAY_PLAIN)
{
    display(vec, 0, vec.size(), numbered);
    return;
}
```

Here we call the sub-range function with a full range of indices and pass along the vector and whether the caller wants it numbered or not as well.

So what does the workhorse look like? It can look a lot like our previous attempts to display a vector:

```
void display(const vector<double> & vec,
             vector<double>::size_type begin,
             vector<double>::size_type end_before,
             bool numbered)
{
    end_before = end_before > vec.size() ? vec.size()
                                         : end_before;
    for ( vector<double>::size_type pos{begin};
          pos < end_before;                         // !=
          ++pos )
    {
        if ( numbered )
        {
            cout << pos + 1 << ":  ";
        }
        cout << vec[pos] << '\n';
    }
    return;
}
```

Here we sanity-check the end_before value to make sure it is inside the vector's indices. Then, to be extra careful, we use a strict less-than instead of a not-equal-to test on that upper bound.

The only real change from before is the if to check that the user did/didn't want the display numbered. This prints the position variable *plus one*. This +1 is important because your typical user doesn't want to see a list that's numbered starting at 0. That'll freak them out!

This display function does have a detriment compared to our earlier approach: it forcibly prints the values one-per-line. But adding the three string parameters is left as an exercise to the user. *grin*

Wait! What about that subset thing? We said sub-range or subset. What's the difference and how would we do the subset printing?

A subset is not necessarily contiguous. The values can be spread far and wide amongst those in the vector. Instead of being at positions from 5 to 9, they might be at positions 5, 7, and 8. No particular pattern to them — just these are the ones the user wants to see.

How would that happen? Let's say we had that vector of height information from before and the user wanted to see all the heights that were less than 6. We could gather those like so:

```
vector<vector<double>::size_type> disp_poses;

for ( vector<double>::size_type pos{0};
      pos != heights.size(); ++pos )
{
    if ( heights[pos] < 6 )
    {
        disp_poses.push_back(pos);
    }
}
```

This is a little tricky because the content of the disp_poses vector are size_types from another vector — our heights vector from the last sample program.

Once this is done, the positions within the heights vector are stored in the disp_poses (short for 'display positions') vector. Once known, we can use those to print the original data at those positions for the user:

```
void display(        const vector<double> & vec,
             const vector< vector<double>::size_type > & poses,
             bool numbered)
{
    for ( vector<
                  vector<double>::size_type
                >
                ::size_type
            pos{0}; pos != poses.size(); ++pos )
    {
        if ( numbered )
        {
            cout << poses[pos] + 1 << ":  ";
        }
        cout << vec[ poses[pos] ] << '\n';
    }
    return;
}
```

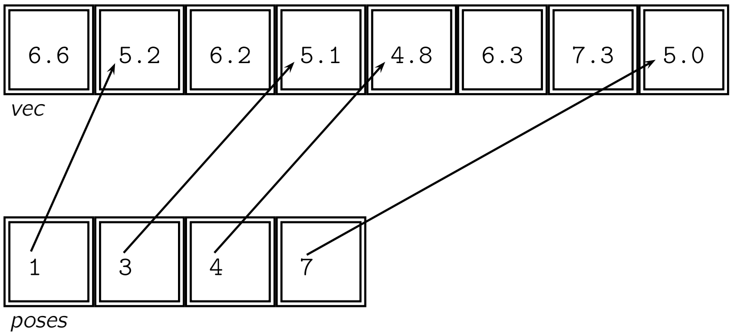
Here we have the vector of information (doubles) coming into the function as well as the vector of size_type positions within that vector. Both are const& to protect them and avoid copies. We also take in whether the caller wants this display numbered or not.

Pardon the spacing on the first argument and the loop head. I was trying to emphasize that the positions vector is positions in the data vector.

This code uses a standard for loop to walk through all the size_type values and use each one to display both the number and its position if numbered. This is the trickiest of the options of coding but it is the most prevalent in existing code since the range-based for loop didn't come along until C++11.

So to get the position of the height desired, we must subscript the poses vector. This, then, can be used to subscript the vector of data. This leads to a nested subscript operation:

```
cout << vec[ poses[pos] ] << '\n';
```



As usual, the inside of the expression is evaluated first and we get the value of the position into the data `vector`. Then this result is used as the subscript into the data `vector` to retrieve the data at the desired position.

As you can tell from the diagram at right, the indices stored in the `poses` `vector` are those of the entries in the data `vector` that are less than 6 just as we'd called for earlier. (Remember that `vector` subscripts start at 0 just like those of `string`s.)

Such use of one container to store positions from another container is called indirect access or indirect indexing. Quite the fancy term for something not so hard after all, eh? Just scary at first glance, but walking through it makes it okay!

But this double-subscripting wasn't technically necessary if we wanted to use a range-based `for` loop instead:

```
for ( vector<double>::size_type vpos : poses )
{
    if ( numbered )
    {
        cout << vpos + 1 << ":  ";
    }
    cout << vec[vpos] << '\n';
}
```

Here we pull the position values out of the `poses` `vector` via the range-based `for` loop and use those to display the numbering and data — still via subscripting.

There is another tool possible to use here, too. We saw the `auto` keyword used to automatically deduce the `return` types in a structured binding for receiving a `pair` or `tuple` from a function (section 4.6.2.1). Here we can use it instead of having to type out the whole base type of the `vector` the range-based `for` loop is running through. Since the base type in this situation is rather bulky, it is to our benefit to make it just `auto` instead.

Some people are against this usage and say you should always know and type out for clarity the full data type involved. But here the data type is just above in the function head if we want to know it. I say go for it!

The syntax of this use is just:

```
for ( auto vpos : poses )
```

Nothing else in the code need change and so our function becomes:

```
void display(        const vector<double> & vec,
             const vector< vector<double>::size_type > & poses,
             bool numbered)
{
    for ( auto vpos : poses )
    {
        if ( numbered )
        {
            cout << vpos + 1 << ":  ";
        }
        cout << vec[vpos] << '\n';
    }
    return;
}
```

Note that this still is using indirect indexing/access but it has been hidden by the range-based for loop.

6.4.1.2 Insert and Remove

We talk of insertion and removal together because they are almost inverse processes. We compare/contrast the two to see this at the end.

6.4.1.2.1 Insertion

When inserting a new element into the list, where should it go? (a) at the end of the list, (b) in the middle of the list somewhere, or (c) in the front of the list. If we are adding the element to the end of the list, the code is clear: push_back is our tool. But the other two situations bear more consideration and study.

It might seem odd at first that we would care where we put new data, but soon we'll talk about sorting the data — putting it into some kind of order. Then we'll need to move the data around to put them into the desired order. Or it might be as simple as a new step in a recipe to make it just the right way needs to go between two steps that were there already.

Let's try to insert the new data into the first slot in the vector first. This is an easy, known location. We'll need to do two things: make room and then move the data over to clear out the spot we want to use. We have to make room at the rear, you see, because the vector class doesn't grow at the front — only the back. And if we want the data in the front, we'll have to move every piece of old data over to make room.

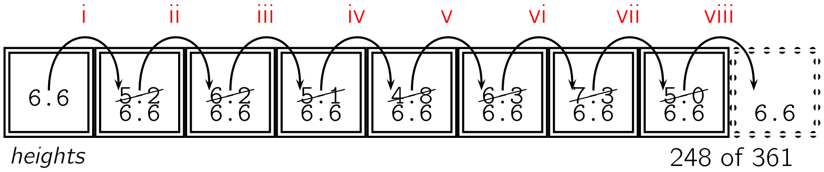
To start, let's resize the vector:

```
heights.resize(heights.size()+1);
```

This will give us the one more slot we need to put in this new value that's come along.

Then we'll move every old piece of data over. But we have to be very careful! If we move the data in the wrong order, we could lose almost all of it!

As you can see in the diagram at the right, if we start at the first element and move it over before any others, for instance, we'll overwrite the



second element. Then, moving forward, we would take that copy and overwrite the third element. And so on until all the data was overwritten except the first element which would be filling the whole vector. The dotted box is the newly-resized box just added to the vector.

So we definitely have to move the data forward but we also need to start at the end of the vector and work our way backward. This sounds harder than it is, so don't worry too much about that. We do have to be careful of two things, however.

The first thing isn't that bad: we can't move the data from the actual end of the vector forward. After all, we just resized it and the last slot is empty. Plus, there's nothing after it to move it to. So, that would be doubly bad.

The second issue is a little trickier. How do we stop after moving the first slot's data to the second slot? If we stop when the index reaches 0, we'd not move that first slot's data at all. If we try to go past 0, the unsigned nature of the size_type would make it turn uber-positive instead of just dropping to -1.

For example, this code:

```
for ( vector<double>::size_type i{2}; i >= 0; --i )
{
    cout << i << '\n';
}
```

if run on a 32-bit system would end up printing the results to the right.

```
2
1
0
4294967295
4294967294
. . .
```

And it would keep going forever! Luckily our code would be subscripting a vector inside the loop and it would crash pretty fast at those offsets/indices.

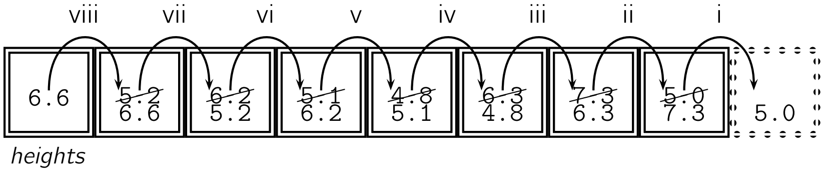
But we'd rather not crash, right? So what can we do? Well, we can stop when we reach that 4 billion-ish value by checking if i+1 is 0 or not. Yep, it's that +1 to the rescue again! It is the best friend of the unsigned integers.

So, the code we end up with is this:

```
for ( vector<double>::size_type i{heights.size() - 2}; i + 1 > 0; --i )
{
    heights[i + 1] = heights[i];
}
```

Note how we didn't start the moves at the last element (heights.size() - 1) but the next-to-last element (heights.size() - 2). Remember that this is because the last slot is still empty having just resized the vector.

But this will pull off the task shown in the diagram at right. (Note that the Roman numbers are no longer red like in the earlier diagram.)



Moving all the data over to make room for our final job, putting in the new value:

```
heights[0] = new_value;
```

I'm just pretending that new_value is the variable that holds this mythical value at this point in the code.

Next we turn our attention to insertion of a new value at some arbitrary position in between the first and last slots of the vector. We'll call this position the target. Here is the code to place a new value in the target position of the vector:

Alternative to +1

There is an alternative to the +1 in the loop. We could have walked through the destination slots instead of the source slots. This would have meant we could stop at 1 instead of 0 and not had that wrap-around problem after all.

```
heights.resize(heights.size() + 1);
for ( vector<double>::size_type i{heights.size() - 2}; i + 1 > target; --i )
{
    heights[i + 1] = heights[i];
}
heights[target] = new_value;
```

This looks suspiciously familiar and if we ran it side-by-side through a good difference checker (see section 4.5.1.2 for more on such software) we'd see that the only change is from 0 to target! Thus we can code both situations together in a single shot by just setting target to 0 when we want to insert at the front of the vector. Putting this with a helper if, we get the whole insertion package! Let's go ahead and change the name of the vector to something more generic and put it in a function while we're at it:

```
void insert(vector<double> & vec, vector<double>::size_type target,
            double new_value)
{
    if ( target < vec.size() )
    {
        vec.resize(vec.size() + 1);
        for ( vector<double>::size_type i{vec.size() - 2}; i + 1 > target; --i )
        {
            vec[i + 1] = vec[i];
        }
        vec[target] = new_value;
    }
    else
    {
        vec.push_back(new_value);
    }
    return;
}
```

And here's an overloaded inline helper for adding a new item just arbitrarily at the end of the vector:

```
inline void insert(vector<double> & vec, double new_value)
{
    insert(vec, vec.size(), new_value);
    return;
}
```

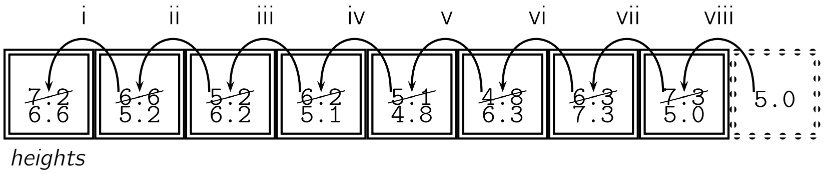
6.4.1.2.2 Removal

When removing an old element from the list, where was it at? (a) at the end of the list, (b) in the middle of the list somewhere, or (c) in the front of the list. If we are deleting the element from the end of the list, the code is clear: `pop_back` is our tool. But the other two situations bear more consideration and study.

Building on our previous work, we see that we will need to scoot all the elements after the removed one down toward the beginning of the `vector`. In fact, this moving will cause the removal of the target data in the first place! Just like moving the data over in the wrong order during insertion overwrote all those good data with a copy of the one from the front, moving the next piece of data that follows the removal target value down will overwrite it and it will be removed!

The only thing is, that would leave two of the data that followed the removal position and the `size` of the `vector` would remain unchanged. So we must not only continue to move data down but in the end either `resize` or `pop_back` to remove the current last element and shrink that `size`.

But let's be careful! What order should the data be moved down? After a modicum of thought, we realize our first instinct is right: move them from right to left and move right to get the next mover. This will look like the diagram at the right. (This time the dotted box is the soon-to-be-removed element.)



This code would look like this:

```
for ( vector<double>::size_type pos{1}; pos < heights.size(); ++pos )
{
    heights[pos - 1] = heights[pos];
}
heights.pop_back();
```

Again, building on our earlier experience with insertion of new data, we feel confident that this approach will work for other removal positions as well. Let's say the removal point is `target` as we did with the insertion point. Then the above code would change to:

```
for ( auto pos{target+1}; pos < heights.size(); ++pos )
{
    heights[pos - 1] = heights[pos];
}
heights.pop_back();
```

Not bad at all. But a little thought and a few experiences later, we'll come to realize that it is also easy enough to extend this algorithm to removing a sub-range of values from the `vector`. We just move the data following the deletion region down by the number of deleted spaces instead of just 1 space.

Let's call the upper bound on the removal region `keep_this`. Then the `count` of removed elements would just be `keep_this-target`. This is because `target` is to be removed and `keep_this` is a strict upper bound on the removal sub-range. (Again, no `+1` as we did with random numbers because the upper bound is not inclusive.)

So we'd get:

```
count = keep_this - target;
for ( auto pos{keep_this}; pos < heights.size(); ++pos )
{
    heights[pos - count] = heights[pos];
}
heights.resize(heights.size() - count);
```

We did have to change the pop_back to a resize after all since we were removing more than one item now. And the pos had its initial +1 removed because keep_this is exclusive. But otherwise, not much of a change!

However, maybe we should add some sanity checking and generalize it into a function for good measure:

```
bool remove(vector<double> & vec, vector<double>::size_type remove_this,
            vector<double>::size_type keep_this)
{
    vector<double>::size_type count;
    keep_this = keep_this > vec.size() ? vec.size() : keep_this;
    bool okay{keep_this >= remove_this};   // since keep_this is already
                                           // <= vec.size(), transitivity
                                           // will protect remove_this, too
    count = keep_this - remove_this;
    if ( okay && count > 0 )
    {
        for ( auto pos{keep_this}; pos < vec.size(); ++pos )
        {
            vec[pos - count] = vec[pos];
        }
        vec.resize(vec.size() - count);
    }
    return okay;
}
```

Thinking about it, this should work for all cases, shouldn't it? If the caller wanted to remove the last element, then remove_this would be vec.size()-1 and keep_this would be vec.size() to indicate one element to be removed. So pos would start as vec.size() — and the for loop would not execute its body. And we'd simply perform the resize! Cool...

But let's make this official with a couple of [overloaded] inline helpers:

```
inline bool remove(vector<double> & vec, vector<double>::size_type remove_this)
{
    return remove(vec, remove_this, remove_this + 1);
}

inline bool remove_last(vector<double> & vec)
{
    return remove(vec, vec.size() - 1);
}
```

I put overloaded in brackets because only one of these functions is an overload. The other is just a helper function that goes with the group. But, as I've said before, we don't shy away from helper functions when it nicely rounds out the capabilities of a family of functions.

6.4.1.2.3 Inverses

We said before that insertion and removal were almost inverses of one another. We should look at that before moving on. Let's look at the core of each algorithm[10] side-by-side:

```
void insert(vector<double> & vec,
            vector<double>::size_type target,
            double new_value)
{
    // ...
    vec.resize(vec.size()+1);
    for ( vector<double>::size_type i{vec.size() - 2};
          i + 1 > target; --i )
    {
        vec[i + 1] = vec[i];
    }
    vec[target] = new_value;
    // ...
}
```

```
bool remove(vector<double> & vec,
            vector<double>::size_type remove_this,
            vector<double>::size_type keep_this)
{
    // ...
    count = keep_this - remove_this;
    for ( auto pos{keep_this}; pos < vec.size();
          ++pos )
    {
        vec[pos - count] = vec[pos];
    }
    vec.resize(vec.size() - count);
    // ...
}
```

Now we can see that insertion starts by resizing the vector to be larger whereas removal ends by resizing the vector to be smaller — inverse operations.

Then the insertion moves data forward starting at the end and moving toward the beginning. But the removal moves data backward starting from an earlier position and moving toward the end. Again, inverses, pretty much.

The only thing that isn't inverse-like is the last bits. The insertion ends by actually inserting the data and the removal begins by counting how many items are to be removed. This isn't very inverse-like, but we don't really remove the data, after all — we just overwrite it.

The point is, that if you remember one of these algorithms, you should be able to rebuild the other from scratch.

6.4.1.3 templates Revisited I

These algorithms — insertion and removal from a vector — are so generic that we'd like to make them work for vectors of other base types. Let's make them into templates!

We'll start with insertion. Before we template, we'd like to make it a bit with some error checking/reporting:

```
inline bool insert(vector<double> & vec, double new_value)
{
    auto orig_size{vec.size()};
    vec.push_back(new_value);
    return vec.size() > orig_size;
}

inline bool insert(vector<double> & vec, vector<double>::size_type target,
                   double new_value)
{
    auto orig_size{vec.size()};
    bool okay{false};
    if ( target == orig_size )
    {
        okay = insert(vec, new_value);
    }
    else
```

[10] Remember that an algorithm is just a plan or set of steps by which a problem can be solved. All of our code counts as an algorithm in some sense. But we especially use this term for functions which are so easily reused as these.

```
    {
        vec.resize(vec.size() + 1);
        okay = vec.size() > orig_size;
        if ( okay )
        {
            for ( auto pos{orig_size}; pos > target; --pos )
            {
                vec[pos] = vec[pos - 1];
            }
            vec[target] = new_value;
        }
    }
    return okay;
}
```

This rearranged some responsibilities and shifted the focus of the `for` loop, so let's examine it carefully. The overloaded helper now does its own work instead of calling the big function. This helps to split the work more evenly and helps us debug should a problem arise because we can see that it is an end-insertion and focus on that code.

Second, we refocused the `for` loop to walk through the destination positions instead of the source positions. This removed the `+1` protection and allowed us to use `auto` to type the walking variable.

We also added error checking in the form of making sure the `vector` grew when it should have. Some people on the net will tell you to check `max_size` before growing instead. This is less than useful as discussed in section 6.3.1.1, however. So we recorded the original `size` and checked the new `size` after the attempt to grow — either by `push_back` or by `resize`.

Finally, we `inline`d the workhorse function because upon reflection, it comes in at just 10 lines. (13 including the `return` and variable declarations, but this are rarely counted. Some would even merge the `okay` initialization and the following `if` to check it making our count 9.) This fits nicely in our guidelines from section 4.5.3 so it is a good call.[11]

Now to `template` it for different base types. This is most likely your first instinct:

```
template <typename BaseT>
inline bool insert(vector<BaseT> & vec, const BaseT & new_value)
{
    auto orig_size{vec.size()};
    vec.push_back(new_value);
    return vec.size() > orig_size;
}

template <typename BaseT>
inline bool insert(vector<BaseT> & vec,
                   vector<BaseT>::size_type target,
                   const BaseT & new_value)
{
    auto orig_size{vec.size()};
    bool okay{false};
    if ( target == orig_size )
    {
        okay = insert(vec, new_value);
```

[11] Pardon the pun there. . .

```
    }
    else
    {
        vec.resize(vec.size() + 1);
        okay = vec.size() > orig_size;
        if ( okay )
        {
            for ( auto pos{orig_size}; pos > target; --pos )
            {
                vec[pos] = vec[pos - 1];
            }
            vec[target] = new_value;
        }
    }
    return okay;
}
```

But it won't completely compile. It has one fail point — the `size_type` argument. The compiler can't figure out that the `size_type` inside the now-`template`d `vector` is a data type and therefore valid for an argument type. We have to nudge it by inserting the keyword — wait for it! — `typename`. You thought this new keyword was just to introduce `template` types above a function, but it has an active use, too! Here is the fix:

```
template <typename BaseT>
inline bool insert(vector<BaseT> & vec,
                   typename vector<BaseT>::size_type target,
                   const BaseT & new_value)
```

Keeping these things in mind, we look over our removal functions and apply our new-found knowledge to them, too:

```
template <typename BaseT>
bool remove(vector<BaseT> & vec,
            typename vector<BaseT>::size_type remove_this,
            typename vector<BaseT>::size_type keep_this)
{
    typename vector<BaseT>::size_type count;
    keep_this = keep_this > vec.size() ? vec.size() : keep_this;
    bool okay{keep_this >= remove_this};   // since keep_this is already
                                           // <= vec.size(), transitivity
                                           // will protect remove_this, too
    count = keep_this - remove_this;
    if ( okay && count > 0 )
    {
        auto orig_size{vec.size()};
        for ( auto pos{keep_this}; pos < orig_size; ++pos )
        {
            vec[pos - count] = vec[pos];
        }
        vec.resize(orig_size - count);
        okay = vec.size() < orig_size;
    }
    return okay;
}

template <typename BaseT>
inline bool remove(vector<BaseT> & vec,
                   typename vector<BaseT>::size_type remove_this)
{
    return remove(vec, remove_this, remove_this+1);
}

template <typename BaseT>
inline bool remove_last(vector<BaseT> & vec)
{
    auto orig_size{vec.size()};
    vec.pop_back();
    return vec.size() < orig_size;
}
```

Here we've added error checks to the shrinking codes and made the remove_last helper do its own work. The error checks are necessarily important as most implementations can't fail to shrink a size, but it isn't a horrific slow-down, so why not?

6.4.1.3.1 An Alternative Approach

There is an alternative to using the typename keyword like this. We could have instead made a templated using alias to help us out! This using alias would look something like this:

```
template <typename Container_Type>
using Size_Type = typename Container_Type::size_type;
```

Note that we are still using the typename keyword, but it is now embedded inside the using alias so we don't have to deal with it all the time.

Also note that we can't do this with a typedef. Only the using aliases. They are not able to be templated.

How do we use it, though? That would look like so:

```
template <typename BaseT>
bool remove(vector<BaseT> & vec,
            Size_Type<vector<BaseT>> remove_this,
            Size_Type<vector<BaseT>> keep_this)
{
    Size_Type<vector<BaseT>> count;
```

It acts in a function-like way to extract the size_type from the given vector. But it also works with strings if we wanted it to. We just probably wouldn't since strings are templates and so don't suffer in the same way.[12]

Should we use this or the raw typename? It is another six-of-one[13] situation. Just pick your favorite or try them both out in different programs to see which you like better.

[12] Full disclosure: string is actually a template instantiation to the char type from basic_string. There is another instantiation called wstring for the wchar_t type we spoke briefly of in section 2.3.1.3.

[13] Six of one, half-a-dozen of another. . .

6.4.1.4 Searching

There are two types of search that are most useful/prevalent. These are linear or sequential search and binary search. We'll look at both of these and discuss their relative advantages and disadvantages.

6.4.1.4.1 Linear Search

Let's start with the most general search algorithm: linear search (aka sequential search). It starts the search for a target value by looking at the first element to see if that's it. If it is, we stop and report success! If it isn't, we move over to the second position and check that value. This repeats until there are no more positions to check or we find the target value. In code, it looks something like this:

```
template <typename BaseT>
    inline typename vector<BaseT>::size_type
locate(const vector<BaseT> & vec,
       const BaseT & find_me,
       typename vector<BaseT>::size_type start = 0)
{
    auto pos{start};                // start at specified position
    while ( pos < vec.size() &&     // not at end of vector  AND
            vec[pos] != find_me )   // not found, yet...
    {
        ++pos;                      // try next one...
    }
    return pos;                     // either place find_me was, or .size()
}
```

I went ahead and templated it. Notice that the BaseT parameter find_me is passed by const& since we don't know at this stage how big that type is. If it is a class type like string, we'd want it protected from copying. And it won't hurt a built-in type to be so protected, either. It is just usually not worth our typing to do this sort of protection to built-in types.

I've defaulted the start position argument to 0 because that is the normal place to begin a search. But if the caller wants, they can specify a different place. This is similar to how the string *find*[14] family of functions are overloaded with an argument specifying where to begin their search.

The result of the function is either the location of the target value or the size of the vector when the target value wasn't found in the container. This allows the caller to check quickly whether the target was found by doing something like:

```
vector<string>::size_type location;
vector<string> stuff;
// fill in stuff from user
location = locate(stuff, "sweet");  // may need string{} around "sweet"
if ( location < stuff.size() )
{
    // "sweet" found
}
else
{
    // error:  "sweet" not present
}
```

[14] Recall these *s are for you to pattern-match against other names — not part of the function's actual name.

As indicated in the comment, some compilers may need help searching a templated vector parameter for a slightly different type of value. Here "sweet" is a string literal and the parameter is expected to be a const string &. Since string literals aren't exactly this type, the match may fail unless we help it by either anonymously constructing a string from "sweet" in the argument or by explicitly instantiating the locate template.

The former is easy enough:

```
location = locate(stuff, string{"sweet"});
```

The second looks like a vector declaration instead of a function call:

```
location = locate<string>(stuff, "sweet");
```

The reason for this odd similarity is that the vector is itself a template. (So is numeric_limits, if we are being transparent about it.) Some other time we may discuss making a whole class a template. But for now, just know that it is a thing and we're using it if not implementing it for ourselves.

Anyway, either of these techniques can help you avoid some template instantiation issues. Typecasting is also of great help here as we discussed when we first went over overloading in section 4.4.2.2. (No, overloading has nothing particularly to do with templates, but the typecasting technique is useful for both to help the compiler over an ambiguity or instantiation failure.)

The basic attributes of this search are that it runs only on base types that can be compared with != and it takes on average half the size of the vector to find what you are looking for. We call this last property linear performance because it is based solely on the size of the data collection and nothing else.

This is sometimes written as $O(n)$ or $\Theta(n)$ depending on how recently your colleague has studied algorithm analysis. *smile* These are pronounced "Big-Oh" and "Big-Theta" of n, respectively.

6.4.1.4.2 Comparison Within a Search

The dependence of the linear search on != is a bit concerning, of course. After all the compiler often complains when we do this or == to floating-point types and our current application involves doubles. And if we have our own class type, we don't have operators overloaded for those yet.

This can be addressed in a couple of ways. The first is to overload the locate template with one or more extra parameters. This is probably best for our floating-point issue because we'll need to know how accurately to match those.[15]

Such an overload might look like this:

```
template <typename BaseT>
    inline typename vector<BaseT>::size_type
locate(const vector<BaseT> & vec,
       const BaseT & find_me,
       const BaseT & epsilon,
       typename vector<BaseT>::size_type start = 0)
{
    auto pos{start};               // start at specified position
    while ( pos < vec.size() &&                   // not at end of vector
            abs(vec[pos] - find_me) > epsilon )   // AND not found, yet...
    {
```

[15] Recall our method of comparing these types is an absolute difference compared to a small value known as an epsilon. This value should be small enough for the application needs and we used a typical value of 10^{-6} before.

