20240110-0300

Ground-up algorithm design with modern C++

Don Blaheta
Longwood University

Spring 2024 edition

Welcome to CS1!

You're currently paging through a document I wrote for my CMSC 160 course at Longwood University. Textbooks are important and valuable, but it's been clear for a while now that they're not $150-valuable, or whatever the typical number is now. This is a course pack that is textbook-ish, but free, and actually follows the order I generally teach things. On the other hand, it's a bit shorter than a standard textbook and not quite as well-filled with exercises and review questions, or reference content; I strongly encourage you to *also* look around for other online resources that click with you.

How to use the course pack

In general, treat the sections assigned for a particular day like you might treat a posted video lecture to watch before class: watching the video, or reading the section, preps you for the content of the day so that you are optimally ready to learn. If the section raises questions in your mind, that's great! Ask them in class—other students probably have the same questions.

When there are programs in the reading, or sections of programs, you should type them in and run them! Merely reading them will give you passive familiarity with some of the programming ideas, maybe, but the only way to really learn this stuff is to actively engage with it, write programs, make mistakes, fix them, and ask lots and lots and lots of questions about all of it.

Also, if you spot any typoes, or any parts that you found confusing and have ideas about why it was troublesome (and/or how I might fix them) please let me know. This is a work in progress. :) (If you're using this book somewhere other than Longwood, you might not know me but your professor probably does! If you give them the book feedback they'll be sure to pass it along.)

—Don Blaheta
Farmville, Va.

Contents

Chapter 1

Algorithms: precise and abstract

v20231026-0100

In this book and this course we'll be developing a number of skills. In addition to the most obvious one—programming—we'll use programming projects to develop ways of thinking that will be of use in larger projects and even outside of computer science: precision, and abstraction.

When you give instructions to another human about how to do something, you can usually rely on them to fill in some details. If the instructions are ambiguous or unclear, they'll either use their own judgement to resolve what to do, or ask for clarification. Computers don't have that kind of judgement, so rather than "asking for clarification", a computer faced with even slight ambiguity in a program will simply reject it and make you fix it. Modern systems are usually pretty good at highlighting *where* the ambiguity is, so you will be able to fix it, but as time goes by, you'll get better at thinking precisely about a task and specifying exactly what you intend to happen.

But sometimes you can provide instructions precisely—clear and correct instructions—while some details are still up in the air, by including how to react to the remaining unknowns. A knitting pattern that can be made in size small, medium, or large, with instructions on what to do differently for each one, can still be perfectly clear and well-specified. Furniture assembly instructions can be written for two similar models by having a segment in the middle that specifies "for model A, do these steps, but for model B, do these steps instead—then return to the shared instructions for both models". We say that such instructions are abstract (rather than concrete) when the writer leaves some of the details open but is still able to fully and carefully specify what should happen in each circumstance.

algorithm

An *algorithm*, in computer science, is a series of instructions that is clear and precise, aiming to achieve a goal, which is guaranteed to finish once applied.[1] All algorithms must be precise, by definition, and virtually all interesting algorithms are at least partially abstract, working flexibly in reaction to different users or times of day or locations or some other source of variation.

Precision and abstraction are part and parcel of any programming effort, and in this course you will develop both thinking skills. Now let's write some programs!

[1] Informally, the word "algorithm" can also refer to ranking strategies used by social media platforms to choose what to display to their users. To relate this back to its formal definition, we can consider the task or goal as something like "given a particular user at a particular time, identify the next post or video that they should be served".

Chapter 2

Output, input, and variables

v20231222-0100

In this chapter, we'll write and execute our first programs.

2.1 Hello world, output, compile-and-run

In the book that introduced the C programming language—a predecessor of the C++ that you'll be learning here—authors Brian Kernighan and Dennis Ritchie presented a short C program that printed the text "hello, world", intended as the shortest and simplest program in that language. Since then, it has become conventional for a programmer's first program in *any* language to be one that simply produces the text "Hello, world!". Simple, but it ensures that the tools you're using let you type in a program, and run it, without any additional fuss.

In a project on `codeboard.io`, or any other environment that lets you type in C++ programs, enter the following program (some of which may already be filled in for you):

```
// First "hello world" program
#include <iostream>
using namespace std;

int main()
{
  cout << "Hello, world!" << endl;
  return 0;
}
```

Program fragment 2.1: First “hello world” program

Attend carefully to detail here: although some details (spacing, for instance) can be changed without altering the meaning of the program, most of the program needs to be typed in *precisely* as shown above, or it won’t work correctly.

Once you’ve typed in your program, most C++ programming environments require two distinct steps to execute your program: first *compile*, then *run*. The compiling step verifies that what you’ve typed is in fact a valid program—not that it’s 100% correct but at least that it is clear and will do something—and the *compiler* translates it into a machine language that the computer can run directly. Running the program, then, actually executed the machine language according to the instructions you’ve written.

compile

run

compiler

Try it now! Compile your program, and then run it. If you’ve typed it exactly correctly, the compiling step will run with no errors and the running step will put “Hello, world!” on the screen. If the compiling step gives error messages, look carefully at what you typed, particularly on the lines that the error messages indicate. Fix any problems and try it again!

Once you’ve got a program that successfully produces “Hello, world!”, don’t move on just yet. Starting from a definitely-working program, it can be highly instructive to intentionally make some changes and see their effect. Try changing something about the program, and then re-compile it. Read the error message, if any; since you made the change yourself, you know exactly what you typed to cause that particular message. Much like a scientific experiment, you can vary tiny details of your program and see how the compiler reacts to the change, as a way to build your understanding and mental model of how the compiler works.

2.2 What are all those pieces?

The program you typed in was eight lines long and it will be a while before all of them will be fully explained. For the next few chapters, all your programs will begin with the same few lines, and the differences will be in the very first line describing the program, and the line or lines between the curly brackets `{ }`.

The identifier `cout`, usually pronounced "cee-out", indicates that something is being sent "out" as *output* from the program to the screen, or sometimes as *printing*.[1] The identifier `endl` marks the end of the line of output. A semicolon will appear at the end of most instructions that you write for the computer. And the arrows (`<<`) mark out the different elements of the output stream, pointing towards the `cout` because here the information is being sent towards the output.

`cout`
output
printing
`endl`
`<<`

2.3 Input and output

The program from the previous section runs exactly the same every time and always produces the same message, but most interesting programs will behave differently, in reaction to some information provided by a user. We refer to that information as *input*, moving from the user *into* the program, by analogy with the output that moves *out* from the program back to the user.

input

Edit your existing project or start a new one, and type in the following program, and compile and run it:

[1] The term "printing" is a callback to a time—many decades ago now—when *all* output was printed on paper, but the term stuck and we use it even when "printing" to the screen.

```
// Program with input and output
#include <iostream>
using namespace std;

int main()
{
  cout << "Type your name:" << endl;

  string firstname;
  cin >> firstname;

  cout << "Hello " << firstname << endl;
  return 0;
}
```

Program fragment 2.2: Program with input and output

When you run it, notice that it produces the first output (“Type your name: ”) and then seems to halt. It’s waiting for you to provide information into the system. Once you type a name, and press enter, it will then continue to the final statement, which prints a greeting using that name.

Run the program a few times, trying different inputs to see how it reacts, and identify the limits of the program so far. What does it do if the name includes punctuation, like a hyphen or apostrophe? What does it do if you start the input with spaces? What if there are spaces in the middle of the name?

Now try changing the program itself. The two lines in the middle read the user’s name as input—what happens if those lines change? Can you read two different names? How can you modify the final output command to print more than one name?

2.4 Variables and types

You might have noticed an interesting difference between output and input in the program from the previous section. Output is triggered with the use of `cout`, arrows, and some information to send. But input is indicated by the use of `cin` (pronounced “see-in”) and arrows (`>>`) towards a placeholder to be used later, and that placeholder needs to be set up in advance.

`cin`

`>>`

The placeholder is called a *variable*—its value might be vary, be different, the next

variable

time the program runs—and can be thought of as a *named value*, or more precisely, a named *container* that holds a value that we'll want to make use of later. We don't know in advance what name a user will type in, so we refer to it in the abstract as `firstname`. Any time you read input from a user, you'll use a variable to do so. Its name should be descriptive, if possible; it can't include spaces or punctuation but it can include the underscore character ('_') if you find that useful to visually separate "words" in a longer variable name. It can also contain digits like `1` or `2` as long as it doesn't start with them.

Setting up a variable is referred to as *declaring* the variable, and is done by indicating the *type* of value that the variable will hold. For now there are three possible types of values that we'll work with:

declaring

type

`string`	General text, anything you can type on a keyboard
`int`	Integers: whole numbers and their negatives
`double`	Other numbers: if you'll need fractions or decimals[2]

In the previous program, the requested name wasn't a number, so we declared it as a `string`. Compare to the following program (type it in and try it!):

```
// Doing a little arithmetic
#include <iostream>
using namespace std;

int main()
{
  cout << "Type a whole number to triple:" << endl;

  int number;
  cin >> number;

  cout << "Triple " << number << " is " << number*3 << endl;
  return 0;
}
```

Program fragment 2.3: Doing a little arithmetic

We'll see more about arithmetic operators in a few more chapters, but for now use the rule of thumb that if a value is one that you'll want to do math on, you should declare it as a number (either `int` or `double`), and if you'll just want to hang onto the text without doing math, it's just text (a `string`).

[2] The name "double" for this type may be a little surprising and is not obviously related to its purpose; it got its name because it can store numbers *twice* as precisely as an earlier standard type named "float". The full name for the data standard that `double` corresponds to is a "double-precision floating point number".

Chapter 2 Sidebar: Errors

v20220822-0115

If you were very, very careful—and a little lucky—you might have typed in that first program without errors. Even if you did, back in Section 2.1 you were encouraged to actively break your program in a variety of ways.

It's really important to emphasise that error messages are your friend here, not your enemy. Every programmer, even the most experienced ones, make errors in their code very day; the essence of software development is in discovering how the program ought to be written by trying things, observing them, and fixing them. Every time you get an error message, take it as constructive feedback on where to look and, maybe, what's wrong.

Any compiler error will give you a filename and line number where it detected the problem, and generally some summary of the problem. Take both with a grain of salt: the line number is often *close* but sometimes a little after where the problem actually is. And the identification of the problem is the compiler's best guess, but even if it suggests a fix, you should keep your brain turned on and see if the suggestion makes sense. Maybe it's right! But maybe not.

That's why it's a good idea, early in your C++ programming career (i.e. now), to take an actually working program and try breaking it in little ways, because then you know exactly what you changed—and now you will have a better idea what the error messages are really pointing to. Start a notebook (on paper or maybe in a file somewhere) of error messages, connecting the wording that seems confusing with what the actual problem actually was. Then, when you get that error message again in a less-controlled situation, you'll have a better guess how to fix it. Taking notes like this will also let you ask better questions later, about what various words mean or how to interpret certain phrases. (Writing down the message exactly *also* makes it easier to do an internet search for what it might mean!)

Chapter 3

Conditional execution

v20231219-1730

In this chapter we'll see how to make a program execute certain instructions only under certain conditions, rather than every time the program runs.

3.1 Comparison, `if`, `else`

In most programs of any interesting depth, there will be at least some behaviour that will be different depending on what exactly the user types. If you envision the pattern of executing instructions as flowing from the first instruction to the end—sometimes called a *flowchart*—then conditional execution represents a fork in the road.

flowchart

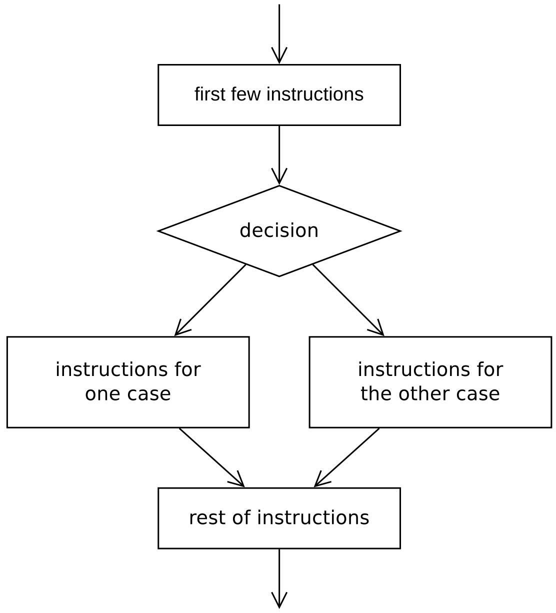


Figure 3.1: A fork in the road

A C++ program corresponding to this flow pattern needs to specify how to make the decision—for now, it will be a comparison between a variable and some other value—and then the instructions to follow in each case. Consider (and try typing in!) the following program:

```
// Demonstrating if and else
#include <iostream>
using namespace std;

int main()
{
  cout << "Type any integer:" << endl;

  int number;
  cin >> number;

  if (number < 0)
  {
    cout << "Your number was negative!" << endl;
  }
  else
  {
    cout << "That definitely wasn't negative." << endl;
  }

  cout << "That's all for now." << endl;
  return 0;
}
```

Program fragment 3.1: Demonstrating `if` and `else`

Several parts are as we've seen before, but focus now on the eight lines beginning with the keyword `if`. That line contains the comparison condition itself; then there are some instructions in curly brackets to follow if the comparison is true, and some other instructions for if the comparison is false.

The keyword `if` is, in C++, always followed by a pair of parentheses with a condition to check. Here, we see an inequality between the value in a variable (`number`) and a literal exact value (`0`). The less-than and greater-than comparisons on numbers work just like you'd expect from math class.

`if`

Next, we see a *block* of instructions, that is, lines enclosed by curly brackets `{ }`. We've already seen that curly brackets enclose the entire program, but blocks like these also represent smaller pieces of a program that are controlled by *control statements* such as `if`.

block

control statements

Then we see the keyword `else` followed by another block. The word "else" exists in

`else`

English but usually isn't used quite in this way; a better translation into informal English might be the word "otherwise", indicating the alternate path to take when the first path isn't chosen. But the word "else" is just four letters long—less to type—and more importantly it's standard across all programming languages with exactly this meaning, so `else` it is.

3.2 `if` by itself, equality

Sometimes the flow of the instructions involves simply deciding whether or not to do something:

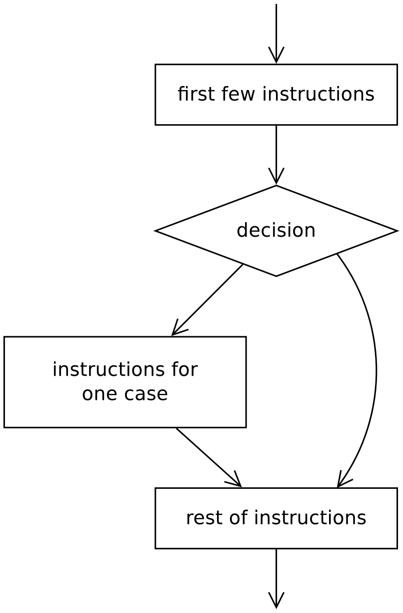


Figure 3.2: Instructions on only one branch

If the "else case" is simply to do nothing at all (and then continue on to the shared rest of the instructions), you can certainly still be explicit about that, and leave the `else` block as an empty pair of curly brackets, but you can also simply omit the word `else` and the associated block. Consider the following program fragment:

```
string name;
cin >> name;

if (name == "Nobody")
{
  cout << "That's an unusual name." << endl;
}

cout << "Hello, " << name << endl;
```

Program fragment 3.2: Using `if` without `else`

(Note that starting now the program examples will sometimes not include the entire program—when you type them in you'll have to type in the standard first few lines yourself. Refer back to earlier program examples if you need to!)

This program expects the user to type a name, and only in the special case where that name is literally `"Nobody"` it will make a comment about the name, but then in all cases (*including* if the name was `"Nobody"`) it will greet the user by name.

There are a few other things to observe about this conditional statement. First, the comparison itself: when the condition is an equality comparison, we write it with `==` (sometimes read aloud as "double-equals" or "equals-equals")—it's important that you use `==` instead of `=` for this, because `=` does something a little different, which we'll see in the next few chapters.

`==`

Second, notice that the name `"Nobody"` is written in double-quotes. As we've previously seen in `cout` statements, double-quotes are a way to tell the program to use a literal exact sequence of characters, and indeed we will refer to this construct as a *string literal*. (Unlike some other programming languages, in C++ a string literal must be surrounded by double-quotes, as single-quotes do something a little different.) By contrast, the identifier `name` is *not* surrounded by double-quotes, because we don't want to refer to the exact sequence N-A-M-E but rather whatever value is stored in the variable `name`.

string literal

All four of these changed `if` lines represent a subtle and slightly different error in the above program:

```
if ("name" == "Nobody")

if (name == Nobody)

if (name == 'Nobody')
```

```
  if (name = "Nobody")
```

Try typing each one as a replacement for the original `if` line, and then compile and run the program to see if you can see how its behaviour changes!

3.3 Multi-way decisions: `else if`

The basic functionality of `if` and `else` enables us to make either-or decisions about how to proceed, but it is frequently the case that we have three (or more!) paths to manage:

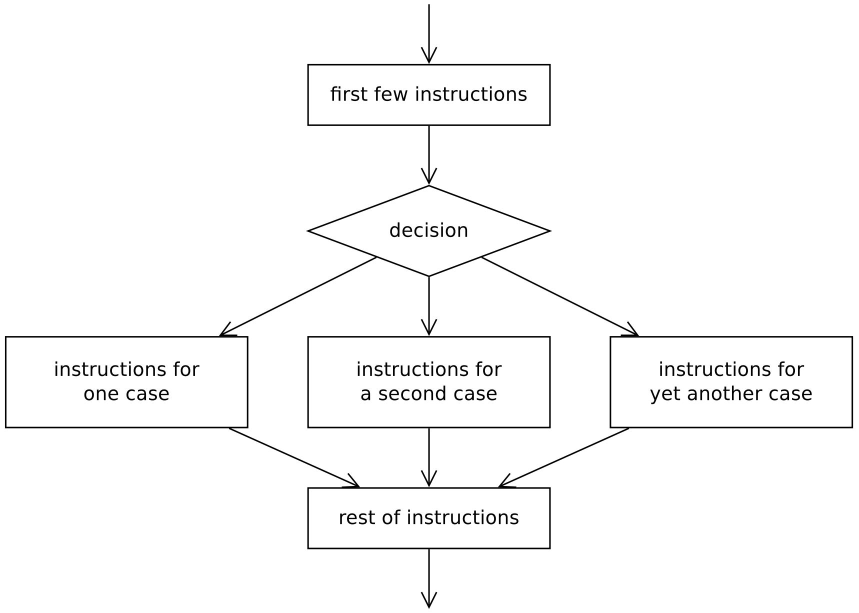

first few instructions
decision
instructions for one case
instructions for a second case
instructions for yet another case
rest of instructions

Figure 3.3: A three-way decision...

To encode that into a program, we reframe it as a series of two-way decisions: are we in the first case? otherwise, are we in the second case? And so on:

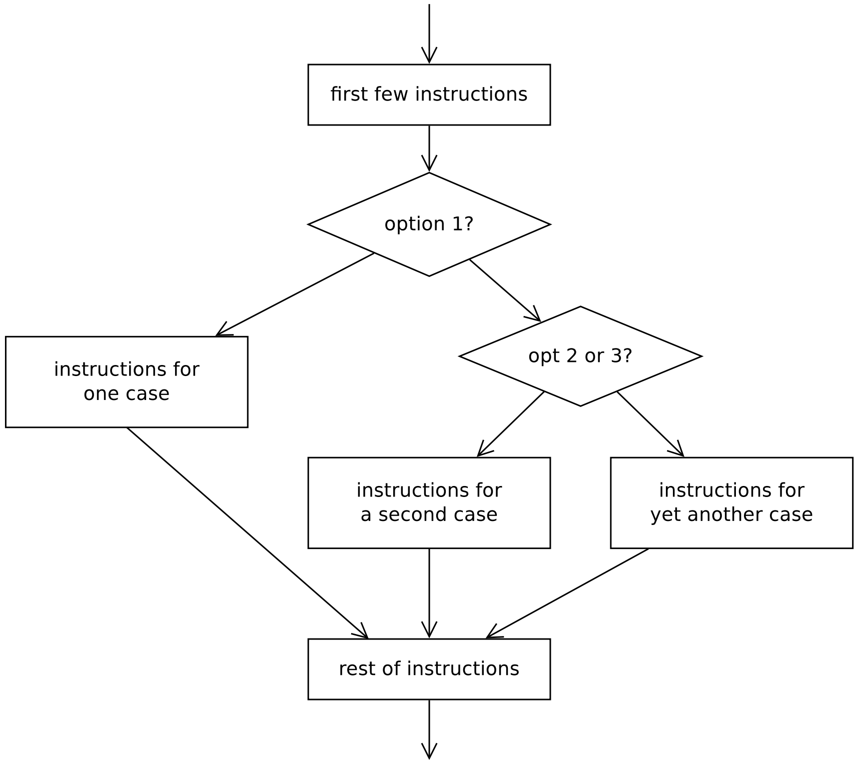

first few instructions
option 1?
instructions for one case
opt 2 or 3?
instructions for a second case
instructions for yet another case
rest of instructions

Figure 3.4: ...restructured as two two-way decisions

You might notice from the use of “otherwise” in that informal description that our C++ solution to this will involve the use of `else`. The following code fragment shows a three-way decision being made:

```
int age;
cin >> age;

if (age < 13)
{
  cout << "Your teenage years are ahead of you." << endl;
}
else if (age < 20)
{
  cout << "You're a teenager!" << endl;
}
else
{
  cout << "You are no longer a teenager." << endl;
}
```

Program fragment 3.3: Using `else if` for multi-way decisions

There's no real limit to the number of clauses you can chain together like this. For a five-way decision, you could write the initial `if`, three separate `else if` clauses, and then the final `else`. You could even make a twenty- or hundred-way decision (but it would be tedious to type and there probably is a better way to do it).

There's something else interesting about the condition on the `else if` line, though. Not all people under 20 are teenagers, but the condition on that line is simply `age < 20`. Why does that work? Look back at the flowchart: we only even *ask* this second question if the answer to the first question was "no" (or "false"). So the comparison `age < 20` is only being performed in the cases where `age` is *at least* 13.

If what's intended is a multi-way decision like this, it's important that all the middle clauses in the chain are written with `else if`. To see why, try changing that middle line to simply

```
if (age < 20)
```

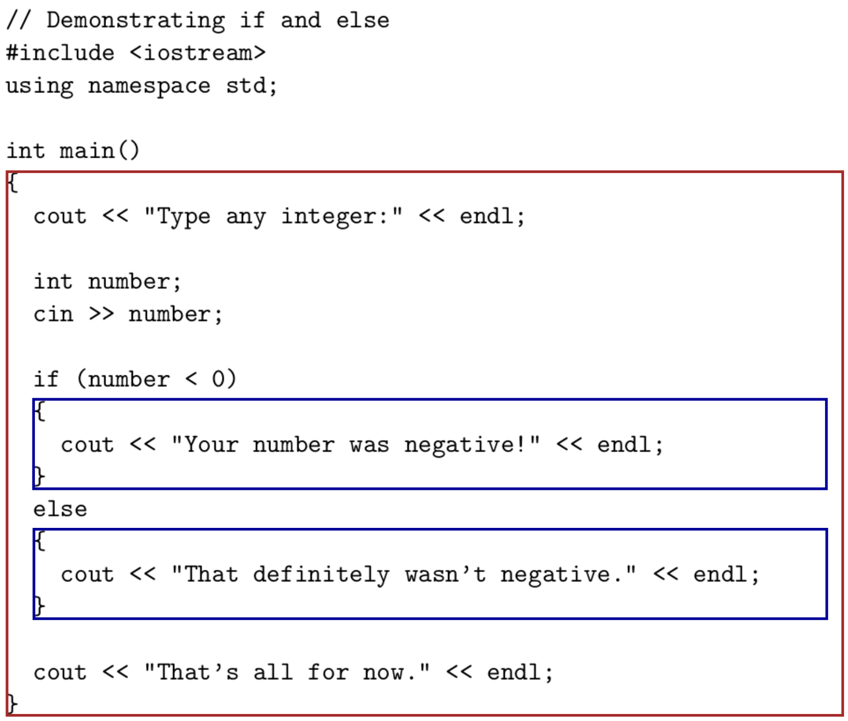
and try running it with various different ages to see how it behaves.

