1

Working with strings

Chapter 0 looked closely at a tiny program, which we used to introduce surprisingly many fundamental C++ ideas: comments, standard headers, scopes, namespaces, expressions, statements, string literals, and output. This chapter continues our overview of the fundamentals by writing similarly simple programs that use character strings. In the process, we'll learn about declarations, variables, and initialization, as well as something about input and the C++ `string` library. The programs in this chapter are so simple that they do not even require any control structures, which we will cover in Chapter 2.

1.1 Input

Once we can write text, the logical next step is to read it. For example, we can modify the `Hello, world!` program to say hello to a specific person:

```
// ask for a person's name, and greet the person
#include <iostream>
#include <string>

int main()
{
    // ask for the person's name
    std::cout << "Please enter your first name: ";

    // read the name
    std::string name;      // define name
    std::cin >> name;      // read into name

    // write a greeting
    std::cout << "Hello, " << name  << "!" << std::endl;
    return 0;
}
```

When we execute this program, it will write

```
Please enter your first name:
```

on the standard output. If we respond, for example,

```
Vladimir
```

then the program will write

```
Hello, Vladimir!
```

Let's look at what's going on. In order to read input, we must have a place to put it. Such a place is called a ***variable***. A variable is an ***object*** that has a name. An object, in turn, is a part of the computer's memory that has a type. The distinction between objects and variables is important because, as we'll see in §3.2.2/45, §4.2.3/65, and §10.6.1/183, it is possible to have objects that do not have names.

If we wish to use a variable, we must tell the implementation what name to give it and what type we want it to have. The requirement to supply both a name and a type makes it easier for the implementation to generate efficient machine code for our programs. The requirement also lets the compiler detect misspelled variable names—unless the misspelling happens to match one of the names that our program said it intended to use.

In this example, our variable is named `name`, and its type is `std::string`. As we saw in §0.5/3 and §0.7/5, the use of `std::` implies that the name, `string`, that follows it is part of the standard library, not part of the core language or of a nonstandard library. As with every part of the standard library, `std::string` has an associated header, namely `<string>`, so we've added an appropriate `#include` directive to our program.

The first statement,

```
std::cout << "Please enter your first name: ";
```

should be familiar by now: It writes a message that asks for the user's name. An important part of this statement is what isn't there, namely the `std::endl` manipulator. Because we did not use `std::endl`, the output does not begin a new line after the program has written its message. Instead, as soon as it has written the prompt, the computer waits—on the same line—for input.

The next statement,

```
std::string name;      // define name
```

is a ***definition***, which defines our variable named `name` that has type `std::string`. Because this definition appears within a function body, `name` is a ***local variable***, which exists only while the part of the program within the braces is executing. As soon as the computer reaches the `}`, it ***destroys*** the variable `name`, and returns any memory that the variable occupied to the system for other uses. The limited lifetime of local variables is one reason that it is important to distinguish between variables and other objects.

Implicit in the type of an object is its ***interface***—the collection of operations that are possible on an object of that type. By defining `name` as a variable (a named object) of type `string`, we are implicitly saying that we want to be able to do with `name` whatever the library says that we can do with `strings`.

One of those operations is to ***initialize*** the `string`. Defining a `string` variable implicitly initializes it, because the standard library says that every `string` object starts out with a value. We shall see shortly that we can supply a value of our own when we create a `string`. If we do not do so, then the `string` starts out containing no characters at all. We call such a `string` an ***empty*** or ***null*** `string`.

Once we have defined `name`, we execute

```
std::cin >> name;      // read into name
```

which is a statement that reads from `std::cin` into `name`. Analogous with its use of the `<<` operator and `std::cout` for output, the library uses the **`>>` *operator*** and `std::cin` for input. In this example, `>>` reads a `string` from the standard input and stores what it read in the object named `name`. When we ask the library to read a `string`, it begins by discarding ***whitespace*** characters (space, tab, backspace, or the end of the line) from the input, then reads characters into `name` until it encounters another whitespace character or end-of-file. Therefore, the result of executing `std::cin >> name` is to read a word from the standard input, storing in `name` the characters that constitute the word.