```
        ++pos;                        // try next one...
    }
    return pos;                       // either place find_me was, or .size()
}
```

This overload is still a template but has an extra required parameter specifying the epsilon for the absolute difference test. We cannot default this argument because then it could be called with exactly the same number of parameters of the same types. This would cause the compiler no end of headaches. As it is, we might need some typecasting to make the compiler tell the vector base type from its size_type.

The other technique deals with template specialization and is discussed below in section 6.5.1.

6.4.1.4.3 Binary Search

Linear search works with data that is in any order and can be compared to see if it is equal or not. That's pretty general and useful. But, if we add a couple of more requirements, we can make a search algorithm that is **much** faster.

First we require that the data be organized into order — sorted we call it. This would generally require and we further enforce that the data be comparable by at least a less-than-like operation in addition to the equal requirement that linear search already imposed.

With these two extra stipulations in place, we can work out a new algorithm: binary search. This algorithm is so named because it splits the list in two halves at each step. One half is to be ignored and the other is to be looked deeper into.

How can we ignore half the data? Since the data is in order, we can see if the target is the same as the middle piece of data. If it is, we are done. If not, we check if the target is less than the middle piece of data. If it is, then the target is also less than all data that follow the middle and so we need not look at any of them. If it isn't, then the target is clearly greater than all the data before the middle and we can eliminate them from consideration.

This process continues in the half chosen until only a single piece of data remains. That data is either the target or not and we return that information to the caller.

There are a couple of ways to implement binary search. I'll discuss the classic way here and you can explore another way and a tweak on both versions on your own.

Here is basic binary search (albeit templated):

```
template <typename BaseT>
    inline typename vector<BaseT>::size_type
binary_search(const vector<BaseT> & v, const BaseT & target)
{
    auto mid{v.size()},            // set up for empty issue
         bot{mid}, top{mid};       // init required for auto
    if ( ! v.empty() )
    {
        bot = 0;
        top = v.size() - 1;
        mid = (bot + top) / 2;  // average for middle
        while ( bot < top &&        // more elements to search thru
                v[mid] != target )  // not what we want
        {
            if ( v[mid] < target )  // target falls above middle
            {
                bot = mid + 1;      // move bottom up just past middle
            }
            else  // v[mid] > target  -- target falls below middle
            {
                top = mid;          // move top down to middle
            }
            mid = (bot + top) / 2;
        }
    }
    return mid;   // item is either at mid or not or size if v was empty
}
```

As we said, we continue to cut the search space in half as long as there is at least one piece of data and the middle element isn't the target. To adjust the bounds, we see if the target is greater than the middle (i.e. the middle is less than the target) and move the bottom boundary up to just past the middle. If it isn't, we move the upper boundary down to just the middle instead.

The reason for this difference is that the `mid+1` is moving the search position toward the end of the `vector` but the integer division for the middle is slightly moving the search position toward the beginning of the `vector` — due to truncation. These basically balance out. But if the `top` were to go all the way to `mid-1` then it would unbalance the system. I've seen this kind of thing cause a perfectly good binary search skip over the target value! Not the behavior we want, of course.

There are two tweaks that are fairly common for a binary search algorithm. The first is to implement it with all < operations instead of having a mix of != and <. This tweak isn't terribly hard but does demand a helper `bool` to trigger the end of the loop and an extra branch to adjudicate the equal case. This is a useful idea because it limits the requirements we are placing on the data and therefore on the programmer making the list.

The second is to `return` either the place the data is at or the place the data would have been had it been there. This is very helpful to those who want to keep their data sorted by inserting new items in their proper place every time. In fact, this idea is the basis for one of the sorting techniques we cover in a little bit (section 6.4.1.5.3).

Both of these techniques are left as exercises for the intrepid reader.

6.4.1.4.4 Comparison of the Searches

The performance of linear search is summarized in the following table.

`v.size()`	**Average Comparisons Until Found**	**Comparisons Until NOT Found**
8	4	8
64	32	64
1024	512	1024
4096	2048	4096
1048576	524288	1048576

The first column is supposing a certain length of list stored in the `vector`. The second column counts the average number of comparisons it takes to find a target in that list. And the third column counts the number of comparisons it takes to realize a target isn't in the list at all.

For the binary search, the numbers are a bit better:

`v.size()`	**Average Comparisons Until Found**	**Comparisons Until NOT Found**
8	4	4
64	7	7
1024	11	11
4096	13	13
1048576	21	21

Wow! Those numbers are **WAY** better!

It turns out that the performance of linear search is a linear function of the number of comparisons done whereas the performance of binary search is a logarithmic function of the number of comparisons done. This measure comes about from a process known as algorithm analysis. More on this topic can be found in a second semester course or in a course on discrete mathematics.

But, remember that the list must be sorted for binary search to work. How much effort is that going to take? Well, the easy sorts we'll learn in this book are quadratic in nature. So they cost quite a lot of time to perform. Looks like linear search wins after all. . .

Maybe not! There are two factors in binary search's favor. One is that many applications gather and sort their data once and then perform lots of searches on the data. Thus even a quadratic sort can be negligible in the face of thousands of logarithmic searches. Another is that there are other sorting algorithms out there. Many are better than quadratic, in fact. We won't learn those until another term, but they exist and so we have a plan to upgrade if necessary at that time.

6.4.1.5 Sorting

As mentioned before, sorting is the act of placing the data into order. This could be in any of ascending, descending, non-decreasing, or non-increasing orders. The difference in the first pair and the second is, of course, whether duplicates are allowed or disallowed. Ascending and descending imply that there are no duplicates in the data whereas non-decreasing and non-increasing imply there can be duplicate values in the data.

So how do we do this? Well, there are three fundamental techniques that all programmers should know: bubble sort, selection sort, and insertion sort. These sorts may be slow as noted in the search performance discussion above (section 6.4.1.4.4), but they are used in everyday situations where data is small or speed of coding is more important than the speed of the running application. There are more advanced sorts that run quite a bit faster, but they are more complicated than we want to take on at this point in our programming development.

6.4.1.5.1 Bubble Sort

The approach taken by bubble sort is to make the data float or percolate its way into a new position — its final resting position if possible.[16]

It does this by comparing next-door neighbors to one another and seeing if they are out of order compared to how we want them to be in the sorted order. If they are, we'll `swap` them like we did when we first learned reference arguments (section 4.4.1.2). Now we do this for every pair of next-door neighbors in the `vector` using a `for` loop.

Turns out that this process will put a single element into its final resting position in the sorted list per pass. But there are `size` elements to put into place overall. So we must also place another loop around our neighbor-visiting `for` loop to repeat those passes until all the elements are put into place.

If the `size` of the `vector` is n, then it will take $n-1$ passes to put all the data into place. This is because the n^{th} piece of data is already in its right spot as long as the other $n-1$ pieces of data were put into their places by a full pass. Think about it in the small. Let's say you had 4 pieces of data.

[16] I actually view this as more of a wave effect, but I was out-voted at the geeks convention and it stayed 'bubble' sort. *shrug*

Each pass puts one of them into its final resting place. After 3 passes, 3 of them are in their final place. Where is the fourth one? It must also be in its rightful place! There's nowhere else for it to go, after all.

Finally, it turns out that the one-per-pass guarantee will also let us stop early if not all the data were out of place to begin with. If only two pieces of data were out of sorts,[17] we would only have to make two passes to put them to their proper positions. Then the rest of the passes could be skipped!

How would we detect that only two pieces of data were out of place? Well, we can't easily. But what we can do is tell that the third pass did no work. What? That's right, wait until the third pass is done instead of the second and say, "Hey! That pass didn't swap anything. Let's stop now." How can we tell this? Our old friend bool who remembers when things have or have not happened during the run. We put a bool in the branch with the swap call to flag that we made a swap. If we haven't, we stop the outer loop before the next pass.

In an extreme case, this would mean making only one pass through an already sorted list to verify this fact and then stopping. Such behavior is called short-circuiting because it cuts the loop off early when a special situation is detected.

What's all this look like? Like so:

```
// put the data in vec into non-increasing order
template <typename BaseT>
inline void good_bubble(vector<BaseT> & vec)
{
    using vecSize = typename vector<BaseT>::size_type;
    vecSize loop = 0;
    bool done = false;
    while ( loop+1 < vec.size() &&    // max passes to put size items in order
            ! done )                  // had to swap last round
    {
        done = true;     // hope we are done this time around
                         // the last item has no next neighbor
        for ( vecSize cur = 0; cur + 1 != vec.size(); ++cur )
        {
            if ( vec[cur] < vec[cur + 1] )      // if cur and neighbor out of order
            {
                swap( vec[cur], vec[cur + 1] );  // swap
                done = false;                    // we moved stuff, not done, yet
            }
        }
        ++loop;
    }
    return;
}
```

As mentioned before, this ends up being quadratic on average, but can be faster due to the done bool tracking placements. It does require a helper swap function, but we've done that before — even in templated form (section 4.6.1).

6.4.1.5.2 Selection Sort

Selection sort tries to improve on bubble sort in a particular way. Although they are both quadratic algorithms in terms of comparisons done, selection sort aims to move less memory around to get the order right. Bubble sort, you see, moves a quadratic amount of memory around. And if each element is

[17] Pardon the pun. . .

more than a simple built-in type — say a string or other class object (see section 6.5) — then this could mean a lot of movement, indeed!

Does selection sort succeed? Why, yes, it does! It only moves a linear amount of memory to order the n items in your vector.

How does this work!? Well, selection sort starts by finding the item in the list that should go first — the largest element for a non-increasing order as we did before. Once found, the algorithm swaps it with whatever is currently in its proper spot. Then we find the second largest value in the list and do it again. Over and over until $n - 1$ items have been placed into their proper spots.

That actually sounds kinda cool. Let's look it over in code:

```cpp
// find largest item in vector and return its position
template <typename BaseT>
inline typename vector<BaseT>::size_type
largest(const vector<BaseT> & vec, typename vector<BaseT>::size_type from = 0)
{
    auto max = from;  // assume first is largest
    for ( auto at = from + 1; at < vec.size(); ++at )  // go thru ~rest~ of list
    {
        if ( vec[at] > vec[max] )    // this one bigger than largest?!
        {
            max = at;           // re-think our decision!
        }
    }
    return max;
}

// vector is placed into in sorted order -- non-ascending
// (decreasing)
template <typename BaseT>
inline void good_select(vector<BaseT> & vec)
{
    using vecSize = typename vector<double>::size_type;
    auto max = largest(vec);                              // find largest
                  // walk through size - 1 positions -- last is already placed
    for ( vecSize cur = 0; cur + 1 != vec.size(); ++cur )
    {
        if ( max != cur )    // if it wasn't in the right place
        {
            swap( vec[cur], vec[max] );    // swap it out
        }
        max = largest(vec, cur + 1);     // what's the next largest?
    }
    return;
}
```

But that's two functions! Yep. We broke the finding of the i^{th} largest item off into a helper function. This is also a reusable function as finding the largest thing in a vector comes up a good deal, too. Let's walk through both algorithms in turn.

The largest function starts looking at the position indicated to start from by the caller. (This defaults to 0 on our initial call.) We assume that this value is the maximum value in the entire vector and, indeed, it is the maximum of those we've inspected so far!

As we move from the next element to the end of the vector, we check each element to see if it is greater than our current maximum value. Since we've actually stored the position of the maximum element, we have to subscript to do this check. If the new element is larger, we change the position we've recorded as that of the maximum element. This continues until the last item of the vector is checked. At this point we return the position of the largest value in the vector — after the position indicated to start from, at least.

The selection sort routine, in its own right, asks for the overall largest value in the vector and then starts a loop through all but one element. This is because each element looped over will be a potential destination of a swap to put an element in place — largest first down to smallest. Since when all but one element is swapped into place, the last one will be in its proper place already, we can stop the loop one shy of the vector's size.

Inside, if the largest value isn't in its proper position — the current position, we swap it there. Then we ask for the next largest value by searching from just after the cur position where we just put the largest value.

Overall, we move at most one pair of memory locations per sort loop and so we end up with a linear amount of memory moved to place the items into order. Again, this will be of most benefit when the data items in the vector are larger than built-in types.

6.4.1.5.3 Insertion Sort

Bubble sort and selection sort both looked at the list as a whole and put together the sorted list in an all-inclusive way. Insertion sort takes a different approach altogether. It looks at the first element as a small but already sorted list. Then it brings into view/account the second element and finds where it should go amongst the already sorted list — aka before or after the first element. If after, it is already there so we do nothing special. If before, we hold onto the second element outside the vector and scoot the first element over to make room and then insert the old second element into the first slot.

This process continues bringing in the third, then fourth, etc. elements until all elements have been inserted into their proper place in the sorted portion of the vector. This is still quadratic since inserting an item in a list is linear and we are doing that once per new element for all the elements — save the first which was already sorted by itself. And it does move a lot of memory with the shifting of elements for each insertion — a quadratic amount, in fact.

So what's so good about this approach? Well, it can be amortized over the course of the user entering the data. Amor-what? Amortized. It means basically that we can average out the quadratic cost of the algorithm over the input of the data — a linear process — to get a linear feel to our sorting process.

How does that work? Well, the user enters a piece of data and we push it onto the vector and call insertion sort. This returns immediately saying the single element is sorted. Then the user enters the second piece of data. We again push it onto the list and call insertion sort. This time it may need to insert the new data in front of the older item. And we do this once per input. The user feels a slightly longer linear lag at each input rather than a single quadratic lag at the end of their input sequence. If we waited to bubble sort a list of 100 items, for instance, it would be a lag of 10000 magnitude. But by insertion sorting each new item as it comes in, they feel no more than a 100 lag each time. That's much nicer.

Okay, enough theory! Let's see that code! It looks like so:

```
/*
 * Place the new element into the head of the sub-vector
 * from before_me (inclusive) to end (also inclusive!) of
 * the given vector.  The current element at that position
 * (as well as any which follow it) are shifted toward the
 * end of the sub-vector.  A value of true will be returned
 * if the sub-vector was non-empty; false will be returned
 * otherwise.
 */
template <typename Base_Type>
inline bool insert(vector<Base_Type> & vec,
                   typename vector<Base_Type>::size_type before_me,
                   const Base_Type & new_item,
                   typename vector<Base_Type>::size_type end)
{
    using vecSize = typename vector<Base_Type>::size_type;
    bool okay = before_me < end;
    if ( okay )
    {
        for ( vecSize pos = end; pos > before_me; pos-- )
        {
            vec[pos] = vec[pos - 1];
        }
        vec[before_me] = new_item;
    }
    return okay;
}

// vector is placed into in sorted order -- non-ascending (decreasing)
template <typename Base_Type>
inline void good_insertion(vector<Base_Type> & vec)
{
    typename vector<Base_Type>::size_type next, dest;
    Base_Type holder;
    next = 1;                       // next unsorted item
    while ( next < vec.size() )
    {
        holder = vec[next];           // hold new value
        dest = next;                  // start where it was
        while ( dest > 0 &&              // look amongst the already sorted
                vec[dest - 1] < holder ) // for before whom the new guy goes
        {
            --dest;
        }
        if ( dest != next )           // not already in place?
        {
            // insert 'im in front of dest
               //      in-front-of    what   end-of-insertion-range
            insert(vec,    dest,     holder,    next);
        }
        ++next;                       // move to next item
    }
    return;
}
```

The trickiest part is the search for the destination. This is the while loop nested inside the outer one. It walks backwards through the preceding elements to the beginning of the sorted portion looking at each item to see if it is less than the held copy of the newly viewed value. If so, we move on. If it is greater than or equal to, we stop because that is the element this new one should be placed to precede!

(Remember, we are sorting into non-increasing order in this sort — largest to smallest, basically. So all smaller elements should follow us in the list — we'll insert in front of them. The first larger or equal value should precede us — we'll insert after them.)

Online you'll find versions of this algorithm that merge the search for the destination spot with the shifting done in our helper function. This is a good idea in theory, but in practice it slows them down as testing has shown.

One other thing to point out is that it can be important that we search for the new spot backwards like this. If we searched forwards like so:

```
dest = 0;
while ( dest < next &&            // look amongst the already sorted
        vec[dest] > holder )      // for before whom the new guy goes
{
    ++dest;
}
```

we would possibly jump a value over several identical values if the list's data is not all unique. This can be important for reasons of stability which is discussed shortly in the sorting algorithm comparison (section 6.4.1.5.5).

6.4.1.5.4 Reversing a Sort's Order

As noted before, the order of sorting might be important to the user or even another algorithm — like binary search. In fact, our implementation of binary search required the data to be in non-decreasing order and then we made all our sorts non-increasing! How can we fix this?! Well, it isn't difficult. We just change a < or > here and there and all is well with the world.

Which ones? The `if` protecting the `swap` in bubble sort would change its < to a >. The > in `largest` to find the maximum element for selection would change. (Of course this would also demand changing this to a `smallest` function and making `max` into `min` instead. If we didn't, the other programmers on our team might be a bit stumped by our logic and results!) Finally, the > in the search for a destination in `good_insertion` would change.

But, this need only be done if you really need the implementation to work well with another algorithm like binary search. If the user just asks for the sort to be reversed (by clicking on a column head or the like), we can just print the `vector` backwards from how we currently have it sorted! Never resort when they ask for a reversal! That's slow and silly.

6.4.1.5.5 Comparison of the Sorts[18]

So how do these sorts compare to one another? What about those more advanced sorts we mentioned before? How do those compare — even though we won't learn them yet? I've summarized these answers in the table below. It is quite complicated, though. There are eight columns and four footnotes, for instance. So be sure to read the explanation that follows it!

[18] Pardon the pun.

Sort	Average Comparisons	Average Swaps	Worst Comparisons	Extra Memory	Amortizes	Short-Circuits	Stable
Bubble	n^2	n^2	n^2	0	no	yes	yes
Selection	n^2	n	n^2	0	no	no	no
Insertion	n^2	n^2	n^2	0	yes	no	yes*
Quick	$n \log_2 n$	$n \log_2 n$	n^2†	0	no	no	no‡
Heap	$n \log_2 n$	$n \log_2 n$	$n \log_2 n$	0	no	no	no
Merge	$n \log_2 n$	$n \log_2 n$	$n \log_2 n$	n§	no	no	yes

*If you use a backward linear search for insertion location.
†Only if not randomized. If randomized worst comparisons is $n \log_2 n$.
‡Only if randomized to avoid quadratic worst comparisons. If not randomized, it is stable.
§Only when sorting a `vector` or `array`. Other containers can avoid this excess memory.

The way we generally tell the simple sorts apart is average swaps, amortization, short-circuiting, and stability. Average swaps was talked about above and I think is pretty clear. The next two were described in their particular sorts' sections. Stability, on the other hand, is a generally desirable feature in a sort that makes earlier sorting results stick around when a new sort is applied. Before we get into it more deeply, however, let's look at the difference between short-circuiting and amortization.

There may at first seem no difference between short-circuiting and amortization, but there is a subtle one. A bubble sort can stop early after putting a single out-of-place item into position no matter where the item was originally. Insertion sort can only do this if we put the out-of-place item in the last slot and only perform one pass of the outer loop.

In fact, this is such an oft-used technique that it is sometimes coded for by moving the body of the outer loop into a helper function that takes the `next` position as a parameter in addition to the `vector`. This, then, can be called with the index of the last element to make a single pass. Perhaps we'd even default it to a special value to make it simpler for our caller to use:

```
// When next is '-1', we use the last position of the vector, otherwise we
// use the specified position as the new position for this pass.
template <typename Base_Type>
    inline void
insert_sort_one_pass(vector<Base_Type> & vec,
                     typename vector<Base_Type>::size_type next = -1)
```

Although `-1` is an invalid value for a `size_type`, this just wraps around the `unsigned` circle of doom[19] to the maximum. The alternative would be to use `numeric_limits` to find the max value and that would be really heavy typing. Keep this possibility in mind, though, if your compiler complains about the `-1` default initializer. Just in case it troubles you, it would look like this:

```
// When next is the max of the size_type, we use the last position of the
// vector, otherwise we use the specified position as the new position for
// this pass.
template <typename Base_Type>
    inline void
insert_sort_one_pass(vector<Base_Type> & vec,
                     typename vector<Base_Type>::size_type next =
                         numeric_limits<typename
                             vector<Base_Type>::size_type>::max())
```

Now back to stability. To see this concept most clearly, we need a bit of data with multiple attributes like a `class` object or even, perhaps, a `string`. We can then sort the data on one part of the data and follow it up with a resort on another part. This is analogous to sorting a spreadsheet by one column and then on another.

[19] See section 2.4.2 for a refresher on `unsigned` arithmetic.

How does all this relate to stability? Well, if the sort is stable, it can guarantee the relative order of the prior sorted values amongst like-valued items in the latest sort. Let's look at an example. We'll start with this table of values:

ID	Month	Sales
432	8	189
123	8	116
555	8	203
432	9	105
123	9	43
555	9	187

Here the data is already sorted on the months of the sales for the sales personnel. Sadly, we can't actually witness the stability of the sort by just sorting on sales figures next. This is because the sales figures are all unique.

ID	Month	Sales
123	9	43
432	9	105
123	8	116
555	9	187
432	8	189
555	8	203

But if we further sort on the sales personnel IDs, we can see the stability within equal ID values:

ID	Month	Sales
123	9	43
123	8	116
432	9	105
432	8	189
555	9	187
555	8	203

See how the sales figures are still sorted within the equal ID groups? If this is guaranteed by the sorting algorithm, it is said to be stable. This can make a presentation much nicer and more readable and so is highly desired. (As a side benefit here we also note that September sales are always lower than August sales no matter the sales person.)

But even when we are able to use the more advanced sorts, we'll be able to distinguish situations to use one versus another. Here we rely a bit on stability, but also on worst number of comparisons and how much extra memory the algorithm uses.

Quick sort has two footnotes that might need explanation. It turns out that there are [at least] two forms of quick sort. The basic one has a quadratic worst number of comparisons when the input is just right. But by randomizing the data right off, we can avoid this problem. Of course, by randomizing the data, we mess up any prior order information and lose stability. So the entries in the table represent the worst of both worlds. Only one of these will be the case at any given time for any particular implementation. If randomizing, the worst comparisons will be $n \log_2 n$ and you won't be stable. If not randomizing, the worst comparisons will be n^2 but you will be stable. You have to decide which is more important: avoiding that worst possible input situation or stability.

The footnote on merge sort mentions that the n extra memory it takes — basically an entire extra copy of the container it is sorting — only happens when sorting a `vector` or `array`. Since these are our only two containers, this might seem weird at first. But there are other containers that we'll learn about in future courses. An example is a linked list. Here we don't store the data in a single contiguous block of memory but strew it about throughout the computer's memory and just 'point to' each successive piece from the current one. This lets us avoid the extra memory by just changing where we 'point' instead of. . . well, let's not get into all that right now.

6.4.2 An Example Application

Again, this program comes to more lines of code than we want to repeat here (just over 700!), so I'm placing it on a website for you. Here is a re-envisioning of the heights program as a heights list manager.

The program consists of a main application and three supporting libraries. The `display` overloads were left as non-`template`s in the application instead of being put in the `listops.h` library. So were specific `locate` and `remove` functions that didn't seem general enough to put in a library. All library functions are `inline` and `template`d for speed and ease of use.

As always, download the code and play with it in your compiling environment to see how it functions. Then tweak it and see if you can make it do something differently. Always learn from any new experience!

6.5 A class As a Base Type

It sometimes happens that a `vector` needs to contain not built-in type values but `class`-type values. This could be `string`s, `Die` objects, or any `class` you've created or found.

The main confusion new programmers have with this is the use of the dot operator right up against the subscript operator. For instance, if we wanted to take a `substring` from an element of a `vector` full of `string`s, we could do this:

```
vector<string> vec;
// fill up vec...
// later, after vector<string>::size_type i has a value:
vec[i].substr(0, 5);
```

This would pull 5 characters from the `i`th entry in the `vector` starting at position 0 in the `string`.

6.5.1 templates Revisited II

Let's say we wanted to search for a `Die` object in a `vector`. That `class` doesn't have a `!=` operator. What can be done? We can do a technique called specializing the `template`. This involves writing a version of the `template` function specifically for the desired type so that the compiler will use it as a more perfect match than the general `template` as it parses the call.

Here is an example for a `Die` `class` `vector` and the `locate` linear search template above:

```
template <>
    inline vector<Die>::size_type
locate<Die>(const vector<Die> & vec,
            const Die & find_me,
            vector<Die>::size_type start)
{
    auto pos{start};                 // start at specified position
    while ( pos < vec.size() &&          // not at end of vector
            ! vec[pos].isSame(find_me) )   // AND not found, yet...
    {
        ++pos;                       // try next one...
    }
    return pos;                      // either place find_me was, or .size()
}
```

Here we fill in all the `BaseT` references with `Die`, leave the angle brackets on the initial `template` header empty, and put `Die` inside new angle brackets between the function name and the argument list. This last is optional, but useful in some contexts so it isn't a bad habit to get into.

Inside we change the != code to use the isSame method of the Die class. This makes it so we can use the locate function to search for a Die object inside a vector of them.

Now when the compiler sees a Die-based vector being passed to the locate function, it will call the specialization instead of trying to use the more general template.

Do note that the default value for the last parameter wasn't specified here. That's because of the one-default rule we discussed back when we first saw default arguments. The default value has to be on the first head the compiler sees of the function and that was the general template rather than this specialization. They are considered two parts of the same function in a way.

If we wanted to, we could change this up from a full isSame comparison to check just whether the number of sides in the Die was what we were looking for:

```
while ( pos < vec.size() &&           // not at end of vector
        vec[pos].get_sides() !=       // AND not
        find_me.get_sides() )         // found, yet...
```

or we could make this a non-template overload of the original locate template:

```
    inline vector<Die>::size_type
locate(const vector<Die> & vec, const long & find_me,
       vector<Die>::size_type start = 0)
{
    auto pos{start};               // start at specified position
    while ( pos < vec.size() &&                 // not at end of vector
            vec[pos].get_sides() != find_me )   // AND not found, yet...
```

Here is a proof-of-concept driver for the locate templates and overloads:

```
#include <iostream>
#include <vector>
#include <string>
#include "die.h"

using namespace std;

template <typename BaseT>
    inline typename vector<BaseT>::size_type
locate(const vector<BaseT> & vec,
       const BaseT & find_me,
       typename vector<BaseT>::size_type start = 0)
{
    auto pos{start};                // start at specified position
    while ( pos < vec.size() &&     // not at end of vector  AND
            vec[pos] != find_me )   // not found, yet...
    {
        ++pos;                      // try next one...
    }
    return pos;                     // either place find_me was, or .size()
}

template <typename BaseT>
    inline typename vector<BaseT>::size_type
```

```
locate(const vector<BaseT> & vec,
       const BaseT & find_me,
       const BaseT & epsilon,
       typename vector<BaseT>::size_type start = 0)
{
    auto pos{start};               // start at specified position
    while ( pos < vec.size() &&                   // not at end of vector
            abs(vec[pos] - find_me) > epsilon )   // AND not found, yet...
    {
        ++pos;                    // try next one...
    }
    return pos;                   // either place find_me was, or .size()
}

template <>
    inline vector<Die>::size_type
locate<Die>(const vector<Die> & vec,
            const Die & find_me,
            vector<Die>::size_type start)
{
    auto pos{start};               // start at specified position
    while ( pos < vec.size() &&            // not at end of vector
            ! vec[pos].isSame(find_me) )   // AND not found, yet...
    {
        ++pos;                    // try next one...
    }
    return pos;                   // either place find_me was, or .size()
}

    inline vector<Die>::size_type
locate(const vector<Die> & vec,
       const long & find_me,
       vector<Die>::size_type start = 0)
{
    auto pos{start};               // start at specified position
    while ( pos < vec.size() &&                 // not at end of vector
            vec[pos].get_sides() != find_me )   // AND not found, yet...
    {
        ++pos;
    }
    return pos;
}

int main(void)
{
    vector<double> values;
    vector<double>::size_type loc;
    vector<string> labels = { "default", "thirds", "d20" };
    // here we construct the Die objects for a vector using curly brace
    // lists like we do for the vector itself; the empty list calls the
    // default constructor, the third one calls the only-size constructor
    vector<Die> stuff = { {}, { 3, 1.0/3, -2.0/3 }, { 20 } };
    Die target{20};
    vector<Die>::size_type at = locate(stuff, target);
    if ( at == stuff.size() )
    {
        cout << "No such Die!\n";
    }
    else
    {
        cout << "Found " << labels[at] << " at " << at << ".\n";
    }
    at = locate(stuff, 6L);
    if ( at == stuff.size() )
    {
        cout << "No such Die!\n";
    }
    else
    {
        cout << "Found " << labels[at] << " at " << at << ".\n";
    }
    values.push_back(2.49999);
    for ( vector<double>::size_type v = 0; v != 10; ++v )
    {
        values.push_back(stuff[1].roll());
    }
    loc = locate(values, 2.5, 1e-6);
    if ( loc == values.size() )
    {
        cout << "No such double!\n";
    }
    else
    {
        cout << "Found 2.5 at " << loc << " for 10^(-6).\n";
    }
    loc = locate(values, 2.5, 1e-2);
    if ( loc == values.size() )
    {
        cout << "No such double!\n";
    }
    else
    {
        cout << "Found 2.5 at " << loc << " for 10^(-2).\n";
    }
    return 0;
}
```

All of the templates and overloads play nicely together. We could make a library out of them, but the Die focused specialization and overload wouldn't fit with the more general ones. Those would either stay in the 'app' or go with the Die library itself or an application-specific library for helper functions.

6.6 Container Members of a class

It often comes to pass that we need to place a container inside a class because each object of that class will have/own a bunch of data of a common type. This design is effective but the implementation is deceptively intricate. It particularly affects how we construct, access, and mutate the container member.

Here we'll use a vector for our example, but the same basic principles would apply to an array except possibly for the mutation. There you wouldn't worry about the array gaining new members since we typically create those pre-filled.

6.6.1 Constructors, Accessors, and Mutators

6.6.1.1 Built-ins and class Members

class members of built-in types typically have an accessor and mutator each, no special helpers, and are parameterized in the extra constructor(s) for the class. Members of a class type (including string) are handled similarly.

Our only exception was when there is a binding/intimate relationship between two (or more) members such that altering one will have an impact on the other(s). In such cases, we used a common mutator for the related members rather than separate ones for each member. They still had individual accessors, however. (That was way back in section 5.2.2.2.1, remember?)

As an example, we might have this class representing students in a course tracks the students' names amongst many data items:

```
class Student
{
    string name;
    // ...
};
```

Here are sample accessor and mutator methods for this string member of the Student class:

```
const string & get_name(void) const
{
    return name;
}

bool set_name(const string & n)
{
    name = n;                  // normally we'd error check here,
    return true;               // but names are not really validatable
}
```

Note the const& on the getter's return type. This makes it so that the caller can't accidentally change the object sent back without great trouble and avoids the copy normally associated with a value return.

Remember that we always return a bool from a mutator to show success versus failure. Even here where we can't really validate the data we like to keep the pattern for the programmer using the class as consistent as possible.

Constructors, likewise need to treat the member variable in their member initialization list (or use an in-class initializer):

```
Student(void)
    : name{}
{
}
Student(const Student & c)
```

```
    : name{c.name}
{
}
Student(const string & n)
    : Student{}
{
    set_name(n);
}
```

Remember not to re-invent the wheel: delegate to the default constructor rather than redoing its initialization. The more member variables there are, the more this will help you in your coding.

Also note that I'm not scoping any of these definitions because they are all going to get inlined by placing them inside the class definition.

6.6.1.2 But With a Container. . .

If a class has a member which is of a vector (or array) type, however, we are dealing with data of a different ilk altogether! The built-in types and class types (even string) all mean to have objects of their type taken as single items. A vector, on the other hand, is meant to be a sequence of *individual* items. Therefore, the accessor, mutator, and constructor patterns for vector type members of a class must take this collection-of-individuals nature into account.

To explore this, let's add a vector to store the Student's grades:

```
class Student
{
    string name;
    vector<double> grades;
    // ...
};
```

6.6.1.2.1 Accessors

The simplest way to do so is to have the accessor (and mutator) for the vector member each take a size_type parameter to designate which item in the vector's sequence the caller wishes to access or mutate.

Notice the difference between the name (above) and grades members with respect to their access patterns:

```
double get_grade(vector<double>::size_type which) const
{
    return which < get_grade_count()     // grades.size()
         ? grades[which]
         : -1.0;                   // min_flag - 1.0
}

vector<double>::size_type get_grade_count(void) const
{
    return grades.size();
}
```

The string member here — name — was accessed as a single entity/thing. The vector member — grades, on the other hand, is accessed item by item. If a grade we don't own is accessed, we report its value to be -1.0 so that the careful programmer *can* notice that something has gone awry. In addition, we provide a helper function which returns the number of items in the grades vector to the programmer using this class (probably calling the get_grade method in a for loop bounded, we hope, by a call to get_grade_count to avoid running into that -1.0 value).

> **That Flag Comment**
>
> The comment about a min_flag deserves a little note. Basically, teachers keep track of special reasons students miss an assignment. Some track more situations than others. We used to just put a star or dagger in a paper gradebook. But that's hard to do in a computer storing numeric grades. So we put in flag values instead. Maybe -1.0 means that the student didn't show up and hasn't made arrangements to make up the assignment yet. Maybe -2.0 means the student has made arrangements already but hasn't made up the assignment yet. Etc. To make sure the programmer knows what negatives are allowed and which aren't, we'd have to add a min_flag member variable and allow it to be constructed and mutated at least. Then we'd use one less than min_flag as the error code in our accessor.

6.6.1.2.2 Validation and Information Hiding

Note how the get_grade validates the which argument by calling the get_grade_count function rather than directly calling the grades.size() function. This protects us from any underlying change in the implementation of the grades data.

After all, right now vector may be our only choice, but next semester or the one after that we may decide to change to a different storage medium. Then there may be a different way to find out the number of grades than '.size()'. By calling on get_grade_count, we limit our reliance on implementation knowledge and isolate those details into individual functions.

In fact, if I was really pushing this, I'd have begun our class' public area with a typedef or using alias as well:

```
class Student
{
    string name;
    vector<double> grades;
public:
    typedef vector<double>::size_type GradePos;
    using GradeSize = vector<double>::size_type;

    double get_grade(GradePos which) const;
    GradeSize get_grade_count(void) const;
};
```

Now the functions themselves are protected from any change in the underlying data storage technique with these typedef and using names.[20] The programmer using our class just has to refer to Student::GradePos rather than vector<double>::size_type whenever they are talking about a particular grade within the Student's record. (Or Student::GradeSize when they are referring to the number of grades the Student has amassed.)

In fact, this is exactly the mechanism by which and pretty much the reason why the vector and

[20] We normally wouldn't do both techniques in the same place, but I was just reminding you what each looked like. Also, I like having the two type names separate for semantic reasons.

string classes do this themselves with size_type! (Similarly with the time function and the time_t data type and the array class and the size_t type.)

6.6.1.2.3 Mutators

Mutation patterns are similarly adjusted from those of a non-container member. Note again the difference between the name mutator above for the name string member and this mutator/helper pair for the grades member:

```
bool set_grade(GradePos which, double g)
{
    bool okay = false;
    if ( which < get_grade_count() )
    {
        grades[which] = g;   // validate element itself?  >= min_flag
        okay = true;
    }
    else if ( which == get_grade_count() )
    {
        okay = add_grade(g);
    }
    return okay;
}

bool add_grade(double g)
{
    GradeSize old_size = get_grade_count();
    grades.push_back(g);   // validate element itself?   >= min_flag
    return old_size < get_grade_count();
}
```

Notice that the string member — name — was mutated all at once whereas the vector member — grades — is mutated on an item-by-item basis. It even requires a helper function to add new items to the vector. Our mutator does take the gentlemanly route of calling the add_grades method when the caller tries to modify the next grade that *would* exist. The alternative would be to skip the else-if branch altogether and just have the if branch and return. Some would debate us on this decision, but I wanted to show you and give you the chance to make up your own mind.

On the other hand, the error checking in add_grade is the right way to go. Other programmers would have you code something like this:

```
bool add_grade(double g)
{
    bool okay = false;
    if (get_grade_count() < grades.max_size())
    {
        grades.push_back(g);   // validate element itself?   >= min_flag
        okay = true;
    }
    return okay;
}
```

But this uses the poorly implemented max_size function and makes us think about the implementation again — removing that thin layer of design sanity we'd just achieved.

6.6.1.2.4 Constructors

Finally, constructor patterns for a vector element. These are oddly simpler than other types of members. Because the class is assumed to manage the vector member for its entire life-time, we don't typically need to construct in other than an empty manner. The one different case — copy construction — is handled nicely by the vector's own copy constructor. For instance:

```
Student(void)
    : name{}, grades{}
{
}
Student(const Student & c)
    : name{c.name}, grades{c.grades}
{
}
```

Also note that we don't have a constructor to take in a vector from the outside. If you do have other members which need construction, just default construct the vector member in their alternative constructor's member initialization list (or the in-class initializer or delegate to the default constructor):

```
Student(const string & n)
    : Student{}
{
    set_name(n);
}
```

The name can be initialized via a parameter, but the grades should not. We will manage the grades vector from inception to death.

6.6.1.3 Full Example

I've once again posted the full example (over 150 lines) on the book's website for you to download and peruse at your own pace alongside the discussion.

Most of it is just as we've coded above, so no surprises there. But I did add input, output, and an averaging function to the Student class. I also wrote a little main to take it for a drive.

The input and output deserve a little attention as they are the most complicated design we've had to date. This class has a lot of data — even at only two members! And that data is treated specially so some care should be taken in reviewing this interaction.

The output is pretty straightforward and will guide our input efforts. Again, not only do the two take a terse approach with no prompting or labeling as much as possible, but they also parallel one another so that the input could potentially read in exactly what the output produces. This is on purpose and will be put to good use shortly (chapter 7).

As you can see in your editor, the output function is on lines 35-51. It begins by sending the name member to cout followed by a newline. This is important for later as it means a getline will be needed to read it back into the class. We put the name on a line by itself because we expect it to contain space-separated name components — first, last, maybe middle, etc.[21] If we'd expected their name to just be one word, we could have just displayed it with just a space or tab.

Why the spacing if it isn't a multi-word entity? Well, because the grades must follow and we don't want them to run into the Student's name. Speaking of the grades, when we display this Student, do

[21] Etc here might entail a salutation or nickname in parentheses or quotes.

they have any scores accrued yet? It may be early in the semester and we are just printing a roster for taking attendance. What should we do if get_grade_count is 0?

This is a pretty deep question and better programmers than I have struggled with it for many years. There are basically two approaches: display nothing at all or display that there are no grades. The latter is more readable for an end-user, but we aren't always in this for their benefit. Sometimes programs send output such that it is suitable for input into another program instead of for a user to read. When this happens we say they are connected by a 'pipe' and the output of the first is piped to the second's input. Sadly, this is all we are going to say about this at this time, but keep an eye out for this topic in a future computer science course!

For the moment, we've gone with the user-friendly version and we'll have to make input look out for this situation. So we either print a space-separated list of the Student's grades or the message "No grades\n" when there aren't any. Each possibility is followed by a newline, notice.

So when we get to the input function on lines 139-162 starts out with its getline into a local string before calling the mutator for the name. Then it starts trying to read grades one at a time into a local double. This continues until cin fails or a newline is detected. if the loop ends in failure, we assume it is reading a phrase like that displayed by the output function above. To get this out of the way of future inputs, we reset the fail flag with clear as usual and then ignore the line also as normal.

But then we do something quite peculiar. We call clear with an argument! Normally we don't pass clear anything — just call on it to do its thing. But that just defaults, it turns out, to setting the input stream cin to a good state — making it happy enough to go about further inputs. Here we are passing an argument to make it go back into that failure fugue it was in a moment ago. This will make it more clear[22] to the caller of input that something went wrong.

It is our way to flag them down and pay attention to the fact that we didn't receive any grades during this input. Even if they don't pay attention to our bool return that tells them we were successful or not, they are going to be forced to pay attention to the fact that cin is now upset and won't go on.

So what is this argument? Well, the name of the flag constant that signals to cin that it is in a fail state is failbit. It was created in the ios_base class just like our formatting flags. There are four states cin or any input stream can be in: good, fail, eof, and bad. I've highlighted them like that because they are also the names of functions that report true/false that the stream is in that state at the moment.

The corresponding flags are the same names but with 'bit' appended. This is because the flags only take up one bit of space. Since cin has so many flags to deal with, they chose not to use separate bools for each but to combine them into a single integer space which can hold many such flags at once. (It turns out a stream can be both bad and failing at the same time, for instance!)[23]

Finally, let's look at the main on 109-137. It doesn't actually use our carefully crafted i/o routines but instead takes it upon itself to design a UI with a more friendly face. It prompts for and reads the name for a Student and sets this into its object. Then, since we are a driver as well as a handy application, we get the name back out with the getter to test that function as well. (Normally we'd just keep using our local copy instead of pulling it back out of the object like that.)

Next we prompt for the Student's grades making sure the possessive has an appropriate s on it if need be but leaving it out if it is not required. (Remember that possessives don't need an s if the noun already ends in an s.) Just a quick ?: suffices for this, so it is an easy way to make our program seem more intelligent.

Then we use a standard fail-terminated loop to read in a sequence of grades for the Student and add each one to our object in turn. When that loop ends, we clean up the "quit" or whatever they typed to cause the failure on cin and make a little report of the Student's status in the course.

[22] Pardon the pun. . .

[23] We won't discuss bit flags at any greater depth here, but it will almost certainly come up in a later course.

The get_average helper/utility function from the class (lines 97-106) is hardly worth looking at, but it does have one thing we might point out. It will report the double value NaN when no grades have yet been added. This is a natural consequence of dividing a 0 sum by a 0 count in a computer. And, as a secondary statistic, an average doesn't technically exist when there weren't any numbers to average. So NaN — 'not a number', remember — is a good value to use here.[24]

6.6.2 A Vignette

TBD — students example for vector in a class in vector As an aside, let's get a little creative and think a little more abstractly for a moment. We usually don't have just one Student at a time in a course so why so in our program? Let's collect together multiple Students in a vector. At first glance this isn't a big deal:

```
vector<Student> roster;
```

Now we have a roster capable of holding more than a single Student at a time. So what? Well, think about it. Our roster vector has inside it Student objects. Each of those Student objects has inside it both a name string and a grades vector (holding doubles). So we've got vector whose elements hold further containers. It's almost two-dimensional!

But let's just look at the whole driver/application. The Student class itself hasn't changed at all, so no worries there. But there's a new function called read_student and the main is all new, too.

6.6.2.1 Producer/Factory Functions

The roster is first 'filled' with the user's Students in a nice little yes/no loop. Here we utilize the read_student function inside our push_back onto the roster. So what is this new function all about? Its comment says it is a factory or producer function. That means it manufactures or produces objects for us to use. In this case it is producing Student objects one at a time by reading their data from the user.

A deeper look at the function (lines 130-154) reveals the same code we used in our main in the last program! If you pull the two out and run them through your difference checker[25] you'll actually find a tiny change. I alternated from a char literal apostrophe for the possessive to a literal string. Not much, but just a reminder that there is often more than one way to solve a problem and that you need to escape a quote inside its own kind of delimiter but not vice-versa.[26]

This is another instance of a fancy word for a relatively simple concept. Just like we had with composition and class design — making one class with at least one member variable of another class type. A factory or producer function is just a function that fills in an object somehow — from input or randomly — and returns it to the caller.

After the main fills its roster vector, it makes a little table of the gathered Students and their averages. It isn't a wonderful table. It could use some width or setw love for sure. But it gets the job done for a driver-style application.

6.6.2.2 Local Scope and Looped Objects

The only evident note is whether or not to use Student or auto in the range-based for loop that fills in the table. We definitely want the const& on it either way to avoid copying each Student as we go by.

[24] Also we static_cast the number of grades to a double for the division because some computers have a vector size_type that is too big to fit in a double and the compiler will warn us of this situation without it. This shouldn't be a problem here because we don't expect the number of assignments in a single course to be a number with as many as 19 or 20 digits. I'd guess 20-30 is pretty high, in fact.

[25] See section 4.5.1.2 for more on that.

[26] I.E. the single quote inside double quotes doesn't need escaping but it did inside single quotes. Likewise for double quotes inside either single or double quotes respectively.

But I'd err here on the side of using Student as the code inside relies heavily on the methods from that class. I usually leave auto for situations where there is a bulky size_type or the like involved. (We'll see more crazy situations soon enough!)

There are, however, a couple of other things we can learn while we are here. Let's first just look at the table loop itself. We've used a range-based for loop, but what if we'd gone traditional? It's head would have looked like this:

```
for ( Student::GradePos s = 0; s != roster.size(); ++s )
```

Note how we had to use Student as the scope for the GradePos using alias. I mentioned this earlier, but it is so much more impactful to see it in live code. All-in-all, the range-based for was still the best choice.

Within the loop we printed each Student's name, but what if that was causing our table to be misaligned? Perish the thought! We could make it less likely by using simpler parts of the name like initials or just first names or some such. Maybe a combination of these.

So we could easily get the initial or first name with these constructs:

```
s.get_name()[0]
s.get_name().substr(0, s.get_name().find_first_of(" \t"))
```

But that is a bit messy. We could make a local variable to hold a copy of the name, but storing it as two variables — one inside and one outside the Student — would waste space. So, instead we can use a reference variable:

```
const string & whole = s.get_name();
whole.substr(0, whole.find_first_of(" \t"))
```

Here, whole is made as a reference to the get_name result rather than a copy of it. Then this is utilized to pull out the first name in a subsequent operation.[27]

The main difference of the reference variable as compared to a reference argument is that it can never be reset to refer elsewhere. A reference argument gets set every time you call the function.

But, like a function, scopes like loops and branches get reset as you enter them. A particularly good place to use one, therefore might be inside our for loop:

```
for ( const Student & s : roster )
{
    const string & whole = s.get_name();
    cout << whole.substr(0, whole.find_first_of(" \t")) << "\t\t"
         << s.get_average() << "\t\t"
         << s.get_grade_count() << "\n";
}
```

Pulling out the last initial is left as an exercise for the interested reader. But I'll give you a hint: find_last_of.

6.7 Parallel Containers

Parallel containers are a way to link data together with a simple subscript. We mentally place two or more vectors side-by-side like so:

[27] Yes, the substr is making a copy of part of the name, but at least it isn't the whole thing.

```
    +-----+         +-----+         +-----+                 +-----+
    |     |         |     |         |     |                 |     |
    +-----+         +-----+         +-----+                 +-----+
    |     |         |     |         |     |                 |     |
    +-----+         +-----+         +-----+                 +-----+
    |     | <--i--> |     | <--i--> |     | <-- ...i... --> |     |
    +-----+         +-----+         +-----+                 +-----+
    |     |         |     |         |     |                 |     |
    +-----+         +-----+         +-----+                 +-----+
    |     |         |     |         |     |                 |     |
    +-----+         +-----+         +-----+                 +-----+
    |     |         |     |         |     |                 |     |
    +-----+         +-----+         +-----+                 +-----+
    |     |         |     |         |     |                 |     |
    +-----+         +-----+         +-----+                 +-----+
```

As a simple example to give you a clearer idea, let's look at storing times in a pair of parallel `vector`s:

```
  h +------+------+------+------+------+------+------------+
  r |  12  |  13  |  15  |   9  |   5  |  10  |    ....    |
  s +------+------+------+------+------+------+------------+

  m +------+------+------+------+------+------+------------+
  i |  45  |   2  |  18  |  27  |  42  |  53  |    ....    |
  n +------+------+------+------+------+------+------------+
```

Here we have two `vector`s named `hrs` and `min`. These were probably declared like so:

```
vector<short> hrs, min;
```

Then, after filling in `hrs` and `min` to resemble the above sample or with whatever we need, we can display the times like so:

```
for ( vector<short>::size_type i = 0; i != hrs.size(); ++i )
{
    cout << hrs[i] << ':' << min[i] << '\n';
}
```

Both containers must stay the same length at all times and information in like offset positions must be about the same entity from the problem domain.

This practice is steeped in tradition and is still in use today, but it can be fraught with peril! If you ever make a change to one container without making an analogous change to the other container, they are no longer parallel! The data items in the one container are misaligned with those in the other container and you may never get them back into alignment again!

You will also find this technique used in older code that you'll be expected to maintain without updating it. It shouldn't really be considered in a modern design.

Still, if you are careful, it can be done. . .

6.7.1 Grades Example

As implied above, we're going to adjust our `Student` `class` to allow the teacher to use weighted `grades`. These weights are entered alongside the scores and used when calculating the average. To view the code, please see the website here as it is a bit long for this text.

Now, we can see the declarations of the parallel vectors on line 11. They are kept in sync all through their lives from initialization in the constructors to additions during a run to the end of the object's scope. Let's look at this process.

In the constructors on lines 17-25, grades and weights are constructed side-by-side. This keeps them in sync as a new object is constructed. (The final constructor delegates to the default one so it works here, too.)

In the input function on lines 33-43, two doubles are read in for each assignment and add_grade adds both the supposed grade and its associated weight. This parallels[28] the output function (lines 45-54) which displays grades[g] and weights[g] as the loop goes through all elements in them.

One other point here is that I went ahead and simplified the input and output functions to allow us to inline the input function. There is no longer a "No grades\n" phrase printed on the grades line when no grades have been entered — that line is simply not printed at all. This was a somewhat troubling design to begin with and the process works much cleaner without it.

This also makes things work out better for transitioning to file storage when we get there (chapter 7). And it doesn't look that bad on-screen, either. Just a little less friendly. But no one ever said that class-provided i/o would be pretty. We just said we'd print the entirety of the held data and we hold no grades in this situation.

Even though they are kept in sync, they can still be accessed separately as seen in lines 61-69. This access doesn't change either vector and so is safe.

set_grade on lines 82-96 takes both a grade and a weight as its parameters. The weight, if not specified by the caller, defaults to 1.0. This is fine and actually makes the get_average work just like an unweighted system if that's what the teacher wants. But either way the element updates happen back-to-back to keep things aligned.

Finally, add_grade on lines 98-104 takes the two necessary data and does push_backs to both vectors to keep them in sync. Once again the weight will default, but that's fine as long as both of our vectors stay aligned and the same length all their lives.

get_average (lines 106-115) also has an upgrade. It takes a bool parameter and uses it to decide whether to weight the grades during the averaging. If averaging is indicated, we multiply the grade by the associated weight. If not, we multiply it by 1.0. And at the return we divide either by the sum of the weights or by the number of grades again at the indication of the bool parameter. The only inefficiency is that we always sum the weights whether this sum is needed or not.

Normally we make constants for bool parameters to avoid their magical literals. This time is no exception. They just follow the class instead of preceding the function as usual. The reason is that making constant members of a class is a little tricky. And if I had put these above the class definition they wouldn't have made as much sense being so out-of-context. The drawback is that we couldn't use the proper constant to initialize the default parameter to get_average. *shrug*

The main is virtually unchanged. We just added a column to our table to report both the weighted and unweighted averages.

Unlike the Student::input function, read_student (lines 152-175) is using the fail loop technique to read in the grade/weight pairs. This way the user can enter just an invalid value for the grade and not have to enter in anything for the weight. (This isn't technically a problem, but it is an opportunity to see this idea in action.)

[28] In a different way than the vectors are parallel, of course.

6.7.2 Grades Revisited

But all that parallelism is quite tedious. And it can get worse! What if we wanted to sort the Students grades into non-ascending order? We'd have to take both vectors through the process and swap the aligned positions in both when a move was indicated or the alignment would be kaput! What if an assignment is being dropped and we remove it from grades but not from weights? *BOOM* These kinds of things are easy to forget and really mess things up.

To avoid these problems we simply don't use parallel containers any more. Instead we use a nested struct or class and make a single vector of that base type. This collects the 'parallel' data together in a single place and makes sure we can't lose a piece or misplace a piece during insertion, removal, or sorting operations.

Take a look at this technique in action in the example here.

In this code you can see this technique in action on lines 34-42. Here I've chosen to use a struct even though this makes the data public by default. This struct is used only inside the Student class and so we won't expose the public data to any other programmers. We just have to be careful ourselves not to make bad changes to the data in our class code.

We've also added to the struct a constructor to ease building these entities during insertion and update of vector entries. This constructor also does double-duty by having parameters that all default. This allows it to act as, in fact, three constructors in one! It can take two parameters, one parameter, or even 0 parameters. So it is two overloaded constructors and a default constructor all rolled into one. Pretty slick, eh?

We see the usage of this constructor on lines 123 and 136 in the set_grade and add_grade functions respectively. We anonymously construct a WeightedGrade object and either assign it to a grades position or push it onto the back of the vector.

There is a note on line 136 about a new function called emplace_back. This function will take the parameters for a base type constructor and construct the newly added element in place rather than us creating the object and then having it copied to the new slot. This can be more efficient when it is possible to do and so is highly recommended.

We also see the effects of the struct idea in output, in the grade and weight getters, and in get_average on lines 83 & 86, 99, 104, and 158-9 respectively. Here we see that we use the dot operator right against the subscript brackets like we did with a class base type but this time we have member variables because of the public nature of the struct.

Also I've moved the constants for the get_average argument inside the Student class on lines 150-1. This makes sure the programmer using our class won't have a name conflict with our choice of constant names. It even gives us a chance to simplify the names to make them read better. The only cost is that the programmer using our class will now have to access their names with Student:: scope overrides.

But what is with that static keyword on them? This is so that all objects instantiated of the Student class will share these same definitions of these constant values.[29] If we left off the static keyword, they would just be constant members of the class and each object instantiated of the class would have its own versions of them.

Wait, what? Well, think of a class to represent a State with a constant member sales_tax_rate. This member wouldn't be a shared constant — not all states have the same tax rate, right? It would be a constant that each object would initialize on its own — to a rate proper for its local economy and budget needs.

Further, such a plain constant member could not be initialized at its declaration because not all objects would share the same value for the constant. Therefore, it must be initialized in the member initialization list of the constructors. This happens before the object is fully fleshed out and before it can

[29] static doesn't have to be just for constants, however. We'll use it for other things in future designs.

start to use these values in other code. Once you hit the body of the constructor, it's too late! There code could try to use the constant member before it got initialized! Thus it has to be initialized in the member initialization list.

What if your constant is larger than a built-in type — some class object that needs to remain constant? Then we need to separate initialization from declaration! Freaky for a constant, I know, but that's what the standard requires in this case. So declare the constant inside the class definition like so:

```
class Class
{
    // ...
    static const ________ const_member;
    // ...
};
```

(The run of underscores should be filled in with the actual type of the constant member.) And then initialize it in the implementation file (or just outside the class if you haven't put it in a library — yet):