3.4 Nested conditionals

The last variant we'll see in this chapter is something called *nesting*, that is, putting

nesting

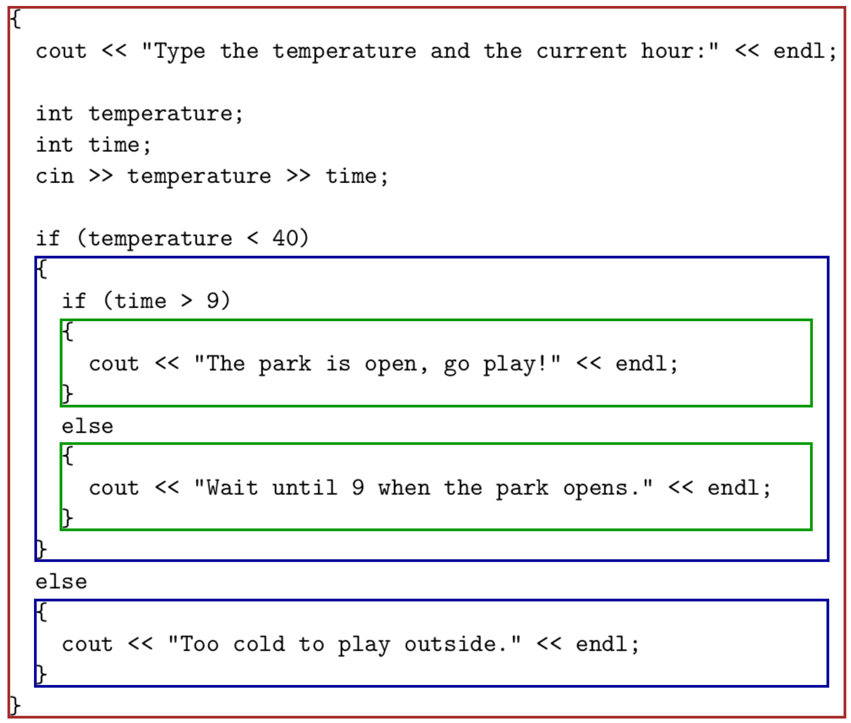
one block inside another. In some sense, we've already seen an example of this: everything between the curly brackets under `int main` is a block, the block that contains all the program instructions; and the curly brackets under `if` and `else` are also blocks, but inside the larger block. One way to visualise this concept of blocks is to imagine that each curly bracket is marking two corners of an invisible box, whose contents are the instructions inside the block. Here's a visual of that, with the first program from this chapter:



```
// Demonstrating if and else
#include <iostream>
using namespace std;

int main()
{
  cout << "Type any integer:" << endl;

  int number;
  cin >> number;

  if (number < 0)
  {
    cout << "Your number was negative!" << endl;
  }
  else
  {
    cout << "That definitely wasn't negative." << endl;
  }

  cout << "That's all for now." << endl;
}
```

Figure 3.5: Showing boxes around blocks of code

We've already said that the contents of an `if` block can be multiple instructions; in fact it can be arbitrary code, including other `if` blocks. *Anything* that can go inside a program (the red box in the diagram) can go inside an `if` block or any other block. When the contents are themselves a conditional statement, the usually-invisible boxes might look something like this:



```
int main()
{
  cout << "Type the temperature and the current hour:" << endl;

  int temperature;
  int time;
  cin >> temperature >> time;

  if (temperature < 40)
  {
    if (time > 9)
    {
      cout << "The park is open, go play!" << endl;
    }
    else
    {
      cout << "Wait until 9 when the park opens." << endl;
    }
  }
  else
  {
    cout << "Too cold to play outside." << endl;
  }
}
```

Figure 3.6: Nested conditionals with boxes drawn

The green (innermost) boxes are nested inside of one of the blue (middle) boxes in just the same way the blue boxes are nested inside the outermost red box. And while the boxes may be visually satisfying, all the actual structural information in that program is carried entirely by the curly brackets themselves; with a little practice, you'll start to see the boxes even when they're invisible:

```
int main()
{
  cout << "Type the temperature and the current hour:" << endl;

  int temperature;
  int time;
  cin >> temperature >> time;

  if (temperature < 40)
  {
    if (time > 9)
    {
      cout << "The park is open, go play!" << endl;
    }
    else
    {
      cout << "Wait until 9 when the park opens." << endl;
    }
  }
  else
  {
    cout << "Too cold to play outside." << endl;
  }
}
```

Program fragment 3.4: Nesting conditionals inside conditionals

Incidentally, the notion of blocks as “boxes containing instructions” is not just marked by the curly brackets, but also by visual indentation. Human readers of your programs—including yourself—will find it much easier to see those invisible boxes if the “contents” of each box are indented a little more than the stuff outside the box. Good program editors will help you with this, but try to develop the habit yourself as well, when writing programs on paper or on the whiteboard. The compiler might treat this as the same program as the previous one:

```
int main()
{
cout << "Type the temperature and the current hour:" << endl;

int temperature;
int time;
cin >> temperature >> time;

if (temperature < 40)
{
if (time > 9)
{
cout << "The park is open, go play!" << endl;
}
else
{
cout << "Wait until 9 when the park opens." << endl;
}
}
else
{
cout << "Too cold to play outside." << endl;
}
}
```

Program fragment 3.5: BAD: Nested conditionals without indentation

but it's much, much, much harder for a human to read. Practicing making your indentation match your intent will help you in developing the mental models of how your programs are structured, which in turn will help you write better and more complex programs.

Chapter 3 Sidebar: Pseudocode and comments

v20231221-2330

The examples in this book so far have been exact programs or program fragments to type in and work with, written in valid C++ (or intentionally demonstrating C++ errors!). But when you are drafting your own program, it's usually hard to jump directly to the final product, and so we instead make use of the concept of *pseudocode*.

pseudocode

First, let's define some other terms. A *program* is a complete sequence of instructions for a computer to do something (at least, once you get it working). It's written in a programming languege like C++, and it gets automatically translated (by the compiler) into some underlying machine language in order to actually be executed. We sometimes work with smaller pieces of programs, which we can refer to as program fragments or, commonly, *code*. Code, in this sense, is a mass noun (we don't talk about "codes" or "a code"), but it's related to other meanings of the word "code" in that it's required to follow certain rules about how to write it if you want it to correctly be translated as intended. Once it is correctly written, code is precise and unambiguous, and requires no direct human thinking in order to determine exactly what it means or does.

program

code

This provides us with a contrast to what is widely called *pseudocode*. The prefix "pseudo-" indicates it's related to and similar to "code", but isn't quite all the way there. It has no formal grammar or syntax—informally, it often looks a bit like code, but to highly varying degrees. Pseudocode is used in two main situations:

pseudocode

- when planning or first-drafting a program, to avoid having to focus on fussy details of code syntax
- when communicating an algorithm strictly to other humans, to keep the focus on the interesting parts and not the syntax

Crucially, the point is to shift focus away from the syntax details. That's useful when you're first learning a programming language and don't deeply know all the details yet, but it remains useful, and computer scientists at all levels will, at least sometimes, plan and communicate using pseudocode.

What does it look like? That's largely up to you. Consider this program that we've already seen:

```
// Demonstrating if without else
#include <iostream>
using namespace std;

int main()
{
  string name;
  cin >> name;

  if (name == "Nobody")
  {
    cout << "That's an unusual name." << endl;
  }

  cout << "Hello, " << name << endl;
  return 0;
}
```

Program fragment 3.6: Demonstrating `if` without `else`

Almost any pseudocode will strip out repetitive pieces that are "always there" unless particularly relevant; and might omit some things like variable declarations or semicolons (or might include them out of habit). This would be one valid pseudocode representation of that program (shown in computer font because it's staying sort of close to C++):

```
cin >> name;

if name = "Nobody"
{
  cout << "That's an unusual name." << endl
}

cout "Hello, " << name << endl;
```

Program fragment 3.7: Pseudocode for the if-without-else example

As C++ code, this would have a number of notable errors in it, but as pseudocode it's absolutely valid—and can be a useful first step to writing the program "for real".

At the other end of the spectrum, pseudocode can be lightly formatted English text, one step up from a paragraph description. Here is the same program in a more English-y pseudocode:

```
read name
if the name is 'Nobody':
    print that it's unusual
print a greeting using name
```

Figure 3.7: Also pseudocode for the if-without-else example

Even in near-English, we use line breaks and indentation and words like "if" and "else" in pseudocode to maintain some precision about the algorithm, but to most human readers, the above would be a clear and unambiguous specification of the algorithm that is encoded by the program we wrote above.

Comments

Pseudocode is useful for early drafts of programs, or for communicating algorithmic ideas to other humans, but it's entirely separate from the actual program.

Isn't it?

Well, it doesn't have to be. Sometimes there are ideas about the program, in addition to those directly encoded into C++, that you want to convey to other human readers of the program. They could be additional C++ statements that you are thinking about adding, or that were previously in the program; or maybe they could just be English narrative about what you're planning or why you made some programming choice. Every programming language has some mechanism, generally called "comments", that lets you add stuff to the program file that you don't intend for the computer to read.

In fact, you've already seen them. In every listing so far of an entire program, the first line has been a brief description of the program, starting with two slashes:

```
// First "hello world" program
// Program with input and output
// Doing a little arithmetic
// Demonstrating if and else
// Demonstrating if without else
```

Program fragment 3.8: Examples of first-line descriptions

In fact *any* line that you write that begins with two slashes will be completely ignored by the compiler, so that you can safely write notes to yourself and other humans there. It's always a good idea to make the first line of a program a brief description like this, but you can also use it elsewhere in your program.

If you put the double slash anywhere in a line, the compiler will read the first part of the line and ignore the rest, which makes it convenient for leaving notes specifically about that line:

```
  if (name == "Nobody")         // special case name
  {
    cout << "That's an unusual name." << endl;
  }
```

Program fragment 3.9: Commented statement

When you type comments into a program editor, they're often changed to a fainter color, and sometimes a different font, to help indicate that they are not one of the runnable parts of the program.

Chapter 4

Expressions

v20231221-2330

In this chapter we'll explore *expressions*, which are ways of combining variables and/or literal values into increasingly complex computations. The word "expression" in this sense has the same meaning as it does in mathematics, and much of the content here might be review for you, especially if you've worked with spreadsheets or graphing calculators. Make sure to read carefully, though, because some of it is likely to be new!

expressions

4.1 Variables and values

We've already seen *variables*, as a way to read in values from the user and write algorithms to abstractly refer to those values, whatever they might be. It is worth reiterating here that while we may informally treat a variable as a name for a specific value, a better mental model is that the variable is a *named container*, which values can be placed in. Accessing the variable (in a `cout` statement or other expression) gives us access to the value it contains.

variables

One reason this is important is that declaring a variable only sets up that container, but doesn't actually provide it with a value. In the brief fragment

```
int age;
cin >> age;
```

Program fragment 4.1: Declaring a variable and reading a value

the first of the two lines only declares the variable, with contents unspecified and unknown, as on the left in this diagram:

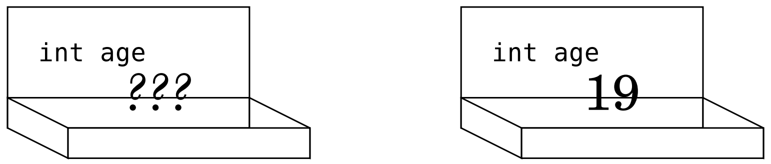


Figure 4.1: A newly declared variable, and a variable with a value

and then after the second line, the variable contains a value such as 19, as on the right. Notice that the diagram indicates the "contents" of the newly-declared variable as *???*: there's something there, but we don't know what it is. It might be 0, it might be 19, it might be some arbitrary number in the millions. C++ makes no guarantees about what it will be—which is why it's important to put a definite value in the variable as soon as possible after declaring it.

4.1.1 Naming values directly

Sometimes it is useful to attach a name to values other than those read from the user, and variables will be used in these cases as well; and since we won't have to wait for the input from `cin`, we can assign a value on the very line that we declare the variable, using `=` as follows:

`=`

```
string sentence = "This is a longer sentence, with spaces in it.";
int quarts_per_gallon = 4;
```

Program fragment 4.2: Using one line to declare a variable and give its initial value

This syntax is sometimes called the *declare-and-init* pattern, because in addition to declaring the variable, it also provides the initial value to go in the box. (It's an "initial" value because, eventually, we'll see how and why you might want to change them later on. But for now we'll leave them unchanged from that initial value.)

declare-and-init

Recall that when we were comparing values for equality, we used a symbol comprised of two equals signs "`==`"; that usage is in contrast with this one for a single equals sign "`=`", immediately assigning a value to the variable when it is declared. Use double equals to *ask* about a value, and single equals to *make* it be the value.

4.2 Order of operations

In an expression like

$$2 + 3 \times 4$$

we might plausibly imagine two ways to evaluate it: first add to get 5, then multiply by 4 to get 20; or, first multiply to get 12, then add to 2 to get 14. To make expressions like this unambiguous—in this case preferring the latter interpretation, yielding 14—mathematicians have long agreed on a standard *order of operations*, which you may have learned under the acronym "PEMDAS":

order of operations

P	parentheses
E	exponents
MD	multiplication and division
AS	addition and subtraction

Figure 4.2: The classic order-of-operations table

This same set of rules, expanded a little bit, holds true in programming languages as well. Note that the above table puts M and D on the same line, and A and S as well; for operations in the same level of the table, we proceed left-to-right in the expression. So for an expression like

$$1 \div 2 \times 4 - 3 + 5 \times 6$$

we do all the multiplying and dividing in a left-to-right pass (*not* necessarily with multiplication first):

$$\begin{gathered} \underbrace{1 \div 2} \times 4 - 3 + 5 \times 6 \\ \underbrace{\frac{1}{2} \times 4} - 3 + 5 \times 6 \\ 2 - 3 + \underbrace{5 \times 6} \\ 2 - 3 + 30 \end{gathered}$$

and then all the adding and subtracting, also left-to-right (*not* necessarily with addition first):

$$\begin{gathered} \underbrace{2 - 3} + 30 \\ \underbrace{-1 + 30} \\ 29 \end{gathered}$$

In a program, you're not likely to have lengthy expressions with only literal values (though it's permitted)—typically you'll have at least some variables. The standard order of operations still applies, so multiplication and division (written * and / respectively) happen before addition and subtraction:

```
cout << "Result: " << 100 - 3 * depth + 2 << endl;
```

Program fragment 4.3: Multiplication comes first

Here the value of depth would be multiplied by 3, then that would be subtracted from 100, and finally 2 added to the result before printing it to cout. If you want the addition to happen before the subtraction, you can always force that to happen using parentheses:

```
cout << "Result: " << 100 - (3 * depth + 2) << endl;
```

Program fragment 4.4: Parentheses affect order-of-operations

As expressions get more complicated, it can also be sometimes useful (in addition to or instead of parentheses) to use variables to name some intermediate value in the computation, both to make the computation itself clearer and perhaps to give it some context for a human to understand:

```
double adjusted_depth = 3 * depth + 2;
cout << "Result: " << 100 - adjusted_depth << endl;
```

Program fragment 4.5: Named intermediate values

4.2.1 Warning: no exponent operator

Perhaps surprisingly, C++ does not have an operator to raise numbers to another power. Perhaps more surprisingly, it *does* have an operator represented by the caret (^) character, but it does something completely different. If you try

```
cout << (5 ^ 2) << endl;     // This is NOT an exponent
```

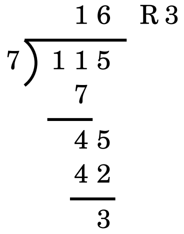
you'll find that the result is 7 rather than 25. So it's valid C++, it compiles, it runs, and it does something unrelated to exponentiation.[1] If you know the caret as an exponent operator from somewhere else, be careful not to use it in C++!

4.3 Division, integer and real

Quick quiz: what's three divided by two?

If you said "one and a half" (or three halves, or 1.5, or anything in that vein), congratulations, you are thinking about division like basically everyone else. But in many programming languages, including C++, there's a little more to the story.

Think back for a moment to when you first learned about division, before you knew about fractions or decimals. Perhaps you learned to write out long division following a pattern like this:



or perhaps your technique was different, but look at that top line for a moment. Presented with a division problem that didn't have an exact whole-number solution (7×16 is 112, so $115 \div 7$ won't divide an exact number of times), we give the answer in two parts: the whole-number quotient (here 16), and the remainder (here 3). This style of division is called *integer division*, and is important enough to many computational problems that programming languages provide it as an option, distinct from *real division* (or *real-number division*), which produces a single result that might be fractional.

integer division

real division

Here's how it works in C++ and many related languages: if the expressions on both sides of the division operator (`/`) are of the `int` type, C++ will perform integer division and the result will be just the whole-number quotient. If either or both of the expressions are of the `double` type, it will perform real division and the result will be a `double`.

The easiest way to force real division, if that's what you want, is to use a `double` variable in the computation, or to explicitly add a "point zero" after a literal number, or to multiply something by `1.0`, all of which ensure that part of the computation is a `double`. Type in the following expressions and see what they do:

[1] The operation it's performing is called "bitwise XOR", which is rather outside the scope of this course.

```
cout << 13 / 4 << endl;
cout << 13.0 / 4 << endl;
cout << 13 / 4.0 << endl;

int first;
cin >> first;

cout << first / 4 << endl;
cout << first / 4.0 << endl;
cout << first * 1.0 / 4 << endl;

double second;
cin >> second;

cout << second / 4 << endl;
```

Program fragment 4.6: Comparing integer division and real division

On the other hand, if what you want actually is integer division, you probably also want a way to get the remainder. This is provided with an operator variously called *remainder* or *modulus* (or just *mod*), and represented with a percent sign (`%`). It has nothing in particular to do with percentages, it's just a convenient symbol on the keyboard that looks a little bit like the division sign. Using integer division with remainders is useful to identify digits of a number, for dividing items into sets while keeping track of leftovers, and for asking whether numbers are evenly divisible by each other.

remainder

modulus

`%`

```
int number;
cin >> number;

cout << number / 10 << endl;
cout << number % 10 << endl;

if (number % 2 == 0)
{
  cout << number << " is even" << endl;
}
```

Program fragment 4.7: Uses of the mod operator

4.3.1 Warning: `double` values are approximate

When you write expressions that produce `double` values, you should be aware that waht is stored in the system is not necessarily an exact number, but might be an approximation. Similar to how 0.33333 is not *exactly* $\frac{1}{3}$, the result of real division (or other kinds of computation with `double` values) will often be very close to, but not *exactly* the mathemetically expected answer. For many purposes it's close enough, and when the results are printed out, C++ will try to present them in a way that hides any inexact approximations.

The main place you need to watch out for is if two `double` values are being tested for equality. Try typing in the following program fragment:

```
cout << 10000.0 / 6         << endl;
cout << 10000.0 / 6     * 6 << endl;
cout << 1.0 / 6 * 10000     << endl;
cout << 1.0 / 6 * 10000 * 6 << endl;

if ( 10000.0 / 6  ==  1.0 / 6 * 10000 )
{
    cout << "Exactly equal" << endl;
}
else
{
    cout << "Not the same" << endl;
}
```

Program fragment 4.8: Demonstrating inexactness

The first four statements play out as you might expect: the expressions `10000.0 / 6` and `1.0 / 6 * 10000` appear to both evaluate to the same number, and when you multiply each one by 6 you get the original 10000 back. Nevertheless, at least on most systems, the conditional statement will print that the two values are *not* the same, because the internal representations got rounded slightly differently.

The lesson here: if you will need to test equality with `==`, try to figure out how to do the computation with only integers; and if you must use `double` computations, try to avoid exact equality testing, or you might get some unexpected results.

4.4 Functions (and library headers)

There are a handful of mathematical functions that are not associated with an operator (such as logarithms or trigonometric functions, or just rounding to the nearest integer), or whose operator doesn't show up on a standard keyboard (such as square root, often written $\sqrt{\ }$). For these, we'll use mathematical *function* notation, with the name of the function followed by the value it's operating on, in parentheses. For instance,

function

```
cout << cos(0) << round(3.75) << sqrt(2) << endl;
```

Program fragment 4.9: Some math functions

But there's one more thing we have to do first, before it will compile and run. C++ comes with a lot of tools provided in its *standard library*, including mathematical operations and functions, but we have to tell the compiler that we'll be using the math parts of the library by including a *library header* at the top of the program. We've already seen one library header, `<iostream>`, which indicates we'll be doing input and output ("I/O"); to use the math functions, we have to also include `<cmath>`. This is a short but complete program that uses several of them:

standard library

library header

```
// Demonstrating various math functions
#include <iostream>
#include <cmath>
using namespace std;

int main()
{
  cout << cos(0) << round(3.75) << sqrt(2) << endl;
  cout << floor(3.75) << log(100) << pow(5,2) << endl;

  return 0;
}
```

Program fragment 4.10: Demonstrating various math functions

These are some of the functions provided in the `<cmath>` header that might come in handy at some point:

`<cmath>`

`sqrt`	Square root
`pow`	Exponent (compute x^y)
`round`	Round to nearest integer
`floor, ceil`	Compute floor or ceiling (round down, round up)
`sin, cos, tan`	Trig functions sine, cosine, tangent
`log, log10`	Logarithms (ln, $\log_{10}$)

Figure 4.3: Some useful `<cmath>` functions

Try using them with different values. What happens if you omit the `<cmath>` header from the program?

Note that all of the provided mathematical functions work with `double` values, so the warnings about approximate values apply here as well!

Chapter 5

Collections of things: vectors

v20231222-0100

One of the most important programming concepts is that of the “collection”—a way for a single value to contain multiple other values. Eventually, you will run across other kinds of collection in C++, but our first will be collections that act like lists, which in C++ will be called *vectors*.

vector

The two central facts about lists in general, and specifically about the `vector` types in C++, are that

- they can hold multiple things, with no theoretical maximum size, and
- we can go through them in order.

That is, it may be that we only have five or 17 or 42 things in a particular `vector`, but there’s nothing about the concept of a `vector` that prevents us from having five hundred or five billion elements, depending on what our data happens to look like. And when we process those elements, we can meaningfully think about the “first thing” or “seventh thing” and can predictably and repeatably process them in order.

5.1 Making a vector

For now, we’ll restrict ourselves to making `vector` values directly in the declare-and-init format. (We’ll see other ways to process them, including reading them in from the user, in a few more chapters.) In the same way that we can declare-and-init a variable of one of the simpler types, e.g.

```
string drink = "coffee";
```

Program fragment 5.1: A `string` variable

we can declare a `vector`-type variable and give its initial value:

```
vector<string> drinks = { "coffee", "tea", "water", "soda" };
```

Program fragment 5.2: A vector of strings

The syntax of both these lines is exactly the same: first the type of the variable, then the variable name, a single equal sign, and the value used to init the variable. Similarly:

```
vector<int> lengths = { 3, 72, 5, 1, 5, 54, 3, 28, 154 };
vector<double> weights = { 4.0, 5.8, 2.75, 3.1, 12.5 };
```

Program fragment 5.3: Vectors of numbers

The statement can run to multiple lines if it needs to (just like any other statement):

```
vector<string> rainbow = { "red", "orange", "yellow",
                           "green", "blue", "indigo", "violet" };
```

Program fragment 5.4: Breaking an init statement across multiple lines

And the value need not contain any elements at all!

```
vector<string> zeroWords = { };
vector<int> noNums = { };
```

Program fragment 5.5: Empty vectors

Lists of things can be empty, after all.

5.1.1 Required header

Vectors are standard in C++, but you need to explicitly tell the compiler you'll be using them. At the top of your program file, add the line

```
#include <vector>
```

in addition to any other `#include` lines that you may have up there, in any program where you'll be using vectors.

5.1.2 What's the deal with the <stuff> after the word `vector`?

Previously, we've seen types like `int` or `string`, which are complete and entire types by themselves. An `int` is an `int`, representing an integer value within a defined range and represented by a particular number of bits. But as we've already seen, saying a value is a `vector` doesn't fully specify its type, because there is also a question about the type of the *contents* of the collection. A vector... of what?

We say as a result that `vector` is not quite a complete type, but rather a *type template* or a *templated type*, which requires a *template argument* to identify the type of the contents. The details of this are not important right now, just remember: when declaring a `vector` variable, always follow the word `vector` with the type of the contents, in angle brackets,[1] as shown above. When reading code out loud or otherwise speaking about them, we read the content type with the word "of", so that `vector<int>` is read as "vector of int", and so on.

type template

templated type

template argument

Omitting the content type may not always be a problem. In some cases, the type of the contents might be obvious:

```
vector nums = { 3, -1, 0, 42, 7 };  // Error?
```

Program fragment 5.6: No explicit content type: may be an error

Indeed, since the C++17 standard, the above line would not give an error message, or even a warning, correctly inferring that the content type must be `int` since all of the contained values are `int`s. (Compilers on older standards would give an error about a missing template argument.) However, even on a current compiler it's a good idea to habitually provide the content type:

[1] We use the term "angle brackets" here because they're enclosing something much like square brackets or curly brackets might, but on the keyboard they're exactly the same as the less-than and greater-than signs.

```
vector<int> nums = { 3, -1, 0, 42, 7 };  // Better
```

Program fragment 5.7: Explicitly providing the content type

Because sometimes the type can't be inferred, even on a modern compiler:

```
vector empty = { };  // ERROR: no way to infer content type
```

Program fragment 5.8: BAD: no content type for an empty vector

And the inference system can't read our minds—this might be intended as a `vector<double>` but some of the contents are int literals:

```
vector moreNums = { 5, 7.3, 9 };  // ERROR
```

Program fragment 5.9: BAD: no content type for a vector of mixed numbers

But if we are explicit it works fine:

```
vector<double> moreNums = { 5, 7.3, 9 };  // Works as expected
```

Program fragment 5.10: A vector of numbers

In no situation does C++ permit us to mix-and-match the types of elements in a vector:

```
vector<???> mixed = { 3, 2.5, "Eli", 23, "Audra" };  // ERROR
```

Program fragment 5.11: BAD: a vector of completely mixed values

In this sense, C++ vectors are said to be *homogeneous*, i.e. all the contents in any particular vector must be of the same type.

homogeneous

5.1.3 Review

1. Write a declare-and-init statement for a vector containing the names of the people that live in your apartment or suite or house.

2. Write a declare-and-init statement for a vector containing the ages of the members of your immediate family.

5.2 Processing its contents: loops

In real life, when we are giving someone simple instructions about processing a list, we might typically do so using the word “each” to stand in for each value in turn:

Buy each of the items on this list.

Read out loud each of the names on the list.

Figure 5.1: Simple instructions

If the instructions for what to do with each item are more complex, we typically put the “each” first and refer back with the pronoun “it” (sometimes with additional logic):

With each of the words on this word list,
    *write **it** down and*
    *say **it** out loud.*

For each of the items on this grocery list,
    *check our pantry for **it** and*
    *if we are out of **it**,*
        *buy **it** at the store.*

Figure 5.2: Complex instructions

This kind of pattern in natural language points the way to how we formalise processing lists of stuff in programming languages. The main difference is that a word like “it” works well for humans who have a lot of context (and can ask questions to clarify misunderstanding), but in a programming language we would like to be more precise by introducing a variable name for the “it” we’ll be referring to in the instruction. And, since C++ doesn’t want us to introduce new variable names without giving them a type, we will specify the type of the variable. Thus:

```
vector<string> words = { "correct", "horse", "battery", "staple" };
for (string word : words)
{
  cout << word << endl;
}
```

Program fragment 5.12: A loop to print the contents of a vector

Here, the actual repetition instruction might be paraphrased as “for each word in the word list, print **it** out.” Indeed, when we do read it out loud, we would typically actually begin, “for each `string word` in `words`...”. Though the word “each” is not explicitly written in the C++, this kind of repetition is widely known as a *for-each loop*. The *loop header*, which is the line with the `for` on it, declares a local variable that will first name the first value in the list (and execute all the instructions in the block), then it will name the second value in the list (and execute all the instructions in the block), and so on. The type of this *loop control variable* should be the same as the type of the contents of the vector.

for-each loop

loop header

loop control variable

Here is another example of a for-each loop, which in this case prints out each element of the vector along with its square:

```
vector<int> nums = { 3, 0, -2, 5, 11 };
for (int n : nums)
{
  cout << n << "  " << n*n << endl;
}
```

Program fragment 5.13: A loop to process numbers

Its output would be:

```
3  9
0  0
-2  4
5  25
11  121
```

Just as with `if` and `else` statements, the `for` controls the entire block that immediately follows it, which can contain multiple statements. The statements it contains can themselves be `if` and `else` statements, which by convention we indent a little further to help show that they are “inside” the block:

```
vector<int> nums = { 3, 1, 15, 5, 11 };
cout << "Only bigger than ten:" << endl;
for (int n : nums)
{
  if (n > 10)
  {
    cout << n << endl;
  }
}
```

Program fragment 5.14: Nesting a conditional inside a loop

5.2.1 Review

1. Make a vector of the names of the towns you and your friends live in, and a for-each loop that prints them all out.

2. What would be the output of this program fragment?

```
vector<double> temps = { 0, 20, 25, -40 };
for (double ctemp : temps)
{
  cout << ctemp << "   " << ctemp * 9 / 5 + 32 << endl;
}
```

Program fragment 5.15: Doing something with some numbers

5.3 A few more problems

1. Write a program that has a vector variable with many `int` values in it, and then two loops: the first prints all the negative numbers in the vector, and the second prints all the positive numbers. (Hint: remember that blocks can contain *any* other statements, including other control statements!)

Chapter 6

Strings: looking inside

20231222-0100

We've been using strings for some time now, but in this chapter we'll dive a little deeper into how they work; and strings will provide a mechanism and motivation to explore the concept of methods, and revisit loops.

6.1 Characters and sequences of characters

string

character

We have hitherto treated a *string* as arbitrary blob of text, whether provided as a string literal in the program or typed in by the user, but a more detailed definition is that a string is a sequence of characters. A *character* is, generally, a single element that you can type with a keyboard, such as a letter, or a space, or punctuation; you might know the term from character limits on some social media platforms or communication systems. Some characters are a little harder to type on a standard American English keyboard, such as accented letters, foreign currency symbols, letters and characters from other writing systems, and even emoji, but they do also count as characters and can be part of the contents of a string. In modern C++, characters that are "outside ASCII"[1] will most often be handled using an encoding called UTF-8, and should work ok with a few caveats:

- If you use non-ASCII characters in string literals, you need to precede the double-quote with "u8", as in: `u8"¡Holá!"`
- Not all compilers support non-ASCII in variable names

[1] The American Standard Code for Information Interchange, standardised in the 1960s, by Americans, which is essentially "exactly the characters you could type on a standard American typewriter", which makes it adequate for specifically American English communication but somewhat limiting if you're doing anything else.

- Anything the user types can be stored and printed back out correctly with no special work, but
- Strings that include non-ASCII will generally over-count the number of characters they contain unless you do a little extra work

The type `char` is provided for storing individual characters, and for historic reasons it is essentially defined as exactly one byte, eight bits, and thus in practice a single `char` value can only represent an entire character if that character is part of ASCII. Non-ASCII characters are representable in strings by a sequence of multiple `char` values.

So, variables and values of type `char` in C++ aren't *quite* the same thing as a "character" in modern global usage. That said, it's close enough, for people working primarily in English, that in an introductory context it's a lot easier to say "a `char` is a character" and defer the complexity to later. Examples in the rest of this book will generally stick to ASCII, but other than the character count possibly being off, if you want to write or test your code with non-ASCII characters, go for it!

6.2 Methods

We have already seen two ways that C++ provides for us to indicate how to perform a computation: operators, and functions. As a matter of syntax, operators are represented with punctuation symbols like `/` or `+`, and have two operands to say what to perform the operation on;[2] and functions are represented with names and followed with parentheses that contain some number of parameters (separated by commas if there's more than one). For both operators and functions, the data they work with (operands and parameters) can be either literal data, named variables, or more complex expressions. This diagram reflects that:

[2] Technically, only some operators have exactly two operands (the "binary" operators), and we'll see others later. But for now, exactly two operands.

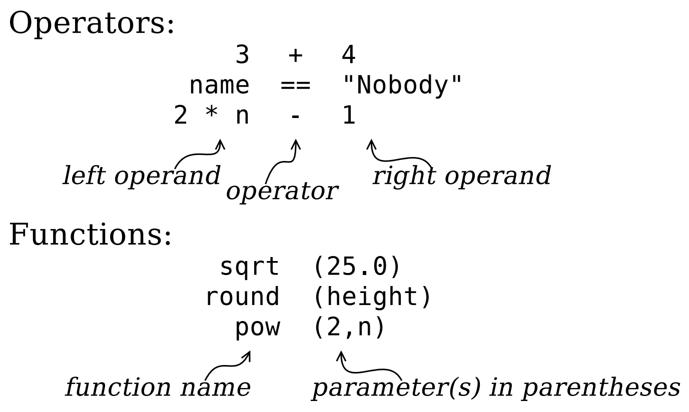


Figure 6.1: Operators and function call syntax

Here we will see the third way that C++ lets us indicate a computation to perform, and it's a bit of a hybrid of the other two. A *method* call, like an operator, is written with something on the left side and something on the right side to indicate the data it will work with, and the computation itself is indicated by the part in the middle. Like a function call, the computation is given a name (preceded by a dot to make it a method call) and the parameter list to its right is always surrounded by parentheses, which must be present even if there are zero parameters. This diagram illustrates three method calls:

method

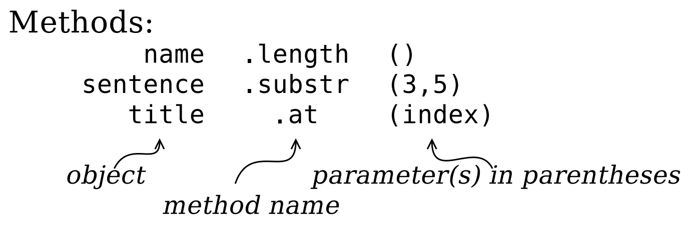


Figure 6.2: Method call syntax

For now, all the method calls we'll see are designed to operate on string objects. The object of a method call can in general be any expression of the appropriate type, but for reasons we'll discuss in a later chapter, methods won't work directly on string literals:

```
"This won't work, sorry".length()
```

so for now we'll only make method calls directly on string variables. The parameters can, as with function calls, be literals, variables, or complex expressions, as long as they are of the correct type.

The diagram above illustrates the three string methods that we'll be using for now:

`.length` gives the number of characters in this entire string (and requires no parameters, but you can't skip the parentheses)

`.substr` extracts a *substring* beginning at a given point in this string and extending for a given number of characters

substring

`.at` extracts a single character at a given location in this string

To see in more detail how they work, type in the following program fragment, try to predict how it would work on the word "howdy", and then run it and see what it does:

```
cout << "Type a word at least five characters long:" << endl;
string word;
cin >> word;

cout << word.length() << endl;
cout << word.substr(1,3) << endl;
cout << word.at(4) << endl;
```

Program fragment 6.1: Demonstrating some `string` methods

Saying that the length of the string was 5 was probably not surprising, but you might not have expected the other two answers:

```
5
owd
y
```

The reason for this behaviour has to do with zero-based indexing, which is standard in C++ and nearly every other modern programming language.

6.2.1 Zero-based indexing

In the string `"howdy"`, there are five characters. We can model the string as a sequence of five cells, each containing a single `char` value, as follows:

0	*1*	*2*	*3*	*4*
h	o	w	d	y

Figure 6.3: A five-character string

Each of those cells is labelled with a number, or *index* (plural “indices” or “indexes”— both are correct and in common use!), that can be used to pinpoint the location of that cell in the string.

index

When a language uses *zero-based indexing*, the index of a character is answering the question “how far from the start of the string is this position?”. In this string, ‘`h`’, the first character, is *at* the start of the string, a distance of zero, so its index is 0. The next letter, ‘`o`’, is one step away from the start, so its index is 1.

zero-based indexing

So in the test program, the line

```
cout << word.substr(1,3) << endl;
```

Program fragment 6.2: Using `.substr`

is asking for the portion of `word` that starts at index 1 and continues for 3 more characters, thus printing “`owd`”. Continuing to the next instruction,

```
cout << word.at(4) << endl;
```

Program fragment 6.3: Using `.at`

is asking for the character at index 4, which in this case is ‘`y`’.

Armed with this knowledge, experiment a little bit with the program:

- Try other five-letter words: predict the output, then run the program and see if your predictions are correct.
- What about longer words?
- What do you expect would happen if you typed a shorter word? Try it and see what it does.
- Change some of the numbers in the `.substr` and `.at` method calls, and then run the program with different words as its input. Are you able to predict the output before you run it?

6.3 Loops and individual char values

Understanding strings as a sequence of separate characters also gives us the ability to process the characters one-by-one. In this sense, a `string` is similar to a `vector<char>`; although the vector can't be as easily processed with `cin` and `cout`, we can write an identical `for` loop for either one:

```
string word;
cin >> word;
for (char ch : word)
{
  cout << ch << endl;
}
```

Program fragment 6.4: Accessing individual characters

Most interesting loops involving strings will want to ask questions about each character and behave conditioned on that. For now, we can ask whether the character is exactly some specific character, using `==`, or whether it belongs to some group of characters, using provided functions.

6.3.1 Comparing to `char` literals

Just as we had string literals earlier, indicated by double quotes, we can also represent *char literals* using single quotes. Any printable ASCII character is represented by typing that character between a pair of single quotes. We can use `==` (and eventually other tools) to compare them:

char literals

```
cout << "Replacing underscores with spaces:" << endl;
for (char ch : sentence)
{
  if (ch == '_')
  {
    cout << ' ';
  }
  else
  {
    cout << ch;
  }
}
cout << endl;
```

Program fragment 6.5: Comparing characters

6.3.2 Functions on `char` values

We've previously seen math functions like `sin` and `round`, that process numbers into other numbers. Another category of function answers questions about values and can be used in `if` statements.

```
// Filtering only the lowercase letters in a word
#include <iostream>
#include <cctype>
using namespace std;

int main()
{
  string word;
  cin >> word;

  cout << "Only the lowercase letters:" << endl;
  for (char ch : word)
  {
    if (islower(ch))
    {
      cout << ch << endl;
    }
  }
  return 0;
}
```

Program fragment 6.6: Filtering only the lowercase letters in a word

Functions that are usable in this way in `if` statements often have names beginning with "`is`", and several of them are provded in the `<cctype>` library header, including:

`<cctype>`

`isalpha`	is the character alphabetic?
`islower`	is the character a lowercase letter?
`isupper`	is the character an uppercase letter?
`isdigit`	is the character a digit from 0–9?
`ispunct`	is the character a punctuation character?

Figure 6.4: Some functions in `<cctype>` to check characters

There are also a couple functions (also in `<cctype>`) that help you to convert case:

`toupper`	produce the character code for uppercase of given letter
`tolower`	produce the character code for lowercase of given letter

Figure 6.5: Some functions in `<cctype>` to transform characters

so you can, for instance, all-caps a word:

```
for (char ch : word)
{
  char ch_up = toupper(ch);
  cout << ch_up;
}
cout << endl;
```

Program fragment 6.7: Printing the all-caps version of a word

6.3.3 Warning: `toupper` produces a number

It might seem that the more obvious way to write that last program fragment would be

```
for (char ch : word)
{
  cout << toupper(ch);
}
cout << endl;
```

Program fragment 6.8: Surprise! Prints numbers

but if you do that (try it!) you'll see that it prints out a lot of numbers instead. For some inconvenient historic reasons, `toupper` and `tolower` technically produce `int` rather than `char` values, and if you print them directly you get those `int` values. If you store them in a `char` first, though, they'll get converted back into the character values you would otherwise be expecting.

This trick also lets us do some other nifty things; lightweight math (adding or subtracting small numbers) done on a `char` value produces an `int` value that can be converted back to `char` the same way as before. Try to figure out what this program is doing:

```
cout << "Type a word:" << endl;
string word;
cin >> word;
for (char letter: word)
{
  char result = letter + 1;
  cout << result;
}
cout << endl;
```

Program fragment 6.9: Does something with the letters of a word

Are there any cases it doesn't handle nicely? How might you fix it to handle those cases?

Chapter 7

Variables that vary, and accumulators

v20231222-0045

In this chapter we'll see why "variables" are called that, and introduce the *accumulator* pattern, a powerful way to use loops to process data.

7.1 Assignment and update statements

Thus far, we have only seen programs where a variable's value never changes (at least, not until you run the program a second time with different input). Here we will see two ways that you can change the value of a variable during the runtime of a program.

Back in Chapter 4, we first saw the use of the = operator in what we called a "declare-and-init" statement, immediately assigning a value to a variable rather than reading its value from the user:

```
int n = 5;
cout << n << endl;
```

Program fragment 7.1: Declare-and-init

assignment

The same operator can be used in a simple *assignment* statement: any variable name, a single =, and then any simple value or other expression of a type that can be assigned to that variable.

```
int a = 5;
int b = 7;
cout << a << " " << b << endl;
a = b;
b = 13;
cout << a << " " << b << endl;
a = 25;
b = 2 * a + 7;
cout << a << " " << b << endl;
```

Program fragment 7.2: Assigning to a variable

Another way to change the value of the variable is to update its value relative to its current value. These *compound assignment* statements, such as **+=**, are all based on one of the arithmetic operators (in this case **+**), and instruct the computer to evaluate whatever expression is on the right side of the assignment operator, and then instead of *replacing* the original value of the variable, use the indicated operation to combine the variable's current value with the right-side result, and store that as the new value of the variable.

compound assignmen

+=

```
int a = 23;
cout << a << endl;
a += 1;
cout << a << endl;
a -= 2;
cout << a << endl;
a += 18;
cout << a << endl;
```

Program fragment 7.3: Updating a variable with compound assignment

7.2 Accumulating values

Armed with your understanding of vector loops from Chapter 5 and your new knowledge of assignment statements, you might be able to work out what this program does:

```
vector<int> numbers = { 3, -2, 0, 12, 4 };

int result = 0;
for (int num : numbers)
{
  result += num;
}
cout << result << endl;
```

Program fragment 7.4: Assigning inside a loop

Type it in and confirm your understanding, and as usual, try modifying parts of the program to see if they change its behaviour the way you expect.

accumulator

This program is a short, simple example of the *accumulator* pattern that will spring up for almost every interesting program we write to loop through a sequence (either a vector or a string) and process its values. The accumulator itself is a variable that accumulates relevant values, step by step, until at the end it has the final result being computed. (Or at least, an intermediate result that will be further processed later.)

Aside from the loop itself, there are three important parts to every accumulator algorithm:

- Declare-and-init the accumulator variable, before the loop
- Update the accumulator, inside the loop (at least sometimes)
- Use the value of the accumulator, after the loop

Figure 7.1: The accumulator pattern checklist

For now, “using” the accumulator after the loop will often just be printing its value. How (and when) to update the accumulator, and what its initial value should be, will depend on the algorithm.

7.3 A few standard loop-accumulator patterns

7.3.1 Sum, average, product, concatenate

One of the most fundamental accumulator patterns is the one that computes a sum, which you have already seen in the last section, recapped here:

```
int result = 0;
for (int num : numbers)
{
  result += num;
}
cout << result << endl;
```

Program fragment 7.5: Computing a sum with a loop

Here, the operation used to update inside the loop is addition, because summing is adding together a whole bunch of numbers, and that addition is performed every single time through the loop in order to add *all* the numbers. The initial value of the accumulator is the “sum” that you get when you haven’t seen anything yet, or if the list is empty. (Not coincidentally, zero is the additive identity, the number that you can add to things without changing them—accumulators that follow this pattern will generally declare-and-init the variable with the identity of whatever operator they use.)

The same general pattern is used for some other computations that are similar.

- To compute an average (mean) of a list of numbers, you add them up and divide by how many of them there are. This will follow the sum pattern exactly up to the “use” part: instead of printing the sum, it prints the average. What should you divide by? How can you do that division correctly? Write or modify a program to compute the average of a bunch of numbers.

- To compute the product of a list of numbers, you’ll be multiplying them instead of adding. What number should you use to init the accumulator? Write or modify a program to compute the product of a bunch of numbers.

- *Concatenation* is a term for combining multiple strings into a single string, and in C++ this operation also uses the + operator (although unlike some other languages, both operands of + must already be strings and they can’t both be string literals). How will you declare-and-init an accumulator for this operation? Write or modify a program to compute the concatenation of a

concatenation

bunch of strings.

7.3.2 Count

Another broad category of accumulator algorithms are the counting algorithms: how many elements of the vector match a specified condition?

```
int result = 0;
for (int num : numbers)
{
  if (num > 100)
  {
    result += 1;
  }
}
cout << result << endl;
```

Program fragment 7.6: Counting matching elements

Here the "update" part of the accumulator checklist is adding 1 to the accumulator, but only *sometimes*. (If we added 1 for every single element in the vector, we'd just get the length of the vector.)

- The values being counted don't need to be numbers. Write or modify a program that counts how many strings in a vector start with an uppercase letter.
- One variant of counting is the scoring accumulator, where instead of always adding 1, some different amounts are added (or subtracted) based on different properties of the value being considered. Write or modify a program that scores a vector of words by awarding 3 points for every word over 5 characters long and 1 point for every word shorter than that.

7.3.3 Max, min

Yet another pattern to be aware of is the max/min pattern, finding the largest or smallest or longest or hottest or otherwise mostest element of a sequence.

```
double highest = numbers.at(0);  // The first element of the vector
for (double num : numbers)
{
  if (num > highest)
  {
    highest = num;
  }
}
cout << highest << endl;
```

Program fragment 7.7: Finding the greatest element

The update is only sometimes, as with the counting loops, but the update is to *replace* the result value rather than to add to it; the condition for the update asks if the current number higher than the highest (so far), and if so, update that result.

The initial value is also interestingly different from the others we've seen. In the previous sections we asked "what would the answer be if the list were empty?", but that's not really a sensible question here. (What is the largest value in an empty list?) On the other hand, the first element in the vector might not be the highest, but it's at least a valid comparison value, and if there's a higher one later, we'll find it.

- The algorithm for finding the smallest number in a list is almost identical. What change do you have to make to the algorithm find the minimum instead? Write or modify a program to find the smallest of a bunch of numbers.

- The comparison operators `<` and `>` work on strings as well as numbers. They work by comparing ASCII values, so things get a little complicated when the strings are mixed case or involve numbers and punctuation, but if all the strings being compared are the same case (all lowercase, or all uppercase), then you can treat them as alphabetic comparisons. Write or modify a program to find the alphabetically-earliest of a vector of lowercase words.

- Eventually we'll see accumulator-loop situations where we don't have the entire vector of values in advance, so we can't init the result with "the first value". If we know enough about the domain of the data (in particular, what are the largest and smallest realistic values that could be in there), an alternative way to init the accumulator is with a much-larger or much-smaller value. Write a program that finds the highest age from a bunch of ages of users of a particular website, without using a value from the vector as the initial value (i.e. choose an appropriate initial value). Modify the program, including the initial value, to find the lowest of a bunch of ages of users of a particular wesbite.

7.3.4 Argmax, argmin

Sometimes what you want is not the max/min value itself, but the value in a sequence that has the maximum or minimum value of some relevant property. For instance, finding the longest string in a sequence:

```
string longestword = words.at(0);
for (string word : words)
{
  if (word.length() > longestword.length())
  {
    longestword = word;
  }
}
cout << longestword << endl;
```

Program fragment 7.8: Finding the *word* with the greatest *length*

This is arguably a subtype of the max/min pattern, but the difference is important enough to be worth pointing out. An *argmax* algorithm is one that, rather than finding "the largest value", instead finds "the value that maximises some specific property" (here, the length of the string). That makes the comparison slightly more complicated, though in a predictable way: *both* sides of the comparison have something (here, the `.length` method) applied to them, and the result of that is what is being compared. The actual update is identical to the base max/min pattern.

argmax

7.4 Constants

One last note for the chapter about variables. All through this chapter we've been seeing ways to usefully *change* the values of variables; but you might recall that we sometimes want to name a value for clarity, but the value should never change:

```
int quarts_per_gallon = 4;        // is this actually going to change?
```

Program fragment 7.9: Named value that won't change

The number of quarts in a gallon won't change! We call such values *constants*. It's useful to indicate that a variable is meant to be a constant, both for human readers to understand the code, and also for the compiler to help make sure that you don't *accidentally* change the value. There are two keywords to do this in C++: `const` and `constexpr`. The difference between the two is subtle (having to do with whether the value is "known at compile time"), but most named constants can be declared `constexpr`, and if the compiler complains, try switching it to `const`. (If the compiler still complains, and you're sure it really shouldn't be changing, double-check that you aren't actually changing it anywhere!) Thus we can say

constants

`const`

`constexpr`

```
constexpr int quarts_per_gallon = 4;    // definitely never changing
```

Program fragment 7.10: Indicating that it won't change

and then use `quarts_per_gallon` anywhere we want to make that measurement conversion, rather than having the unexplained number 4 floating around in our code. They're also nice to use when there's a filtering or counting threshold somewhere:

```
constexpr int minimum_membership_age = 25;

int underage_count = 0;
for (int age : ages)
{
  if (age < minimum_membership_age)
  {
    underage_count += 1;
  }
}
cout << "Number of people too young to join: " << underage_count << endl;
```

Program fragment 7.11: Counting the values below the threshold

Choosing a good name for the accumulator, and using a well-named constant, make programs like this much easier to read and understand.