```
const ________ Class::const_member{....};
```

(The .... is a placeholder for the constructor parameters. Fill them in as necessary for the underscore-replacement type.) Note that since it is outside the class definition, you need to mark it with the class scope or the compiler will think it is another global constant with a similar name...

6.8 More Than One Dimension

Consider if the heights vector we created earlier were not a collection of students' heights but rather a collection of the height measurements for a [single] grade school student's 'grow the plant' project. Why a vector? So that we have a place to store our many measures of the same entity that we collect over the course of the project. This process of collecting individual measures over a period of time also provides us with a need for one-dimensional storage.[30]

```
vector<double> heights;  // collect plant measures during semester
```

So what about the other students in the class? Don't we normally take down their plants' measurements, too? Yes, a teacher would normally record all of these measurements on a huge sheet of construction paper on the wall above the table where all the plants were growing. A chart much like this:

Student	Week 1	Week 2	Week 3	Week 4	Week 5	Week 6	Week 7	...	Total
Ng	.4								
Jesse									
Blondie							2.		
Dagwood									
Beetle									
Doogie									
Jasmine									
Total									

[31]

[30] For the curious, yes, a single simple variable is considered zero-dimensional — a data 'point'. No, I don't think it's a pun. It is just what we call it. *sigh* *shakes head*

[31] Just imagine the rest of it is filled in with numbers.

To emulate this in our program, though, we'd have to use a two-dimensional vector. To achieve this, we'll have to use a vector as the base type of another vector. The code looks something like this:

```
// collect measures during semester
// for all students in class
vector<vector<double> > heights;
```

An Extra Space

What's that extra space between the close angle brackets in the code to the left? That tells the compiler that one template instantiation is closed and the next is beginning. Older compilers needed this space to realize that a >> wasn't extracting data from a stream at that point! Modern compilers no longer need this space, but many programmers still put it there out of habit and a sense of security.

We can visualize this structure (let's cut it to 4 students over 10 weeks of measurements for the sake of brevity) like so:

```
+-----------------------------------------------------+
|    0    1    2    3    4    5    6    7    8    9   |
| +----+----+----+----+----+----+----+----+----+----+ |
| | .4 |    |    |    |    |    |    |    |    |    | | 0
| +----+----+----+----+----+----+----+----+----+----+ |
+-----------------------------------------------------+
|    0    1    2    3    4    5    6    7    8    9   |
| +----+----+----+----+----+----+----+----+----+----+ |
| |    |    |    |    |    |    |    |    |    |    | | 1
| +----+----+----+----+----+----+----+----+----+----+ |
+-----------------------------------------------------+
|    0    1    2    3    4    5    6    7    8    9   |
| +----+----+----+----+----+----+----+----+----+----+ |
| |    |    |    |    |    |    | 2. |    |    |    | | 2
| +----+----+----+----+----+----+----+----+----+----+ |
+-----------------------------------------------------+
|    0    1    2    3    4    5    6    7    8    9   |
| +----+----+----+----+----+----+----+----+----+----+ |
| |    |    |    |    |    |    |    |    |    |    | | 3
| +----+----+----+----+----+----+----+----+----+----+ |
+-----------------------------------------------------+
```

The .4 is at position 0,0. But we'd actually access this with heights[0][0] in our code. The first subscript operation specifies the row (element of the outer vector) retrieving the entirety of the inner vector at that location. And **then** the second subscript specifies the column (element of the inner vector). Likewise, the 2. value is at heights[2][6].

Extending each 'inner' vector's borders out to the corners of the 'outer' vector's containing element, we get a more compact way to render this same layout:

```
   0    1    2    3    4    5    6    7    8    9
+----+----+----+----+----+----+----+----+----+----+
| .4 |    |    |    |    |    |    |    |    |    | 0
+----+----+----+----+----+----+----+----+----+----+
|    |    |    |    |    |    |    |    |    |    | 1
+----+----+----+----+----+----+----+----+----+----+
|    |    |    |    |    |    | 2. |    |    |    | 2
+----+----+----+----+----+----+----+----+----+----+
|    |    |    |    |    |    |    |    |    |    | 3
```

```
+----+----+----+----+----+----+----+----+----+----+
```

Two ideas to keep in mind as the dimensionality of your problem grows (pardon the pun):

- keep your subscript use consistent throughout the program
- keep your inner dimension a consistent `size`/`length` so that the overall structure is rectangular (especially if you are processing the data column-wise)

6.8.1 2D Initialization

Of course, if you know the content, you can use nested initialization lists like so:

```
using vec1D = vector<double>;
using vec2D = vector<vec1D>;

vec2D the_data = { {  1,  2,  3,  4,  5 },
                   {  6,  7,  8,  9, 10 },
                   { 11, 12, 13, 14, 15 } };
```

Note how nicely the `using` aliases hide the 2D nature and they'll even give you help with `size_types` later!

6.8.2 Shaping It Later

How do you make the structure rectangular in the first place? You could `resize` each dimension appropriately:

```
heights.resize(4);
for (auto & row : heights)
{
    row.resize(10);
}
```

This will shape our `heights vector` from above to hold 4 rows of 10 elements each, just as we'd drawn before. (This is, of course, `heights.size()*heights[0].size()` or 40 elements.)

If you somehow knew the shape of the `vector` structure — say it was `my_rows` by `my_cols`, for instance — before even declaring it, you could use this shorter form instead:

```
vector<vector<double> > heights(my_rows, vector<double>(my_cols));
```

Here we use an anonymous `vector` of length `my_cols` to be the model element for the `my_rows` elements of the outer `vector`.

But how would you gather that info before you declared your `vector` variable in the first place? Maybe a support function could help:

```
    inline
vector<vector<double> > make2D(vector<vector<double> >::size_type rows,
                               vector<double>::size_type cols,
                               double elem_to_copy = 0.0)
{
    return vector<vector<double> >(rows, vector<double>(cols, elem_to_copy));
}
```

Note that the size_type for the rows parameter is that from a nested vector type but the size_type for the cols parameter is from a single vector type.

This could be used at any time in the code like so:

```
heights = make2D(my_rows, my_cols);
```

But it is only good for two-dimensional vector structures containing doubles.

If we template this, many callers would need to do an explicit instantiation of make2D like so:

```
heights = make2D<double>(my_rows, my_cols);
```

(It isn't all callers, because we'll be including a defaulted third parameter that will help the compiler deduce the type when present.)

So what does the template itself look like, you say? Like this:

```
template <typename Base_Type>
    inline vector<vector<Base_Type> >
make2D(typename vector<vector<Base_Type> >::size_type rows,
       typename vector<Base_Type>::size_type cols,
       const Base_Type & elem_to_copy = Base_Type{})
{
    return vector<vector<Base_Type> >(rows, vector<Base_Type>(cols,
                                                        elem_to_copy));
}
```

Note the Base_Type{} as the default value for the initial elements. This default constructs a temporary element and uses it to default the third argument.

6.8.3 Two Dimensional Processing

Once you get a two-dimensional structure set up, you'll want to fill it with data, print it back out nicely, and even do stuff with it in between. Most of this 'stuff' is going to be math-based. But not all...

6.8.3.1 Input

We'll want to read in data from the user to fill the 2D structure we've formed. For instance, we could read in all the elements from the user into our heights vector like this:

```
for (vector<vector<double> >::size_type row = 0;
     row != heights.size(); ++row)
{
    for (vector<double>::size_type col = 0;
         col != heights[row].size(); ++col)
    {
        cin >> heights[row][col];
    }
}
```

In fact, once you've constructed or resized your vector to be the proper rectangular shape, you can do all further processing with for loops similar to those used just now for input. The outer for loop goes from row to row. And for each such row, the inner for loop walks from element to element along that row. Therefore, inside this inner for loop, the double subscript [row][col] will access the next element to be processed.

But a pair of range-based for loops might prove simpler:

```
for (auto & row : heights)
{
    for (auto & elem : row)
    {
        cin >> elem;
    }
}
```

Here we reference each row from the heights vector with the outer range-based for loop. Then we take the individual elements from each row in turn — once again by reference — and read data into it. We didn't have to hassle with size_types or incrementing or any of that stuff.

6.8.3.1.1 Unknown Dimension Sizes

But what if we hadn't constructed/resized the vector before input? Could we still input the data somehow? Sure! We'd have to use indeterminate loops, of course, like a while or do. We'd have to use push_back to put in new elements. And we'd need a helper one-dimensional vector to house each row of data before we add it to the 2D structure. It could look something like this:

```
cout << "\nPlease enter your data with non-numbers at the ends of the first"
        " and last rows.\n";
cin >> n;
while ( ! cin.fail() )  // non-number stops row input
{
    row.push_back(n);
    cin >> n;
}
cin.clear();
cin.ignore(numeric_limits<streamsize>::max(),'\n');
data.push_back(row);
cin >> n;
while ( ! cin.fail() )  // non-number stops input of rows
{
    row.clear();
    while ( row.size() < data[0].size() &&  // each has same size
            ! cin.fail() )         // each row could be the last
    {
        row.push_back(n);
        cin >> n;
    }
    // if row.size() < data[0].size() ?
    //     pad out with 0s!
    //         row.resize(data[0].size());
    data.push_back(row);                    // add row to 2D structure
}
cin.clear();
cin.ignore(numeric_limits<streamsize>::max(), '\n');
```

Here we read one row at a time and allow the user to cause a failure on the first and last rows to signal how long the rows are and how many rows there are respectively. After each row is read it is put onto the end of the 2D structure and that temporary vector is cleared before being used to read the next row.

Note particularly the comment after the nested while loop about the size of the last input row being less than that of the rest. If this happens, most applications will call for you to pad out that row to be of equal length to the rest. You can do that with a simple call to resize which will default construct all new elements.

6.8.3.2 General Algorithms

Many algorithms extend naturally to the second dimension. We'll take the example of finding the largest element in the 2D structure. The approach is to reuse our 1D largest finding code in a process quite similar to that original algorithm. Here, for reference, is the one-dimensional largest finding code:

```
template <typename BaseT>
    inline typename vector<BaseT>::size_type
largest(const vector<BaseT> & vec, typename vector<BaseT>::size_type from = 0)
{
    auto max = from;  // assume first is largest
    for ( auto at = from + 1; at < vec.size(); ++at )  // go thru ~rest~ of list
    {
        if ( vec[at] > vec[max] )    // this one bigger than largest?!
        {
            max = at;           // re-think our decision!
        }
    }
    return max;
}
```

To take this to the second dimension, we'll make a new loop similar to this one and at each iteration call on this function to help update our idea of the largest data's position:

```
template <typename BaseT>
inline void largest(const vector<vector<BaseT>> & vec,
                    typename vector<vector<BaseT>>::size_type & max_row,
                    typename vector<BaseT>::size_type & max_col)
{
    using rowSize = typename vector<vector<BaseT>>::size_type;
    max_row = 0;  // assume it is in the first spot -- upper-left corner
    max_col = 0;
    if ( ! vec.empty() && ! vec[0].empty() )  // if there are any data...
    {
        for ( rowSize row = 0; row != vec.size(); ++row )
        {
            auto max_in_row = largest(vec[row]);
            if ( vec[row][max_in_row] >             // new largest value?
                 vec[max_row][max_col] )
            {
                max_row = row;      // remember where we found it!
                max_col = max_in_row;
            }
        }
    }
    return;
}
```

The only thing we didn't do here that we did do in the 1D case is optimize away testing the assumed maximum position. In the one-dimensional case we could do that because we just started testing after that and moved along the single dimension. In 2D we can't do this because we'd have to skip the assumed position in only the first row and not all successive rows. This would lead to lots of extra checks about what position we are in and it isn't really worth all the hassle.

Other 1D algorithms extend nicely to 2D as well: linear search, smallest finding, etc. Even if the algorithm isn't as smooth as these, it will almost certainly reuse its 1D counterpart in some way making it that much easier to develop.

6.8.3.2.1 Playing with Columns

Unfortunately, the same ease of retrieval is not true for columns. You cannot use simple subscripting to access a column of a 2D structure. Instead, you must use a loop to pull out the individual elements and put them into a new vector:

```
col_vec.clear();
for ( vector<vector<double>>::size_type row = 0;
      row != heights.size(); ++row )
{
    col_vec.push_back(heights[row][target_col]);
}
```

Now col_vec is a copy of the target column from the 2D structure heights. Or we could do it with a range-based for loop:

```
col_vec.clear();
for ( const auto & row : heights )
{
    col_vec.push_back(row[target_col]);
}
```

Once extracted like this, we can send the column vector to any 1D processing function we like. If it is a function that changes the content of the vector, we'll have to put the changed data back into the 2D structure:

```
for ( vector<vector<double>>::size_type row = 0;
      row != heights.size(); ++row )
{
    heights[row][target_col] = col_vec[row];
}
```

This can't be done with a range-based for since we need the index to work in parallel between the two columns.

6.8.3.3 Nice Output

To achieve nice output of a 2D structure, we actually need to know at least the largest value in it! This is because 'nice' output of such data requires consistent column widths so everything lines up. To find these widths we need to know how big the data in them are. This could be determined on a column-by-column basis, but each column might end up with a different width! To avoid this, we base the width on the largest thing in the whole vector.[32]

```
inline streamsize int_width(double x)
{
    short sign = static_cast<short>(x < 0.0);
    x = fabs(x);
    return x > 0.0 ? static_cast<streamsize>(floor(log10(x)) + 1 + sign)
                   : 1;    // 0 is 1 integer digit wide
}

void display(const vector<vector<double>> & vec)
{
    double d = largest(vec);  // and smallest?!
    streamsize wide = int_width(d) + 1;   // + desired precision + 1 (for the .)
    cout.precision(0);
    cout << fixed;
    for ( const auto & row : vec )
    {
        for ( auto elem : row )
        {
            cout.width(wide);
            cout << elem << ' ';
        }
        cout << '\n';
    }
    return;
}
```

Here we've embedded our formula for the width of an integer (section 3.7.2.3) into a helper function. The only adjustment here is to make it work for all double values instead of just non-negative ones. The trick was to store a typecasted bool into a short. This makes it 1 if the value is negative and 0 if not. Then this is added to the prospective width to be returned. Also we took the absolute value of the double before taking its logarithm.

The changes noted in the comments allude to the possibility of a desired precision on a floating-point bit of data with actual decimal places. Here we've used a precision of zero to round the values to the nearest integer. If you want a certain precision, you should make it a parameter to the function and handle it in a reusable way.

6.8.3.4 An Example

Although we haven't done much so far, it will be illustrative to look at a whole example. I've placed it on a website as usual so you can download it and look at it alongside these notes.

Most of the code is the same as above, but I've made a couple of adjustments for the full program. For instance, I added using aliases for the vectors on lines 23-27. This simplifies changing the base type for the program's vectors if necessary during development or maintenance. I also added a comment

[32] If you are expecting negative values, you should also find the smallest thing in the vector because negatives can be wider due to their leading minus symbol.

above the int_width function to try and help the coder that follows me understand the use of the log10 function here.

I also broke the reshaping code and the two input methods out into separate functions for easier reuse in later programs. These could have been templated but I left that as an exercise for the reader. So too is putting all of these helper functions into a nice library for easy reuse. (If you do put these routines into a library, make sure you either take the using directives with them or — even better — template them first.)

Finally, I added a defaulted string argument to the display function to act as a title printed above the vector contents. If you want to add display to your library, it won't be easily templated if you are adding the argument for precision. After all, if the base type were integers or strings you wouldn't need that code. And if they were strings, you wouldn't use the int_width function or largest function. You'd use a largest function that found the longest length of the container's strings and then you'd use that directly in the wide calculation. This would require a few specializations or overloads of your templates.

6.8.3.5 Math Usage

As we said above, the algorithms we might want to use with 2D structures are mostly math related. Mathematicians often group numbers into 2D structures they call matrices (matrix singular). Some common uses are:

- equality testing of two matrices
- adding of two matrices
- product of a scalar and a matrix
- product of a matrix with a row or column 'matrix'
- general matrix multiplication
- solution of linear systems by row operations
- multiplicative inverse of a matrix
- regression modeling

A scalar is a term for a single number used in matrix focused discussions. (Weird, right?)

We won't go into anything beyond matrix multiplication here, but you can find more on these topics in a typical linear algebra text or an intermediate to advanced statistics text.

6.8.3.5.1 Equality of Matrices

For two matrices to be equal, they must first be the same shape and then each of their elements must be equal position-by-position. We'd code such a test like so:

```
using vec1D = vector<double>;
using vec2D = vector<vec1D>;
vec2D A, B;
bool equal;
// fill in A and B somehow
equal = false;             // assume they are not equal
if ( A.size() == B.size() && A[0].size() == B[0].size() )
{
    equal = true;     // hopefully they are equal
    vec2D::size_type row = 0;
    while ( row != A.size() && equal )
    {
        vec1D::size_type col = 0;
        while ( col != A[row].size() && equal )
        {
            equal = equal && A[row][col] == B[row][col];
            ++col;
        }
        ++row;
    }
}
```

First we check the shapes of the matrices and then we loop through the elements position-by-position until they are done or not `equal`. Note how we accumulate the equality with an `&&` inside the inner `while`. If the old value was `true` and the new test is also `true`, `equal` will come out `true`. If one of these pieces is `false`, then `equal` will change to `false`.[33]

If the data inside the matrices are `double`s instead of integers, we would need to use an absolute difference check instead of just raw `==`, of course. And if they were of some type like `Die`, we'd have to call the `isSame` function instead of `==`. You get the picture.

6.8.3.5.2 Adding of Matrices

As with equality testing, two matrices to be added need to be the exact same shape. Then their elements are added together position-by-position and the total stored into a new matrix. It can be coded like so:

```
C.resize(A.size());
for ( vec2D::size_type row = 0; row != A.size(); ++row )
{
    C[row].resize(A[row].size());
    for ( vec1D::size_type col = 0; col != A[row].size(); ++col )
    {
        C[row][col] = A[row][col] + B[row][col];
    }
}
```

Here I've assumed the shape check and focused on the adding itself. Note how we `resize` the result matrix as we go. We could have called our `reshape` function beforehand, but this is often just as easy.

6.8.3.5.3 Products of Matrices

There are three types of products involving matrices. We'll talk about each in turn.

Scalar Times Matrix: Here there are no shape concerns since a scalar is a simple number. We just multiply every element by the same number throughout the matrix:

```
C.resize(A.size());
for (vec2D::size_type row = 0; row != A.size(); ++row)
{
    C[row].resize(A[row].size());
    for (vec1D::size_type col = 0; col != A[row].size(); ++col)
    {
        C[row][col] = 16 * A[row][col];
```

[33] Although not strictly necessary here, this `bool` accumulation trick can come in handy, so why not learn it now?

```
    }
}
```

I've again shaped C as I went. I chose the 16 rather arbitrarily.

Matrix Times Row/Column Matrix: Here we are multiplying a 2D matrix by a 1D matrix. Well, in math they call them either $n \times 1$ or $1 \times n$ shaped matrices depending on if it is a single column or row respectively. But we store it as a 1D `vector`. For this to work, the length of the 1D must equal the dimension of the 2D that abuts it in the product. That is, if we are multiplying a 1D times a 2D, the length of the 1D must equal the row count of the 2D. But if we are multiplying a 2D times a 1D, the length of the 1D must equal the column count of the 2D.

As you can tell already, this gets a little tricky. Let's visualize with some help. Here is a row matrix times a matrix:

$$\begin{bmatrix}1 & 2 & 3\end{bmatrix}\cdot\begin{bmatrix}1 & 2\\ 3 & 4\\ 5 & 6\end{bmatrix}$$

Here we have the length of the row is 3 and the height of the matrix is also 3. The other dimension of the 2D matrix doesn't matter.

Here is a matrix times a column matrix:

$$\begin{bmatrix}1 & 2 & 3\\ 4 & 5 & 6\end{bmatrix}\cdot\begin{bmatrix}1\\ 2\\ 3\end{bmatrix}$$

Here the width of the matrix is 3 and the height of the column is 3. The other dimension of the 2D matrix doesn't matter.

The way this works is we walk along the commonly sized dimension of the two matrices and multiply element-wise and add the products. This sum, then, becomes an entry in the result matrix. For the examples above, we have:

$$\overset{1\times3}{\begin{bmatrix}1 & 2 & 3\end{bmatrix}}\cdot\overset{3\times2}{\begin{bmatrix}1 & 2\\ 3 & 4\\ 5 & 6\end{bmatrix}} = \overset{1\times2}{\begin{bmatrix}1\cdot1+2\cdot3+3\cdot5 & 1\cdot2+2\cdot4+3\cdot6\end{bmatrix}} = \overset{1\times2}{\begin{bmatrix}22 & 28\end{bmatrix}}$$

And:

$$\overset{2\times3}{\begin{bmatrix}1 & 2 & 3\\ 4 & 5 & 6\end{bmatrix}}\cdot\overset{3\times1}{\begin{bmatrix}1\\ 2\\ 3\end{bmatrix}} = \overset{2\times1}{\begin{bmatrix}1\cdot1+2\cdot2+3\cdot3\\ 4\cdot1+5\cdot2+6\cdot3\end{bmatrix}} = \overset{2\times1}{\begin{bmatrix}14\\ 32\end{bmatrix}}$$

The code for this would depend a bit on whether we were doing row times matrix or matrix times column. For the former, we would have:

```
if ( row.size() == A.size() )  // 'inner' dimensions match
{
    C.resize(A[0].size());
    for ( vec1D::size_type col = 0; col != C.size(); ++col )
    {
        C[col] = 0;
        for ( vec1D::size_type common = 0; common != row.size(); ++common )
        {
            C[col] += row[common] * A[common][col];
        }
    }
}
```

The code for a matrix-column product would be similar but backwards a bit.

But since these are special cases of general matrix multiplication where one of the dimensions is 1, we don't normally code for them.

Matrix Times Matrix: Here we are talking about two matrices with 2D shape each. The entries in the result are the sums of products along the common inner dimension the two original matrices share. That is, you can only multiply when the number of columns in the first matrix equals the number of rows in the second matrix. The result matrix has the number of rows of the first matrix and the number of columns of the second matrix.

That is:

$$\overset{m\times p}{C} = \overset{m\times n}{A} \cdot \overset{n\times p}{B}$$

The general formula for the elements of C is:

$$c_{i,j} = \sum_{k=0}^{n} a_{i,k} \cdot b_{k,j}$$

Here n is the inner dimension and the lowercase letters with double subscripts represent elements within capital named matrices.

In code this looks like:

```
if ( A[0].size() == B.size() )
{
    C.resize(A.size());
    for ( vec2D::size_type row = 0; row != C.size(); ++row )
    {
        C[row].resize(B[0].size());
        for ( vec1D::size_type col = 0; col != C[row].size(); ++col )
        {
            C[row][col] = 0.0;
            for ( vec2D::size_type common = 0; common != B.size(); ++common )
            {
                C[row][col] += A[row][common] * B[common][col];
            }
        }
    }
}
```

The only unfortunate thing is that we are relying on the vec2D and vec1D size_types to be the same. There is a footnote in the standard that says container size_types should all be the same, but it is only a suggestion in a footnote — not a hard and fast rule. I try to avoid using the common suggested data type size_t because of this. But some situations call for mixes — note parallel vectors before, for instance (section 6.7). We just must do our best.

An Example: Here is a driver program for these ideas on the website. I've simply put the codes in functions and called them from a driver main. I've used a nice technique, though, to initialize the A and B matrices that makes checking the results of addition and scalar multiplication easier (lines 134-151). This code makes the numbers in one matrix complement those in the other matrix — off of 16.

6.8.3.6 Another 2D Structure

One of the simplest 2D structures you can create and which is quite useful is a vector of strings:

```
vector<string> blah;  // each element of the vector is a string
                      // a string is a container/collection of chars
```

A vector of strings is also easier to initialize to particular values (an inner vector would need individual push_backs or resize'ing with future subscripting):

```
blah.push_back(" 1 | 2 | 3 ");
blah.push_back("---+---+---");
blah.push_back(" 4 | 5 | 6 ");
blah.push_back("---+---+---");
blah.push_back(" 7 | 8 | 9 ");
```

6.8.3.7 Parallel 2D Structures

The idea here is having either two 2D structures parallel to one another along a single dimension or having one or more 1D structures parallel to a 2D structure along some dimension. Here is a diagram of the latter:

```
          +----+----+----+----+----+----+----+----+----+----+
          |    |    |    |    |    |    |    |    |    |    |
          +----+----+----+----+----+----+----+----+----+----+
             0    1    2    3    4    5    6    7    8    9
 +----+   +----+----+----+----+----+----+----+----+----+----+   +----+
 |    | 0 |    |    |    |    |    |    |    |    |    |    | 0 |    |
 +----+   +----+----+----+----+----+----+----+----+----+----+   +----+
 |    | 1 |    |    |    |    |    |    |    |    |    |    | 1 |    |
 +----+   +----+----+----+----+----+----+----+----+----+----+   +----+
 |    | 2 |    |    |    |    |    |    |    |    |    |    | 2 |    |
 +----+   +----+----+----+----+----+----+----+----+----+----+   +----+
 |    | 3 |    |    |    |    |    |    |    |    |    |    | 3 |    |
 +----+   +----+----+----+----+----+----+----+----+----+----+   +----+
             0    1    2    3    4    5    6    7    8    9
          +----+----+----+----+----+----+----+----+----+----+
          |    |    |    |    |    |    |    |    |    |    |
          +----+----+----+----+----+----+----+----+----+----+
```

The indices are listed to help you visualize alignments in indexing.

Consider, if you will, that the top vector is filled with names for assignments in a class. The left-side vector is likewise filled with names of students within the class. The 2D structure in the middle is filled with the students grades on the various assignments. The right-side vector, then, holds averages for each student over all assignments. And the bottom vector holds averages for each assignment across all students.

This kind of parallelism is difficult to remove with the use of a struct or class base type without causing redundancy in data storage. We usually just practice great care.

6.8.4 Beyond Two Dimensions

Why go beyond two dimensions? What is storage beyond 2D? Well, we got our first dimension by collecting a scalar (0D) across time or individuals. (We could have also done a 1D model of some situation.) We got our second dimension by collecting such collections across multiple individuals (students, particularly). (We could have also done a 2D model of some situation.)

A third dimension can come from either collecting a 2D model over the course of time or individuals (or from an actual 3D model of some situation). And a fourth can easily come from collecting a 3D model over time/individuals.

Also, from the examples we've seen so far (students by weeks for instance), these dimensions clearly don't have to be physical in nature.

Here are a couple of examples of multidimensional storage designs.

6.8.4.1 Books in a Library

Let's imagine that a vector of strings is a page of text. A collection of pages will form a chapter. A collection of chapters would form a book.

Of course, books are arranged along a shelf and shelves are stacked 4-6 high on a rack. Racks are placed side-by-side along an aisle. And there are many aisles in a single floor of a library (or bookstore).

Of course, someday the library won't have what we want and we'll have to resort to inter-library loan.

Okay. . . that's enough. Finding the book you want could be a ten-dimensional trip!

Here is an attempt to visualize this in code:

```
vector<vector<vector<vector<vector<vector<vector<vector<vector<string>>>>>>>>> libraries;
 //                                                            lines
 //                                                        pages
 //                                                   chapters
 //                                            books
 //                                     shelves
 //                              racks
 //                       aisles
 //                floors
 //         libraries
 // inter-library loan system
```

But we can code this more cleanly with using aliases:

```
using Line = string;
using Page = vector<Line>;
using Chapter = vector<Page>;
using Book = vector<Chapter>;
using Shelf = vector<Book>;
using Rack = vector<Shelf>;
using Aisle = vector<Rack>;
using Floor = vector<Aisle>;
using Library = vector<Floor>;
using InterLibraryLoanSystem = vector<Library>;

InterLibraryLoanSystem libraries;
```

Wow! That's *SOOOOOOOOOOOOOOOOOOOOO* much better!!! And we have all those intermediate types should we need to look at just a `Page` or a single `Book` or the like.

Either way `libraries[5][3][12][6][2][24][10][7][42]` would represent:

```
             The 43rd line
          on the eighth page
          of the eleventh chapter
          of the 25th book
       along the third shelf
          in the seventh rack
       along the thirteenth aisle
          on the fourth floor
          in the sixth library
    of our inter-library loan system.
```

6.8.4.2 Bricks in a Building

Next let's imagine that we have a sonar image of a brick — that's three dimensions of measurement goodness. If we track where this brick is placed within the structure of a house being built. . . and then do it for all the bricks in the house, we'd have a 3D storage full of 3D sonar measurements — a **six** dimensional structure! If we did this for an entire subdivision, we'd have a 2D map of houses which consist of 6D storage structures for a total of **eight** dimensions!

6.9 Wrap Up

In this chapter we've learned a lot about storage of multiple related data. From `array`s for known amounts of data to `vector`s for unknown amounts of data — unknown at design time, that is.

We also talked about many algorithms for processing containers and how to use them to make list management systems for an end user.

Finally we explored using multidimensional containers and their possible uses in data storage design.

Chapter 7

Permanent Storage (aka File Streams)

So far all our data processing has been strictly temporary. The data we worked on was stored in the computer's RAM and went away once the program ended (`return`ed to the OS). It's about time we saved some of this data for the long term. To do that, we'll need to store it in a file on the user's disk (SSD, USB, etc.). In this chapter, we'll be working with files using the concept of streams.

We've been using streams since chapter 2, so it isn't a big deal. But we must get into a slightly different mindset for file streams versus console streams like `cin` and `cout`.

7.1 Getting Up To Speed

We all deal with files on a daily basis: spreadsheets, papers, source code, web pages, applications, configuration, etc. But what is a file? What's it for? Where's it at? What's in it?

7.1.1 What's a File?

First off, there are two basic types of files: text and binary. Binary files are limited to a single system without tedious conversions but are fast with large quantities of data. Text files are portable across platforms but take up more storage and are slower to process.

Even so, text files have become the norm for data processing on a daily basis. The proliferation of JSON, XML, and other textual data formats attests to this. We'll be using text files exclusively in this text.

7.1.2 Where's a File?

When a program uses a file, there are actually two copies of [some of] the file at any given time. There is, of course, the whole thing out on the user's disk. But there is also a bit of it inside a buffer[1] within a stream variable in our program. As with the console streams, this buffer helps adjust for the speed difference between the multi-gigahertz CPU and the somewhat 100 times slower SSD (or an even slower mechanism in some systems).

7.2 Basic File Operations

How do we connect our program to a disk file? How do we use that connection once established? Can anything go wrong while using it? Do we just walk away when we're done with it?

7.2.1 Connecting to Disk Files

Within the realm of text files, there are two ways we use them: input and output. To each of these use-models is assigned a data type: `ifstream` and `ofstream`, respectively. Of course, those types don't come out of nowhere. They are found in the `fstream` library. So make sure you *`#include`* this library when wanting to code file input/output in your program.

Once you know what you want to do with the disk file, you can declare the right type of variable — a file stream variable — and then you are ready to `open` it. Well, almost. You also need to know what the user has called or wants to call the file — its name. The filename, of course, is a `string` and we make a separate variable to hold this information.

This `string` is housing not only the name of the file but also the path to the file. The path can be relative to the current folder/directory or it can be absolute from the top of the drive the user is using. If just the name is given, then it is considered relative to the directory the user ran the executable from.

The basic code is thus:

```
ifstream data_file;
string filename;

cout << "Please enter the name of your data file:  ";
getline(cin, filename);

data_file.open(filename);
```

Note that we use `getline` to read the file's name since most modern users will want to put spaces in their filenames. Also note that it is the stream variable that `open`s the disk file with the given name — not the other way around.

The process for an output file is identical except for the prompt and the type of the stream variable.

7.2.2 Reading from File Streams

To input from an `ifstream`, you can use any tool you've used with `cin` to read from the keyboard: >>, `getline`, `ws`, `peek`, etc:

```
string name;
double salary;
char unit;

```

[1] Much like those of `cin` and `cout`.

```
getline(data_file, name);  // read user's name

// read user's salary
data_file >> ws;
if ( ispunct(data_file.peek()) )    // did they put a dollar sign on it?
{
    data_file >> unit;    // pull in monetary unit
}
data_file >> salary;      // get actual salary
```

Note that it is not necessary to prompt the user when reading from a file stream. The user isn't involved in that transaction and cannot help you there. There's no need to discuss it with them unless you want to report what you read to `cout` afterward.

Also note that we usually code file input to mimic/parallel file output much like we've done when designing basic input/output for a `class`. This helps when a program must later read in what it wrote out on a prior run.

7.2.2.1 File Error Handling

We all know about the streams' `fail` method that reports when a numeric input has failed to translate properly. Well, there are actually three more error states available on streams that we just didn't need before, but it's time to come clean.

Streams not currently experiencing an error are said to be in a `good` state. This is also the name of the method that reports said condition with a `true` result.

The `eof` method reports `true` when a file has attempted to read in past the end of the file. (Turns out there is a special marker there right after the last byte of data and `eof` only `return`s `true` when it has tried to input that marker. Sitting next to it because of a prior read is just **not** enough to do it.)

Finally, the `bad` method reports `true` when a file is experiencing a significant and unrecoverable — often physical/hardware related — error. We can't really deal with these problems, so we also won't be checking for them, either.

In addition to using the above four methods to ask about errors on a stream, you can use the `!` operator on a stream to ask if it is unhappy in any way. This is less typing than asking it if it is `good` and is considered more elegant all around.

Last tidbit: `eof` can't be cleared from `cin`. It is possible to get `cin` into an `eof` state, but you cannot `clear` it like the other states from `cin`. (It can be cleared from a file stream, of course.) How do you set it on `cin`? Well, you type either Ctrl+D on a Unix system like Linux or macOS or Ctrl+Z on a Windows system.[2]

7.2.2.2 The Whole File

So we can read a few bits of data from a file. So what? I've got tons of data in files — megabytes or gigabytes, even! What can my program do about that? We can read it all in!

This is where the `eof` function really shines. It helps us read a file from start to finish. Let's say you had a file full of `double`s and wanted to read them all in for some purpose, how would you do it? The code looks like this:

[2] As usual, press and hold the Control key while hitting the D or Z key to make this happen. Also, it is always `Control` — even on macOS which usually switches to `Command`.

```
double num;
data_file >> num;
while ( ! data_file.eof() )
{
    // use num in some way
    data_file >> num;
}
```

Note the priming style of the loop — reading for both initialization and update of the `data_file` variable whose `eof` state we test in the loop condition.

7.2.3 Writing to File Streams

Similar to input files and `cin`, output files can be written to with any method we've used with `cout`: `<<`, `endl`, `flush`, formatting, etc. (Of course, I still don't recommend `endl`, but it can be used!)

Although we could, we rarely check for errors during a write operation on an output file. This is because these errors tend to be of a `bad` nature and we can't deal with those at all.

As with `class` output methods, never output any labeling along with the data that isn't absolutely necessary. This might include a monetary unit with a salary, parentheses and a comma for a Cartesian coordinate, or a `'d'` and plus/minus for a roll of the dice. Even a single newline after a `string` that contains spacing would be considered necessary.

These ideas also extend to that of paralleling the way data will be/was input. What, after all, if the program is saving a data file now (writing that stuff out to a file) so that on a later run the user can load it (read it in from the file)? If the two processes don't parallel one another, all would be lost!

7.2.4 Disconnecting from Disk Files

Always remember to `close` your `open` file connections when you are done with them! If you don't, you may lose the last buffer contents when the program ends. This could leave your output file shortened or even empty!

Don't forget to `close` your input file streams, too. You won't lose data to buffer truncation, but you will deny the OS necessary resources — file handles. If too many applications lose file handles, the system will be required to reboot.

Persnickety readers of the standard have also forced upon us a situation where it is necessary to `clear` a file stream variable after it is `closed` — at least if we want to [loop back and] use this variable again later in the program.

So, in the code, it would look like this:

```
data_file.close();
data_file.clear();
```

Not hard at all, really. But desperately necessary!

7.2.5 A Whole Example

Here, then, is a whole example program showing these features off for both output and input files. Note that the output file is created for you, but you'll need an input file with data like that shown below the code.

```
#include <iostream>
#include <limits>
#include <string>
#include <fstream>

using namespace std;

int main(void)
{
    double num;
    ifstream data_file;
    ofstream report_file;
    string filename;

    cout << "Please enter the name for your report file:  ";
    getline(cin, filename);

    report_file.open(filename);

    report_file << "\nThis is my report title...\n\n"
                   "And this is a line in my report...\n\n"
                   "And now I'm closing my report with a tag line...\n";

    report_file.close();
    report_file.clear();

    cout << "\nPress <Enter> to try an input file...\n";
    cin.ignore(numeric_limits<streamsize>::max(), '\n');

    cout << "\nPlease tell me the name of your data file:  ";
    getline(cin, filename);

    data_file.open(filename);

    data_file >> num;
    while ( ! data_file.eof() )
    {
        cout << "Read " << num << " from your file!\n";
        // or push it onto a vector or add it to a sum or whatever...
        data_file >> num;
    }

    data_file.close();
    data_file.clear();

    return 0;
}
```

We'll also need a set of data for the input portion to run on. I've called mine `nums.dat`:

nums.dat

```
42 753 -12 0 2
```

Just make a new file in your programming editor, put in some space-separated numbers, and save it with a good name. It doesn't actually need an extension, but I'm traditional that way.

7.3 Intermediate Usage

Those basics will get you through a lot of situations, but if you want to use files very much, you'll run into issues that only these more advanced — and yet not really advanced — tools/techniques can get you out of.

7.3.1 Passing File Streams to Functions

The first thing to remember is our primary design tenet: break a program into smaller and, if at all possible, reusable units of work. And unless the entire use of a file stream is in a single function — not likely — you'll need to pass one to a function from time to time!

When passing a file stream to a function, it must always be passed as a [pure] reference (with a & after its type). This is because, aside from the state reporting functions, almost all other stream methods will change something about the stream object! It might update the buffer position, a state variable inside the object, or even dump/load data to/from the disk file.

So, to avoid accidental use of streams passed by value, all of their copy constructors have been either `delete`d in modern designs or made `private` and broken in older ones. Thus, you cannot take a stream object into a function by value — only by reference. But I felt it bore drilling in.

Also, unless you are actually `open`ing or `close`'ing a file stream inside the function, you might want to use a compatibility type instead of the regular `ifstream` and `ofstream`. A compatibility type is like having a function take a `double` but being able to pass in a `short` instead. This was coercion and we've used it for a long time. But with streams, the types `istream` and `ostream` are similarly compatible with other types of streams.

In fact, you can pass either an input file stream or `cin` to an `istream&` parameter.[3] And you can pass either `cout` or an output file stream to an `ostream&` parameter.

Such a design makes the single function capable of both console and file interactions. Just don't use this technique if your function needs to prompt the user or label something that isn't a report (as opposed to a plain data file).

As a quick example, here is a program that generates a little report (more parameters could be added to make the report more interesting, but let's keep our focus).

```
inline void make_report(ostream & ostr)
{
    ostr << "Now I'm writing my report to the output stream.\n\n"
         << "I wonder if it is a file or the screen...\n\n"
         << "I may never know!\n";
    return;
}

inline void close_file(ofstream & file)
{
    file.close();
    file.clear();
    return;
}
```

[3] Note the reference!

```
int main(void)
{
    ofstream results_file;
    string filename;

    cout << "Enter name for file into which results will be printed:  ";
    getline(cin, filename);

    results_file.open(filename);

    make_report(results_file);

    close_file(results_file);

    cout << "\nBTW, here is a copy of the report I put in your file:\n\n";
    make_report(cout);

    return 0;
}
```

Note how the report is sent to both an `ofstream` and the console (`cout`) with the same function.

There is also a function to close an `ofstream`. This one has to take the stream as a true file stream in order to use `close`. But, thinking about it, we do the same thing to `close` an `ifstream`, don't we? Why not make a `template` out of it?!

```
template <typename FileT>
inline void close_file(FileT & file)
{
    file.close();
    file.clear();
    return;
}
```

Now we could put this in a helper library and reuse it in lots of applications!

7.3.2 Opening Woes

We've heretofore assumed that `open`ing a file automatically succeeded. Nothing could be further from the truth!

Input files can `fail` to `open` if the file doesn't exist under the given name or if the file's permissions don't allow reading data from it. Output files can fail for permissions issues or path issues — not chosen names, however.

To alleviate these simple problems, we can employ a loop to ensure file `open`ing success:

```
ofstream file;
string fname;
cout << "What should the file's name be?  ";
getline(cin, fname);
file.open(fname);
while ( ! file )
```

```
{
    file.close();
    file.clear();
    cout << "\nCannot write to '" << fname
         << "'!!\a\n\nPlease choose "
            "another name (and/or location):  ";
    getline(cin, fname);
    file.open(fname);
}
```

This priming style loop with modifications to the prompt and file type can be used for input file streams, too.

7.3.2.1 Overwriting vs Appending

But there was one more issue with output files. When we `open` an output file stream and that file already exists on the disk, the system feels it should wipe it clean to give you a neat place to work on your output. This is the most dangerous aspect of file `open`ing and involves many decision-making tools to help fix it.

To avoid this problem, we'll have to test-`open` the filename given as an input stream and, if successful, ask the user if they really want to overwrite that file and lose its current content. The alternatives are to append to the file — a special `open` mode, loop back for another name for the file, or to give up and not go on with this operation. That last is sometimes not available depending on the program design and needs.

This looks like this in pseudocode:

```
abort = okay = false;
do
{
    get name
    open for input
    if ( success )
    {
        eek!  overwrite or append or give up or new name?
        abort = gave up;
        okay = overwrite or append;
    }
    else
    {
        yea!  it's safe!
        okay = true;
    }
    close input connection (and clear!)
} while ( ! okay && ! abort );
if ( okay )
{
    open for output (or appending)
    use...
    close (don't forget to clear!)
}
else
{
    sorry...
}
```

So how to open a file for appending? As mentioned, it is a special opening mode. The open method takes a second parameter that defaults for the type of stream to either ios_base::in or ios_base::out. But, if we need to, we can pass our own value for this parameter as ios_base::app to choose appending new data to the end of the current file instead of overwriting it like the out flag would.[4] To use it you just change your normal open like so:

```
file.open(fname, ios_base::app);
```

Now when you write more data to file, you'll add to the user's file instead of destroying the old data first.

7.3.3 Advanced eof

When reading a whole file, the eof loop above is classic. But it won't work in all situations. It doesn't always work for files with no newline at the end of the last line of data, for instance. And there are other styles of eof loop that you'll find on the Internet and think they must be better because they're from the Internet! But they'll hose you in the end, too.

Here's a test program with a chart at the end of it which reports all the combinations and what works/doesn't.[5,6] You'll see from this chart clearly that only the following loop I've nicknamed the 'hacked'[7] eof loop (or 'hacked' while when the eof is understood) will work for all situations.

Here is a variant of this loop for reading data with extraction:

```
data_file >> ws;
while ( ! data_file.eof() )
{
    data_file >> num;
    cout << "number read '" << num << "'\n";
    data_file >> ws
}
```

Technically the ws could be combined with the data read, but I separated them to make the priming more clear. Here we are reading the data with extraction and so we must prime with the extraction of whitespace to make the loop function in all cases.

If we are reading with getline instead, we would instead code this loop like so:

```
data_file.peek();
while ( ! data_file.eof() )
{
    getline(data_file, line);
    cout << "line read '" << line << "'\n";
    data_file.peek();
}
```

Here peek checks for the end-of-file condition for us after the getline grabs its data for processing.

[4] Yes, these are open mode flags much like the formatting flags you learned about so long ago in section 2.6.5.2.

[5] There are also some calls to a method called seekg which is discussed below. Don't mind them until you've read that section.

[6] The directory also contains the test data files used to make the chart. Feel free to make your own and let me know if more data needs to be added to the chart!

[7] I say 'hacked' because we're priming with tricks instead of data reads as usual. See further on. . .

UPDATE! I've recently found a new loop other than the 'hacked' while that will also work in all input situations tested. I've included it in the test program linked above. I call it the OOP while because it actually primes with the input but in such a way that it doesn't miss those edge cases like the traditional eof loop does. It looks like this for extraction:

```
while ( data_file >> num )
{
    cout << "number read '" << num << "'\n";
}
```

This is a little disconcerting at first to see the extraction used as a loop test. But it stems from the chaining nature of extraction and how it always results in the stream just input from. Then, the file stream notices it is in a bool context and reports its error state as in true for all good and false for something's gone wrong.

A similar thing happens with a getline loop as this function also returns the stream it read from:

```
while ( getline(data_file, line) )
{
    cout << "line read '" << line << "'\n";
}
```

Also a little creepy at first, but you get used to it.

7.3.4 Reprocessing a File Stream

Sometimes you'll have read through a file once already and the user will decide to redo that processing with new parameters. We could just close the connection — remembering to clear it as well — and then reopen a new connection to that file. But that is really slow and cumbersome.

It is much easier to just tell the file to start over. How simple? Let's see:

```
filevar.clear();
filevar.seekg(0);
```

Pretty simple indeed!

But what does this new line mean? Well, it speaks of seeking to g of 0. Okay. What's g? Why 0?

Well, the g is short for 'the getting position'. That is, the position at which the file is currently getting information for you. And such positions, it should not come as a great surprise, are numbered starting at 0 rather than 1. So, we are effectively telling the file to start getting data from the very first position rather than wherever it was last.

In modern library implementations, the clear isn't explicitly needed as seekg has been changed recently to automatically call clear for itself. But it is safer to put it in unless you know how up to date your library is.

7.3.5 Sample Code

Anyway, here's a small program fragment with seekg and the 'hacked' eof loop in action:

```
bool again, found;
char yesno;
ifstream contacts_file;
```

```
string filename, contact_sought, contact_from_file;

cout << "What is the name of your contacts file?  ";
getline(cin, filename);

contacts_file.open(filename);
while ( ! contacts_file )
{
    contacts_file.close();
    contacts_file.clear();
    cout << "\n\aUnable to open file '" << filename
         << "' for reading!\n";
    cout << "\nPlease give me a new contacts file name:  ";
    getline(cin, filename);
    contacts_file.open(filename);
}

do
{
    cout << "\nWhat contact name would you like to find?  ";
    getline(cin, contact_sought);

    contacts_file.clear();
    contacts_file.seekg(0);

    found = false;
    contacts_file.peek();
    while ( contacts_file.eof() )
    {
        getline(contacts_file, contact_from_file);
        if ( contact_sought == contact_from_file )
           // or:  contact_from_file.find(contact_sought)
           //          != string::npos
        {
            found = true;
            cout << "Found '" << contact_from_file
                 << "' in file!\n";
        }
        contacts_file.peek();
    }
    if ( ! found )
    {
        cout << "\nI'm sorry we couldn't help you this"
                " time...\n";
    }
    cout << "\nWould you like to search for another contact"
            " in this file?  ";
    cin >> yesno;
    cin.ignore(numeric_limits<streamsize>::max(), '\n');
    again = tolower(yesno) != 'n';
} while ( again );

contacts_file.close();
contacts_file.clear();
```

Here we are reading from a file containing one contact name per line and searching for the user's desired contact within these lines. If found, we report it but if not, we let them down gently. Then there's a yes/no loop to repeat the process — `seekg`'ing back to the beginning for the new sought contact.

7.4 Wrap Up

In this chapter we've learned how to store and retrieve information with disk files. This allows us to keep data between runs of the program in a more-or-less permanent way.

This involves basic stream input and output we've already practiced with the console streams. But it also involved a few new tools like connecting to a disk file and disconnecting from it when we were done.

Later on, we learned to pass file streams to functions for help processing and to reposition the input position to the start of a file. We even took care of some persnickety issues with `open`ing disk files and `eof` looping.

Appendices

Appendix A

Setup

This appendix will help you set up your local environment for compiling C++ programs — an integrated development environment (IDE) — and a connection to a Unix server if your school provides one.

Rather than show a specific setup for each IDE for each platform, however, I'm going to advocate the blanket use of VS Code on all platforms. This will make things easier if you do decide or need to move back and forth between multiple platforms in your programming. It doesn't have its own compiler, though, so we'll have to install one of those alongside it on each platform.

The instructions below are broken down into sections based on your desired operating system. Feel free to scroll down or simply click here to jump straight to your OS: Windows, macOS, Linux, and ChromeOS.

A.1 Windows

You've chosen your OS wisely. This is one of the most popular environments around. It is also very fun to work in as it has the most games available for it! (Wait, isn't that counter-productive?)

As to job potential, there are probably more Windows shops out there than anything else. So don't worry! Your time learning Windows skills will certainly be rewarded.

Now let's get you set up for programming!

A.1.1 IDE

Most people, when working on coding, want a nice integrated development environment where they can do all their tasks at once. This can be done with Microsoft-provided tools, but we'll take a different route to give you a portable experience in case you have to work on other platforms in the future.

A.1.1.1 The MinGW Diversion

We'll start by installing the MinGW compiler utilities via a suite called MSYS2. Scroll down to 'Installation' and download the proffered `.exe` file. Don't worry with the signature check unless you are into that kind of thing. Run the installer, hit 'Next' three times to accept the defaults, and finally 'Install'. Finally click 'Finish' to exit the installer and run `MSYS2`.

Once in, you'll receive a little window with a couple of lines that say your computer's name and your user name and ends in a cute little dollar sign. This is a command window typical on a Unix machine — but you're getting one on Windows — congratulations! Here you can enter any basic Unix commands. We'll run the package manager: `pacman` — cute, eh?

To do so, type this:

```
pacman -Syu
```

at the dollar-prompt and hit Enter — note the capital 'S'! This will start a few parallel checks and download them after you hit Enter to accept the default 'Y' response.[1] If you want to more easily complete this process, you can pin `MSYS2` to the taskbar before hitting Enter a second time.

`MSYS2` will close after that is finished, but we need to do more in that dollar-prompt! Run `MSYS2` again from either the taskbar or your Start menu. Run the above command again to get other necessary components.

This time the `MSYS2` terminal doesn't close so you can go on to the next step. Now type:

```
pacman -S --needed base-devel mingw-w64-x86_64-toolchain
```

(Sadly, even this simple command won't copy-paste well from the PDF. Be extra careful typing it!) When you hit Enter, many packages will be listed. You don't technically need them all, but I'd just accept the bunch for ease of use.

Now we need those tools in our Windows `PATH` — where Windows looks for executable commands by default. This can be done by running Control Panel, type 'edit path' in the 'Search' bar, and click the first resulting suggestion. Here we can double-click the 'Path' variable in the top window. A new window appears with all your current `PATH` entries listed out. Click 'New' to the right and enter:

```
C:\msys64\mingw64\bin
```

and hit Enter twice. (Note there are no spaces in there!)

Now go to the Start menu and run the command prompt (`Cmd`). At its terse greater-than-prompt, type:

```
g++ --version
```

to verify you have all you need for this book. (Don't worry, it actually depended on quite a few of the other installed items. It wasn't just that one we needed.) If it says command not found, we've got issues. Please talk to your instructor right away.

A.1.1.2 VS Code Time!

Now go to the VS Code site and find the 'Download' button. (For me it was on the upper-right side of the screen just to the right of the Search field.) From the choices, choose the left-most one under the big Windows logo. This will get you the installer. It also takes your browser to a 'Getting Started' page

[1] Sometimes typing Y or y at this prompt will fail. It is not repeatable and they can't track it down.

just for Windows. We'll just use our own steps below, though. No need to read that now — or maybe ever. (But it can't hurt to bookmark it for later. . .)

But, before you go below, double-click the installer in its download location — or however you like to run a newly downloaded installer.

Click 'I agree...' and then 'Next' three times. I turned on the 'Create a desktop icon' box before my next 'Next' — up to you. Now click 'Install' and finally 'Finish'. This will run it and you can now proceed to section A.5. But don't forget to come back sometime and do the Unix software setup if you are using that at your school!

A.1.2 Unix Server Connection

If you are working in a Unix environment at your school, then I've got the instructions for you here to make that happen. We'll need two tools: one for entering commands like compiling and execution of your programs and one for file transfer to either back up your programs from the Unix server or upload your local VS Code creations to the Unix server.

A.1.2.1 Command Processing

We'll start here by downloading a nice terminal. Windows does include the app we need, but it isn't very pretty and Windows users like pretty, right? So we'll download a nice one. It's called `Putty`. You can get it from the author's site. Amongst other links and information, you'll see the link next to 'Download' that says 'Stable'. Click that. While there is a lot to the Putty suite of programs, we only need one file. Scroll down to the 'Alternative binary files' section and click the first 'putty.exe' link. This is the 64-bit version for a typical Intel machine. If you've got something different, you'll be used to picking the right thing or you can ask your instructor for help.

Once that's downloaded, move it from where it went (your `Downloads`, perhaps?) to the `Desktop` or somewhere easily accessible. You can also make it part of your taskbar by right-clicking its icon once you run it and selecting Pin to taskbar. Go ahead and run it now and we'll configure it a little.

There really isn't much to configure, but we'll save a session to make it easy for later. `Putty` actually starts in the 'Session' tab, so that's perfect. Now fill in the 'Host Name' bar with the name of your school's Unix server. This should be three words separated by dots and the last one is probably `edu`. The 'Port' field should already say 22 which is exactly what modern Unix servers want. This is for an SSH connection — the Secure SHell protocol. This makes all information sent across the connection encrypted for safety. (You'll also notice that SSH is selected in the radio buttons below the 'Host Name' and 'Port' bars.

Now type a name for your school in the 'Saved Sessions' line. Something you can remember what this is for even after you've collected several dozen Unix machine names. *grin* *chuckle* Just kidding. But make it a good name anyway. Once done, click the 'Save' button at the right side.

Right now you can connect to your Unix server with either an Enter or clicking 'Open' or by double-clicking the saved session name in the box below 'Saved Sessions'. Next time you use `Putty`, you'll probably just double-click to connect.

One other bit of configuration bears noting: the scrollback buffer. This is relatively short at first — just a couple of thousand lines. And some programs will give **LOTS** of errors and warnings. So we want to probably increase this to 10,000 or even 100,000. This setting is located in the 'Window' tab. To get there, look to the left and see the tree of tabs available. Click 'Window' and just below the middle of that tab is the 'Lines of scrollback' input. Just change it to some reasonably large value. No millions or anything! To save this change we have to return to the 'Session' tab, make sure your saved session is selected still, and click the 'Save' button. If your saved session wasn't still selected, you will probably have to 'Load' it, change the scrollback setting again, and return here to 'Save' that change.

Now, to test this out, you’ll need to know your Unix server’s fingerprints.[2] Your instructor or IT folk should have given them to you for reference. If you don’t check them here you may subject yourself to a man-in-the-middle attack![3] This fingerprint check is only done for the first connection to the new server. Once you answer ’yes’, they are saved with the session and always checked automatically from here in.

When prompted, enter your username for the school’s Unix server at the ’login at’ prompt and then your password at the next prompt. You hit Enter after each one — not Tab as some interfaces expect. Note that your password won’t show up as you type for extra security. You’ll have to be a really good typist and trust that you’ve done it right. *smile*

Once in, you’ll receive a little window with a line that says your computer’s name and your user name and ends in a cute little dollar sign. This is a command window on a typical Unix machine. Here you can enter any of the commands your instructor/IT staff gave you to edit, compile, or run programs on the Unix machine.

When you are done, type `exit` and hit Enter to log out of the connection.

A.1.2.2 File Transfer

But someday you’ll want to download those files you edited on the remote server or upload those you’ve developed in your VS Code environment locally. To do this you’ll need a good file transfer program/app. The best one I’ve found for Windows is WinSCP. The SCP part is short for Secure Copy Protocol. It is also the typical command used in a Unix environment for secure copying of information — albeit in all lower case.

Once at the site, you can click the ’Download Now’ button — it’s big and green about the center of your window, perhaps. The information on the next page tells us lots of crazy details but note that it integrates well with `Putty`! That’s nice. Again, hit the big green ’Download WinSCP’ button. This time the download actually starts. Save it in a known place (your `Downloads`, perhaps?). Now go there and run it.

Whenever I have to install it afresh, I just click ’Accept’ and then ’Next’ twice and finally ’Install’. After a few files are installed, it notices you have `Putty` configured already and offers to import those sessions — say ’Yes’ and then ’Okay’. When the final dialog comes up to ’Launch’ and/or ’Open’ the website, you can uncheck the website open button. I’ve got your back!

Clicking ’Finish’ will run `WinSCP` and it opens to the connection window. There you’ll see your `Putty` session and a ’New Site’ icon. Double-click your session and you’ll be prompted for your ’Username’ and then for your ’Password’. If there’s a window in between, your Unix administrators have set up a special message for those who connect to the machine. Just click the ’Never show this banner again’ box and then click ’Continue’. At this point, you’ll have two panels. The left one shows files from your own machine and the right one shows files from your Unix account.

The left one defaults to your `My documents` folder. This can be changed by double-clicking on the .. entry at the top of the file list. This entry stands for the folder above the currently displayed one. If, on the other hand, you want to go into some other folder that is already shown, double-click that one, by all means. The same goes for the right panel. Once you’ve navigated one side or the other to the files you want to transfer, just click, Shift-click, or Ctrl-click the files to select them. Grab the selected files and drag them to the other panel and let go. Make sure you aren’t hovering over a folder when you let go! That will transfer the files to that folder instead of the displayed folder.

One quick but important fact before we close down `WinSCP`: don’t try to double-click a PDF to view it from inside `WinSCP`. It will try to view it as a plain text file which really is not the case! Instead either find it in the file manager or right-click and choose Open to launch it into your favorite PDF viewer.

To end the session, just click the corner X button. But this first time it’ll bring up a dialog. Here we’ll check ’Never ask this again’ in the lower-left corner and click ’Yes’.

[2] Of course computers have fingerprints! People are always touching them! *chuckle*

[3] I actually like the moniker ”monkey-in-the-middle attack”, but it just isn’t as popular for some reason.

Don't worry about finding WinSCP again, it saved an icon on your Desktop for your convenience. *smile*

A.2 macOS

You've chosen your OS wisely. This is one of the most stable and secure environments around. It is easy to work with and has a sturdy Unix undercarriage.

Now let's get you set up for programming!

A.2.1 IDE

Most people, when working on coding, want a nice integrated development environment where they can do all their tasks at once. This can be done with Apple-provided tools, but we'll take it a step further to give you a portable experience in case you have to work on other platforms in the future.

A.2.1.1 The Xcode Diversion

First install Xcode from the App Store. I know, I know: I said we would be doing VS Code. But remember, it doesn't have a compiler of its own. And do you know how you get a compiler on a Mac? That's right: you install Xcode. It's annoying, but it works.

The icon for Xcode is an A like the App Store's but in blueprint form — like we're building it, get it — with a hammer on top. In my search it was the second hit right after a story item called "What the Heck is Xcode?". Click the 'Get' button and wait. And wait. And wait. It has been known to take hours. I hope you got yourself some coffee and went potty before you clicked. Well, I suppose you have time to do it now. Go on, I'll wait for it for you.

There, now just drag that to the Applications folder for safe keeping. Now open a Terminal. This app, if you've never used it, is located in the Utilities folder inside the Applications folder. Once opened, you'll receive a little window with a line that says your computer's name and your user name and ends in a cute little dollar sign. This is a command window on a typical Unix machine.[4]

Now type the following at this dollar-prompt:

```
xcode-select --install
```

This should start installing the command-line tools for Xcode. These are what we need to run underneath VS Code. But don't worry, they don't take nearly as long as Xcode itself did.

If you had already installed Xcode and done something from the command-line, this step would report an error asking you to run a Software Update. In that case, you'd have to log in — with your Apple user name and password — at Apple's developer site. This would give you access to download the latest set of command-line tools. Click 'Account' in the upper-right-ish of the screen. Log in. Verify yourself with two-factor authentication. And then you can get the latest command-line tools. If you are told there are no operating systems to download even though you clearly just clicked the 'Download Tools' button, just click 'More' in the upper-right. This should bring you to Xcode and such. Scroll down a bit until you hit the "Command Line Tools for Xcode xx.x". The version number should agree with what you just got from the app store, if not, scroll until you find one that does. All older versions are available here for download.

The command-line tools are much smaller than the full Xcode, as I said. Just double-click the .dmg file[5] that results. This will do a verify and then open up. Double-click the .pkg file inside to install the command-line tools themselves. You'll probably have to give permission with your Mac password at some point.

[4] You did know your Mac was Unix under the hood, didn't you? *smile*

[5] I believe that stands for Disk iMaGe. It's like a little downloadable USB drive.

A.2.1.2 VS Code Time!

Now go to the VS Code site and find the 'Download' button. (For me it was on the upper-right side of the screen just to the right of the Search field.) From the choices, choose the right-most one under the big Apple logo. This will get you the Universal binary that should run on Intel or M1 Macs just fine. It also takes your browser to a 'Getting Started' page just for Macs. We'll just use our own steps below, though. No need to read it now — or maybe ever. (But it can't hurt to bookmark it for later. . .)

But, before you go below, unzip the file you downloaded by double-clicking it. Drag the resulting app (with a blue ribbon on its icon) to your application tray for easy access. Now you can run it and proceed to section A.5. But don't forget to come back here to do the Unix software setup if your school uses such!

A.2.2 Unix Server Connection

If you are working in a Unix environment at your school, then you are in luck by being on a Mac! The tools you need to connect to your school's Unix server are built in for command processing and you only have one download to make if you want to transfer files from your Mac to the server and/or back.

A.2.2.1 Command Processing

We begin by opening a Terminal. This app, if you've never used it, is located in the `Utilities` folder inside the `Applications` folder. Once opened, you'll receive a little window with a line that says your computer's name and your user name and ends in a cute little dollar sign. This is a command window on a typical Unix machine.[6]

At this cute little dollar-prompt you can type commands to the computer like the `ssh` command. This is short for Secure SHell. It will give you a nearly identical dollar-prompt except it'll be one connected securely — encrypted — to a remote Unix machine. You just need the machine/host name for the school's Unix server and its fingerprints for verification.[7]

To connect to your school's Unix server, just type this at the command prompt:

```
ssh youraccount@hostname
```

Of course, you must replace `youraccount` and `hostname` with the name of your account on your school's server and that server's host/machine name respectively. When the connection has been established for the first time, `ssh` will prompt you with a set of fingerprints. Verify these against those given to you by your school's IT site or your instructor. If they don't match, you may be subject to a man-in-the-middle attack![8]

When prompted, enter your password for the school's Unix server. Note that your password won't show up as you type for extra security. You'll have to be a really good typist and trust that you've done it right. *smile*

Now you can type any commands your teacher told you about to edit, compile, or run files on the Unix system. If you need more help, I recommend finding a good Unix tutorial you like. I'm partial to text references, but lots of younger folk enjoy a good YouTube video. *shrug* To each their own. . .

To disconnect from the remote machine, just type:

```
exit
```

at the dollar-prompt. This will place you back in your own `Terminal` on your own Mac. Before you quit the `Terminal` app, though, you might want to pin it to your app tray if you'll be using it a lot.

[6] Yes, I said all this before, but some people don't read both parts!
[7] Of course computers have fingerprints! People are always touching them! *chuckle*
[8] I actually like the moniker "monkey-in-the-middle attack", but it just isn't as popular for some reason.

You can pin a currently running app by right-clicking its icon in the tray, selecting Options and finally Keep in Dock. You can then drag it to your happy location amongst the other apps, if you want.

Once you're done, you can just command + Q as usual to exit the app.

A.2.2.2 File Transfer

But someday you'll want to download those files you edited on the remote server or upload those you've developed in your VS Code environment locally. To do this you'll need a good file transfer program/app. The easiest one I've found for my Mac is FileZilla. When you go there, you'll want the Client — not the Server. But don't grab the big green button for macOS right away! Instead, click the little text at the bottom that reads "Show additional download options". You want to do this because the green button gives you 'value-added' software add-ons. Once on the next page, you can click the first link for the `.tar.bz2` file. This is a compressed format similar to `.zip` which you may be more familiar with.

Once this downloads, double-click it to uncompress it. Then drag the resulting app to your app tray or `Applications` folder as you see fit. Now run the app and let's get configuring!

First let's declutter the interface a bit. Go to View and click Local directory tree to undo the checkmark by it. Then go back into View — see the checkmark is gone? — and click Remote directory tree. Two panels just disappeared from the main window and that's less to look at. Depending on how much it annoys you, you can also go back into the View menu and click to uncheck the Transfer queue panel.

On the left side now is your local directory. I believe your home directory is the default, but it's been a long time since I was able to do a fresh install, so it may be your `Documents` folder now. You can see little folder icons next to any subfolders/subdirectories and little pieces of paper next to other files. At the top of the subfolders list is one called `..` — yes, just two dots. This represents the 'parent' directory of the current one — i.e. the one above this on the way to the `Macintosh HD` and then the machine itself! Sorry, got a little carried away. But, for instance, if you were in your `Documents` directory, you'd double-click `..` to go to your home directory.

Time to connect to the school's Unix server! Above the directory/file panel a little ways is the connection bar. This has places for the Host (aka host name or machine name), your username on the school's Unix server, your password there, and the port to connect with. You should have been given the first three by your teacher or IT personnel. The port will be 22 for an `ssh`-style connection. When used for file transfer like this one will be, we usually call it an `sftp` connection — Secure File Transfer Protocol.

Once you've filled in all the spots, click the 'Quickconnect' button. This should start a scroll of information between the connection bar and the file panel. You'll be prompted for two things: whether to save your password — I wouldn't, but it's your computer/choice — and to check the fingerprints of the remote machine. As with the `ssh` in the `Terminal` above, you should have been given fingerprints by your instructor or IT staff. Make sure it is right to avoid that monkey in the middle!

Once connected, you'll see your remote account's files in the right panel next to your local ones. Now you can navigate each side by double-clicking folders (including `..` if needed) to find your program source codes. Once you find them, just drag and drop them to the other side. If you let go over a folder, you'll place the files inside it instead of just where the other panel was at. I say 'files' plural because you can use shift-click or command-click to grab multiple files at once. `FileZilla` may ask you if it is okay to copy the files, if so, just check the "Don't ask again" box and say 'Okay'.[9]

When you are done, just command + Q and next time you open up `FileZilla` it will remember where you've been. Click the drop-down arrow next to 'Quickconnect' to find your history. Just choose the one that starts with `sftp://` and has your user name and the school's host name in it and `FileZilla` will connect anew!

[9] Aren't those things annoying?!

A.3 Linux

You've chosen your OS wisely. This is one of the most stable and secure environments around. It is also very educational for Computer Science students and fun to work in!

As to job potential, there are more Linux shops out there than people realize. So don't worry! You time learning Linux skills will almost certainly be rewarded.

Now let's get you set up for programming!

A.3.1 IDE

Most people, when working on coding, want a nice integrated development environment where they can do all their tasks at once. This can be done with free Linux tools, but we'll take it a step further to give you a portable experience in case you have to work on other platforms in the future.

A.3.1.1 The g++ Diversion

First we must install the actual Linux compiler for C++: g++.[10] We can do this from the command-line or from a graphical interface. The instructions for this vary from distribution to distribution, but I'll show you how it is done on my Ubuntu box and you can easily look up how to install packages on your own distro.

The graphical package installer (manager) on Ubuntu is Synaptic. (I had to look this up as I had taken it out of my app tray/launcher long ago.) Its icon is a box/package with a big green arrow pointing down in the corner. Click this to get started. You may have to enter your password to prove you are worthy to install things. If it doesn't give you this option, you might have to revert to the command-line. Once you get started, put 'g++' in the search and select the top hit. You do not need the `multilib` version at this juncture.

And on the command-line (see A.3.2.1 below for how to open a command prompt), we simply run these two commands one after the other:

```
sudo apt-get update
sudo apt-get install g++
```

and hit Enter when it asks to install other packages as well.

Once this is done, you'll have g++ installed and can make sure by running:

```
g++ --version
```

in the terminal. It should show 9 or higher on a typical system these days. 10 is a little better, but if it didn't come with your default package management system, you don't wanna take on installing it by hand!

Now it is time to put an IDE in front of this puppy!

A.3.1.2 VS Code Time!

To get VS Code, you must visit the VS Code site. It isn't handled by the system package manager. (Although it will update automatically after it is installed!)

Once there, find the 'Download' button in the upper-right side of the screen just to the right of the Search field. From the choices, choose the middle one under the big Linux Tux logo (the penguin!). You'll actually have to choose if you are on a Debian-based system or a Red Hat style system. If you

[10] Well, there is also `clang++`, but `g++` is the typical recommendation unless you need something particular.

don't know, you'll have to wing it and come back to get the other one should you fail this time around. It also takes your browser to a 'Getting Started' page just for Linux. We'll just use our own steps below, though. No need to read it now — or maybe ever. (But it can't hurt to bookmark it for later. . .)

Find where you downloaded it and either double-click it from your file manager or run this from an Ubuntu command-prompt:

```
sudo dpkg --install ...
```

Fill in the name of the downloaded file for the ... here. To find the installed application can be daunting. On my Ubuntu box, for instance, when I hit the command/Windows key on my keyboard, a menu of apps comes up and I can search. Typing in 'code' — the executable name for VS Code — gives just the one hit. You can also type `code` into the terminal if you've just installed from there.

Now go below to the VS Code configuration section (A.5) for how to set this environment up. But don't forget to come back here and set up the Unix server connection software if your school uses such!

A.3.2 Unix Server Connection

If you are working in a Unix environment at your school, then you are in luck by being on Linux! The tools you need to connect to your school's Unix server are built in for command processing and you only have one download to make if you want to transfer files from your Linux box to the server and/or back.

A.3.2.1 Command Processing

Now open a terminal. This app, like you've never used it, is located in various forms on various distributions. For instance, when I hit the command/Windows key on my keyboard, a menu of apps comes up and I can search. Typing in 'terminal' brings up some eight choices — and I don't think that is all of the ones that are on my system! Once you pick one and open it, you'll receive a little window with a line that says your computer's name and your user name and ends in a cute little dollar sign. This is a command window on a typical Unix machine.

At this cute little dollar-prompt you can type commands to the computer like the `ssh` command. This is short for Secure SHell. It will give you a nearly identical dollar-prompt except it'll be one connected securely — encrypted — to a remote Unix machine. You just need the machine/host name for the school's Unix server and its fingerprints for verification.[11]

To connect to your school's Unix server, just type this at the command prompt:

```
ssh youraccount@hostname
```

Of course, you must replace `youraccount` and `hostname` with the name of your account on your school's server and that server's host/machine name respectively. When the connection has been established for the first time, `ssh` will prompt you with a set of fingerprints. Verify these against those given to you by your school's IT site or your instructor. If they don't match, you may be subject to a man-in-the-middle attack![12]

When prompted, enter your password for the school's Unix server. Note that your password won't show up as you type for extra security. You'll have to be a really good typist and trust that you've done it right. *smile*

Now you can type any commands your teacher told you about to edit, compile, or run files on the Unix system. If you need more help, I recommend finding a good Unix tutorial you like. I'm partial to text references, but lots of younger folk enjoy a good YouTube video. *shrug* To each their own. . .

[11] Of course computers have fingerprints! People are always touching them! *chuckle*
[12] I actually like the moniker "monkey-in-the-middle attack", but it just isn't as popular for some reason.

To disconnect from the remote machine, just type:

```
exit
```

at the dollar-prompt. This will place you back in your own terminal on your own Linux box. Before you quit the terminal app, though, you might want to pin it to your app tray if you'll be using it a lot. You can pin a currently running app by right-clicking its icon in the tray, selecting Lock to Launcher. You can then drag it to your happy location amongst the other apps, if you want.

Once you're done, you can just 'exit' as you did above but this time to exit the app.

A.3.2.2 File Transfer

But someday you'll want to download those files you edited on the remote server or upload those you've developed in your VS Code environment locally. To do this you'll need a good file transfer program/app. The easiest one I've found for my Linux machine is FileZilla. But don't go there except for help/reference. Use your package manager to download/install the app. If you like the graphical manager, just search for 'filezilla' and click 'Install'. If you like the command-line installer, the package name is typically 'filezilla' and it will install at least one other package for a helper library as well. On my Ubuntu box I ran:

```
sudo apt-get install filezilla
```

and then hit Enter to accept the extra package as well.

Next run the app from the command-line or the menu or a search. That all depends on how you like to do things and how you have your system configured. I ran mine from the command-line:

```
filezilla
```

Once it is running, you can right-click it in the tray and choose Lock to Launcher to keep it handy for later up/downloads.

Now let's get configuring!

First let's declutter the interface a bit. Go to View and click Local directory tree to undo the checkmark by it. Then go back into View — see the checkmark is gone? — and click Remote directory tree. Two panels just disappeared from the main window and that's less to look at. Depending on how much it annoys you, you can also go back into the View menu and click to uncheck the Transfer queue panel.

On the left side now is your local directory. I believe your home directory is the default, but it's been a long time since I was able to do a fresh install, so it may be your `Documents` folder now. You can see little folder icons next to any subfolders/subdirectories and little pieces of paper next to other files. At the top of the subfolders list is one called `..` — yes, just two dots. This represents the 'parent' directory of the current one — i.e. the one above this on the way to the root directory (`/`)! Sorry, got a little carried away. But, for instance, if you were in your `Documents` directory, you'd double-click `..` to go to your home directory.

Time to connect to the school's Unix server! Above the directory/file panel a little ways is the connection bar. This has places for the Host (aka host name or machine name), your username on the school's Unix server, your password there, and the port to connect with. You should have been given the first three by your teacher or IT personnel. The port will be 22 for an `ssh`-style connection. When used for file transfer like this one will be, we usually call it an `sftp` connection — Secure File Transfer Protocol.

Once you've filled in all the spots, click the 'Quickconnect' button. This should start a scroll of information between the connection bar and the file panel. You'll be prompted for two things: whether to save your password — I wouldn't, but it's your computer/choice — and to check the fingerprints of the remote machine. As with the ssh in the Terminal above, you should have been given fingerprints by your instructor or IT staff. Make sure it is right to avoid that monkey in the middle!

Once connected, you'll see your remote account's files in the right panel next to your local ones. Now you can navigate each side by double-clicking folders (including .. if needed) to find your program source codes. Once you find them, just drag and drop them to the other side. If you let go over a folder, you'll place the files inside it instead of just where the other panel was at. I say 'files' plural because you can use Shift-click or Control-click to grab multiple files at once. FileZilla may ask you if it is okay to copy the files, if so, just check the "Don't ask again" box and say 'Okay'.[13]

When you are done, just Control+Q and next time you open up FileZilla it will remember where you've been. Click the drop-down arrow next to 'Quickconnect' to find your history. Just choose the one that starts with sftp:// and has your user name and the school's host name in it and FileZilla will connect anew!

A.4 ChromeOS

Was this a choice or was it forced on you by circumstances? Yeah, I thought so... Well, let's get you set up for programming!

A.4.1 IDE

Most people, when working on coding, want a nice integrated development environment where they can do all their tasks at once. This can be done with basic Linux tools — even on ChromeOS, but we'll take a different route to give you a portable experience in case you get to work on other platforms in the future.

A.4.1.1 The Linux Diversion

First we'll install Linux on your ChromeOS box. This won't work on terribly old Chromebooks, but not much will, am I right?

Get to your Settings page and look for Linux. It might say "Beta" on it, that's fine. Mine was hidden in Advanced setup under Developers! Click to turn this on and follow the prompts to set it up. The only thing I changed was my name. But when it asks for space, make sure you give it as much as you can spare. I upped mine because I had very little free and it almost ran out during the VS Code configuration! 4.1Gb seems to have satisfied it for now...

When that finishes, you'll be placed into a terminal window. This is a terse prompt that allows you to type commands to your system without clicking yourself to death. It defaults to your chosen name followed by an at sign and then 'penguin'. Then it probably has a colon and a tilde followed by a dollar sign. This is the classic dollar-prompt we talk about in Chapter 2 so much. Now you have one on your Chromebook!

A.4.1.1.1 A Compiler

The next step is to install the compiler utilities from the command-line. Type the following two commands one after the other:

```
sudo apt-get update
sudo apt-get install g++
```

The second command will stop with a Yn prompt. Just hit Enter/return to accept the yes default.

A while later, it will return you to the dollar-prompt. Type this command to check the installation:

[13] Aren't those things annoying?!

```
g++ --version
```

If it says command not found, we've got issues. Please talk to your instructor right away.

A.4.1.2 VS Code Time!

I hope you didn't close the terminal window yet! We need one last thing from there to install VS Code. We need to find out what kind of machine your Chromebook is. Some run on Intel or AMD chips and some run on ARM chips. You have to decide which you have before downloading the right one from the VS Code website. To determine your machine type, just run this command at the dollar-prompt:

```
dpkg --print-architecture
```

Be careful when reading it. 'amd64' looks an awful lot like 'arm64'. But it's an important distinction!

Now go to the VS Code site and find the 'Download' button. (For me it was on the upper-right side of the screen just to the right of the Search field.) From the choices, choose the middle one under the big penguin logo (that's Tux, the Linux mascot). Make sure to select the one from the `.deb` row that matches your architecture. If you were 'arm64', choose 'ARM 64', of course. If you were anything else, choose '64 bit'. This will get you the installer. It also takes your browser to a 'Getting Started' page just for Linux. We'll just use our own steps below, though. No need to read it now — or maybe ever. (But it can't hurt to bookmark it for later. . .)

But, before you go below, double-click the installer in its download location. The system offers to install it with Linux, select Install and then OK.

To run VS Code, go to the dollar-prompt once more and type this command:

```
code
```

You can now proceed to section A.5. But don't forget to come back here to finish setting up software to connect to a Unix server if your school uses such!

A.4.2 Unix Server Connection

If you are working in a Unix environment at your school, then I've got the instructions for you here to make that happen. We'll need two tools: one for entering commands like compiling and execution of your programs and one for file transfer to either back up your programs from the Unix server or upload your local VS Code creations to the Unix server.

A.4.2.1 Command Processing

We'll start here by making your terminal talk to the school's Unix server for processing commands. This is how one typically compiles and executes programs in a Unix environment — from the terminal. If you've closed the terminal, re-open it by swiping up on the task bar at the bottom of your screen and selecting first 'Linux Apps' and then Terminal from there. Its icon looks like a greater-than followed by an underscore for whatever reason.

At the dollar-prompt in your terminal, you can type commands to the computer like the `ssh` command. This is short for Secure SHell. It will give you a nearly identical dollar-prompt except it'll be one connected securely — encrypted — to a remote Unix machine. You just need the machine/host name for the school's Unix server and its fingerprints for verification.[14]

To connect to your school's Unix server, just type this at the command prompt:

[14] Of course computers have fingerprints! People are always touching them! *chuckle*

```
ssh youraccount@hostname
```

Of course, you must replace `youraccount` and `hostname` with the name of your account on your school's server and that server's host/machine name respectively. When the connection has been established for the first time, `ssh` will prompt you with a set of fingerprints. Verify these against those given to you by your school's IT site or your instructor. If they don't match, you may be subject to a man-in-the-middle attack![15]

When prompted, enter your password for the school's Unix server. Note that your password won't show up as you type for extra security. You'll have to be a really good typist and trust that you've done it right. *smile*

Now you can type any commands your teacher told you about to edit, compile, or run files on the Unix system. If you need more help, I recommend finding a good Unix tutorial you like. I'm partial to text references, but lots of younger folk enjoy a good YouTube video. *shrug* To each their own. . .

To disconnect from the remote machine, just type:

```
exit
```

at the dollar-prompt. This will place you back in your own terminal on your own Chromebook. Before you quit the terminal app, though, you might want to pin it to your app tray if you'll be using it a lot. You can pin a currently running app by right-clicking its icon in the tray, selecting Pin. You can then drag it to your happy location amongst the other apps, if you want.

Once you're done, you can just 'exit' as you did above but this time to exit the terminal app itself.

A.4.2.2 File Transfer

But someday you'll want to download those files you edited on the remote server or upload those you've developed in your VS Code environment locally. To do this you'll need a good file transfer program/app. The easiest one I've found for my Linux machine is FileZilla. But don't go there except for help/reference. Use your package manager to download/install the app. If you like the graphical manager, just search for 'filezilla' and click 'Install'. If you like the command-line installer, the package name is typically 'filezilla' and it will install at least one other package for a helper library as well. On my Ubuntu box I ran:

```
sudo apt-get install filezilla
```

and then hit Enter at the `Yn` prompt to accept the extra packages as well.

Next run the app from the command-line or the menu or a search. That all depends on how you like to do things and how you have your system configured. I ran mine from the command-line:

```
filezilla
```

Once it is running, you can right-click it in the tray and choose Pin to keep it handy for later up/downloads.

Now let's get configuring!

First let's declutter the interface a bit. Go to View and click Local directory tree to undo the checkmark by it. Then go back into View — see the checkmark is gone? — and click Remote directory tree. Two panels just disappeared from the main window and that's less to look at. Depending on how much it annoys you, you can also go back into the View menu and click to uncheck the Transfer queue panel.

[15] I actually like the moniker "monkey-in-the-middle attack", but it just isn't as popular for some reason.

On the left side now is your local directory. Your home directory is the default. You can see little folder icons next to any subfolders/subdirectories and little pieces of paper next to other files. At the top of the subfolders list is one called .. — yes, just two dots. This represents the 'parent' directory of the current one — i.e. the one above this on the way to the root directory (/)! Sorry, got a little carried away.

Time to connect to the school's Unix server! Above the directory/file panel a little ways is the connection bar. This has places for the Host (aka host name or machine name), your username on the school's Unix server, your password there, and the port to connect with. You should have been given the first three by your teacher or IT personnel. The port will be 22 for an `ssh`-style connection. When used for file transfer like this one will be, we usually call it an `sftp` connection — Secure File Transfer Protocol.

Once you've filled in all the spots, click the 'Quickconnect' button. This should start a scroll of information between the connection bar and the file panel. You'll be prompted for two things: whether to save your password — I wouldn't, but it's your computer/choice — and to check the fingerprints of the remote machine. As with the `ssh` in the `Terminal` above, you should have been given fingerprints by your instructor or IT staff. Make sure it is right to avoid that monkey in the middle!

Once connected, you'll see your remote account's files in the right panel next to your local ones. Now you can navigate each side by double-clicking folders (including .. if needed) to find your program source codes. Once you find them, just drag and drop them to the other side. If you let go over a folder, you'll place the files inside it instead of just where the other panel was at. I say 'files' plural because you can use Shift-click or Control-click to grab multiple files at once. `FileZilla` may ask you if it is okay to copy the files, if so, just check the "Don't ask again" box and say 'Okay'.[16]

When you are done, just Control+Q and next time you open up `FileZilla` it will remember where you've been. Click the drop-down arrow next to 'Quickconnect' to find your history. Just choose the one that starts with `sftp://` and has your user name and the school's host name in it and `FileZilla` will connect anew!

A.5 VS Code Setup

Now that you have VS Code installed on your platform, you'll want to configure it for best practices. Follow these steps and all should be well:

1) Got to either the Code or File menu — macOS and Windows/Linux/ChromeOS, respectively — and choose Preferences ⟩ Settings under that. Now use the 'Search settings' box to find the following bold items and change them as directed:

    i) **Editor:Detect Indentation** should be unchecked. Having this on can really mess up the next two settings.

    ii) **Editor:Tab Size** should be set to something between 3 and 5 inclusive. The typical 8 is way too deep when we get to nested control structures later on.

    iii) **Editor:Insert Spaces** should be checked. Tab characters save a little disk space, but can really mess up displayed code in other environments.

    iv) **Files:Eol** should be set to \n. This is a platform neutral setting and won't mess up things when moving from a Windows to a Linux venue, for instance.

    v) **Files:Insert Final Newline** should be checked. This is a recent push in the industry for a text file standard. Also makes some code displays look nicer.

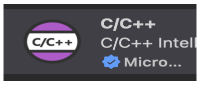
    vi) **Files:Encoding** should be UTF-8. This is an industry standard for text files now.

[16] Aren't those things annoying?!

vii) **Files:Auto Save** should be 'afterDelay'. This is my personal taste, but you definitely don't want it 'Off' which was the default when I installed last! (If you set it to 'afterDelay', you can also set the delay itself in the next item down. It defaults to every second which is really fast. (Note that the delay is set in milliseconds — not seconds.)

viii) **Editor:Rulers** will ask you to 'Edit in settings.json'. Just click that and inside the square brackets (`[]`), type 75 or 80 or whatever your teacher suggests. (I have my students set it to 75 for our local system. 80 is fairly common as well.) This phantom vertical line will tell you visually when you should be wrapping lines of code that get too long for good code display. Save the file but don't close out of it yet, we'll come back to it shortly!

2) Open the Extensions sidebar (the icon is three boxes together and one floating to the side). Here search for C++ in the 'Search Extensions in Marketplace' bar. One is just 'C/C++' — not 'Extensions', not 'Themes', and not 'C++ Intellisense'. Mine looks like the figure at right. Click the 'Install' button. You may have to restart VS Code afterwards.



3) Back over to the `settings.json` file, place a comma after the close square bracket on `"editor.rulers"` if one is not present. Now add this line on the line after that:

```
"C_Cpp.default.cppStandard": "c++17"
```

If any more lines come after it except the close curly brace, place a comma after the `"c++17"` above. (You can use `"c++20"` if you have a compliant compiler for this. Check your documentation or the compiler's website.)
Now save the `settings.json` — but don't close it — we're still not done there!

4) Now open Extensions, click the Search bar, and type rewrap. There are a few, but you want the one with the backwards S and a vertical bar to the side of it and that says **Revived** after it. Install that. You can read more about it in its description later, but for now just go back to the Settings tab and search for rewrap there, too. Check the **ReWrap:Auto Wrap: Enabled** setting so that it actually becomes Enabled. This is very handy for long comments so you won't have to wrap them manually at the ruler we set up before.
Note: you may have to restart VS Code here as well.

5) Now click the stacked sheets of paper icon in the upper-left corner of the window to get to the file explorer. You can then close the Extensions and Settings tabs, too — but leave the `settings.json` file open! Open a new folder by choosing the 'Open Folder' button and choose/create the place you want to write a program.[17] You will have to accede to trust the authors in this folder. You are safe here. Go ahead and trust yourself. *grin* Seriously, make sure you check the 'Trust the authors of all files in the parent folder...' box, too.
Now right-click (two-finger click on a trackpad) the '`settings.json`' in the title bar — the one with the X next to it that would close the file — just don't close it! This brings up a menu of choices of what to do, of course. We want to do the first of the 'Reveal' commands. This will open a file browser window for your OS where the `settings.json` file is located. You should see it and several folders there. (The second one just makes it appear in the File Explorer within VS Code.)
Next click the single sheet with a + on it near the top of VS Code's File Explorer window. It will appear only when your cursor is in the blank area beneath your folder's name — the one you created to place your program above. This wants a name for this new file. Call it `tasks.json` — case is important here! Once you hit Enter/return on the file name, you have a nice blank file to work with. Now we need to put this in it:

```
{
    // See https://go.microsoft.com/fwlink/?LinkId=733558
```

[17] I'd recommend making a first folder for your course or your exploration of this book and then subfolders within this for each of your projects/programs.

```
    // for the documentation about the tasks.json format
    "version": "2.0.0",
    "tasks": [
        {
            "type": "shell",
            "label": "clang++ build active file",
            "command": "clang++",
            "args": [
                "-std=c++17",
                "-g",
                "-stdlib=libc++",
                "-Wall",
                "-Wextra",
                "-Wfloat-equal",
                "-Winline",
                "-Wunreachable-code",
                "-Wredundant-decls",
                "-Wconversion",
                "-Wwrite-strings",
                "-Wcast-qual",
                "-Woverloaded-virtual",
                "-Weffc++",
                "-fno-gnu-keywords",
                "-pedantic",
                "-Wparentheses",
                "-Wshadow",
                "-Wold-style-cast",
                {
                    "value": "*.cpp",
                    "quoting": "escape"
                },
                "-o",
                {
                    "value": "${fileBasenameNoExtension}.out",
                    "quoting": "escape"
                }
            ],
            "options": {
                "cwd": "${fileDirname}"
            },
            "problemMatcher": [
                "$gcc"
            ],
            "group": {
                "kind": "build",
                "isDefault": true
            }
        }
    ]
}
```

Sadly, your mileage may vary for pasting from a PDF. I've had mixed results myself. I'll keep this and the below files on the website for you to copy/paste from or download as you see fit.[18]
Now, if you've set up for Windows, ChromeOS, or Linux, change where it says `clang++` to `g++` (there are two places — lines 8 & 9). If you've set up for Windows, change the `.out` extension on line 36 to `.exe` instead. (And with a compliant compiler, you can change out the 17 on line 11

[18] When clicking to the website link from a web browser, I recommend a right-click to open it in a new tab/window so you don't lose your place in the PDF.

with 20 as before.)

Almost done: if you are on Windows, ChromeOS, or Linux, delete line 13 for the `--stdlib` setting. This is a `clang++` only thing and that is the command-line compiler on Mac. The rest of the settings can stay for either compiler.

Finally, if you are on Windows only, change the `*.cpp` in line 30 to `$(ls *.cpp | % {$_.FullName})` but leave it inside the double quote marks! And then change the `escape` on the next line to `weak` without changing its quotes.

Now make sure you've Saved the file.

Then we want to 'Reveal' this file in the browser. Right-click (two-finger click on a trackpad) the name of the file at the top with the X next to it and choose the first 'Reveal' option again. Now you have two browser windows open — one with the `settings.json` and one with the new `tasks.json`. Go to the `tasks.json` folder and drag it to the `settings.json` folder. This should move the file. If not, you can feel free to delete the one from your just made folder, we won't need it anymore. You can also close the `tasks.json` file in VS Code as it isn't where we put it from there anymore.

6) Next we need to edit the `settings.json` file from before. Go to the last setting — it should be the `rewrap.autoWrap.enabled` we just did — and add a comma after it. Now paste this from either here or the launch file from the website:

```
"launch": {
    "version": "0.2.0",
    "configurations": [
        {
            "name": "clang++ - Build and debug active file",
            "type": "cppdbg",
            "request": "launch",
            "program": "${fileDirname}/${fileBasenameNoExtension}.out",
            "args": [],
            "stopAtEntry": true,
            "cwd": "${fileDirname}",
            "environment": [],
            "externalConsole": true,
            "MIMode": "lldb",
            "preLaunchTask": "clang++ build active file"
        }
    ]
}
```

Again, change `clang++` if necessary on lines 5 (name) and 15 (preLaunchTask) and change the `.out` on line 8 (program) if needed as well. And for Windows, ChromeOS, and Linux also change the `lldb` on the MIMode line (14) to `gdb`.

Save the `settings.json` file and we're done configuring!

7) Finally, we're ready to test this thing! Click the stacked sheets icon to the far left of the window to get back to the File Explorer sidebar. Now move the cursor to the blank area below your folder name again and click the sheet of paper with a + on it at the top of the sidebar. Call this new file `welcome.cpp`. Paste this code in there:

```
#include <iostream>

using namespace std;

int main()
```

```
{
    cout << "\n\t\tWelcome to C++!!!\n\n";

    return 0;
}
```

(If having trouble copy/pasting from the PDF, this file is available on the website as well.) Save and choose Terminal > Run Build Task.... A little window appears below your editor with a bunch of text. If you look carefully, you'll recognize the bits and pieces from the `tasks.json` file we made. If all goes well, you'll see this at the bottom:

```
Terminal will be reused by tasks, press any key to close it.
```

Click in that window and hit Enter/return to close it. Now choose Terminal > New Terminal. This gives you a command prompt in which you can ...wait for it... run commands. We'll run our program. In macOS, ChromeOS, or Linux type `./welcome.out` or for Windows type `./welcome` to run the program. You should see this:

```

                Welcome to C++!!!

```

If anything has gone wrong, please let your instructor know ASAP so they can help you fix the issue. If they need help, they can email me. *smile*

8) One final note about programs. Make sure each program you tackle is in a separate folder/directory from one another or the above techniques won't work.

A.5.1 Normal Workflow

A.5.1.1 Terminal Management

Once you've opened a new terminal window, it should come back each time you reload VS Code. Next time you compile, the compile terminal will overlay it. Just clicking it and hitting Enter as before will close the compile terminal and put you back to your command terminal. There's no need to repeat the new terminal window part each time.

A.5.1.2 File Management

The above setup is geared toward working on a program whose files are all in a folder/directory together. This is helpful once you reach that part of chapter 4 with separate compilation and is not a bad idea otherwise.

Don't expect to place all your C++ programs in the same folder together and all will work well. It would be disorganized and won't work with the above tasks setup or debugging setup.

A.5.1.2.1 Opening Files

Speaking of having multiple files open at once, there are actually two types of open files. If you single-click to open a file, it will be open to view. But if you then single-click to open another file, it will replace the first file! This can get annoying when trying to open multiple files at the same time. To get around it, make sure you double-click to open a file and this will leave it open even if you don't edit in it before opening a second file.

A.5.2 Recommended Extensions

The above are almost all mandatory setups except perhaps the ReWrap extension. But the following are all completely optional. They will, however, make certain things easier to do within the VS Code environment and so are strongly encouraged.

- If your teacher gives PDF handouts or makes you prepare a PDF to hand in to them, the extension ”PDF Preview” is very handy as it let's you view PDF files from within VS Code.
- If you are prone to spelling errors, then the ”Code Spell Checker” extension is your friend. It will check not only your plain text files, literal strings, and comments, but also your variable, constant, and function names. The extension knows many common abbreviations for things so camel-case with short-hand can work without complaints!
- To make the difference checking suggested in Chapter 4 easier, you can install the ”Partial Diff” extension.
  To use this extension, you select a section of code, right-click, and select ”Select for Text Compare” from around the middle of the popup menu. Then select the second section of code you want to the first to, right-click, and select the ”Compare Text with Previous Selection” option — just below the previous option.
  To configure the colors, you need to edit the `settings.json` file. Remember, to open ”Settings” use Ctrl+, or command+,. Then search for 'rulers' and click the ”Edit in settings.json” link. Now, go to the bottom and add a comma after the last thing and then this:

  ```
  "workbench.colorCustomizations": {
      "diffEditor.removedTextBackground": "#FF000055",
      "diffEditor.insertedTextBackground": "#ffff0055"
  }
  ```

  A color box will appear in front of the two color settings. You can then click them to get a color chooser dialog. Note the 55 after the normal color codes is called an alpha channel and can be tweaked as well. I found that tip on StackOverflow.

A.6 Wrap Up

That concludes our setup instructions. If you have any trouble, contact your instructor immediately for help! It is very important that you have a working environment sooner than later in a course like this. Without practice, you don't really learn anything in the long run, after all.

Appendix B

Debugging Tips

We've talked some about debugging programs before:

- Printing values with `cerr` at crucial junctures.
- Using an `ignore` with `cin` to pause the program during a [nearly] infinite loop.
- Using `assert` to test function parameters.
- Using `cerr` to determine which of many function calls caused an `assert` to fail.

But there are software-based debugging systems other than `cerr` and `assert`. Two such systems are `gdb` and `lldb`. These debuggers work to trouble-shoot your program in either a `gcc`-based or `clang`-based build. If you've been working based on the instructions from above for VS Code setup, you should be able to debug your program within that environment with the provided setup. (Windows and Linux were set up to use `g++` from the `gcc` compiler suite and macOS was set up to use `clang++` from the `clang` compiler suite.)

The steps to debugging programs with a debugger are to set watches and breakpoints and to either step into or over lines of code. The exact way to do these things varies from one debugging system to another, but we'll walk you through debugging a program in VS Code as set up above here.

B.1 Starting a Debugging Session

We already set up a debugging session above. We just need to start it going. To do so, choose the Run > Start Debugging.

When I did this the first time, I had to give special permission on my macOS box to use the `lldb` and for my program to access part of my files for whatever reason. I said yes to both and thereafter I was only asked about the file permissions on each rerun. Kind of annoying, but I'm not sure how to turn that on permanently.

B.2 Setting Watches and Breakpoints

Breakpoints are easiest to set. What do they do? Well, they stop your program from running. Sadly, this would be applying the brakes, but we have instead a breakpoint. *shrug* What're ya' gonna do?

This is done by hovering the mouse to the left of the line number in the VS Code editor window. A red dot appears and you just click on it to set that line as a breakpoint. Then, when the program is run in debugging mode within VS Code — not separately in the terminal, it will stop there and let you see the values of variables at that point in the execution. This can be terribly helpful in finding problems during a troublesome project. The variables and their values are located to the left of the editor window where the list of files once was.[1]

There you see variables, watches, the call stack, and breakpoints. The variables panel changes as you move from function to function to reflect the currently visible local variables. The watch panel shows variables or function arguments you'd really like to keep a special eye on. More on that in a bit. The call stack panel shows what functions have been called to this point in the program. You'll note that they are stacked up from oldest on bottom to newest — currently running — on top. This is just as described back in chapter 4.4.1. Finally, the breakpoints panel is just a textual list of the red dots you've made in your code.

Note: to look inside a `string`, `vector`, or `array` `class` variable, you'll have to open the little greater-than arrow after its variables entry during the run.

Unless, you are interested in a specific part of said container. In that case you can set a watch on an expression to reach that part. Just type out something like `v[2]` to see the 3^{rd} slot in a `vector` named `v`, select it, and right-click to "Add watch" on it. Or, while the program is running in debug mode, click the plus sign in the watch panel to add a watch expression.

The expression can look into a container with the `[]` operator or look at a `bool` expression to see if it is `true` or `false` at the moment. It can involve any kind of operations, basically, on any currently known — in scope — variables, constants, or values. This is what makes the watch panel more than the variables panel. Sure, you can watch a variable, but you can also watch any kind of expression involving a variable, too.

B.3 Stepping Through Code

Making the code run during a debugging session can be a bit tricky at first. There are four methods of movement listed in the debug control bar atop the editor window. They are the four icons in light blue. The green circular arrow restarts the program from scratch and the red square stops it right where it is at.

The first icon that looks a little like a play button runs the program until the next breakpoint or the end of the program. The second one with the arrow hopping over the dot will execute the current statement and stop at the next one. This operation is called "step over" in debugging terms. If the statement calls one or more functions, step over will execute them all and then land on the next statement.

The third blue icon, however, is known as "step into". This one looks like an arrow pointing into a dot. This means execute the current statement, but if it contains function calls, dive into them to see how they are working inside as well. On some systems you have to be very careful with step into because it might step into the standard library codes as well as your own. That doesn't seem to be the case on my macOS install of VS Code on top of `clang++` and `lldb`.

But with my Linux install on top of `g++` and `gdb`, it tries to step into the standard code and gives an error at the end about not being able to open a file. In fact, once I tried that to see what it would do, the only time I don't get that error at the end of the program run is when I hit the play button to end things. If I try to step out — even over! — it give the "can't open file" error.

The last button is "step out". I haven't gotten this to work, but it is supposed to run until the current function `return`s.

[1] Don't worry, you can get back to the file browser by clicking the top icon (two overlapping sheets of paper).

B.4 Wrap Up

I hope this short introduction to advanced debugging using an automated tool has helped whet your appetite for this topic.

Appendix C

Essential Unix Knowledge

As implied in the setup appendix above (appendix A), a lot of times introductory programming courses use a command-line interface like the Unix (or Linux) terminal prompt. This appendix attempts to give you the basic knowledge necessary to navigate this environment with relative comfort. We won't talk to compiling commands, as that would require knowledge of your local setup and what compiler(s) are installed. But we'll talk naming, navigation, and essential commands that will get you around the terminal in reasonable style.

C.1 Paths and Filenames

Unlike in a graphical interface (or GUI), the command-line interface deals a lot with path and file names. In a GUI, after all, you just see a dialog box and click on the right one. At best you type most of the filename in an input line when naming the file. But in the terminal we type these things out a lot so it is good to be comfortable with the basics.

First, terminal folk call folders directories. This is an older name but more traditional. And knowing this will make some of the mnemonic names more clear later.

Second, separators that go between folder or directory names and the eventual filename are different in a Unix environment than in Windows. The Unix community uses the same separators as in a web address: `/`. This is the opposite slash as Windows uses — when you even get to see them.[1]

Third, a path relative to the current folder can be entered starting with the subfolder name. But an

[1] But modern Windows is pretty accepting and will transform a path with the Unix/Web separators to its own form on the fly.

absolute path needs a leading / on it to indicate it is with respect to the entire drive.[2]

Fourth, case is SENSITIVE in the Unix terminal. Upper and lower case are considered different and must be entered exactly. So be careful what you name your files if you don't want to have to shift your life away!

Finally, spaces and quotes are allowed in path and file names, but are a little tricky to work with. Let's talk about that specially.

C.1.1 Spaces and Quotes

To use spaces or quotes in a file or folder/directory name, you have to take one of two tactics. The first is to escape the space or quote. No, you don't have to outrun it in a death race. You just place the standard escape character in front of it. We'll learn about the escape character and how it is used in C++ programming in chapter 2 — section 2.2.2. But for now know that it is used at the command-line prompt as well.[3]

What is the escape character and how do we use it? It is a backslash: \. And we place it just before the space or quote we want to make special. You see, normally the space is used by the command prompt to separate 'words' in the input command, so treating them as part of a filename is relatively special. We'll see how quotes are used by the command prompt in a second.

So if you had a file named '`my file`' and you wanted to use a Unix command on it, you could type that name as:

```
my\ file
```

Or, if you prefer, you could surround it in quotes which the command prompt uses to group together space-containing phrases for use as single things in a command:

```
'my file'
```

Double quotes would work just as well, of course. We often just use singles because then we don't have to shift. Seems silly, but when you spend all day typing, you find shortcuts.

And if you need a quote in a filename or path component, you can escape that as well:

```
Jason\'s\ file
```

You cannot, however, just surround a quote in a name with more quotes! They can't even just be escaped! You have to take it to the shift-level. Then you can do a switcheroo:

```
"Jason's file"
```

That's the gist for now, but see below for more on special treatment of spaces and quotes!

C.1.2 Special Folders

There are two special folders/directories that actually exist within every folder on the drive. One represents the parent folder of the current one. It is called `..` — just the two dots. We'll use this later to navigate around the system a bit.

[2] Actually, it is the file system to which many drives could be attached. But more on that in an actual course on operating systems.

[3] Unix was made by programmers for programmers to use and you'll find lots of the same conventions used at its prompt as are used in common programming languages. In fact, the teams that developed Unix and the programming language C overlapped quite a bit.

The second is called . — just a single dot. This represents the current folder/directory. It is used when you want to make sure a file is used relative to the current folder and not with respect to somewhere else on the drive.

C.1.2.1 More on Relative Paths

We spoke earlier about absolute and relative paths. But now we can use the special directories . and .. to make other relative path references like:

```
'./file in this folder'
```

Or perhaps:

```
'../folder beside this one/file over there'
```

These are both relative paths — one relative to the current directory and the other relative to its parent folder.

The parent path can also be used multiple times to dig your way back up a deeply nested folder structure. Like:

```
'../../../folder three up from here/file over there'
```

There is no need, however, to repeatedly use . in a relative path.

C.1.3 Wildcards

Sometimes we want to group many similarly named files together. We can't just shift-click or Windows-click or Command-click them to group them together like we would in a dialog box, though. We have to group them by the commonality of their names. For instance, if you had a set of files all ending in the extension `cpp`, you could specify them all at once with:

```
*.cpp
```

This use of the asterisk indicates to the command prompt that you intend to use all the files whose names end in a period and then the letters `cpp`. The dot isn't even explicitly required and is often left off by speedy programmers:

```
*cpp
```

This wildcard character as we call it can replace any sequence of characters — even spaces or quotes — in a file name or path component. You can, for instance, refer to all subfolders in the current directory whose names start with `lib` by the wildcard pattern:

```
lib*/
```

The ending slash tells the command prompt that you are referring to folder names instead of file names.

There is also the wildcard question mark. This wildcard represents any single character. So if you had five numbered files starting with the word `data` and ending in a `dat` extension, you could group them all into a single command with:

```
data?.dat
```

This would match all of the numbered files `data1.dat`, `data2.dat`, etc.

C.2 Basic Navigation

So what do you do with all these paths and file names? You can do lots of commands with them at the command prompt! Let's look at those dealing with basic exploration, organization, and navigation. Throughout these examples we'll use the sample folder structure at right. Here we have the ~ directory where you initially log in and below it are two more folders. One is for `code` and the other is for `data`. Each has a couple of subfolders below it as well. The `code` ones are organized for the parts of this book and the `data` are just broken down by `old` and `new` data. (The style of the diagram was taken from an actual command-line tool called `tree` that makes what we call directory or folder trees.)

```
~
|-- code
|   |-- Aggregation
|   |-- Appendix
|   `-- Flow Control
`-- data
    |-- new
    `-- old
```

C.2.1 Listing Folder Contents

Let's start with `ls` whih is a command that lists the contents of the specified folder or the current folder if nothing is specified. Like most Unix commands, this is a terse mnemonic for its purpose. We don't really like to type a lot — as you may be gathering.

So, we could do:

```
ls
```

To list the contents of the current directory and we'd see:

```
code        data
```

Or we could do:

```
ls code
```

to list the contents of the specified folder (`code`). Here we'd see:

```
Aggregation      Appendix      'Flow Control'
```

We can even list just a particular file like this:

```
ls file\ name
```

This only lists the file's name, though:

```
'file name'
```

and we already knew that. To see more information about the file, we need to request a longer listing format. This is done with the command-line flag `-l` like so:

```
ls -l file\ name
```

Now we see not only the name but also the permissions, the date of creation, the file size in bytes, and much more!

```
-rwxr--r-- 1 user  user 1317 May 26  2022 'file name'
```

Two other flags we often use with `ls` are `-a` to list all files including those normally hidden and `-h` to list those file sizes in human-readable form with k, m, g, etc. prefixes. If you want more than a single one of these at a time, you can merge them without the dashes like so:

```
ls -lah
```

The order of the flags is irrelevant. Just that they are present after a dash makes them work.[4]

```
drwxr-xr-x 1 user  user 4.0k Jul 28 15:31 .
drwxr-xr-x 1 user  user 4.0k Jul 28 15:06 ..
-rwxr--r-- 1 user  user 1.1k May 26  2022 'file name'
```

I've cut us off after the first file to just show the relevant parts.

C.2.2 Tab Completion

Since we get tired of escaping and quoting all of our spaced names, though, we also invented a tool to help. It is called Tab completion and is pretty straightforward. You type part of a filename and then hit the Tab key. The command prompt then tries to find a file with that part of the name at the start and completes it for you as far as it isn't in conflict with another name.

So let's say you had those five data files from before. You could type just d and hit Tab. The system would then complete `data` for you and beep or at least stop. This indicates that there is confusion at that point and more information is needed. If you hit Tab a second time immediately, the system will display all the files it is considering here so you can see the conflict. Like so:

```
$ ls -l data
data1.dat   data2.dat   data3.dat   data4.dat   data5.dat
```

Here I've added a $ to indicate the command prompt. Many will end with this character while others will use % instead.

Once the list is shown you can type one or more characters to clear up the confusion and hit Tab again. So if you typed 3 and hit Tab, the system would complete `data3.dat` for you.

When the system completes to the end of a filename, it puts a space after the name. When it completes to the end of a folder name, it puts a / afterwards.

C.2.3 Making Folders

But we probably don't want all our files in a single folder. I suppose it is a style of organization, but it isn't much of one — no offense. To make new folders, we have to remember that they are typically called directories in Unix and so the command is `mkdir` to make a directory:

```
mkdir new\ folder\ name
```

[4] If you want to name a file with a dash at the front, you'd have to use an escape or quoting to refer to it at the command prompt due to this convention of prefacing flags with a dash.

Then you can arrange you files to be in different folders based on their purpose. This can be done in a file transfer tool (which could also make folders, probably) like the WinSCP or FileZilla espoused above in the setup appendix for various operating systems. Or you could do it at the command line for convenience.

To make the directory structure in our sample diagram above, we would do the following:

```
$ mkdir code
$ mkdir code/Flow\ Control
$ mkdir code/Aggregation
$ mkdir code/Appendix
$ mkdir data
$ mkdir data/old
$ mkdir data/new
```

Note that the commands just return a new prompt with no message as to success or failure or what was done. Unix, again, is a very terse and programmer-oriented land.

C.2.4 Handling Files

There are two commands to handle files. There is `cp` to copy files leaving one copy where it started and making a new copy in a new location:

```
cp 'current file' 'new location/'
```

And then there is `mv` to move files removing the original and placing it in the desired new location:

```
mv 'current file' 'new location/'
```

As an added bonus, you can use `mv` to rename files that you've decided need a new name as well. Consider it moving the file to a new name:

```
mv 'current name' 'new name'
```

I'll teach you to remove files, but be forewarned: it is typically permanent as there isn't normally a trash can to get things back out of on Unix. The command is `rm` — mnemonic for remove — and you just give it a filename like so:

```
rm 'file name'
```

But remember that there is **NO** undelete! The file is simply removed and that's that!

C.2.5 Changing Folders

Finally, how to you place your terminal into a new folder so that you can use the files you placed there without excessive path typing? Again, recalling the directory thing, we have `cd` to change directories:

```
cd new\ folder
```

Placing a trailing / is optional. You can also use `..` as the destination to move back to the parent folder, of course.

As a special usage, you can also move all the way back to your original login directory by just typing `cd` without a target folder name:

```
cd
```

You can even travel multiple layers at a time like so:

```
cd data/new
```

And Tab completion can complete folder names as well, so feel free to do a `cd` to `code/F1` and hit Tab to finish that out for you. *smile* (As an added bonus, when Tab completing from a `cd` command, only folder names are expanded!)

C.3 Common Commands

There are many common commands in Unix that will help you do certain tasks or utilize files in certain ways as well. Anything from getting free of an infinite loop (see chapter 3) to displaying file contents on the terminal screen and beyond.

C.3.1 Stopping a Runaway Program

If you find yourself with a program run amok, you can stop it by pressing the control key and simultaneously the C key. This closes the errant program rather than copying a selection. Weird, right? (Likewise, Control + V won't paste in a terminal.)

C.3.2 Getting Help

There is a manual in most all Unix environments available from the command prompt. The command to access it is `man` and you just give it a command name and it shows you the manual page one screen at a time right there in the terminal. For instance, you can read much more about `ls` by doing:

```
man ls
```

To get to the next screen, just hit the spacebar. To quit once you've seen enough, hit q.

If you are unsure of a command, you can also use the flag `-k` to find a command that talks about a topic. So if you want to see all the commands related to doing listings, you could do:

```
man -k list
```

Among them you'll find `ls` with a `(1)` after it. This `1` designates the manual section the command is in. If that command is unique, then no section is needed. But if you ever man something that has more details in a later section, the first found section is always displayed! To get to the later section entry, just put the section number before the command name:

```
man 1 ls
```

C.3.3 Displaying Files

If you have a plain text file that you want to display on the terminal screen, you can us the `cat` command:

```
cat 'plain text file'
```

This command name might not seem mnemonic at first, but it kinda is. The creators were considering that displaying the file was like attaching its contents to the end of the screen. So they thought of it as concatenating the contents to the screen. And since that's too big for a command, they shortened it to `cat`.[5]

This is not good to do with a binary file like a compiled program! It can really mess up your terminal to the point you'll have to open a new one.

C.3.4 Paging Long Files

If the file is really long, it will, of course, scroll your terminal up pretty far and you'll have to use the mouse or trackpad to scroll up with it. If you'd like to see it just a screen at a time like the `man` command did, you can use the `less` command:

```
less 'long text file'
```

Again, use spacebar to see another screenful and q to quit before or at the end.

What is this name mnemonic of? Well, the original command was `more` for 'one more screen'. But someone came along with a competing command they thought more beneficial and called it `less` because, as the adage goes, 'less is more'.[6]

C.3.5 Converting Line Endings

Many times you will have created a file on Windows and are trying to transfer it to Unix for use. This works much of the time if you've set up VS Code or some other environment as recommended in the setup (section A.5) to use Unix line endings. But if you didn't, you can still work with these files by running them through the terminal command `dos2unix`. This command is named from the old Microsoft OS from before Windows: MS-DOS.[7]

```
dos2unix 'Windows file to convert'
```

This command will convert the line endings from the Windows 2-byte default to the Unix 1-byte variant. This will help them display on screen correctly with `cat` and also help them be processed by your program as we do in chapter 7.

C.3.6 Formatting Files

Sometimes we've formatted a file so that it won't look nice in certain environments. While the terminal typically wraps longer file lines during a `cat` from one screen line to the next, it often looks odd. And if we are trying to preserve it (see transcript recording below), can even be chopped off when lines are too long. That's why I espoused using a line guide around 75-80 in the VS Code configuration section (A.5).

But if you ended up with a file with longer lines and want it to look good on the terminal screen, you can use the Unix command `fmt` instead of `cat`. It will wrap lines to a given line length automatically and at word boundaries so it looks nice. A blank line in the text file is treated as a paragraph boundary so that you can have more than a single block of text in the file.

This command can be entered as:

```
fmt file\ to\ format
```

for a default 75-length line or you can specify a line length with a flag like so:

[5] Perhaps one of them had a pet they loved? *shrug*

[6] I don't make this stuff up — I swear!

[7] DOS stands for Disk Operating System, if that kind of trivia interests you. *smile*

```
fmt -50 file\ to\ format
```

Here we've specified to wrap lines to about 50 characters long.

If you have control of your Unix machine, you might want to upgrade your toolset to use the `par` command instead. This formats paragraphs really nicely and even does full justification instead of left justification like `fmt` always does. It is a little different in how it works, though:

```
par -j1q0 <'file to format'
```

The `-j1q0` sets you up for full justification if you like that sort of thing. Leave it off it not. If you want a width different than 72, use the `-w99` flag — just fill in the 99 with your desired line width.

The < there is called a redirection indicator and takes contents from a file and redirects it into the given command as if you had retyped all that at a prompt for the program. We could have alternatively used a command-line pipe like so:

```
cat 'file to format' |par -w50 -j1q0
```

The single vertical bar here takes the output from the first command and sends it to the second command as if you had retyped it at that command's prompt.

C.3.6.1 Caveat/Warning

Keep in mind that there is no easy automation for formatting code. There are specialized utilities, but they don't come standard on any platform. Neither `fmt` nor `par` can handle formatting code files!

C.4 Programmer Tools

Some commands are especially useful to programmers like recording a terminal transcript or searching for content in a set of files.

C.4.1 Recording a Transcript

Sometimes a programmer is experiencing something that is hard to describe. If they want another programmer's input on the situation and can't just call them over to look, they might send a screenshot. But sometimes it involves lots of interaction or even dynamic interaction such that a simple screenshot won't do. In these situations a transcript is called for!

The command `script` will record a transcript of the terminal session in a file so that you can later send that file to someone else for review. The file by default is called `typescript`, but this can be changed at the command line. For the default name, just run:

```
script
```

To record to a file other than `typescript`, simply give that filename on the command-line:

```
script 'transcript name'
```

The nice thing about the default name is that you don't have to think about a name and it just overwrites the next time you run `script`. The bad thing about the default name is that it is overwritten the next time you run `script`. If you are preserving a transcript to send to another programmer or the like, having it automatically overwritten might be a bad thing. Worse, you might have two `script` commands running at the same time.

This can happen if while you are recording you realize something else needs to happen and you go do that in another window and when you come back to the terminal you think, "Well, I better start this recording now." It happens a LOT. I see it all the time. And when two script commands are actively writing to the same transcript file, it is a mess!

If the session needs to be more dynamic, you can record timing information with the -T flag. This should go before the filename if that version is used. Then, when the person on the other end gets your transcript, they can use scriptreplay to replay exactly what happened on your terminal in theirs.

BTW, the transcript file is a bit text and a bit binary so it won't look nice with just cat or even fmt. You'll have to do a little more work to prepare it if you want to print it or PDF it. It was designed to work well with scriptreplay, so...

C.4.2 Searching Files for Text

After you've programmed for a while, you'll find that you have many codes you want to reuse in new programs. Chapter 4 on writing functions has lots of tools for easier code reuse, but you still might find yourself with a function name and no idea which file it was in. grep to the rescue!

This command will allow you to search through files for a word or phrase with ease. Just do:

```
grep word *cpp *h
```

This will search through all of your C++ files in the current folder for any occurrence of the given word. Or, if it is a phrase, you can put that in quotes like so:

```
grep 'short or long phrase' *cpp *h
```

And that text will be found instead. The matching line will be printed with some highlighting on what was searched for. If searching more than a single file like we did here, the line will be prefaced by a filename and a colon. If you'd like more context, you can use the flags -C9 for common context or -A9 and -B9 for after and before context. Just change the 9 to your number of lines. So to see 3 lines of context before and 2 after, you could do:

```
grep -B3 -A2 'short or long phrase' *cpp *h
```

To make it a little easier to find in the editor, though, you can also have grep print line numbers with the -n flag.

Or maybe you don't want so much information — just the names of the matching files so you can then load them in the editor and look them over there. This can be achieved with the -l flag (a lowercase L).

But if this weren't enough, we can also use grep to search for fancy patterns! After all, who can remember the exact name of a function written weeks ago — we might just have an idea of how we named it or some inkling of a phrase in its comments.

To handle these situations, we'll need the regular expressions tool. In fact, this is where grep gets its name: global regular expression print.

C.4.2.1 Searching Files for Patterns

Whereas wildcards allowed you to specify groups of files all at once, a regular expression — regex for short — can allow you to match many texts all at once. For instance, you could match any of hello, hallow, or hollow with the regex h.llo.* in your search.

C.4.2.1.1 Regular Expression Basics

In regex patterns, the `.` can match any single character much like a `?` in path or filename wildcards.[8] And the `*` modifier says to match 0 or more of the preceding pattern. So `.*` says 0 or more of any character and plays the role of just `*` in a wildcard match.

But that's just the tip of the iceberg! There are lots more things to match with. You can use groups of characters at a particular position, for instance. So if the above matching `hollow` was a problem, you could exclude it by using `h[ae]llo.*` as your regex. Now only an `a` or an `e` will match between the `h` and the `llo` parts.

The order of the characters in the square brackets is irrelevant — only a single one of them will match at a time unless you put `*` after the close bracket. You can also use ranges of characters in the square brackets like `[0-9a-fA-F]` to match any hexadecimal digit.[9] Note that the match is case sensitive and we had to list lower and upper case A through F to match either.

If you need to have a group include the dash character, you can place it first in the brackets. So this `[-.?]` will match any of dash, period, or question mark. This is also a way to match the dot without it meaning 'any single character'. Another is to use the escape slash. So `Hello\.` will match `Hello` followed by a period. (Again, case sensitive!)

But we've barely scratched the surface! There are ways to represent repetitions of 1 or more, up to *m*, *n* or more, exactly *n*, or even *n* to *m* of the previous pattern. There are ways to group parts of the pattern and repeat them. There are ways to match multiple possible patterns at a time. There are ways to match across line boundaries as well. And so much more!

There are any number of fine tutorials on regex pattern crafting online. Some are video and some are text-based. Use whatever works best for you to learn more on this powerful and fascinating topic!

C.4.2.1.2 regex Everywhere!

Regular expressions are found all over the place, in fact. `grep` uses them to search files from the command line. `less` can use them to search the file being viewed — use / to start a search and n to repeat the search. And even VS Code can use them to search an editor file — see that `.*` off to the side of the Control+F/command+F dialog?

But do be careful — especially when searching for a tutorial — to make sure you are using the right flavor of regex. There are several — almost as many as there are programming languages! `grep` can use any of 3 flavors, in fact. And VS Code uses one for the editor search and another for the search in multiple files dialog. The most popular flavors are Javascript/ECMAscript and Perl/PCRE. But there are others and more coming everyday!

C.5 Wrap Up

In this appendix we've taken a brief overview of the Unix (or Linux) terminal interface. We've discussed issues as basic as forming file and path names and as complex as programmer focused tools. In between we looked at some file handling and other common commands. I hope you put your new-found knowledge to good use soon!

[8] Technically the `.` cannot match an end-of-line character — the one stored for Enter or return and represented by `'\n'` in code.

[9] Hexadecimal is base 16. Since we ran out of digits at 9, we add the letters A through F to represent the next 6 'digits' in this base.

Appendix D

Input and Numeric Formats

Input is a very important aspect of any program so that we can gather the values the user actually wants us to work with this time around. As we implied in the program design section (2.5), we are basically making a general word problem solver: give us a particular word problem and we make a generalized solver for it. So having the numbers and such that the user needs this time is very important.

In this appendix we'll explore more about the input process and numbers in particular.

D.1 The Keyboard Buffer

First off, input doesn't just magically appear in our variables. It starts out as a sequence of keystrokes — `char`acters — sitting in a buffer. What's a buffer? Well, it's a place where things sit waiting to be processed. Sadly, this usage of the word matches no other definition of the word, so I have no parallels for you to draw from.

But envision it as a sequence of `char`acters waiting to be worked on like so:

User Types	Buffer Contents
Hello	`'H'`, `'e'`, `'l'`, `'l'`, `'o'`, `'\n'`
4213	`'4'`, `'2'`, `'1'`, `'3'`, `'\n'`

Note that every buffer ends in a newline keystroke. This is because `cin`'s extraction operator doesn't start working until the user hits Enter/return when they are done typing.

The input of `char` data is pretty straight forward. >> just puts the next non-whitespace keystroke in the `char` variable.

But, as you can see, even numbers are broken down by keystroke before being turned into actual data inside a variable where they can be added, etc. So the next question, then, is how does `cin` do that kind of thing?

D.2 Basic Input of Numbers

Let's take the second example from above and walk through how >> does that translation into an integer for the program to work with. It begins by setting an accumulator to 0.[1] This will be where we add up

[1] Accumulator here is fancy talk for holding onto a sum or total.

the number that the user intended step-by-step.

Next we look at each digit's char in turn and both translate it to a number and put it in the proper place in the accumulator. The translation process could be done with simple subtraction, it turns out. But it is a little weird to subtract char. So we'll use the helper from section 2.4.4 to make our subtraction between ASCII values instead.[2]

Let's call the next digit's char from the buffer next and we'll call its translated value digit. Then what >> does is essentially:

```
digit = static_cast<short>(next) - static_cast<short>('0');
```

This tells us how far from '0' the next digit is. If it were a '0' itself, then it would be 0 away, for instance. If it were the leading digit from above, it would be 4 away. And so on...

Next we need to put this in the proper place in the accumulator. This part is a little tricky sounding at first, but once you see it move a couple of digits along, it comes together.

We start by shifting (multiplying) the accumulator by 10:

```
accumulator = accumulator * 10;
```

Again, this will make more sense after you've done a couple of digits, but for now, we have the initial 0 value times 10 is still 0. Then we add the digit translated above:

```
accumulator = accumulator + digit;
```

For the example from the table above, this gives us 4.

Next we move to the next position in the buffer and repeat.[3] Thus we'll translate '2' into 2, multiply the accumulator by 10 getting 40, and add the 2 to get 42.

Although not complete, things are already starting to take shape. The leading '4' from the user's number is now a step higher in the digit places. It started in the ones place and has now moved to the tens place thanks to the multiplication by 10 of the accumulator. As this process continues, it will continue to shift over until it falls eventually into its final place of thousands.

The process from the beginning, then, is:

Next char	**Translated**	accumulator **Shifted**	**Accumulation**
'4'	4	0	4
'2'	2	40	42
'1'	1	420	421
'3'	3	4210	4213

Neat! Well, I think so, anyway...

D.3 Numeric Formats

So how flexible is this system? That is, what can a number typed by the user consist of? Well, integers can start with a leading plus or minus sign or not. They can then be any sequence of digits. But when it comes to placing it into memory, the accumulator has to fit into the data type's slot. That is, it has to be between the minimum and maximum bounds for the variable's type. These are shown in a table in section 2.3.1.1 for the integer types.

[2] For more on ASCII and the traits of it that make this process work, see appendix E.

[3] We keep going until we reach the first thing that isn't a numeric digit char.

We often represent such specifications in a format known as a regex or regular expression. The regex for the above integer specification is just `[+-]?[0-9]+`. The square brackets here denote a set of values any one of which is allowed. The dash in the second brackets means a range of values. The question mark means the previous set is optional. And the plus means the second set has to have at least one representative but might have more. It's all quite complicated and suitable for a later course to review in more detail. But for now, I just wanted to show you the regex for the floating-point numbers for comparison:

```
[+-]?([0-9]+(\.[0-9]*)?|[0-9]*(\.[0-9]+))([eE][+-]?[0-9]+)?
```

Wow! That's a mouthful! We see a couple of more notations as well here. Basically, the parentheses are used for grouping as usual. The slash as on a `char` escape says the next bit is different than normal. Our problem is that in a regex a period or dot normally matches any single character and we want it to represent itself here. So we escape the typical meaning to make it be itself. Then the asterisks or stars say to match zero or more of the previous item as opposed to one or more like the plus said earlier. Finally, the vertical bar says to match either the left thing or the right thing.

Let's break that down:

```
[+-]?            # optional sign (like on integer)
(                # group for number itself
   [0-9]+        # either 1 or more digits
   (\.[0-9]*)?   #        possibly followed by a . and
                 #                 0 or more digits
 |               #   OR
   [0-9]*        #        0 or more digits
   (\.[0-9]+)    #        followed by a . and
                 #                    1 or more digits
)                # one of these has to be there!
(                # group for scientific notation
   [eE]          # either an e or an E
   [+-]?         # optional sign (as int and above)
   [0-9]+        # 1 or more digits
)?               # sci-not is optional
```

I've added comments off to the side after pound signs as is conventional for these things.

Note that we are including the possibility for scientific notation here and that the `E` normally required to be capital can be lowercase as well. Also note that the number can start with just a dot followed by decimal places and need not have a leading `0` on it. Similary it can end in a dot with no trailing decimal places.

Again, the final accumulated value is subject to fitting into the desired data type. Please see section 2.3.1.2 for more on floating-point data type limits.

Interestingly, these formats are the same as used for the compiler when looking at literals in source code!

D.4 Wrap Up

This was, sadly, a mere brief delving into this topic of the input buffer and particularly how numbers are translated from a sequence of `char`acters to actual numbers we can work with in arithmetic ways. You'll learn more in later courses, though, so don't fret!

Appendix E

Character Encoding

How are letters and symbols stored in the computer? Are numbers always numbers? In this appendix we'll answer such questions!

The answer to the first question is fairly complex so we'll break that down below. The answer to the second is simply no. Numbers start their lives — whether at the user's keyboard or in our own source code — as sequences of individual `char` keystrokes. They are then converted as per the process gone over in appendix D.3.

So that brings us to the question of how are the individual digits stored in the computer? That falls right in line with the first question!

There are actually many answers and which is right depends on your system and sometimes your needs on that system. There are three main players in the field of mapping human letters, digits, and symbols to binary memory values: EBCDIC, ASCII, and Unicode. While others exist, they are less prominent and you may never encounter them.

E.1 ASCII

ASCII is the American Standard Code for Information Interchange. It is essentially a chart by which the letters, symbols, and digits we hold dear are mapped to numeric values for computer memory storage. When a keystroke is hit, it maps to one of these values via the chart. When such a memory value is displayed, it is mapped via the chart to a display form.

As you can see via the link above, ASCII comprises the English alphabet in both upper and lower case forms, the 10 Arabic digits for forming numbers, and various punctuation marks and other symbols we find useful on a daily basis. It even has slots for spacing characters like the Spacebar, Enter, and Tab keys. Essentially, it has the keyboard values and a few others.

But there are also a few items below the Spacebar called control characters. These are/were used for controling various things between parts of the computer. Some were primarily used for modem communications and may still be used in some communication applications today. Others were intended to control aspects of printing on paper. Since we do little of either today, these codes are little used anymore. But they are still there because ...well, why remove them?

E.2 EBCDIC

EBCDIC is the Extended Binary Coded Decimal Interchange Code.[1] It is another chart that maps human symbols to binary memory values. It is primarily used on older mainframe computers whereas ASCII and Unicode are mostly used on personal computers like a Windows™ box, a Mac™, or a Chromebook™.

It's arrangement was quite different and can give some difficulty in common operations like detecting what group a character belongs to or changing the case (upper versus lower) of a letter. More on that in a bit (section E.4). This is because of the gaps (see the grayed out spaces in the chart above) between parts of the alphabet for instance.[2]

E.3 Unicode

The Unicode Standard (a set of translation tables that aim to be a universal collection of all human symbols) is for more advanced programs that need heavy internationalization. It is represented in C++ by the `wchar_t` data type and the associated wide strings (`wstring` for the `class` type). A full discussion is beyond the scope of this text, but please see online for more detailed readings on the matter.

The main thing to know about it is that its first — bottom-most — table is essentially the ASCII table although it is called the Latin-1 or Basic Latin subset.

E.4 Contiguous Runs

The main nice thing about both the ASCII and Unicode systems is the contiguous runs for things like the alphabet and the digits. This makes detecting that we are alphabetic or even which case or a digit easier and even makes it easier to convert upper and lower case back and forth when necessary. The fact that the digits are contiguous in both of these systems and in EBCDIC makes converting from characters to numbers easier as well.

E.5 A Stern Warning

Don't try to remember the numeric versions of the symbols, letters, etc — that's for uber-geeks. Normal programmers — GOOD programmers — use single-quoted literals in their source code to refer to particular character values they need. So use `'a'` instead of 97, for instance. One reason is that the numeric valules change from system to system — ASCII to EBCDIC, notably. If you worry at all about portability of your code — and you should! — then don't code with numeric values for characters. Always use single-quoted literals instead.

E.6 Wrap Up

Hopefully this exploration of different ways to encode human symbols in the computer has proven interesting. Feel free to explore more on your own!

[1] Slightly redundant there, eh?

[2] While the gaps contain valid codes, they don't align with the ASCII table and that was the purpose of the chart I linked above. If you read elsewhere on the page it explains the other bits of the code in more detail.

Appendix F

Timing Program Events

The idea of this appendix is to talk about automating the task of finding out how long certain parts of the program are taking to execute. There are different methods available depending on how far you've come in your studies. Let's start basic with the ctime function: time(nullptr).

F.1 Using time(nullptr)

Since time(nullptr) returns the number of seconds since a fixed point in the past — the computer epoch of January 1, 1970, we can use two readings from it to tell how far apart they are and therefore how long the task between the two readings took!

Um, what? Just call the function twice — once before and once after the code is run. Then subtract to find the number of seconds elapsed. Oh. . . why didn't you say that in the first place?

```
time_t start, end;

start = time(nullptr);

// do code to be timed

end = time(nullptr);

cout << "That took " << end-start << " seconds.\n";
```

But, since most code events are blindingly fast, we might not have even a single second for some of them. We'll need to do one of two things with this idea: time lots of repetitions of the event and take an average or find out how many times the event can run in a single second and take the reciprocal.

F.1.1 Method One

The first way is to average many repetitions of the code event we want to time:

```
time_t start, end;

start = time(nullptr);

for (long i = 0; i != LOTS; i++)
{
    // do code to be timed
}

end = time(nullptr);

cout << "That took " << (end-start)/static_cast<double>(LOTS)
     << " seconds.\n";
```

You just have to decide how much LOTS should be. You might even make this a program input and adjust it on different runs to see if you get better results.

F.1.1.1 Dealing with Data Re-Organization

One thing to watch out for, of course, is making the timed event in the loop consistent. For example, if you were trying to time a sorting algorithm with a short-circuit condition (like bubble sort), timing this sort directly 10000 times would really only time the whole algorithm once and the single short-circuit loop 9999 times. A single loop through a `vector` isn't very slow — nearly instantaneous for most `vector`s. So your timing is likely to still be 0.

To adjust for this circumstance, you can do one of two things: replace the original `vector` contents between sorts or randomly shuffle the data between sorts. Although it may not seem like it, either of these approaches will give you comparable timings. The problem is, this copying or shuffling must be done inside the timing loop and so your timing will reflect not only the sorting done, but the `vector` re-organization as well. To compensate for this, simply time the re-organization separately and subtract this from your 'sort' timing:

```
time_t start, end;
double reorg_time;

start = time(nullptr);
for (long i = 0; i != LOTS; i++)
{
    // re-organize vector
}
end = time(nullptr);
reorg_time = (end-start)/static_cast<double>(LOTS);

start = time(nullptr);
for (long i = 0; i != LOTS; i++)
{
    // re-organize vector
    // sort vector
}
end = time(nullptr);

cout << "That took " << (end-start)/static_cast<double>(LOTS) - reorg_time
     << " seconds.\n";
```

Even if you are comparison-timing many sorts, the re-organization timing only needs to be done once.

You can also improve your timing by making the data larger — not the actual values, but the number of data elements. If the thing you are timing is `vector` processing, for instance, you can increase the number of elements in the `vector` that are being processed.

F.1.2 Method Two

This method of timing program execution also uses the `time(nullptr)` function from the `ctime` library, but it takes a different approach. In the previous method, we ran our code many, many times and calculated how many seconds this took — on average. Another way to look at this idea is that we aren't sure how many times it will take running our code to make a second, so we've instead just run our code long enough to make many seconds and divided out how many times we ran to get an average:

$$\frac{seconds}{runs} = \text{sec/run}$$

However, since we expect our code to run multiple times each second, we could simply run it enough times so that the current second changes. Then, by counting how many runs that took (i.e. how many runs we did in a second), we can calculate:

$$\frac{1}{\frac{runs}{second}} = \text{sec/run}$$

That's just what we wanted and it only took us a second instead of the several/many seconds the previous method took!

```
double reorg_time;
unsigned long count;
time_t start;

count = 0;
start = time(nullptr);
do
{
    // re-organize/setup
    count++;
} while (time(nullptr) == start);
reorg_time = 1/static_cast<double>(count);

count = 0;
start = time(nullptr);
do
{
    // re-organize/setup
    // do code to be timed
    count++;
} while (time(nullptr) == start);

cout << "That took " << 1/static_cast<double>count - reorg_time
     << " seconds.\n";
```

F.1.2.1 Adjusting for System Differences

Well, sort-of... In fact, because of all the things computers are always doing, we'll not likely get an accurate reading all the time. To fix this, we should take several such readings and then find the median number of times the code ran in a second.[1] Even so, this method can still be done in a consistent number of seconds (say 9-10) instead of the more arbitrary time that the previous timing method took.

We should still take account of re-organization/setup time for this method, but we can use this method to time both the setup and actual code running. *smile*

So, what might this look like? Something like this, perhaps:

```
vector<unsigned long> counts;
double reorg_time;
unsigned long count;
time_t start;

for (short rep = 0; rep != 9; rep++)
{
    count = 0;
    start = time(nullptr);
    do
    {
        // re-organize/setup
        count++;
    } while (time(nullptr) == start);
    counts.push_back(count);
}
// sort counts vector
reorg_time = 1/static_cast<double>(counts[4]);

counts.clear();
for (short rep = 0; rep != 9; rep++)
{
    count = 0;
    start = time(nullptr);
    do
    {
        // re-organize/setup
        // do code to be timed
        count++;
    } while (time(nullptr) == start);
    counts.push_back(count);
}
// sort counts vector
cout << "That took " << 1/static_cast<double>(counts[4]) - reorg_time
     << " seconds.\n";
```

And as your knowledge grows, you'll likely find more and more ways to time events on different systems.

[1] See the elsewhere for information about taking a median of a set of data.

F.2 Wrap Up

No matter what way you choose to time a program event — some bit of code, always remember to have fun!

About the Author

Jason James graduated with his BS in Computer Science (minor in Applied Mathematics) and his MS in Computer Science (Theory emphasis) both from the then University of Missouri at Rolla (UMR) now MST (Missouri University of Science and Technology). While working on his PhD (Artificial Intelligence emphasis; ABD), Jason taught introductory programming to engineering students using both FORTRAN and C++. He also taught Freshman and Sophomore topics in Computer Information Systems at an adjunct campus of Columbia College during this time.

When he moved to Chicagoland and the wonderful world of William Rainey Harper College, Jason focused on teaching Freshman and Sophomore Computer Science courses and the occasional mathematics course — especially Discrete Math. During his twenty years at Harper, Jason has also taken over as chair of the Computer Science department; served on committees for Curriculum, Academic Standards, and Testing and Placement; co-mentored the Robotics/Engineers club; and started a Computer Science club last Fall (hopefully converting to an ACM Student Chapter this Spring), but teaching remains his focus and heart.

Jason maintains membership in the international Computer Science group: the ACM (Association of Computing Machinery). He also keeps up his membership in the IEEE (Institute of Electrical and Electronics Engineers, Inc.). In both he is especially interested in their education-focused sub-groups. Jason also tries to attend conferences of the CCSC (Consortium for Computing Sciences in Colleges) whenever he can get away.

When not playing Dungeon & Dragons with his wife and friends, he and his wife take care of their two beautiful sons and two ornery cats. In his *spare* time, Jason enjoys developing formulae for counting the results of dice rolls — with an eye toward a 'nice' standard deviation formula; using the LaTeX type-setting system to prepare lecture supplements and exams; creating lecture supplements and assignments online; and the development of a calculator language & its interpreter which are being applied to classroom management software (such as a gradebook and an exam analyzer). He's also recently taken to writing — converting those lecture supplements into an OER textbook for [at least] his students at Harper College.