Chapter 8

I/O revisited, and general loops

20231222-0045

In this chapter we'll talk about more general forms of looping, and to illustrate and motivate them we will explore some different common input patterns.

8.1 Reading data until it's done

First, let's look at a method of vectors that we haven't seen yet. In previous sections, when we used vectors, we created them with their full contents at the top of the program, and then just made use of the (unchanging) values in the vector. With the `.push_back` method, which adds new elements to the vector after it's been created, we can build vectors based on user input, and then process that data with the standard loop patterns we've seen.

`.push_back`

If the data will be read from the user, we can consider creating a vector with no contents at all:

```
vector<int> numbers = {};   // created empty
numbers.push_back(3);
numbers.push_back(-7);
```

Program fragment 8.1: Simple use of .push_back

This example isn't yet very useful, but it points the way to a powerful pattern: if we can read *as much data as we need*, then the same program, without recompiling, can run with different data series every time it runs.

One more piece: expressions to read in data actually do two things. In addition to putting the input data into a variable, they produce the boolean value `true` if they were successfully able to read. If not, they evaluate to `false`. That makes the following pattern make sense:

```
vector<int> numbers = {};
int num;
while (cin >> num)
{
  numbers.push_back(num);
}

int sum = 0;
for (int n : numbers)
{
  sum += n;
}
cout << sum << endl;
```

Program fragment 8.2: Reading until we run out of input

Let's deconstruct that: First, it creates the vector `numbers`, empty. It declares a variable `num` that will be used to read in each number one at a time. The expression `cin >> num` here does double-duty, both reading in the value, once each time through the loop, and controlling when the loop is done. Inside the loop, every time a value is successfully read, it is added as an element of `numbers` using the `.push_back` method. Finally, a subsequent loop processes the data (in this case, with a standard summing algorithm).

When you run the program, then, you need some way to signal that the input is done. Some simplified environments dont' support that, but if you are working in a command-line environment or a full emulator of one, typically Ctrl-D on a line by itself signals that there is no further input to be read, terminating the loop and continuing on to whatever statements come after it.

- Try writing a program that reads all the words in the input, and then prints how many of them were more than four letters long.
- Try writing a program that reads all the integers in the input, and then prints all of them that are between 5 and 20.

8.1.1 A few important vector methods

In Chapter 6 we saw a few useful methods that `string` values have, and then in the last chapter we saw that `.at` also works on vectors. Here are a few other useful vector methods

`.at` extracts a single value at a given location in this vector

`.push_back` adds a given value to the end of the vector (modifying the vector!)

`.size` gives the number of elements in this entire vector (and requires no parameters)

`.back` returns the last element of the vector (equivalent to using `.at` with the index of the last element; but requires no parameters)

`.pop_back` removes the last element from the vector (modifying the vector!)

8.2 while: general looping

`while`

So what exactly is `while` doing? We've previously seen a `for` loop that accessed each element of a vector or string in turn, but `while` is giving us something much more general: it simply controls a block of code and specifies the conditions under which the loop keeps running.

That's actually a little backwards from how we usually do it. In instructions intended for humans—cooking recipes, knitting patterns, assembly instructions—we usually specify repetition with words like *until*, indicating when to *stop*, and if we're being formal, we talk about *stopping conditions*. But in C++ and virtually all other programming languages, we use `while` instead of "until", indicating a "keep-going condition". This makes the semantics of loops very similar to conditionals: test the condition, and if it evaluates to true, do the stuff in the body of the statement. The difference is that an `if` statement executes the body at most once, while the `while` statement will evaluate its condition after every time the body runs, and as long as the condition remains true, the loop keeps running. These flowcharts show the logic of the two:

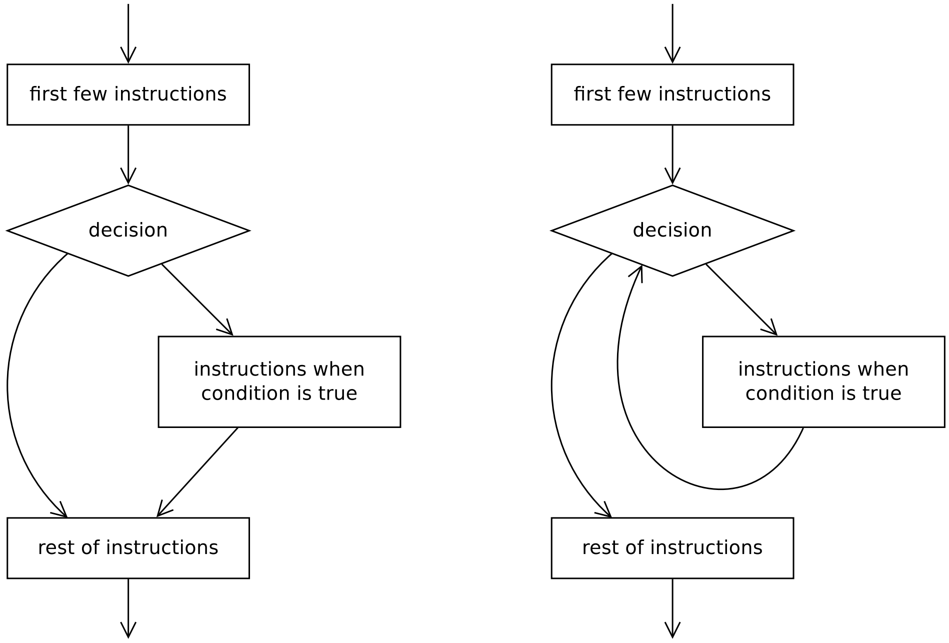


Figure 8.1: An if statement and a while statement, side-by-side

In general, when building any `while`-based loop, the keep-going condition needs to be something that will change over the life of the loop—eventually becoming false. If it is a condition that is always true, the program will never terminate!

8.2.1 An exception: "sentinel" stopping conditions

The exception to that rule is that it's possible to stop a loop in the middle, using a `break` statement. When the loop is built to be stopped in the middle with a `break` statement, it can be ok to leave the loop condition unchanging, and indeed to *always* have the loop condition be true—as long as we're sure the stop condition will be reached eventually.

`break`

A very common reason to use this is when an input loop is meant to stop as soon as it sees a certain special *sentinel* value, as in the following program:

sentinel

```
vector<string> words;
string word;
while (true)
{
  cin >> word;
  if (word == "STOP")
  {
    break;
  }
  words.push_back(word);
}
// further processing after this point
```

Program fragment 8.3: Reading words until one is STOP

This pattern is sometimes referred to as *loop and a half* because only “half” of the loop executes on its last time through—it reads the input, and once it confirms that it is the string `"STOP"`, it skips to the statement after the loop, not executing the `push_back` statement. A flowchart for this pattern clearly shows the loop-and-a-half structure:

loop and a half

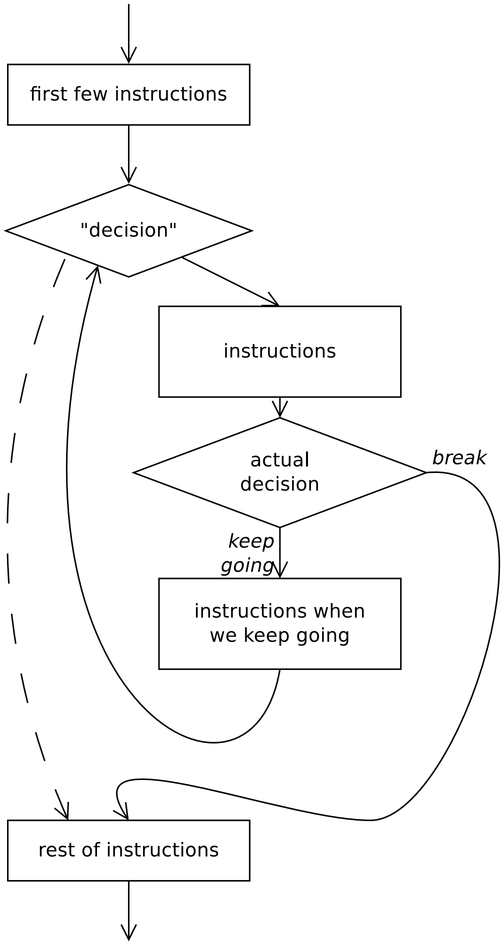


Figure 8.2

The dotted line and the quotes around the first "decision" show that the while condition is always true and the loop never really ends there—the real loop control is on the `if` statement that may decide to `break` out of the loop. Some stylebooks advocate against the use of `break` statements, but especially when input is involved, there are times when it makes the cleanest code—but only if the loop is a short one (under ten lines or so). If the `break` statement is deep in the middle of a very long loop, it makes it harder to read and reason about the program.

- Try writing a program that keeps reading numbers until one of them is negative, then printing the smallest of them (not including the negative).

8.3 Other general loops

Although we've used data input as a motivation for needing a general `while` loop, another broad category of loop is when the loop control is based primarily on numbers counting up (or counting down).

This program fragment is a simple but typical example of a loop involving neither input nor an existing collection of data, but prints a message a number of times based on the value of a variable `n`:

```
int i = 0;
while (i < n)
{
  cout << "Hi!" << endl;
  i += 1;
}
```

Program fragment 8.4: Printing a message n times

The variable `i` here represents something like "how many times has the loop run so far?"—initially zero, and when it has run `n` times, we stop.

Just about any time we use a `while` loop, there is one variable in particualr that changes its value over the lifetime of the loop and which is used to decide whether the loop needs to keep running. This is often called the *loop control variable* (LCV).

loop control variable

The following three-element list can serve as a checklist of sorts for most general loops involving `while`:

- declare and init the LCV, or otherwise set it up, *before* the loop
- establish the keep-going condition, involving the LCV
- make progress on the variable, *inside* the loop (often at the bottom), *always*

Unlike in the accumulator checklist, where updating the accumulator might only happen sometimes, the LCV needs to make progress, somehow, every single time the loop runs. If it didn't, then that would mean the next run of the loop would be identical to the current one—still not making progress—and thus that the loop would never terminate. The progress doesn't have to be an increase; it could be a decrease, or input from `cin`, or a change to the data being processed, or some more subtle change of condition, as long as it gets closer to the keep-going condition eventually being false.

For example, this fragment takes a pre-existing string named `phrase` and prints it out backwards:

```
int i = phrase.length() - 1;
while (i >= 0)
{
  cout << phrase.at(i);
  i -= 1;
}
cout << endl;
```

Program fragment 8.5: Iterating from back to front

The loop control variable, `i`, gets an initial value before the loop starts, is tested in the loop condition, and is updated on the last line of the loop—by *decreasing* it, which brings it closer to 0.

- Try writing a program that reads two numbers from the user and then counts, starting at the first number and stopping at the second.
- Try writing a program that reads a number from the user and then prints the even integers up to that number (remember that the update doesn't have to go up by 1 each time).

8.4 `do - while` loops

The last of the loop constructs is the `do-while` loop, so named because it begins with the keyword `do`, and the `while` condition comes *after* the loop.

`do-while`

```
int i = 5;
do
{
  cout << "Hi!" << endl;
  i -= 1;
} while (i > 0);
```

Program fragment 8.6: Using a `do-while` loop

For every time through the loop after the first, this is exactly equivalent to the other kind of `while` loop, since the condition is tested between the end of one repetition and the start of the next. But by putting the test at the end,

- the loop always runs at least once, even if the condition is false to begin with (because it's not tested before the first trip through the block), and
- the loop condition can use a variable that does not get a value until sometime during the loop.

The flow of the do-while loop is as shown below:

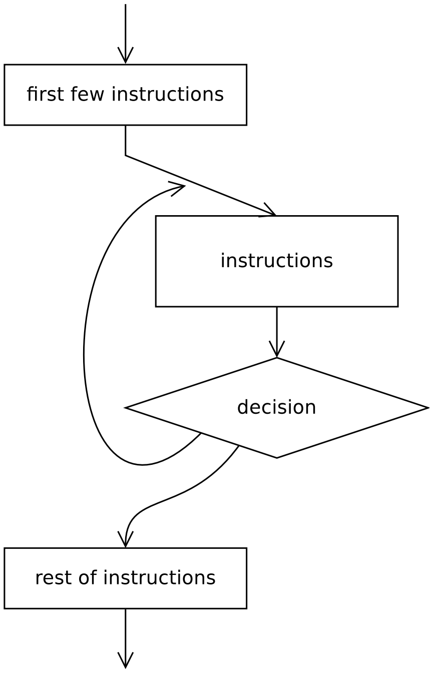


Figure 8.3: Flow of a `do-while` loop

C++ and a number of other languages have some version of a do-while loop, with the condition tested at the end. However, in practice, a loop-and-a-half design tends to be more common and more useful.

Chapter 9

Functions

20231219-1815

For some time now, we've known how to call functions that are defined in the libraries. In this chapter, we'll see how to define our own.

9.1 Functions that accept and return one value

We've already seen functions like `sqrt` that process numbers, perhaps in a statement such as this:

```
double side = sqrt(area);
```

Program fragment 9.1: The `sqrt` function

Here, `area` is the function's *parameter*, the value that the function will be operating on. Functions can have one or more parameters, or even zero parameters (we'll see those a bit later). The value produced by the function call is its *return value*; imagine that the function call is sending an assistant out to figure out the answer, and when the assistant *returns* they have the answer, to be used in the rest of the computation.

parameter

return value

Suppose we wish to write our own function `halve` that might be called as follows:

```
double result = halve(n);
```

Program fragment 9.2: Using `halve`

The first line of our function will specify information about how the function will interact with its parameters and return values; the rest will be a block with instructions on how to perform the computation. Halving a number is simple, so the full definition is short:

```
double halve (int number)
{
  return number / 2.0;
}
```

Program fragment 9.3: Defining `halve`

The first part of the first line designates the *return type* (here, `double`); then the name of the function; and finally, in parentheses, the list of *formal parameters* with both their type and their name. The block, sometimes referred to as the *function body*, can have arbitrary C++ code in it, and can use the parameters by name as variables. The function should contain (for now, always as the last line) a *return statement* that evaluates an expression of the appropriate type to return.

return type

formal parameters

function body

return statement

Functions need to be declared before they are used. Declaring is slightly different than defining, but for now we'll say that the function definition has to be earlier in the `.cpp` file than any code that calls it. That gives us a plausible complete program file such as the following:

```
// Defining and using a function to halve integers
#include <iostream>
using namespace std;

double halve(int number)
{
  return number / 2.0;
}

int main()
{
  int n;
  cin >> n;

  cout << "Half of " << n << " is " << halve(n) << endl;

  return 0;
}
```

Program fragment 9.4: Defining and using a function to halve integers

Notice anything interesting about `main`? We've been defining our programs using `main` since the very beginning without talking about what `main` really is. It's a function! When the operating system runs your program, what it's really doing is calling the `main` function, which has no parameters[1] and which returns an `int`. This is also why our sample programs have a `return 0;` as their last line. The return value of `main` can be used to signal the operating system that something went wrong. Because `main` is special, technically you can omit the return statement from it, in general, but it's a good practice to leave it in there.

- Try writing a function `triple` that computes triple the value of a given integer, and use the function appropriately in the rest of the program.
- Try writing a function `spaces` that counts how many spaces are in a given string, and use it appropriately. (What are its parameter and return types?)

[1] Strictly speaking, it could have parameters—these are called *command-line parameters*—but we won't be using that feature for now.

9.2 Functions on vectors

The type of function parameters is not restricted to simple types like `int` and `string`; it's perfectly acceptable for vectors to be parameters. Many of the programs we've written in previous chapters, which work on vectors and print their result, are adaptable into functions that perform the same computation and then return it:

```
int count_high (vector<int> numbers)
{
  int result = 0;
  for (int num : numbers)
  {
    if (num > 100)
    {
      result += 1;
    }
  }
  return result;
}
```

Program fragment 9.5: A function to count values above a threshold

Indeed, the main difference, other than "return instead of printing", is that return statements *can* usefully occur somewhere other than the end of the function.

9.2.1 Early return

When a program is executing, the keyword `return` doesn't only indicate what the return value will be—it is also an instruction to return that value, immediately. If it is not the last line of the function, all futher computation inside the function is terminated.

To see why this might be useful, consider the function `find_first_negative`, which is intended to return the first negative number in a vector, or zero if there aren't any:

```
double find_first_negative (vector<double> numbers)
{
  for (double num : numbers)
  {
    if (num < 0)
    {
      return num;
    }
  }
  return 0;
}
```

Program fragment 9.6: Returning from inside a loop

Let's start at the bottom. Consider why that final return statement is necessary: if there are no negative numbers in the vector (including the case where the vector is empty!), the `if` is never true, so we'll return zero. But we don't want that statement to execute if a negative number is actually found! Moreover, if we find *multiple* negative numbers, we don't want the later ones to interfere. Conveniently, then, the return statement inside the loop will execute immediately when it finds a match; and when it executes, it drops everything else on the floor and immediately ends the function by returning the found value.

9.3 Functions and good design

Essentially everything we might want to write a function to do *could* be done without a function. So why bother with them?

We've already seen how to use named intermediate values to make a computation easier to design and easier to read: they *encapsulate* a small, well-defined piece of a computation so that the reader (and programmer) can better make sense of them, and verify them. Functions are used in much the same way, with even more separation of the different parts of a computation. Functions can confer this "named value" benefit even on multi-line computations, perhaps involving conditionals or loops or both.

encapsulation

Functions provide an additional important benefit to a well-designed program, too. By making the program more *modular*—with distinct, well-defined parts—we make those parts reusable. As your programs get more complex, if the same computation is used in defferent parts of the program, it can be defined once in a function and

modular

called from wherever else in the program it's needed. Indeed, the same function can be kept and used again in other programs, if it's sufficiently abstract or general-purpose. The most important and reusable functions can get bundled into *libraries*, to be used by programmers elsewhere.

libraries

The shorter a function is, the more likely it is to be readable, testable, and reusable. A good rule of thumb is that if a function—including `main`!—is longer than about a dozen lines (maybe twenty or so if there's a lot of input and output), you should think about whether there's some part of it that might make sense as a nameable separate function.

9.3.1 Function order and headers

Since good design is tightly related to whether a program is legible to other programmers, it's sometimes important to think about what order functions are presented in the file. In our program example earlier, `halve` was defined first, then `main`. Some programmers find it more communicative to put the high-level overview first and then put the details, the helper functions, afterward. That would naturally lead to a program like the following (some lines omitted):

```
// Defining and using a function to halve integers
#include <iostream>
using namespace std;

int main()
{
  ... halve(n) ...
}

double halve(int number)
{
  // ... detail omitted ...
}
```

Program fragment 9.7: Defining main first—compiler error

This means that as the compiler scans the file, when it gets to the line that refers to `halve`, that function has not yet been defined, nor even mentioned—and the C++ compiler will thus reject it. To get around this problem, and let programmers arrange their programs in the order they find clearest, C++ lets us declare a function before we define it, providing only the *header* of the function (also known as its

header

signature—the first line, before the open curly bracket, indicating the return type, name, and parameters) in a sort of table-of-contents that lets the compiler know a definition of that program is on its way. Function declarations can go at the top of the file (or, eventually, in a separate file) before any of the functions are defined:

signature

```
// Defining and using a function to halve integers
#include <iostream>
using namespace std;

// Function headers
double halve(int number);

// Function definitions
int main()
{
  ... halve(n) ...
}

double halve(int number)
{
  // ... detail omitted ...
}
```

Program fragment 9.8: Defining main first—fixed

It's a good habit to always add all your function headers to the top of the program file, even if the compiler isn't insisting on it—again, for human readers, it provides a sort of table-of-contents. But, be careful: if you have declared a function, and call it, but never get around to defining it, the compiler will loudly reject it with something called a *linker error*:

linker error

```
/usr/lib64/gcc/x86_64-suse-linux/7/../../../../x86_64-suse-linux/bin/ld: /tmp/ccG8Yi
/home/dblaheta/intro/halving.cpp:8: undefined reference to `halve(int)'
collect2: error: ld returned 1 exit status
```

Figure 9.1: A typical linker error

The fix to this kind of linker error is to make sure that all promised functions have actually been defined (and have the signature that was promised for them).

9.4 Can we write functions that return nothing?

If the point of a function is to consume parameter values and produce return values, there is an interesting question of how to think about things that the function does *other than* computing the return value.

Any effect a function has other than its return value is formally known as a *side effect* of that function. It's possible for functions to read input or print to the screen, or modify files, or to modify values of variables elsewhere in the program, and all of these are side effects. A side effect is not bad (unless it's unexpected or incorrect, of course), but they can make a program a little harder to reason about if a function has side effects on top of its primary task of returning a value.

side effect

However, sometimes one of those effects is really the intended job of the function: "print the entire contents of the vector, one item per line" is a very well-defined task, and writing a function to do that is probably a reasonable design choice. The output is still referred to as a side effect—again, this is a technical term—but it is not surprising or unexpected because it's part of the description of the function.

Indeed, it's so much the main purpose of the function that it doesn't make sense to return a value at all. When writing a function that will have no return value, we still need to put *something* as the return type, a sort of placeholder, and for this we use the special typename `void`. A *void function* is one that explicitly does not return anything (and if you try to return a value from it, the compiler will reject that as an error). So our print-the-whole-vector function might look like this:

`void`

void function

```
// prints the full contents of the given vector to cout, one element per line
void print_words (vector<string> words)
{
  for (string word : words)
  {
    cout << word << endl;
  }
}
```

Program fragment 9.9: Printing the contents of a vector to `cout`

9.5 What about a function with no parameters?

It's certainly syntactically valid to write a function that takes zero parameters, but the first version that suggests itself isn't very interesting:

```
// always returns "hello"
string always_hello ()
{
  return "hello";
}
```

Program fragment 9.10: Why would you do this?

Normally, if we want a function to compute something, we want that computation to be different each time it's run, and the way to make that difference is to make use of the parameter(s) to control it.

The parameterless function can potentially be useful, though, if the variation comes from outside the function, in the form of a side effect. There are at least two useful side-effect-based ways to control a function, that might be used in a function with zero parameters. The first is input: if you read from `cin`, and return the result with or without processing, that would be a reasonable use of a function with no parameters:

```
// reads a name consisting of two "words" as a single string
string read_full_name()
{
  string first;
  cin >> first;
  string last;
  cin >> last;
  return first + " " + last;
}
```

Program fragment 9.11: A function to read from `cin`

Another side effect that might make a parameterless function useful is accessing a random number generator. In the `<cstdlib>` header, C++ provides a builtin function `random()` that returns a different number every time it's run, ranging from 0 up to about 2 billion on most systems.

`random()`

That's a big number, and typically when you want randomness in a program, it's to choose between a small number of options. A standard trick for turning a large random integer into a small one is to use the remainder operator—if you divide a big number by, say, 4, the remainder is about equally likely to be 0, 1, 2, or 3.

Encapsulating that computation into a function is a nice way to simplify the code that calls it:

```
// generates a whole number from 1 to 6, inclusive
int roll_die()
{
  return (random() % 6) + 1;
}
```

Program fragment 9.12: Generating a random number

9.5.1 Seeding the random number generator

If you typed in the function from the last section and tried to use it, you might have noticed that the sequence of die rolls is the same every time you run the program—not very random! A full discussion of random number generation is outside the scope of this book, but it's important to know that random numbers are generated by starting with a *random seed* and then using that number to generate other numbers. You only provide the seed once per program, with a call to `srandom`, and as long as the seed is different each time the program runs, the random numbers will be unpredictably different each time also.

random seed

A conventional, quick way to do this is using the `time` function found in `<ctime>`, which when called as `time(nullptr)` gives the number of seconds that have passed since January 1, 1970. That number changes every second! (What on earth is `nullptr`? We'll have to defer the details for now, but briefly, the function offers some more advanced options, and `nullptr` is a way for us to decline to use them.) Our resulting program might thus be:

```
// Rolls three dice
#include <iostream>
#include <cstdlib>
#include <ctime>
using namespace std;

int roll_die();

int main()
{
  srandom (time(nullptr));  // Call once in any program where random is used

  cout << "Three random die rolls:" << endl;
  cout << roll_die() << endl;
  cout << roll_die() << endl;
  cout << roll_die() << endl;

  return 0;
}

int roll_die()
{
  return (random() % 6) + 1;
}
```

Program fragment 9.13: Rolls three dice

On some compilers, the extra included headers `<cstdlib>` and `<ctime>` might not be required (they may get automatically included by `<iostream>`) but it's a good idea to include any headers corresponding to functions you know you'll be using.

Chapter 10

Conditionals revisited: boolean logic

20231222-0115

We've been using logic and comparisons since we first introduced `if`, but we'll formalise and develop them further in this chapter, letting us express more complex logical decisions and encapsulate them into functions.

10.1 Boolean expressions

Expressions that are designed to work as conditions are referred to as *boolean*, after the mathematician George Boole (who was among the first to study them). He observed that if a type of value can only have two possible values—in C++ we write them as `true` and `false`—there are some interesting properties that arise when operating on them and combining them. We have already seen a few kinds of boolean expressions, which we'll now explore in a little more detail.

boolean

`true`

`false`

10.1.1 Boolean operators

There are six comparison operators in C++, all of which work on numbers, strings, and characters:

$<$	`<`	Less than
$>$	`>`	Greater than
$=$	`==`	Equal
$\leq$	`<=`	Less than or equal
$\geq$	`>=`	Greater than or equal
$\neq$	`!=`	Not equal

Figure 10.1: The six standard comparisons

All six are probably familiar as mathematical operators, but in C++ (and many other programming languages) they are typed with keyboard characters as shown in the table above. When used on numbers (either `int` or `double`, or a combination), they carry their usual mathematical meaning. When used on characters and strings, they indicate whether one value is earlier than the other, according to a version of alphabetical order as indicated in Section 7.3.3.

As we have seen, these comparisons can be used in the conditions of `if` and `while` statements.

```
while (n < 5)
if (word == "STOP")
if (ch <= 'Z')
```

Program fragment 10.1: Examples of using comparison operators

10.1.2 Boolean functions and variables

We have also seen some specific examples of *boolean functions*, functions such as `isupper` that can be called to serve as conditions:

boolean functions

```
if (isupper(letter))
```

Program fragment 10.2: Using a boolean function

Methods can also be boolean, as with the (not previously mentioned) method `.empty` of every `string` value:

```
  if (line.empty())       // equivalent to: if (line == "")
```

Program fragment 10.3: Using a boolean method

We can also define our own boolean functions, using the typename `bool` as the return type; the return statements should either return `true` or `false` explicitly, or return some other boolean expression:

`bool`

```
bool lower_or_nonletter (char ch)
{
  if (isletter(ch))
  {
    return islower(ch);
  }
  else
  {
    return true;
  }
}
```

Program fragment 10.4: Writing a boolean function

Once defined, it can be used in any condition:

```
  for (char letter : phrase)
  {
    if (lower_or_nonletter(letter))
    {
      cout << letter;
    }
  }
```

Program fragment 10.5: Using a programmer-defined boolean function

Indeed, the `bool` type is one much like the others we know (like `int` and `string`), and can be used in the same ways, including as the type of a variable, which is occasionally useful for pre-computing and storing the result of a boolean expression, and later using it as a condition:

```
bool negative = n < 0;
// ...
if (negative)
{
  // ...
}
```

Program fragment 10.6: Using a boolean variable

but is perhaps most useful when signaling loop stopping conditions when a `break` statement is not suitable:

```
bool more_left = true;
while (more_left)
{
  // do some stuff
  if ( ... )
  {
    more_left = false;
  }
  // other stuff
}
```

Program fragment 10.7: Boolean variable as loop condition

10.2 Combining boolean expressions

It is often the case that the logic of an algorithm requires multiple conditions to be combined—multiple conditions on the same value, or conditions on multiple different values. In English, we would say that one condition must be true *and* another must also be true; and similarly we use the words *or* and *not* to combine and modify conditions.

So also in C++. The operators `and` and `or` can sit between any two other boolean expressions, and the operator `not` can be used before any boolean expression, to build a more complicated expression.

`and`
`or`
`not`

```
if (x > 0 and y > 0)  //runs when both variables are positive
  // ...

while (not ispunct(ch))  //until the character is punctuation
  // ...

while (isletter(ch) or ch == '-')  //when ch is either a hyphen or a letter
  // ...

if (n > 0 and 1 / n > 0.25)    // what if n is zero?  See below
  // ...
```

Program fragment 10.8: Using `and`, `or`, `not`

The last example in that listing illustrates an important aspect of the boolean operators known as *short-circuiting*. On first inspection, it might seem that the expression runs the risk of crashing the program when `n` is zero, due to a divide-by-zero error. But it is in fact safe, because the boolean operators are guaranteed to be evaluated left-to-right, and to stop once the outcome is known. Here, in the case when `n` is zero, the expression `n > 0` is false, so the overall expression cannot possibly be true, so the right side is never evaluated. Similarly, an expression like this is safe:

short-circuiting

```
if (str.empty() or str.at(0) == '+')
  // ...
```

Program fragment 10.9: Using short-circuited `or`

Here, in the case where the string `str` is the empty string, the expression `str.empty()` is true, and so the overall expression is guaranteed to be true, and so it does not evaluate the right side (which would be an error since there is no character at index 0). A consequence of this is that reversing the order of the operands of `or` (or `and`) is not, in general, safe. The following version of the code fragment may seem logically equivalent:

```
if (str.at(0) == '+' or str.empty())
  // ...
```

Program fragment 10.10: BAD: the left side of the `or` can cause an error

But because the operands are evaluated from left to right, the computer would attempt to evaluate `str.at(0)` even on an empty string with no character at index 0. Be careful with this!

An important side note: though the rest of this book will continue to use these forms (`and`, `or`, `not`), each has a punctuation-based form as well, which is precisely equivalent, and is also widely used:

`and`	`&&`
`or`	`||`
`not`	`!`

Figure 10.2: Alternate forms of boolean operators

If you do use the punctuation-style boolean operators, it's critical that you use the doubled forms as shown; the operators `&` and `|` do exist but mean something slightly different, and will usually compile and sometimes seem to work (at least in some cases), but will lead to subtle errors.

10.2.1 Warning: arithmetic "between" expressions

In mathematical contexts, it is common to indicate that a value falls within a range using an expression of this general form:

$$0 \leq i < n$$

Figure 10.3: A valid and natural mathematical comparison

However, the seemingly similar boolean expression is a trap:

```
if (0 <= i < n)    //WARNING: won't do what you expect
  // ...
```

Program fragment 10.11: BAD: combining comparison operators the wrong way

A few programming languages (notably, Python) do permit this notation in its mathematical meaning, and some others will mark it as a syntax error; but many, including C++, will compile it, but it won't work as expected. (The reason it compiles is that boolean values can be treated as numbers, so the first comparison (here `0 <= i`) evaluates to either 0 or 1, and that numeric value is then compared to `n`. Not what you'd want.) To perform the equivalent comparison in C++ (and most other programming languages), you need to explicitly make both comparisons and combine them with a boolean operator:

```
if (0 <= i and i < n)   // this one works
  // ...
```

Program fragment 10.12: The right way to combine comparisons

10.2.2 Warning: multiple comparisons to the same variable

Sometimes we want a program to behave in a certain way when a variable holds any of a small set of values. It is natural to express the condition in English as something like this:

> If the answer is either "Y" or "y", ...

Figure 10.4: A natural multiple-comparison question to pose

and so we might naïvely (but incorrectly) translate this into C++ as

```
if (answer == 'Y' or 'y')    //WARNING: always evals to true
  // ...
```

Program fragment 10.13: BAD: multiple comparisons the wrong way

Another trap! It compiles, but it definitely does not do what you want it to. (Here the reason is the inverse of the issue in the previous section: numbers and other values can be treated as boolean values, so the right side of the `or`, which is simply `'y'` by itself, is converted to a boolean (specifically, `true`) and so the overall condition is always true. Not what you'd want.) To perform the intended comparison, always ensure that *both sides* of any boolean operator are themselves boolean expressions:

```
if (answer == 'Y' or answer == 'y')    // this one works
  // ...
```

Program fragment 10.14: The right way to make multiple comparisons

10.3 Truth tables

In this section we dive a little deeper into the boolean operators introduced in the previous section.

The `and` operator works about the way most people would expect it to: the overall expression is true exactly when both the expression to the left of `and` and the expression to the right of `and` are true. One way to express this is through the following *truth table*:

truth table

`and` Right Left	T	F
T	T	F
F	F	F

Figure 10.5: Truth table for `and`

The corresponding table for the `or` operator is similar, but with a possibly surprising value in the upper-left quadrant:

`or` Right Left	T	F
T	T	T
F	T	F

Figure 10.6: Truth table for `or`

When working with program logic (or with propositional logic in general), the `or` operator is to be understood as an *inclusive* or, meaning that it's true if its left or right operand are true, *or both.*

Finally, for completeness, the table for `not`:

`not` Right	T	F
	F	T

Figure 10.7: Truth table for `not`

This one looks a little different because `not` only has a single operand, to its right, rather than having one on each side.

10.4 Complex boolean expressions

Because the boolean operators both require boolean expressions and evaluate to boolean values, they can be combined in arbitrarily complex combinations, using multiple operations. For example,

```
if (index >= 0 and not isletter(word.at(index)))
  // ...

if ( (x > 0 and y > 0)
    or (x == -1 and y == -1)
    or (not check_bounds) )
  // ...

while ( remaining > 0 and
        (process(numbers, x) or other_process(numbers, y, z)) )
  // ...

return 0 <= i and i < size and 0 <= j and j <= counter;
```

Program fragment 10.15: Examples of complex boolean expressions

As boolean expressions get complex, it's often a good idea to use parentheses to indicate what order to evaluate them. There *is* a standard order of operations—`not` is evaluated first, then `and`, then `or`—but if you are uncertain, the parentheses make it unambiguous. Note that in the above listing, removing the parens in the second `if` would keep the logic the same, but removing the parens within the condition of the `while` would change the outcome; it is currently of the form

```
    r > 0   and ( p(n,x)   or   o_p(n, y, z) )
```

but the no-paren version

```
    r > 0   and   p(n,x)   or   o_p(n, y, z)
```

is actually equivalent to

```
  ( r > 0   and   p(n,x) ) or   o_p(n, y, z)
```

As expressions get more complex, in order to understand how they evaluate, it's sometimes useful to work with another form of truth table (which can work with more than two expressions). Here we write the simplest parts of the boolean expression as column headers, then a double line, then their combined form; then in each row, the values to the left of the double bar represent the "inputs" to the system, and we evaluate them to eventually show the "output". For the expression `m<2 and n>5`:

`m<2`	`n>5`	`m<2 and n>5`
F	F	F
T	F	F
F	T	F
T	T	T

Figure 10.8: Computing a simple `and`

In more complex expression, we add a column header for each progressively larger combination. Applying this to the condition from the `while` statement above,

`r>0`	`p(n,x)`	`o_p(n,y,z)`	`p(n,x) or o_p(n,y,z)`	`r>0 and (p(n,x) or o_p(n,y,z))`
F	F	F	F	F
F	T	F	T	F
F	F	T	T	F
F	T	T	T	F
T	F	F	F	F
T	T	F	T	T
T	F	T	T	T
T	T	T	T	T

Figure 10.9: Combining with multiple boolean operators

We can use this style of truth table to emphatically illustrate how evaluating the version without parentheses will differ:

`r>0`	`p(n,x)`	`o_p(n,y,z)`	`r>0 and p(n,x)`	`r>0 and p(n,x) or o_p(n,y,z)`
F	F	F	F	F
F	T	F	F	F
F	F	T	F	T
F	T	T	F	T
T	F	F	F	F
T	T	F	T	T
T	F	T	F	T
T	T	T	T	T

Figure 10.10: Combining in a different order

The bottom half of the table has the same outcomes, but the third and fourth rows yield a different result, so we know the two expressions are not equivalent.

10.4.1 DeMorgan’s Law

There are many logical rules that have been noted to explain and record how different boolean expressions might be proven to be equivalent. (A complete analysis is known as *boolean algebra* and will generally be found in a discrete math class—a working familiarity with the various equivalence rules will be very useful for you as programs get increasingly complex.) One in particular needs a little of our attention, because so many people find it counterintuitive that it can easily lead to logical bugs if you’re not aware of it, and it tends to turn up in some form almost any time you write complex expressions that have negation (`not`) in them.

boolean algebra

DeMorgan’s Law tells us that if we negate a whole expression that contains a boolean operator, like this:

DeMorgan’s Law

```
    if ( not( isletter(ch) or isspace(ch) ) )
```

that it is NOT equivalent to naïvely “distribute” the negation:

```
    if ( not isletter(ch)  or  not isspace(ch) ) )  //NOT THE SAME
```

but it IS equivalent to distribute the negation if you *also* change the operator:

```
    if ( not isletter(ch)  and  not isspace(ch) ) )  //equiv to original
```

This may seem surprising. But consider: if it is not the case that a character is either a letter or a space—then the character is definitely not a letter, and also definitely not a space. But if we consider the middle example, marked NOT THE SAME, we’ll see that expression will always be true, since for any given character, either it’s a non-letter, or (if it is a letter) it’s a non-space.

We can prove it with truth tables:

`lett`	`spc`	`lett or spc`	`not (lett or spc)`
F	F	F	T
T	F	T	F
F	T	T	F
T	T	T	F

Figure 10.11: Truth table for original expression

`lett`	`spc`	`not lett`	`not spc`	`not lett and not spc`
F	F	T	T	T
T	F	F	T	F
F	T	T	F	F
T	T	F	F	F

Figure 10.12: Truth table for equivalent expression

10.5 Testing with complex booleans

When trying to test your program or function to confirm that it works, it's important to provide effective test coverage, making sure that you're not letting some likely-to-occur case slip by. If the processing is largely numeric, sometimes just two or three cases are enough, and certainly we can't seriously test our programs with *every possible number*. But with boolean logic... there are only two possible values for every simple boolean expression, and even when we combine them, there are often a manageably small number of distinct cases. Sometimes we can literally test all of them!

Even if the inputs themselves are not boolean, you can sometimes use the idea of truth tables to help you with case analysis and test case building. Two rules of thumb here:

- If you've got a truth table, or part of one, to help you think about the problem, make sure that every section of the truth table (maybe not every single line) is represented somewhere in your test cases.
- As you think about your algorithm, if you have `if` statements (and especially if you also have `else if` blocks), make sure that every "branch" of the conditional is represented somewhere in your test cases.

Sometimes that means you'll have three or five or more tests, but that's ok! Don't add tests purely for the sake of adding tests, but if you can identify cases that you're writing program code for that aren't showing up in the tests, it's appropriate to write more tests to cover them.

Chapter 11

Structures

v20231221-2015

Up to this point, we've seen five core types[1] plus `vector`s that can contain one of the others. There are many things out in the world that those core types can model by themselves, and vectors let us model large collections of values.

But sometimes—frequently, even—the real-world data we have to handle is a little more complex than singular values disconnected from anything else. For instance, when modeling locations on a discrete 2D cartesian plane, we need to provide coordinates, often written as an (x, y) pair. These could in some circumstances be specified as two separate variables, but what if we want to keep a collection of multiple locations? What if we wanted to return a location from a function (which can only return one value at a time)? Even when we *can* use separate variables to track related data, it sometimes hinders human reading of the program if they have to keep track of separate things, and always update them together or print them together or whatever.

Once you think to look for it, this kind of multi-part data is everywhere. Books have titles and authors (and maybe some other things). Automobile brands have a make and a model. Customer info typically will have a name and an account number. Street addresses have a whole bunch of distinct pieces.

C++ lets us model this kind of data by bundling it together as a single value, with component parts that can be accessed by name; the overall value is called a *struct* (after the keyword that builds it) or an *object*, and the component parts are usually called *fields*.

struct

object

fields

[1] `int`, `double`, `string`, `char`, and most recently `bool`

11.1 Defining a new `struct` type

To define a new `struct`, we need to specify:

- the name for the overall type, and
- a type and name for each field.

Since we want to be able to use these new types pretty much everywhere, they are defined *outside* of any function, typically near the top of a `.h` file named for the type.

Consider the example described above about modeling locations on a discrete 2D cartesian plane. In a file named `Location.h`, we could type

```
/** Models cartesian coordinate integer x,y pairs */
struct Location
{
  int x;
  int y;
};
```

Program fragment 11.1: A struct definition for cartesian coordinates

By convention (not universal among C++ programmers, but fairly common), we begin the name of the type with a capital letter to indicate that it is distinct from types provided with the language and standard library. Syntactically, the word `struct` and the curly brackets are mandatory, as is the semicolon after the closing bracket—unlike statement blocks, which are not followed by semicolons.[2]

Notice that each line within the `struct` looks just like declaring a local variable. That's not an accident! The syntax is intentionally the same: first the type, then the name, then a semicolon. Another word for *field* is *instance variable*, after the fact that each separate *instance* of the struct can have its own separate value for that field.

instance variable

instance

Let's see how this works in practice.

Once we have defined the type, we can declare and init variables of the type:

[2] The reason for the semicolon here is largely historical and has to do with variable declaration. Most modern compilers will at least give you a good error if you forget it.

```
Location origin = Location{0,0};
Location quad1 = Location{3,4};
```

Program fragment 11.2: Some `Location` examples

The only new syntax here is that the initial value for the variables uses the name of the type, and then specific values for its fields inside curly brackets, a bit like we learned to do when initialising vectors.[3] The values in the curly brackets represent the values to assign to the fields of the object, in the order they were declared.

If we want to access an individual field of an object, we follow the name of the object with the name of the field, e.g. "`origin.x`". So to print out the contents of the `quad1` variable we might write a statement like

```
cout << "quad1 is at (" << quad1.x << "," << quad1.y << ")." << endl;
```

Program fragment 11.3: Accessing a `Location` value

which in this case would print

```
quad1 is at (3,4).
```

Of course, if we only ever accessed the contents of one of these struct values by separately checking each field, they wouldn't bring much advantage. In the next section we'll see one important way to use them.

11.1.1 Review

1. Write code to define a struct to model items stocked in a grocery store, which have names and prices.
2. Write code to define a struct to model full names, which can be separated into first names and last names. (This is an oversimplification! See the end-of-chapter exercises to work towards fixing it.)

[3] Indeed, in this context we can use the exact same syntax as we did for vectors, e.g.

```
Location quad3 = { -5, -12 };
```

but by preceding the curly brackets with the type name we can use it in a few other places too, as we'll soon see.

11.2 Structs as function parameters

We've long been able to write functions that take multiple parameters, so we could imagine declaring a function such as this one:

```
/** determines whether the location corresponding to the given x and
  * given y value lies on either axis */
bool on_axis (int x, int y);
```

Program fragment 11.4: Header for function processing locations, without using a struct

The downside to such a design is that the function description is a bit clunky and the parameters are not intrinsically linked to each other. However, now that we have a coherent single type called `Location` for modelling locations, we can rewrite this header as

```
/** determines whether the given Location lies on either axis */
bool on_axis (Location loc);
```

Program fragment 11.5: Header for function processing `Location` values

The semantics of the function should be fairly clear from the description:

The expression	should be
`on_axis(Location{0,0})`	`true`
`on_axis(Location{0,7})`	`true`
`on_axis(Location{-3,0})`	`true`
`on_axis(Location{3,4})`	`false`
`on_axis(Location{5,-12})`	`false`

Note that in order to specify these test cases we can make concrete values of type `Location` on the spot by making use of the name notation we used when initialising variables. Of course, if we have the variables declared (as from the previous section) we can use those too:

The expression	should be
`on_axis(origin)`	`true`
`on_axis(quad1)`	`false`

since variables like `origin` and `quad1`, declared as being of type `Location`, can be used anywhere a `Location` object is required, including as the parameter to `on_axis`.

Defining the function requires no new syntax beyond that introduced in the previous section to access the individual fields:

```
bool on_axis (Location loc)
{
  return loc.x == 0 or loc.y == 0;
}
```

Program fragment 11.6: Defining `on_axis`

Consider further a function designed to compute the distance between locations. Each location has two coordinates, so a function that tried to do this before structs would need *four* parameters, and the opportunities for confusion would be many. But with our new struct, we can declare the function as follows:

```
/** computes the distance (Euclidean) between two given Locations */
double distance (Location first, Location second);
```

Program fragment 11.7: Description and header for `distance`

Reusing the previously-named values for convenience:

The expression	should be
`distance(origin, origin)`	`0.0`
`distance(origin, Location{0,7})`	`7.0`
`distance(origin, Location{-3,0})`	`3.0`
`distance(origin, quad1)`	`5.0`
`distance(quad1, origin)`	`5.0`
`distance(Location{-2, 3}, Location{3, -9})`	`13.0`
`distance(Location{3, 3}, Location{4, 4})`	$\sqrt{2}$ or about `1.414`

Values that line up vertically or horizontally are easy to compute by hand, and some carefully-chosen Pythagorean triples make most of the rest work out nicely, but it's important to note that answers will not in general work out to integer values!

The body of the function again makes use of the dot to access fields, but is otherwise just an implementation of the Pythagorean/Euclidean distance formula:

```
double distance (Location first, Location second)
{
  int xdiff = first.x - second.x;
  int ydiff = first.y - second.y;
  return sqrt(xdiff * xdiff + ydiff * ydiff);
}
```

Program fragment 11.8: Defining `distance`

11.2.1 Review

1. Based on the grocery item struct you wrote in the previous review, write a function that determines whether a given grocery item costs more than $5.

2. ... and another function that computes the total cost of a given item when a given tax rate is applied.

3. Based on the full name struct you wrote in the previous review, write a function that builds a string view of a given full name in “Last, First” format.

4. ... and another function that builds a string view of a given full name in “First Last” format.

11.3 Structs as function return values

It’s not only possible to use struct values as input (parameters) to a function; we can also use them as output (return values) from a function. Indeed, this usage is arguably even more important, since we *can* always take in additional parameters, but can only return one value from a function—so if we want to in effect return two things, we have to bundle them into a single (hopefully meaningful) object.

Returning a struct object from a function is syntactically no different than returning any other type of value from a function: Declare it as the return type in the header, and write a `return` statement (or more than one) that produces a value of that type.

For instance, this function header continues the `Location` example from the previous sections:

```
/** chooses which of two given Locations is closer to (0,0). */
Location closer_to_origin (Location a, Location b);
```

Program fragment 11.9: Header of function that returns `Location`

Its header looks just like other functions we've written, except that the return type is `Location`. The following table gives some illustrative test cases for the function. Note that the expected results are written in a human-readable (x, y) format rather than something that might be directly typed into a program.

The expression	should be
`closer_to_origin(origin, origin)`	(0,0)
`closer_to_origin(origin, quad1)`	(0,0)
`closer_to_origin(Location{5,0}, Location{0,7})`	(5,0)
`closer_to_origin(Location{0,7}, Location{5,0})`	(5,0)
`closer_to_origin(Location{4,0}, quad1)`	(4,0)
`closer_to_origin(Location{-4,0}, quad1)`	$(-4,0)$
`closer_to_origin(Location{-6,0}, quad1)`	(3,4)

In this case, the return statement can be simply returning one or the other of the function's parameter values. (Note also that we can make very effective use of the `distance` function from the previous section!)

```
Location closer_to_origin (Location a, Location b)
{
  constexpr Location origin = Location{0,0};
  if (distance(origin, a) < distance(origin, b))
  {
    return a;
  }
  else
  {
    return b;
  }
}
```

Program fragment 11.10: Defining `closer_to_origin`

That need not always be the case. This next function will need to construct a brand-new `Location` value on the fly.

```
/** computes the Location that is the reflection through (0,0) of the
  * given Location. */
Location reflection_through_origin (Location loc);
```

Program fragment 11.11: Description and header of `reflection_through_origin`

Reflecting through a point just means finding the other point that's exactly the other side of the reflection point (here, the origin). That is,

The expression	should be
`reflection_through_origin(Location{5,0})`	$(-5, 0)$
`reflection_through_origin(Location{0,7})`	$(-7, 0)$
`reflection_through_origin(quad1)`	$(-3, -4)$
`reflection_through_origin(Location{-5,-12})`	$(5, 12)$
`reflection_through_origin(origin)`	$(0, 0)$

The algorithm here may now be obvious: to reflect through the origin, just negate both components. The implementation of the function thus creates a value to return using the same `Location{...}` syntax as we used when initialising variables, except this time the values for the fields are themselves computed with mathematical expressions:

```
Location reflection_through_origin (Location loc)
{
  return Location{-loc.x, -loc.y};
}
```

Program fragment 11.12: Defining `reflection_through_origin`

In general, the expressions used for the fields of the newly-built struct object can be arbitrarily complex, as long as they produce values of the appropriate type for that field (in this case, `int`, since both `x` and `y` are declared as `int` values).

It's important to note that if a function is required to produce a value, simply modifying an existing value (such as a parameter) won't be enough. If we had instead tried to do this:

```
Location reflection_through_origin (Location loc)
{
  loc.x = -loc.x;           // WON'T WORK
  loc.y = -loc.y;
}
```

Program fragment 11.13: BAD: Function tries to work by modifying parameter

we would have had a few problems. First of all, it says it returns a value, but doesn't do so! Even if we redeclared the function void, however, we run into the issue that (as with the core types), calling a function makes a *copy* of the parameter values, so any modifications to the parameter variables are local to the function and are lost when the function ends. Effectively, this implementation attempt would change some internal values, and then drop everything on the floor, having no effect on anything outside the function.

11.3.1 Review

1. Still building on the earlier full name example, Write a function that produces the full name associated with a given string written in "First Last" format. Assume there is exactly one space in the parameter.

2. Write a function that produces the full name associated with a given string written in "Last, First" format. Assume there is exactly one comma and one space in the parameter.

3. Still building the earlier grocery item example, write a function that produces a grocery item just like a given one, except with a price that is 25 cents higher.

4. Write a function that makes a grocery item to model a "pack" of a given number of a given other grocery item, with a name reflecting that fact and a price that is the correct multiple of the original. For instance, if one soda costs 30 cents, a pack of 12 might be called a "pack of soda" costing $3.60.

11.4 Vectors of structs

Yet another place that we can usefully use these new struct-type values is as contents of a vector. We can make a vector of objects directly in the initialisation of a value:

```
  vector<Location> collection = { Location{5,7}, Location{-1,4},
                      Location{3,10}, Location{2,-5}, Location{-4,-4} };
```

Program fragment 11.14: A vector of `Location` objects

which might be appropriate for building test cases. More typically (in practice), they would be read in from a file or from the user, or produced as the result of another process (which we'll see more of later).

Once we know that such vectors exist, we can write functions to process them. For instance,

```
/** finds the location in the given vector that is furthest from (0,0)
  * @precondition locs is not empty */
Location furthest_from_origin (vector<Location> locs);
```

Program fragment 11.15: Processing a vector of `Locations`

The expression	should be
`furthest_from_origin( { quad1 } )`	$(3, 4)$
`furthest_from_origin( { origin, quad1 } )`	$(3, 4)$
`furthest_from_origin( { Location{-6,0}, quad1, Location{0,7} } )`	$(0, 7)$
`furthest_from_origin(collection)`	$(3, 10)$

Writing a function that processes a vector of struct objects is not substantially different from the vector-processing functions we've written before. If we need to drill down and access the fields of the objects, we can use dot notation to do so; we can also make use of other functions that process the individual struct objects, where applicable.

```
Location furthest_from_origin (vector<Location> locs)
{
  Location result = locs.at(0);  //error if locs is empty!
  for (Location loc : locs)
  {
    if (closer_to_origin(result, loc))
    {
      result = loc
    }
  }
  return result;
}
```

Program fragment 11.16: Defining `furthest_from_origin`

11.4.1 Review

1. Write a function that counts the number of full names in a given vector that match a given first name.
2. Write a function that finds the full name in a given vector that is alphabetically first (by last name).
3. Write a function that computes the total cost of a given vector of grocery items.
4. Write a function that determines whether any of the grocery items in a given vector have a given name.

Chapter 12

Functions producing and modifying vectors

20231221-2045

We've previously studied vectors, and we've studied functions, and in this chapter we'll study functions that produce vectors. In some sense this will be mostly review—though there are some new details about how functions work under the hood, and a little bit of new syntax—but it deserves its own chapter because the patterns we see when accumulating vectors are very important and may seem substantially different from the accumulator patterns we've already seen.

12.1 Map

map

The *map* pattern in programming is related to the mathematical concept of a *mapping*, and involves transforming some given vector of values into another vector, of equal length, that "maps" each original value into a corresponding different value. For instance, this function maps words into their corresponding lengths:

```
vector<int> word_lengths (vector<string> words)
{
  vector<int> result = {};
  for (string word : words)
  {
    result.push_back(word.length());
  }
  return result;
}
```

Program fragment 12.1: Computing lengths of a whole vector of words

For instance,

The expression	would evaluate to a vector containing
`word_lengths({"hi", "there"})`	`{ 2, 5 }`
`word_lengths({"a", "b", "c"})`	`{ 1, 1, 1 }`
`word_lengths({ })`	`{ }`

Figure 12.1: Test cases for `word_lengths`

The pattern of the function is quite similar to the sum pattern from when we first learned about accumulators; the biggest difference here is that the accumulation is into a list, using `.push_back`.

There are a few reasons to particularly identify and use this pattern. First, once identified, we can see that the only major difference between any two mapping functions is the parameter to `.push_back`, so once a programmer knows that a mapping is to be done, most of the function follows a very standard format.

The other is that a function like this can be used as part of a sequence of operations, each relatively standard, that might be easier to assemble as “components” than it would be to write as a single monolithic program. For instance, assuming the `Location` struct defined in the previous chapter, one typical task might be to find the average x value of an entire vector of locations. Using a function that first collects the x values:

```
vector<int> x_values (vector<Location> locs)
{
  vector<int> result = {};
  for (Location loc : locs)
  {
    result.push_back(loc.x);
  }
  return result;
}
```

Program fragment 12.2: A function to collect x values from a vector of locations

produces a vector of numbers that can be passed as the parameter to any standard averaging function that takes a vector of numbers, leading to a simple statement like

```
  cout << average(x_values(locs)) << endl;
```

Program fragment 12.3: Using `x_values`

12.2 Filter

Out next pattern is the *filter* pattern, which takes a given vector as parameter and produces a brand-new vector that contains some, but not all, of the elements in the original. Some condition serves to "filter" the values, keeping some and excluding others from the final result. For instance, the following function filters a vector to keep only the locations that are in the first quadrant, i.e. neither coordinate is negative:

filter

```
vector<Location> only_quadI (vector<Location> locs)
{
  vector<Location> result = {};
  for (Location loc : locs)
  {
    if (loc.x >= 0 and loc.y >= 0)
    {
      result.push_back(loc);
    }
  }
  return result;
}
```

Program fragment 12.4: Filtering first-quadrant values in a vector of locations

This pattern is similar to the counting pattern we've already seen: looping through the entire vector, it uses an `if` statement to select only some of the values in the vactor for further processing.

Where the map pattern produces a vector with the same length, but potentially a different content type, a function that purely follows the filter pattern will produce a vector with the same content type—since the values are copied from the original—but will, in general, be shorter than the original. It's certainly possible to write a single function that combines aspects of both map and filter patterns—using an `if` statement to select some elements of the vector, and then also mapping the element to some other related value—but using one map function, then a filter function, can provide the same result, with a very straightforward pattern-based implementation. Following on the earlier example, we might compute the average x value of all first-quadrant locations in the list without writing any more functions than we already have:

```
  cout << average(x_values(only_quadI(locs))) << endl;
```

Program fragment 12.5: Chaining functions together

12.3 Pass-by-value ("copy") semantics and reference parameters

At this point we need to dig in a little deeper on how function parameters work. When a function is called, under normal circumstances C++ follows a *pass-by-value* semantics, which means that the value that is seen inside a function is actually a *copy* of the value that was sent as the parameter. This can potentially matter for two different reasons: modifications to the value are purely internal to the function (because the modifications are on a copy of the original), and the copying itself may be expensive.

pass-by-value

Why would the copying be "expensive"? Imagine for a moment that the vector we used as a parameter had a thousand, or a million, or a billion elements. In that case, calling a function in the usual pass-by-value form would require first copying every element, even before any of the actions of the function are taken. When a parameter is something simple like an int or double, the act of copying is quick, but when a value could be arbitrarily large, like a vector or even a string, it's preferable to avoid copying unless it's specifically necessary to the algorithm.

In C++, we have another option. If you *pass by reference*, the body of a function can be written identically, but it operates on the original value instead of on a copy, saving the time and trouble of making the copy. To designate a parameter as pass-by-reference, follow the typename with an ampersand (`&`):

pass by reference

```
int count_negative (vector<int>& numbers)
      // ...
```

Program fragment 12.6: Passing by reference, version 1

This confers an advantage—it does not require making a copy—but the very fact that it is operating on the original value may also be a disadvantage: modifying the original value, even if inadvertent, is now possible. Furthermore, if we mark a value as a reference parameter, the compiler will require all values passed as parameters to be variables that can be modified (and not, for instance, literal values constructed on the spot).

As a result, in cases where the parameter is not meant to be modified by the function, and we are only using reference parameters for reasons of efficiency (avoiding an unnecessary copy), it's a good idea to designate it as a *const reference*, preceding the type with the keyword `const`. For example,

const reference

```
int count_negative (const vector<int>& numbers)
      // ...
```

Program fragment 12.7: Passing by const reference

This is the preferred way of declaring any parameters that are strings, vectors, or any user-defined type—anything more complex than a single number—when there is no need to modify the parameter value inside the function. It retains all the advantages of references (no need to copy) without inviting any bugs from accidentally modifying something. Furthermore, because it's guaranteed that no changes will be made, the compiler permits literals, values that cannot be modified, to be passed as parameters:

```
  cout << count_negative ( {-1, 2, 4, -3, 0, -5} ) << endl;
```

Program fragment 12.8: A valid call to `count_negative`

When in doubt, specify parameters that are vectors as const references; remember that doing so requires both the keyword `const` (indicating it won't be modified) and the ampersand `&` (indicating it is a reference).

12.4 Adding and removing values

Sometimes, though, we want to write a function whose job is specifically to modify the contents of a vector. In such a case, we *can't* use pass-by-copy semantics, because if we just modified the copy, the modification wouldn't persist and the function wouldn't be very useful. And we also can't use const references, since that would prevent the modifications that are part of the intent of the function.

In its simplest form, this could just modify a vector by adding or removing something at the end of the vector, using `.push_back` or `.pop_back`:

```
// Precondition: given vector is not empty
void duplicate_last (vector<int>& values)
{
  int value = values.at(values.length() - 1);
  values.push_back(value);
}
```

Program fragment 12.9: Modifying a vector in a function

Other than the ampersand in the header, there's nothing in the syntax that is any different from other vector-processing code that we've written; without that ampersand, the function would compile and run, but the modification to the vector wouldn't "stick" once the function ended, since the operation would be performed on a copy of the vector.

But adding a single element to the end of a vector is both simple and a bit uninteresting. We could add *all* the elements from another vector:

```
void add_all (vector<string>& words, const vector<string>& others)
{
  for (string word : others)
  {
    words.push_back(word);
  }
}
```

Program fragment 12.10: Copying the contents of one vector into another

Note that the second parameter, `other`, is declared `const`, because we do not plan on modifying it.

What if we want to remove a value from the vector, but not at the end of the vector? If our only tool for removing an element is `.pop_back`, we need to do a bit more work to remove something in the middle while keeping everything else in the same order. If we mean to remove an element at a given index, all the elements after it need to effectively "move over" by one. One at a time, we'll replace an element with (a copy of) the element next to it, until it is the very last element that is duplicated—and then we can safely call `.pop_back` to remove it.

```
void remove_element_at (vector<double>& values, int rm_index)
{
  int i = rm_index;
  while (i < values.size() - 1)
  {
    values.at(i) = values.at(i+1);
    i += 1;
  }
  values.pop_back();
}
```

Program fragment 12.11: Removing an element, shifting the others over

Take a moment to trace out for yourself how this function runs when it has a five-element vector and it removes something at index 1, and then try it again with a different five-element vector and remove something at index 4.

12.4.1 Case study: inserting elements

With that in mind, try to sketch out the algorithm for a function `insert_element_at`, which should insert into a given vector of strings a given value at a given index. As with the previous example, depending on the index, you may need to move a number of existing elements to make the rest of the algorithm work. Trace your algorithm to see if it works before continuing on!

It is likely, if you haven't thought about this problem before, that your initial attempt is modeled after the `remove_element_at` implementation, and looks something like this:

```
void insert_element_at (vector<string>& values,
                        const string& value, int ins_index)
{
  int i = ins_index;
  while (i < values.size() - 1)
  {
    values.at(i+1) = values.at(i);
    i += 1;
  }
  values.at(ins_index) = value;
}
```

Program fragment 12.12: BAD: Inserting elements, still buggy

This version has many of the required elements, but is incorrect in two main ways. First, it never calls `.push_back`, so the size of the vector never grows! But as you think about where the call to `.push_back` might go, the other, more subtle problem may present itself: what value do you need to add at the *end* of the vector, and where in the function should that action go?

After the insertion, the new last element of the vector will be the same as the old last element[1]—meaning that what needs to be passed to `.push_back` is a copy of the existing last element. And, more importantly, moving the *last* element has to happen *first*... because if the copying proceeds in the order shown in the above implementation, the value originally at `ins_index` will be repeatedly copied on top of everything else, and all the other values are lost. But if the last element is copied, that leaves a spot for the next-to-last element to be copied, and so on—we need to iterate from back to front:

[1] Kudos if you spot the exception to this statement—stay tuned

```
void insert_element_at (vector<string>& values,
                        const string& value, int ins_index)
{
  values.push_back(values.at(values.size() - 1));
  int i = values.size() - 1;
  while (i > ins_index)
  {
    values.at(i) = values.at(i-1);
    i -= 1;
  }
  values.at(ins_index) = value;
}
```

Program fragment 12.13: BAD: Much better, but still has a bug

Try tracing that version with a problem where you need to insert an element into a 5-element vector at index 2. Then, try to identify where the remaining bug is. What are some meaningful edge cases to try here?

(Really, try to spot the bug yourself before you keep reading!)

The case to be looking for here is the one where the insertion index is at the very end of the vector. Even in that case, the above implementation will often end up with the correct result after some unnecessary work—it doesn't need to copy the old last element, but at least after calling `.push_back` the vector is set up for the last assignment statement to correctly place the inserted element. But the function will actually fail if the vector is empty, even if the insertion is (validly!) at index 0, because that first line will try to grab the "current last element", which doesn't exist.

So this version, which treats inserting at the end of the vector as a special case, is cleaner and more correct:

```
void insert_element_at (vector<string>& values,
                const string& value, int ins_index)
{
  if (ins_index == values.size()
  {
    values.push_back(value);
  }
  else
  {
    values.push_back(values.at(values.size() - 1));
    int i = values.size() - 1;
    while (i > ins_index)
    {
      values.at(i) = values.at(i-1);
      i -= 1;
    }
    values.at(ins_index) = value;
  }
}
```

Program fragment 12.14: Inserting an element

In general, if you're modifying a vector, you need to think about how its elements need to move around—maybe hardly at all, maybe a lot—and plan accordingly.

Chapter 13

Indexed loops, nested loops, and 2D data

20231222-0100

In this chapter we'll return to the mechanics of writing loops, and introduce a new style of loop that gives us more control over iteration, enabling some more complex interactions with our data.

13.1 Indexed `for` loops

The first type of loop we saw, back in Chapter 5, was a `for` loop designed to access every element of a vector (and later, every character in a string). Later, we saw `while` loops, which gave a more general kind of loop construct; but we never really used them to access an already-existent vector, because the loop patterns we were using worked fine with `for`, which was simpler.

Sometimes, though, we will see an algorithm that requires more fine-grained control over the iteration, perhaps by skipping certain elements, or iterating in an unusual order, or accessing adjacent elements, or something else that requires knowing the *index* of the element as well as its value. Because we can use the `.at` method of a vector, we can use a `while` loop to produce each valid index in turn, and use them to access the elements. (The remove and insert algorithms in the previous chapter were examples of this.) This technique is sometimes called an *indexed loop*.

indexed loop

For instance, an algorithm to find the indices of all the empty strings in a vector requires knowing the index of the elements as we look at them! Here is a version of that algorithm:

```
// e.g. { "abc", "", "", "def", "" } should produce { 1, 2, 4 }
vector<int> empty_indices (const vector<string>& strs)
{
  vector<int> result = {};
  int i = 0;
  while (i < strs.size())
  {
    if (strs.at(i) == "")
    {
      result.push_back(i);
    }
    i += 1;
  }
  return result;
}
```

Program fragment 13.1: Finding matching indices, take 1

Here, `i` is the loop control variable—init before, compare as keep-going condition, and make progress each time through the loop—as well as being an index of the `strs` vector. When a match is found, the index is stored in the accumulator and eventually returned.

Another algorithm, to determine whether a vector of values is in ascending order—each element greater than the previous one—is naturally implemented by directly comparing each adjacent pair of elements:

```
// e.g. { 1.0, 2.5, 2.6, 37.0, 37.00001, 50.0 }  should be true
// and  { 1.0, 2.5, 2.5, 37.0, 36.999, 50.0 }  should be false
bool is_ascending (const vector<double>& values)
{
  int i = 1;
  while (i < values.size())
  {
    if (values.at(i-1) >= values.at(i))
    {
      return false;
    }
    i += 1;
  }
  return true;
}
```

Program fragment 13.2: Checking ascending order, take 1

Here the LCV begins at 1 because the loop accesses at both index `i-1` and `i` (so starting at 0 would access a negative index!), but otherwise the loop control looks pretty similar.

In fact, this LCV pattern is so common and so standard, it enjoys direct language support to help you make sure you don't forget any of the three parts.

13.1.1 C-style `for` loops

Recall that the Loop Control Variable (LCV) pattern introduced in Section 8.3 has three parts:

- declare and init the LCV, or otherwise set it up, *before* the loop
- establish the keep-going condition, involving the LCV
- make progress on the variable, *inside* the loop (often at the bottom), *always*

In C++ there are two different loop types that involve the keyword `for`, and the older of the two—dating back to the C programming language from the 1970s and thus often known as *C-style* ***`for`***—is one that simply encapsulates this LCV pattern. To use it, write the keyword `for`, and then in parentheses, the three parts of the LCV pattern, separated by semicolons. The following two loops have almost precisely identical effects:

C-style ***`for`***

```
int i = 1;
while (i <= 10)
{
  cout << i << endl;
  i += 1;
}

for (int i = 1; i <= 10; i += 1)
{
  cout << i << endl;
}
```

Program fragment 13.3: Two loops that each print 1-10

The first clause of the C-style `for` is expected to init the LCV (usually including declaration, in which case its scope is precisely the loop[1]), the second is a condition that is evaluated repeatedly to decide whether to execute the loop again, and the third is an update statement to be executed at the end of the loop.

Note that last bit carefully—the *end* of the loop.

It's really important to remember that this `for` loop was and is seen as shorthand for the `while`-based LCV pattern, which virtually always does its update at the end of the loop, and so that's where the update happens in the shorthand version too. If we executed the `i += 1` at the top of the loop in the above example, the numbers would start at 2!

Let's revisit the `empty_indices` algorithm with the new shorthand:

[1] This fact leads to an extremely subtle way that the two loops above can differ in their effects, and the only reason I had to write "almost" in the description—can you spot it?

```
vector<int> empty_indices (const vector<string>& strs)
{
  vector<int> result = {};
  for (int i = 0; i < strs.size(); i += 1)
  {
    if (strs.at(i) == "")
    {
      result.push_back(i);
    }
  }
  return result;
}
```

Program fragment 13.4: Finding matching indices, take 2

Although it doesn't provide any capability not already present in the base `while` loop, the conciseness of the one-line LCV pattern (and the ability to see all three parts in a quick glance) has made the C-style `for` loop ubiquitous, and worth training your eyes to read fluently.

In fact, there's one more piece of shorthand you should learn to read as well.

13.1.2 The increment (and decrement) operators

Particularly in the LCV pattern, but also in a number of longer patterns and algorithms, we see the most common use of the `+=` operator to add exactly 1 to a variable. Because it's a special case, and so common, a special operator `++` was defined just to handle this "plus one" operation, also known as *incrementing*. Anyplace you would write

`++`

incrementing

```
value += 1
```

you can instead, equivalently, write

```
++value
```

which doesn't even really save much space on the page but a lot of people prefer writing it, and in most contexts you'll see it much more frequently than `+= 1`. The increment operator can also be written after the variable it is affecting:

```
value++
```

and when it is being used as a simple statement or as the update in a C-style `for` loop, these two forms are effectively equivalent.[2]

Though somewhat less common, the mirror operation known as *decrementing* also has a shortcut operator:

decrementing

```
value -= 1
```

can be replaced with

```
--value
```

with no change in meaning.

The loop format and the increment operator let us rewrite our `is_ascending` function as follows:

```
bool is_ascending (const vector<double>& values)
{
  for (int i = 1; i < values.size(); ++i)
  {
    if (values.at(i-1) >= values.at(i))
    {
      return false;
    }
  }
  return true;
}
```

Program fragment 13.5: Checking ascending order, take 2

An experienced programmer is likely to use those forms when writing a program, so you should get comfortable reading them.

13.2 Nested loops

We've seen before that `if` statements can "nest" inside each other, and of course we have many examples of an `if` statement inside a loop, so it may not be entirely surprising that we can also write one loop inside another, forming a *nested loop*

nested loop

[2] In more complex uses, outside the scope of this course, they behave slightly differently, so if you try to build complicated one-liners that involve `++`, read up on "pre-increment" and "post-increment" first.

pattern.

One of the simplest nested loops we can write is one that prints a small rectangle on the screen:

```
for (int m = 0; m < 5; ++m)                      @@@
{                                                @@@
  for (int n = 0; n < 3; ++n)                    @@@
  {                                              @@@
    cout << "@";                                 @@@
  }
  cout << endl;
}
```

Figure 13.1: A simple nested loop, and what it prints

Before you continue, try to work out why it prints in that pattern. Why not horizontally? What would happen if the `endl` were inside the inner loop, in addition to or instead of where it is now? What if it were outside both loops?

Pushing the envelope a little, we can use some `char` values to shed some light on how this works:

```
for (char ch = 'J'; ch <= 'M'; ++ch)             JJJJJJJ
{                                                KKKKKKK
  for (int n = 0; n < 7; ++n)                    LLLLLLL
  {                                              MMMMMMM
    cout << ch;
  }
  cout << endl;
}
```

Figure 13.2: A nested loop with letters, and what it prints

Each full trip through the inner loop prints seven copies of the current value of `ch` from the outer loop, then one newline, then the next letter is up to bat. There is no special meaning to a nested loop—it works just like any other, and the entire loop body is run from start to finish (even if it contains other loops that need to run in their entirety) before moving on to the next iteration.

13.3 An application of nested loops: sorting

A very frequent need for data is that it be put in order, usually to make some future processing more efficient (or at least to make it prettier or more readable in the output). This process is referred to as *sorting* the data, and there are literally

sorting

hundreds if not thousands of distinct sorting algorithms, many of them in widespread use. An interesting property of the sorting problem is that in general—unless your data already happens to luckily be in order—you can't sort a vector in one single-pass iteration through that vector. Because you don't know what order it's in to begin with, you might have to compare each element to many other elements to find where it's supposed to go.

So one strategy is to use two nested loops to do the sorting. The *selection sort* algorithm is not the fastest possible sorting algorithm, but it is straightforward and it gets the job done. Its basic strategy is:

selection sort

```
Target the first position in the vector
Loop through the rest of the vector to find the smallest value
If there's a value smaller than one in the target position, swap places
Advance to the next position and repeat
```

Figure 13.3: Selection sort pseudocode

That is, it is always *selecting* the smallest remaining value, and placing it into its final, sorted position in the vector. Turned into C++ code, it looks something like this:

```
void selection_sort (vector<double>& values)
{
  for (int target_index = 0; target_index < values.size(); ++target_index)
  {
    int smallest_index = target_index;
    for (int i = target_index + 1; i < values.size(); ++i)
    {
      if (values.at(i) < values.at(smallest_index))
      {
        smallest_index = i;
      }
    }
    if (smallest_index != target_index)
    {
      swap (values.at(smallest_index), values.at(target_index));
    }
  }
}
```

Program fragment 13.6: Selection sort

The outer loop here sets `target_index` to each index of the vector in turn. Inside the body of the loop, we see a slight variation on the min/max algorithm we learned in Chapter 7: instead of keeping the smallest *value* in the accumulator, we keep the *index* of the smallest value. Once the loop ends, the accumulator has the index of the smallest value in the rest of the vector; and if that's even smaller than the value currently at the target index, we swap it. Technically, the only thing here that is new is the `swap` function itself: you do need to `#include <utility>` to use it but it causes the two given values to change places.

`swap`

To get a better feel for how it works, trace through the algorithm with a given set of starting values, such as $\{6, 3, 2, 5, 4\}$—if possible, use physical pieces of paper or other objects (playing cards are great for this) to give you a feel for how the values move around in the vector as the algorithm runs. Why do we init `smallest_index` to `target_index` instead of 0? Why does `i` start at `target_index + 1`? What does the algorithm do if the values are already in order? What happens if you run the algorithm on a vector with just one element? Or no elements?

13.4 Two-dimensional data

Sometimes it's the case that data can be best understood as two-dimensional. Perhaps it consists of measurements of different locations in the world, laid out in a grid with (x,y) coordinates. Perhaps it's once-a-month measurements that relate both to different months in the same year, but also to different years. Whatever the context, we can think of two-dimensional data as a grid of values that can be accessed by multiple coordinates, instead of by one single index to mark its position.

There are a few ways to represent 2D data in C++, each with their pros and cons. The one we'll work with now is the most flexible, but a little tricky to set up. The underlying idea here is that if a vector of numbers arranges that data in one dimension, then a vector *of vectors* can arrange it in two. That is, use the type `vector<vector<double>>` to represent the grid of data.[3] As with one-dimensional data, it is possible to declare-and-init the values directly into the vector; here, we know that each element of the 2D value is itself a vector of data, so we enclose each one with curly brackets:

```
vector<vector<double>> grid = {
  { 3.0, -4.5, 1.0 , 0.75 },
  { 1.2, 31.4, 0.25, -2.8 }
};
```

Program fragment 13.7: Example of 2D data

This way of getting values into a vector-of-vectors presents them pretty strongly as row-based, that is, each single vector contained in the larger grid represents a horizontal "row" of the grid:

[3] Some very old C++ compilers required an extra space in the declaration, like so: `vector<vector<double> >`. You'll still see programs floating around online that follow this convention, but the space hasn't been required since the 2011 standard.

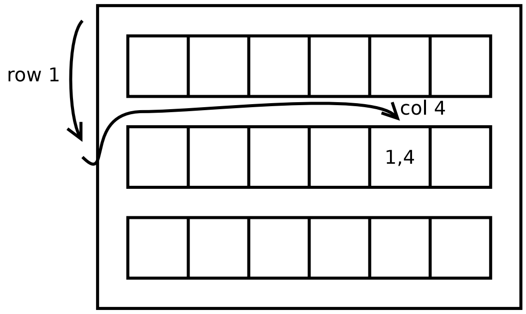


Figure 13.4: A vector of vectors, viewed as a row-column system

In a row-based model, the first coordinate thus designates the row, and a second coordinate would designate the column. For instance, on the data assigned to `grid` above, `grid.at(0).at(2)` would be 1.0, and `grid.at(1).at(3)` would be −2.8. A nested loop to iterate through them would be well-advised to use variables with names like `row` and `col` (or `column`) to make the iteration pattern easier to follow:

```
double sum = 0;
for (int row = 0; row < grid.size(); ++row)
{
  for (int col = 0; col < grid.at(row).size(); ++col)
  {
    sum += grid.at(row).at(col);
  }
}
```

Program fragment 13.8: Computing the sum of all values in `grid`

Note the upper bound on the inner loop: the `size` of the whole grid is in this case the number of rows, 2, so to get the number of columns we need to ask *the whole row vector* how many values it has. (Depending on what else is going on in the program, we could also have variables with names like `width` or `num_cols` that we could use for loop bounds as appropriate.)

Another way to set up 2D data is with the contents being vertical; the first coordinate chooses an “x value” and the second is a “y value” showing how far down to go:

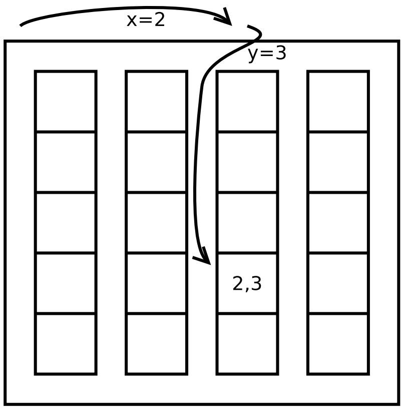


Figure 13.5: A vector of vectors, viewed as an x-y system

How do you tell which way to interpret the 2D value in a particular program? *Only by documentation and context,* because both are valid ways to interpret the data. Setting up an x-y-based 2D value most easily accomplished if you know in advance how big it will be, in which case the initial setup can establish both dimensions when the vector gets its initial value:

```
vector<vector<int>> xy_grid =
    vector<vector<int>>{width, vector<int>{height, 0}};
```

Program fragment 13.9: Assumes `width` and `height` are already defined

Here, we see that the variable `xy_grid` is given a value that is of the correct type and with `width` elements, each of which is itself a newly-constructed `vector<int>` that has `height` elements, each of which is (for now) the value `0`. That is, a width×height 2D value whose individual `int` elements are all set to our requested default value, zero. Next, we would read in or generate the values to go in each position (or perhaps our problem really does want everything to start out at zero, conveniently).

(This pattern for setting up a vector-of-vectors is not only for the x-y systems—but if you use it for a row-column system, remember that if *row* is the first coordinate, then *height* is the first bound!)

Yet *another* way to set up a 2D value is if you're reading data from the keyboard or a text file and don't know in advance how large it is. If the data is presented one row at a time, you can read it in and add it, one row at a time, using `.push_back` to add entire rows:

```
vector<vector<double>> grid = {};  // no rows yet
string line;
while (cin >> line)   // assumes no spaces in line!
{
  vector<double> row_data = {};  // no numbers yet

  /* code to process line to put values into row_data */

  grid.push_back(row_data);
}
```

Program fragment 13.10: Building 2D data one row at a time

(Unless the data is single individual characters for some reason, the processing to turn one line of characters into a vector of data is a little tricky for a short example, so we're handwaving past that part here.)

Practice problems

1. Define a variable with a grid of integers that has 4 rows and 8 columns. Write a loop that finds the smallest number in each row. Try doing it both with and without writing a helper function.

2. Build a 11x11 grid that holds the value of the expression $2x + \frac{y}{2}$ at each integer coordinate. After building the grid, print the grid.

3. Read a word-search grid from the keyboard or a file that contains nothing but uppercase letters; in the input, each line will be entered with no spaces, just the letters, e.g.

   ```
   THATR
   HISOU
   EREWO
   REEDY
   ```

   After reading the whole grid, print it back out, spaced so that individual letters are easier to see, e.g.

   ```
   T  H  A  T  R

   H  I  S  O  U
   ```

```
E  R  E  W  O

R  E  E  D  Y
```

Chapter 14

Strings, characters, and formatting

20231222-0045

In this chapter, we'll revisit strings and string processing for a little bit of a deeper dive into how they work and how they're represented.

14.1 Special characters

Up to this point, if we wanted to indicate that something we printed should move to the start of the next line, we used `endl`, which was not exactly a string or character, just a special keyword that could be inserted into an output stream:

```
cout << "One line of output" << endl;
cout << "Second" << endl << "and third lines" << endl;
```

Program fragment 14.1: Printing multiple lines

A more compact and versatile (and occasionally more efficient) way to do this is to make use of a special "newline" character, built in to ASCII, whose job is not to print a visible letter or mark on the screen, but rather as a *control character* that indicates to a terminal, browser, or device that it's time to start a new line. In other words, it has exactly the same role as `endl`, but since it is officially a character it can be stored in `char` variables and as parts of `string`s.

control character

We *can't* simply type the newline into a string by pressing the Enter key, though; you may have already noticed that code like this will yield a syntax error:

```
 cout << "One line of output
Second
and third lines
";
```

Program fragment 14.2: BAD: Attempting multiple lines in a single string

Instead, we substitute a special sequence “`\n`” whenever we want a newline character:

```
 cout << "One line of output\nSecond\nand third lines\n";
```

Program fragment 14.3: Including newlines within the string

Although in the program file itself we press two keys and type two characters, a backslash[1] and the lowercase letter N, this is understood by the compiler as representing a single character in the string. The string `"hi\n"` is three characters long, not four.

You can even use it in a character literal:

```
  char newline = '\n';
```

Program fragment 14.4: Newline character in a `char` literal

The compiler will complain if you put multiple characters between single-quotes (because single quotes designate a single `char` literal value) but because the backslash-then-N combination represents just one character (a control character) that’s fine.

There are a few other characters you can enter into strings and character literals using this backslash notation:

[1] It’s important that this is a backslash, not a forward slash. A forward slash is used for division operations and comments. To keep them straight, remember that if someone was walking left to right on your page like the words do, a forward slash would be them leaning forward: `/` and a backslash would be them leaning back: `\`

`\n`	Newline
`\t`	Tab
`\'`	Single quote
`\"`	Double quote
`\\`	Backslash

Figure 14.1: Some backslash-based characters

The tab character is sometimes useful for lining up output on the screen—on most terminals, by default, it inserts 1-8 visual spaces to line up with other tabbed content. The others on this list are not because the characters are ASCII control characters, but because they would normally indicate other things: this is the only easy way to include a double-quote character into a string literal!

```
cout << "She said, \"Hello!\", and then walked out.\n";
```

Program fragment 14.5: Including double-quote characters inside a string

14.2 Formatting

Sometimes it's useful to make your program's output visually line up into columns. This is less important than it once was (in the days when financial reports were printed on line printers and a lot of business software was in text windows), which is why it's been put off to Chapter 14 and is just a tiny section, but it can be handy. The simplest way to line up columns involves tab characters, but they only let you line up on 8-character boundaries and are of little help when your output varies in width by more than that.

Giving a little more control, C++ provides *stream manipulators* as a way to indicate to an output stream (such as `cout`, though stay tuned for some other options) that the very next output should use a specific number of characters, and to add extra spaces if necessary to make it line up. For instance, assume the following short example vector:

stream manipulators

```
struct Measure
{
  string item;
  int quantity;
  double measurement;
};

vector<Measure> data = {
  Measure{"first", 32, 3.0625},
  Measure{"long-named second", 4, 17.5},
  Measure{"third", 1024, 2.125}
};
```

Program fragment 14.6: Data for the next few examples

The following loop just prints out the data without formatting (output shown below)

```
for (Measure m : data)
{
  cout << m.item << ' '
       << m.quantity << ' '
       << m.measurement << endl;
}
```

Prints:

```
first 32 3.0625
long-named second 4 17.5
third 1024 2.125
```

Figure 14.2: Basic output

But if we use the setw manipulator (requires the header <iomanip>) to set the "width" that each piece is printed in, the necessary spaces are added:

setw

<iomanip>

```
for (Measure m : data)
{
  cout << setw(20) << m.item
       << setw(6) << m.quantity
       << setw(8) << m.measurement << endl;
}
```

Prints:

```
               first    32  3.0625
  long-named second     4    17.5
               third  1024   2.125
```

Figure 14.3: Formatted

That's a bit more legible, but the default right-justification is a little weird for labels, which we can fix with another manipulator:

```
for (Measure m : data)
{
  cout << left << setw(20) << m.item
       << right << setw(6) << m.quantity
       << setw(8) << m.measurement << endl;
}
```

Prints:

```
first                   32  3.0625
long-named second        4    17.5
third                 1024   2.125
```

Figure 14.4: Left- and right-justified

Note that the use of `left` applies to all future stream output until it's reset with the use of `right`.

`left`

`right`

Finally, we might want to tidy up that column with the `double` values. Often when we print decimals, we want the decimal points to line up, and we want them to show the same number of digits after the decimal point even if they end in zeroes. For this we combine the `setprecision` manipulator and the `fixed` manipulator, which together indicate we want the number of digits to consistently print after the decimal:

`setprecision`

`fixed`

```
for (Measure m : data)
{
  cout << left << setw(20) << m.item
       << right << setw(6) << m.quantity
       << setprecision(3) << fixed << setw(8) << m.measurement << endl;
}
```

Prints:

```
first                   32   3.062
long-named second        4  17.500
third                 1024   2.125
```

Figure 14.5: Decimal precision

14.3 Line-based input and streams

We've known since the earliest days of the course that when we read in strings, we only get one "word" at a time. That's because the `>>` operator, when presented with a `string` variable, is *whitespace-delimited*, meaning that it stops whenever it sees any of the whitespace characters (space, tab, or newline). But there *is* a way to read an entire line all at once: a function called `getline`. It takes two parameters, exactly the same parameters in the same order as the `>>` operator:

whitespace-delimited

`getline`

```
string word, line;
cin >> word;
getline(cin, line);
```

Program fragment 14.7: Two ways to read from `cin`

In fact, a call to the getline function can be used as a drop-in replacement anyplace that we've seen `>>` used, including as a loop condition:

```
string line;
while (getline(cin, line))
{
  cout << line.at(0) << endl;
}
```

Program fragment 14.8: Print the first letter of each line

14.3.1 Warning: mixing `>>` and `getline`

If your program reads nothing but string variables, and all those strings are meant to be typed in one per line, then you could simply replace all uses of `>>` with calls to `getline`, and the program would work identically except that now those strings could have spaces in them (as indeed most kinds of string input, like names or countries or addresses or sentences, can have).

But if the input format requires a mix of strings and numbers, or a mix of line-based strings and single-word strings, simply mixing `>>` and `getline` will lead to some subtle bugs unless you counteract them. The following code has a clear intent but it will not work as designed:

```
int num1, num2;
cin >> num1;
cin >> num2;

string firstline, secondline;
getline(cin, firstline);
getline(cin, secondline);

cout << num1 << ' ' << num2 << endl;
cout << firstline << endl << secondline << endl;
```

Program fragment 14.9: BAD: Subtle input bug

If you run the program, and type a number (and hit enter) and another (and hit enter) and then some words (and hit enter), it won't even wait for a second line of words, and it will print the words you typed as the *second* line of input. (Try it!) Why would that be?

Remember that reading input from `cin` always involves reading up to a point and then stopping, leaving the rest of the input for the next access of `cin`. It turns out that all of the `>>` input techniques, regardless of what type they're reading, skip over any initial whitespace, then read the number or char or word they're looking for, and then stop as soon as they find more whitespace—leaving the whitespace to be read by the next `cin` access. But `getline` always reads from wherever it starts, up to and including the next newline character (or the end of the input). That's a problem when a `>>` is followed by a `getline`. Suppose you tried the following input for that program:

```
42 -127
blah blah blah
additional stuff
```

Figure 14.6: Possible intended input for that program

This shows the "remaining input in `cin`" after each of the input statements, with space characters shown visibly as "␣" and newline characters as "`\n`":

Before input:
`42␣-127\nblah␣blah␣blah\nadditionalstuff\n`

After first input `cin >> num1` reads 42:
`␣-127\nblah␣blah␣blah\nadditionalstuff\n`

After second input `cin >> num2` skips the space and reads -127:
`\nblah␣blah␣blah\nadditionalstuff\n`

After third input `getline` gets *only* the newline:
`blah␣blah␣blah\nadditionalstuff\n`

Still waiting in `cin` after fourth input:
`additionalstuff\n`

Figure 14.7: As the input proceeds

This will be a problem in any program that needs to mix use of `>>` and `getline`. There are a few solutions that involve consuming the "extra" newline when you switch from `>>` to `getline`, but in many ways the cleaner solution involves *only* using `getline` for reading from `cin`, and then further processing any lines that have numbers or multiple words that need breaking up. One way to do that is to use streams, as demonstrated in the next section.

14.4 Streams

We've occasionally referred to `cin` and `cout` as *streams* (and it's right there in the name of the header file `<iostream>`), but now we'll see that the particular power of the stream concept is that they aren't the *only* streams we can use, and once we create a stream, it behaves just like `cin` or `cout`. *streams*

First, file streams. Instead of reading from the keyboard and printing to the screen (or using external operating system features to redirect them), sometimes we want to read or write an actual file on the hard drive. Type some words in a file called "`sample.txt`" and then the following code will read them and print them back, one per line:

```
// Read and print every word from the file "sample.txt"
#include <iostream>
#include <fstream>
using namespace std;

int main()
{
  ifstream file_in = ifstream{"sample.txt"};

  string word;
  while (file_in >> word)
  {
    cout << word << endl;
  }

  return 0;
}
```

Program fragment 14.10: Program to read and print from a file

Note that it requires the use of the `<fstream>` header, and that the type of the variable is `ifstream` (that is, an *input file stream*); but the actual file stream is a value stored in a variable and uses the exact syntax we've always used for that. The filename used to create the `ifstream` value need not be a string literal, but could instead be a string variable or any other expression that produces a string, as long as it's an actual filename that exists. From then on, the stream variable (here `file_in`) can be used exactly as `cin` would be (including with `getline`).

input file stream

The pattern is similar for *output file streams*, just using the `ofstream` type:

output file streams

```
  ofstream file_out = ofstream{"data.txt"};

  for (int val : values)
  {
    file_out << val << endl;
  }
```

Program fragment 14.11: Printing to a file

For output files, the file need not already exist (and in fact if it *does* exist it will be overwritten). Once created, any output file stream can be used in exactly the way that you would use `cout`.

Another type of stream that you can use is the *string stream*, which lets you treat a string inside your program as if it's a tiny little file that you can read from or write two. You'll need the `<sstream>` header, and then you can do this:

string stream

```
istringstream str_in = istringstream{"some words to read"};
string word;
while (str_in >> word)
{
  cout << word << endl;
}
```

Program fragment 14.12: Using string stream to "read" from a string

which would print the four words from the stream, each on their separate line. They are also useful for resolving situations where line-based input includes numbers or otherwise requires processing, as in the problematic example in the previous section:

```
string templine;
getline(cin, templine);
istringstream numbers = istringstream{templine};

int num1, num2;
numbers >> num1 >> num2;

string firstline, secondline;
getline(cin, firstline);
getline(cin, secondline);

cout << num1 << ' ' << num2 << endl;
cout << firstline << endl << secondline << endl;
```

Program fragment 14.13: Fixed input bug using string stream

Since all accesses of `cin` use `getline`, the newlines are handled uniformly, and while `templine` is still a string, it can be processed into the numbers by way of an `istringstream`.

In the other direction, we can use string streams as if they were a tiny output file in order to build strings using `<<` and other stream-based tools, without actually printing them to the screen; once we've "output" everything we want into the stream, we use the `.str` method of the stream to access the resulting string:

```
string format_string (GroceryItem item)
{
  ostringstream result = ostringstream{};  // initially empty
  result << left << setw(20) << item.name;
  result << right << setw(4) << item.quantity;
  result << fixed << setprecision(2) << setw(6) << item.price;
  return result.str();
}
```

Program fragment 14.14: Building a result string using string stream

14.5 C strings

The `string` type that we've been using is not the only string-representing type in C++. Dating from its predecessor language C, there also a type `const char*` (or just `char*`, read aloud as "tchar star"), which is also known as a *C string* due to its provenance. It is still used in some C++ programs when they need to interact with libraries designed for C—and any time you use a string literal. As we mentioned back in Chapter 6, string methods don't work on string literals, and the reason is that the actual type of every string literal is actually `const char*`. In most circumstances the compiler is able to automatically convert the C strings into C++ `string` values as needed, but in method calls, concatenations, and a few other places, it needs a little help. You can always explicitly construct the `string` object, just as we might do with a struct value we'd defined ourselves, and when explicitly constructed as a `string` you can use methods freely:

`const char*`

C string

```
string{"This works"}.length()
```

Moving in the other direction, if you do interact with a library that is designed to interoperate with both C and C++ (and maybe also some other languages), it may expect text-type parameters to be given as `const char*` rather than `string`. Conveniently, there is a `string` method `.c_str` that does the conversion:

`.c_str`

```
old_library_call(word.c_str())
```

Why did the designers of C++ invent a new **`string`** type rather than sticking with the old one? Put simply, the old way is pretty primitive. The value of a C string is nothing but an address in memory where the text begins—it doesn't have any other information about the string, not even how many characters long it is! The only way to find the end of the string is to scan forward until you find a special *null character*, a byte with all zero-bits that doesn't correspond to any other ASCII value. (That's different from `'0'`, which is the character for the digit zero. To represent the null character, use the number `0` by itself, or explicitly say `'\0'`.) If it's a long string, just asking how long it is requires computational work to scan for the end. If a bug in a program misplaces the null character, the string can appear to be much longer than it should be, and such an error may give rise to a *buffer overflow*, a serious security problem.

null character

buffer overflow

There are still times when you need to use a C string, and you definitely should know they're out there (not least because it will make certain error messages make more sense). But all else equal, you should probably continue using C++ **`string`** values wherever possible.

Chapter 14 Sidebar: `printf`

v20231222-0100

An older style of formatted output, a little less flexible than stream manipulators but much more compact, uses the old C function `printf` (which stands for "print formatted"). To access it, include the `<cstdio>` header (for C standard I/O). It has an unusual prototype that is written

`printf`

`<cstdio>`

```
  int printf (const char* format, ...);
```

because while it *must* have a first "format string" parameter, it can have any number of parameters after that—depending on the contents of the format string.

The format string can contain arbitrary characters, which will be printed exactly as they are except when the special percent-sign marker is used to indicate a "fill-in-the-blank" placeholder. (This usage of the percent sign has nothing to do with either percentages *or* the mod operator.) In its most simple form, the percent sign is followed by a single letter (formally known as a *conversion specification* indicating what type of value it will be replaced with:

conversion specificat

The sign	is replaced with a...	whose type is...
`%s`	string	`const char*`
`%f`	floating-point number	`double`
`%d`	decimal (base 10) integer	`int`
`%c`	character	`char`

Figure 14.8: Some common conversion letters for `printf`

Each subsequent parameter then is used to fill in the placeholders, in order. At its simplest, you can thus lay out a line to be printed in a visually readable form, without being cluttered by the expressions used to access the values, as in this statement:

```
printf ("Product: %s   (%d @$%f each)\n",
            item.name.c_str(), item.quantity, item.price);
```

Program fragment 14.15: Using printf

(note the required `.c_str()`, because `printf` needs a `const char*`), which in a loop might print something like

```
Product: can of soup   (10 @$1.39 each)
Product: 12-pack of soda   (3 @$5.49 each)
Product: jar of peaches   (1 @$2.5 each)
```

Figure 14.9: Output

which could instead be generated with the equivalent

```
cout << "Product: " << item.name
     << "   (" << item.quantity
     << " @$" << item.price << " each)\n";
```

Program fragment 14.16: Equivalent operations using `cout`

But just as with our `<<` output we could insert some manipulators to control the formatting, `printf` lets us insert additional specifications between the percent sign and the conversion letter to give information about left/right justification, field width, and decimal precision. The revised print statement

```
printf ("Product: %-20s (%2d @$%5.2f each)\n",
            item.name.c_str(), item.quantity, item.price);
```

Program fragment 14.17: Using printf with widths and precision

would, given the same data loop as the earlier example, yield the following output:

```
Product: can of soup          (10 @$ 1.39 each)
Product: 12-pack of soda      ( 3 @$ 5.49 each)
Product: jar of peaches       ( 1 @$ 2.50 each)
```

Figure 14.10: Output

and corresponds to the following stream output statement:

```
cout << "Product: " << left << setw(20) << item.name
     << " (" << right << setw(2) << item.quantity
     << " @$" << setw(5) << fixed << setprecision(2) << item.price
     << " each)\n";
```

Program fragment 14.18: Equivalent operations using `cout`

A lot of software out there (including a number of other programming languages!) will use `printf`-style output formatting, so it's important to have at least a passing familiarity with it. There are also a *lot* of other options for the format string that are not detailed here, if you're interested in diving deeper.

Chapter 15

Arrays and pointers

20231222-0100

In this chapter we'll see a little more of the internals behind how complex data can be stored, and how data is organised in memory. We'll talk about two related concepts (arrays and pointers) and a *lot* of new-to-us syntax that C++ inherited from C; both can be used in modern C++, and sometimes are (especially when interacting with older code), but many of their most common use-cases have been largely taken over by other language features like strings, vectors, and references, which you've seen, and managed pointers, which you haven't. It's important to know about these concepts and recognise the syntax, but a true deep-dive into the use of arrays and pointers is outside the scope of this book.

15.1 Arrays

An *array* in C++ is a bit like a vector: it holds a sequence of zero or more values of the same type. However, relative to a vector, an array has several important limitations:

array

- An array's *capacity*, the number of elements it could eventually hold, is specified at the moment the array is declared, and cannot be changed later *capacity*
- Arrays do not have any associated fields or methods, so we can't call `.size` or similar to get info about the array
- In general, at any moment the number of values stored in the array is less than the max capacity of the array, so we have to think about (and track) *size*, the number of actually-stored values in the array, separately from capacity *size*
- Because they don't know their own size, they also can't prevent the user from accessing or writing memory after the end of the array (a *buffer overflow*, a *buffer overflow*

common source of security vulnerabilities and irritating run-time errors)

The main upside to using arrays is that you have more fine-grained control over exactly how many bytes of memory you are using, but in modern C++ code this is rarely of much benefit when set against the limitations. Unless you have a very specific reason to choose arrays, vectors are probably a better bet.

15.1.1 Syntax

The syntax for declaring an array breaks a rule established in Chapter 4: the type of the variable does include the fact that it's an array, but the order of the statements is that the type of the *contents* comes before the variable name, but the *array* part comes after, usually also with the array's capacity as a compile-time constant. All three of the following statements declare an array of `int` with the capacity for 5 values:

```
int values[5];
int others[5] = {8, 7};
int more[] = {9, 8, 7, 6, 5};
```

Program fragment 15.1: Creating arrays

The square brackets may only be left empty if the array is immediately initialised, and the capacity is set according to the number of values in the array. If the capacity is explicitly given, and there is no initialisation or only some of the values are provided, the remaining cells of the array are *not* initialised,[1] and contain whatever bits happened to already be in those bytes of memory. If you want them to have a particular value (such as zero), make sure to write code to explicitly make that happen.

Functions that take arrays as parameters may either include or omit the capacity, but if present the capacity must be a (correct!) compile-time constant. Since arrays don't "know" their own size, array parameters are often accompanied by separate parameters to track the number of values they hold. For instance:

[1] This statement is a slight simplification; but don't *count on* array values being initialised.

```
int sum_nums (int values[], int how_many)
{  /* ... details omitted ... */ }

double average (double weights[20], int n)
{  /* ... details omitted ... */ }
```

Program fragment 15.2: Function headers using array parameters

The syntax with square brackets is also used to access the individual values of the array. In this case, the square brackets contain an expression that evaluates to an `int` index that indicates which position to access or assign a value to:

```
cout << values[2] << endl;

for (int i = 0; i < n; ++i)
{
  weights[i] = 0.0;
}
```

Program fragment 15.3: Accessing array elements

15.1.2 Square brackets on vectors and strings

Now is probably a good time to point out that the array syntax for accessing elements can also be used on most other collection types, including vectors and strings:

```
string greeting = "Hello!\n";
cout << greeting.at(0) << endl;
cout << greeting[0] << endl;

vec.at(i) = value;
vec[i] = value;
```

Program fragment 15.4: Using square brackets

The difference is not purely aesthetic, though. Using the array syntax says something like “pretend this is actually an array, and if I provide an index that’s out of range, just access it anyway”, while the `.at` syntax that we’ve been using will first verify that the index is a valid one. That can help catch some subtle bugs, by halting the program and alerting you to the bad access rather than blindly stumbling on. Consider these two incorrect loops over a vector named `nums`:

```
int product1 = 1;
for (int i = 0; i <= nums.size(); ++i)
{
  product1 *= nums[i];
}

int product2 = 1;
for (int i = 0; i <= nums.size(); ++i)
{
  product2 *= nums.at(i);
}
```

Program fragment 15.5: BAD: accessing out of bounds

The first one runs and will in general produce an incorrect answer (often zero, though not always). If you didn’t test enough you might not even notice the error, and debugging it might be challenging. The second one runs and halts, every time, notifying you that `i` is outside the valid range for `.at` (and specifically, that it’s equal to the size of the vector), which helps narrow your debugging focus, correctly, to the loop bound.

But you’ll see an awful lot of C++ code out there in the real world that uses square brackets. As long as you’re careful to check your bounds, that will work just fine.

15.2 Pointers

A *pointer*, in C++, is literally just an address in memory, a virtual arrow pointing to some particular byte that says “some relevant value is stored in that byte (and subsequent bytes)”. Because more than one arrow can point at the same spot, pointers provide a way for different parts of a program to update and access the same value—much as with references, but more general. Pointers also can be used to express (and update!) relationships between objects that are more complex than “direct sequence of values” and may not even be known in advance.

pointer

A reference, which we discussed in Chapter 12, can also be thought of as an arrow, permanently connecting the reference name to the originally-declared value that it refers to. By contrast, a pointer is itself a value; it is best illustrated with *one* box that models the pointer itself, whose value is an arrow (representing the memory address); and then a *separate* box that the arrow points to, containing the pointed-to value.

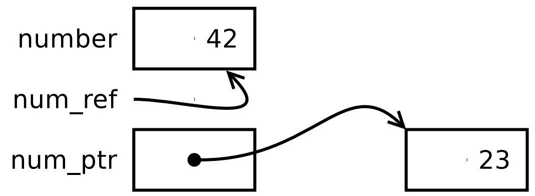


Figure 15.1: An int, an int reference, and an int pointer

Because they can point to values that are non-local (and may not have any name at all), pointers are often used as a way to establish values and objects that will usefully continue to exist after the current function is completed—as long as *someone* still has a pointer to that value.

15.2.1 Syntax

Declaring a variable to be of a pointer type requires placing an asterisk after the name of the type it will point to.[2] There are a number of ways to get a pointer value to assign to a pointer variable, but the most straightforward for us will involve the keyword `new`:

`new`

```
int number = int{23};             // or just ... = 23;
int* num_ptr = new int{23};

string word = string{"example"};  // or ... = "example";
string* word_ptr = new string{"also"};

Location loc = Location{3, 4};    // or ... = {3, 4};
Location* loc_ptr = new Location{-5, 12};
```

Program fragment 15.6: Allocating objects with `new`

[2] Important caveat: if you declare your pointer variables like "`int* np;`" as recommended here, it's important to only declare one variable per statement. The statement "`int* np, mp;`" will not do what you expect. If you declare multiple variables on a line, there's additional subtlety that is outside the scope of this book.

These examples show three pairs of declare-and-init statements, in each case first declaring a variable to directly hold a value of the type and second declaring a variable to hold a *pointer* to that type. It's generally possible to skip the `int{...}` and `string{...}` when dealing with regular values, but they are required with `new`, and I've included them here to show the parallel with other types. They would set up memory something like this:

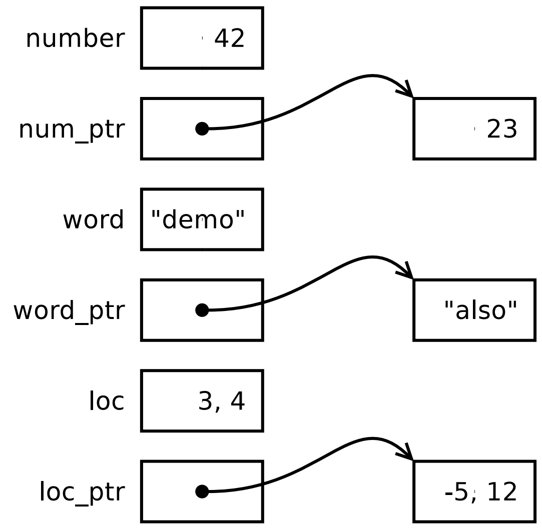


Figure 15.2: The above statements, diagrammed

Pointer access, that is, accessing the values that pointers point to, comes in two flavours. If you want to access the entire value (as is typically the case with simpler values like strings and numbers), precede the variable name with an asterisk. To access a *field* or *method* of a value through a pointer, replace the period (usually used for field/method access) with a two-character arrow "`->`". `->`

```
cout << number << endl;
cout << *num_ptr << endl;

cout << word << endl;
cout << *word_ptr << endl;

cout << word.length() << endl;
cout << word_ptr->length() << endl;

cout << loc.x << endl;
cout << loc_ptr->x << endl;
```

Program fragment 15.7: Accessing values set up in previous listing

Finally, there are two different kinds of assignment/update that are relevant here: an assignment to the *pointer variable itself* will change the arrow and make it point elsewhere (without affecting the existing value it pointed to, which might have other things pointing to it). Thus the following statements:

```
num_ptr = new int{-57};
*word_ptr = "upd";
loc_ptr->x = 8;
```

Program fragment 15.8: Assignments involving pointers

might be diagrammed as follows (former values crossed out with red lines, new values shown in green):

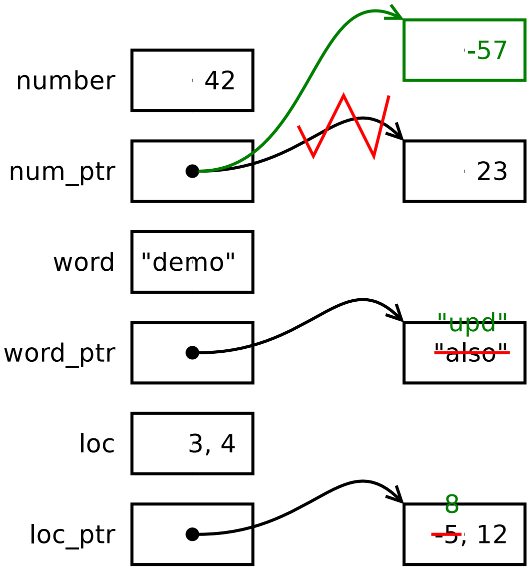


Figure 15.3: The above updates, diagrammed

Note, by the way, that the allocated box containing 23 is unchanged and, in this case, has nothing left pointing to it. The final piece of syntax is that everything that is allocated with `new` should, eventually, be de-allocated with `delete`:

`delete`

```
delete num_ptr;
delete word_ptr;
delete loc_ptr;
```

Program fragment 15.9: De-allocating memory with `delete`

But even after doing that, the box with 23 remains allocated memory, and can now never be deallocated. This is referred to as a *memory leak*, and if it happens at scale it can really slow down a computer. Keep track of your memory and make sure to use `delete` when you're done with it![3]

memory leak

15.3 Arrays as pointers (and vice versa)

As explained in the last two sections, there are differences in how arrays and pointers are declared, and how they allocated memory; but once created, they can be *used*

[3] Or use managed pointers, such as `unique_ptr` or `shared_ptr`, but the details of those are beyond the scope of the book.

interchangeably in most contexts. There are two particular ways this comes up most often: function parameters and value access.

Any function with a parameter specified as an array type can instead be given a variable (or expression) of the corresponding pointer type. Similarly, a function whose parameter requires a pointer, can be called with a corresponding array-type value. To interpret an array as a pointer, we simply treat it as the address of its first element. To interpret a pointer as an array, we imagine that the value it points to is just the first in the array of multiple allocated memory locations.

As a consequence of that, the syntax used to access the two is also basically interchangeable. If we understand a value as a pointer, we tend to use the asterisk to "access *that* value", while if we understand it as an array, we use `[0]` to "access *the first* value", but as these are precisely the same operation, they're formally interchangeable!

```
cout << others[0] << *others << endl;   // same value twice
cout << *num_ptr << num_ptr[0] << endl; // same value twice
```

Program fragment 15.10: Pointers and arrays

The most common place you're likely to see this is when working with C strings. We learned in the last chapter that they also are known as "`char*`", which we now know as a pointer to `char`; but if we want to access individual characters in the string (which doesn't have an `.at` method!), we use the square-bracket array notation.

```
char* phrase = "another example";
int i = 0;
while (phrase[i] != '\0')  //keep going until we see the "null character"
{
  cout << phrase[i] << endl;
  ++i;
}
```

Program fragment 15.11: Using pointers as arrays

Though usually declared as a pointer, C strings are virtually always *accessed* as arrays.

15.4 2D arrays

When we talked about two-dimensional data in Section 13.4, we worked with vectors of vectors, and all the options probably felt a little clunky. Another option would be to use true arrays, and C++ provides an option that will at first seem obviously superior; making space for a 2D array of `double` with 2 rows and 4 columns is done in just one line:

```
double grid[2][4];
```

Program fragment 15.12: Declaring a 2D array

If we have the exact data at compile time, we can declare-and-init it in the obvious way:

```
double grid[2][4] = {
    { 3.0, -4.5, 1.0 , 0.75 },
    { 1.2, 31.4, 0.25, -2.8 }
};
```

Program fragment 15.13: Example of 2D data in array form

We could also loop through and set values individually as we read them in, as in one of the options presented for vectors of vectors.

One reason this won't work for certain kinds of problems is the same problem as with one-dimensional arrays: you have to know, at compile time, exactly how large to make it. If your data is coming in while the program runs, and the grid can be of different sizes each time (or is just not known in advance), you'd *at least* need to allocate a larger-than-needed capacity, and then keep track of the number of rows and columns, and it starts to seem less appealing.

The other reason that 2D arrays are a bit irritating in C++ is that declaring them in function parameters requires, again, knowing their size in advance. Recall from earlier in this chapter that an array parameter can omit the size:

```
int sum_nums (int values[], int how_many)
{  /* ... details omitted ... */ }
```

Program fragment 15.14: Function header with array parameter

But if you tried the seemingly-obvious analogue for 2D arrays, it won't work:

```
int process_grid (double grid[][], int rows, int cols)
{  /* ... details omitted ... */ }
```

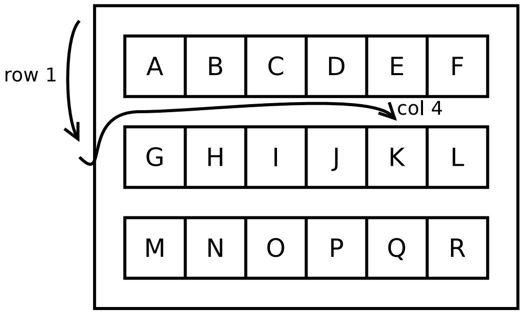
Program fragment 15.15: BAD: Generates a compiler error

To see why, note that 2D arrays, like arrays in general, are really just pointers to the start of the space. If we create a 2D array of letters and access it as follows:

```
char grid[3][5] = {
    { 'A', 'B', 'C', 'D', 'E', 'F' },
    { 'G', 'H', 'I', 'J', 'K', 'L' },
    { 'M', 'N', 'O', 'P', 'Q', 'R' }
};
cout << grid[1][4] << endl;  // prints K
```

Program fragment 15.16: A slightly larger 2D array example

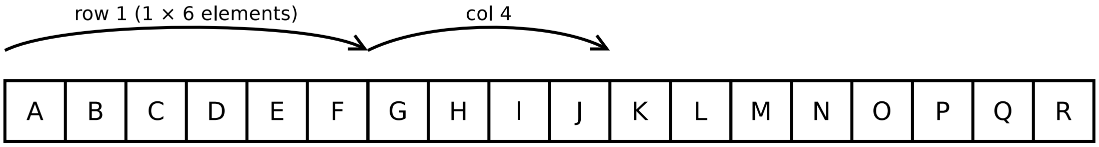
we imagine that it is accessing things like in this diagram, just like in a vector of vectors:



A	B	C	D	E	F
G	H	I	J	K	L
M	N	O	P	Q	R

Figure 15.4: BAD: A faulty mental model of the above access

But in fact, all 18 elements of the 3×6 array are stored in one linear sequence; jumping to the start of “row 1” requires skipping over exactly 6 elements, because there are 6 elements per row. So this is a better way to think of the access `grid[1][4]` when working with a true 2D array:



A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R

Figure 15.5: A better model

So the reason the function parameter can’t leave the sizes unspecified is that if it didn’t know how many columns this 2D array had in it, it wouldn’t know how many spaces to jump over to get to the start of “row 1”.

The upshot of all of this is, if you’re writing modern C++ code, unless you are interacting with some old existing code or a C library or something that really really requires 2D arrays as such... probably just use some other data structure (like vectors of vectors) instead.

Chapter 16

Constructors and other methods

20230419-1815

If you continue to future CS courses, you will learn about the concept of *classes*—an extension of structs—and their use in a paradigm known as *object-oriented design*. In this chapter, we'll get a preview of some of those ideas, presenting the idea of a *constructor* and talking a little bit about how methods work and are implemented.

constructor

16.1 Setting up the data fields

We've seen in previous chapters that after a struct type is defined, an instance of a struct can be created using the typename, curly brackets, and initial values for the fields.

```
Location quadI = Location{3, 4};
Location origin = Location{0, 0};

// and in a function that returns a Location, maybe:
return Location{xval, yval};
```

Program fragment 16.1: Making some structs

What's really happening here? The typename indicates to the compiler what struct type is being used, so it can go look up what the fields are, and what types they have, so that when specific actual values are provided, the compiler can implement instructions that will store those values. For the `Location` type, it knows that the contents are two `int` *fields* named `x` and `y` respectively (see definition in Chapter 11), so it can accept two `int` *values* and use them to fill in the boxes, as shown:

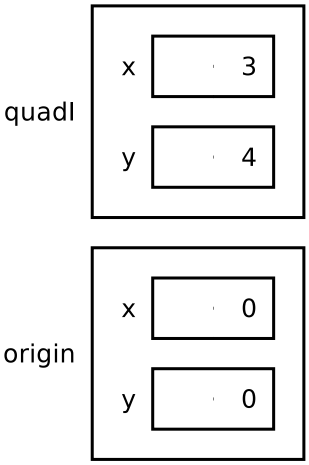


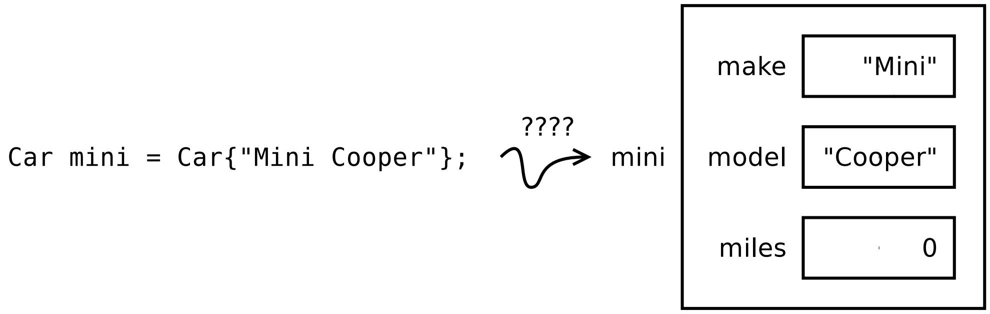
Figure 16.1: Memory model of some of the data in that code segment

But what if the way we acquired the data for a struct was going to usually be in some slightly different format? Consider the following struct definition:

```
struct Car
{
    string make;
    string model;
    int miles;
};
```

Program fragment 16.2: Another example struct: `Car`

Let's say that the general case for `Car` values would be that we were provided a single string with both the make and the model, like `"Mini Cooper"` or `"Jeep Grand Cherokee"`. It would certainly be possible to write a function that took a string like that and broke it up into make and model and returned a `Car` object, but what if we wanted to write something like what's on the left to produce this corresponding model:



```
Car mini = Car{"Mini Cooper"};
```

Figure 16.2: Intended data stored

We can make that happen! By including a *constructor*, a function-like definition *inside* the struct, we can write code to provide the initial values for every field in the struct. The main differences between a constructor and a regular function are:

constructor

- They are written[1] *inside* the struct definition
- The "function name" is the same as the struct typename
- There is no return type, not even `void`
- Between the header and the body of the constructor, each field is given its initial value (which may be further updated in the body)

So in this case, our constructor might be written as follows:

```
    Car (string make_model) :
        make{""}, model{""}, miles{0}
    {
        int spc = make_model.find(' ');
        make = make_model.substr(0, spc);
        model = make_model.substr(spc+1);
    }
```

Program fragment 16.3: The `Car` constructor, written *inside* the `Car` struct

Note that the fields are not re-declared inside the constructor: we are still inside the curly brackets of the overall struct, so the field definitions are "in scope" and we can refer to and update them.

[1] or at least declared; this chapter is an introduction to constructors and will present only a portion of how they can be written and used

Note that once *any* constructor is defined for a struct, C++ no longer provides the automatic ability to simply provide each field's value, so with only the above constructor definition, if we still wrote

```
Car used_mustang = Car{"Ford", "Mustang", 93205};
```

Program fragment 16.4: Another `Car` value

the compiler would reject that as an error. The good news is, we can provide as many distinct constructors as we want, as long as they are each have different number/type of parameters. In this case, we could add

```
    Car (string mk, string mod, int mls) :
        make{mk}, model{mod}, miles{mls}
    {
    }
```

Program fragment 16.5: Additional `Car` constructor

which, when I provide it with two strings and an int, behaves exactly like the automatic version: it sets the value of each field to the value of the provided arguments, in order, and then does nothing further to modify them. Many constructors will look roughly like this; the initialisers in the second line are able to make use of any of the provided named parameters, or compile-time constants, or some combination thereof, and the body of the constructor only needs to include additional code if there is additional computation to be done to set up the field values.

Putting that all together, here is the full struct definition:

```
struct Car
{
    string make;
    string model;
    int miles;

    Car (string make_model) :
        make{""}, model{""}, miles{0}
    {
        int spc = make_model.find(' ');
        make = make_model.substr(0, spc);
        model = make_model.substr(spc+1);
    }

    Car (string mk, string mod, int mls) :
        make{mk}, model{mod}, miles{mls}
    {
    }
};
```

Program fragment 16.6: The `Car` struct

And some examples that show how the different constructors get called:

```
  Car mini = Car{"Mini Cooper"};
  Car used_mustang = Car{"Ford", "Mustang", 93205};
```

Program fragment 16.7: Examples of `Car` values

16.2 A few future strands to explore

This is where we'll wrap up the course, but having introduced constructors, we can close with a few notes about what might come next:

- Once you start writing explicit constructors, you can call them as if they were functions, using parentheses instead of curly brackets: `Car("Jeep Grand Cherokee")`. Using the parentheses won't work if you're relying on the implicit constructors as we did with `Location` and other structs. (In pre-2011 versions of C++, parens were the *only* way to call a constructor, and there is still a lot of new code written in this older style.)

- In addition to constructors, you can write code that is even more function-like (with a return type, a different name, and no list of initial values for fields) inside the struct definition. In fact, it will look *exactly* like a function definition except that it can refer to the struct fields that are declared earlier in the struct. That's how you define your own *methods*, which you can then call on your own objects just like calling `.substr` or `.push_back` or the other methods we've learned for strings and vectors.

methods

- Once you get to writing your own methods, you open the possibility of data fields that get an initial value in the constructor, are modified and accessed by other methods, and that in fact you don't want to access *except* by the various methods. This is referred to as *private* data, and *access control* is an important concept in software engineering.

private

access control

- As the number of methods increases, it becomes increasingly unwieldy to put all their definitions in the `.h` file where the struct is defined; there is additional syntax that lets you merely *declare* the methods inside the struct definition, and then fully *define* the methods in a corresponding `.cpp` file, just as we've done for other kinds of functions.

- If you're doing all of that, you're probably ready to write your types as *classes* instead of structs, and the paradigm you're writing in would then be properly referred to as *object-oriented programming* or just *OOP*.

classes

object-oriented progr

OOP

That's all, folks!

Index