The input operation has another side effect: It causes our prompt, which asks for the user's name, to appear on the computer's output device. In general, the input–output library saves its output in an internal data structure called a ***buffer***, which it uses to optimize output operations. Most systems take a significant amount of time to write characters to an output device, regardless of how many characters there are to write. To avoid the overhead of writing in response to each output request, the library uses the buffer to accumulate the characters to be written, and ***flushes*** the buffer, by writing its contents to the output device, only when necessary. By doing so, it can combine several output operations into a single write.

There are three events that cause the system to flush the buffer. First, the buffer might be full, in which case the library will flush it automatically. Second, the library might be asked to read from the standard input stream. In that case, the library immediately flushes the output buffer without waiting for the buffer to become full. The third occasion for flushing the buffer is when we explicitly say to do so.

When our program writes its prompt to `cout`, that output goes into the buffer associated with the standard output stream. Next, we attempt to read from `cin`. This read flushes the `cout` buffer, so we are assured that our user will see the prompt.

Our next statement, which generates the output, explicitly instructs the library to flush the buffer. That statement is only slightly more complicated than the one that wrote the prompt. Here we write the string literal `"Hello, "` followed by the value of the `string` variable `name`, and finally by `std::endl`. Writing the value of `std::endl` ends the line of output, and then flushes the buffer, which forces the system to write to the output stream immediately.

Flushing output buffers at opportune moments is an important habit when you are writing programs that might take a long time to run. Otherwise, some of the program's output might languish in the system's buffers for a long time between when your program writes it and when you see it.

1.2 Framing a name

So far, our program has been restrained in its greetings. We'd like to change that by writing a more elaborate greeting, so that the input and output look like this:

```
Please enter your first name: Estragon
**********************
*                    *
* Hello, Estragon! *
*                    *
**********************
```

Our program will produce five lines of output. The first line begins the frame. It is a sequence of `*` characters as long as the person's name, plus some characters to match the salutation (`"Hello, "`), plus a space and an `*` at each end. The line after that will be an appropriate number of spaces with an `*` at each end. The third line is an `*`, a space, the message, a space, and an `*`. The last two lines will be the same as the second and first lines, respectively.

A sensible strategy is to build up the output a piece at a time. First we'll read the name, then we'll use it to construct the greeting, and then we'll use the greeting to build each line of the output. Here is a program that uses that strategy to solve our problem:

```
// ask for a person's name, and generate a framed greeting
#include <iostream>
#include <string>

int main()
{
    std::cout << "Please enter your first name: ";
    std::string name;
    std::cin >> name;

    // build the message that we intend to write
    const std::string greeting = "Hello, " + name + "!";

    // build the second and fourth lines of the output
    const std::string spaces(greeting.size(), ' ');
    const std::string second = "* " + spaces + " *";

    // build the first and fifth lines of the output
    const std::string first(second.size(), '*');

    // write it all
    std::cout << std::endl;
    std::cout << first << std::endl;
    std::cout << second << std::endl;
    std::cout << "* " << greeting << " *" << std::endl;
    std::cout << second << std::endl;
    std::cout << first << std::endl;

    return 0;
}
```

First, our program asks for the user's name, and reads that name into a variable named `name`. Then, it defines a variable named `greeting` that contains the message that it intends to write. Next, it defines a variable named `spaces`, which contains as many spaces as the number of characters in `greeting`. It uses the `spaces` variable to define a variable named `second`, which will contain the second line of the output, and then the

program constructs `first` as a variable that contains as many `*` characters as the number of characters in `second`. Finally, it writes the output, a line at a time.

The `#include` directives and the first three statements in this program should be familiar. The definition of `greeting`, on the other hand, introduces three new ideas.

One idea is that we can give a variable a value as we define it. We do so by placing, between the variable's name and the semicolon that follows it, an = symbol followed by the value that we wish the variable to have. If the variable and value have different types—as §10.2/176 shows that `strings` and string literals do—the implementation will ***convert*** the initial value to the type of the variable.

The second new idea is that we can use `+` to ***concatenate*** a `string` and a string literal—or, for that matter, two `strings` (but not two string literals). We noted in passing in Chapter 0 that `3+4` is `7`. Here we have an example in which `+` means something completely different. In each case, we can determine what the `+` operator does by examining the types of its operands. When an operator has different meanings for operands of different types, we say that the operator is ***overloaded***.

The third idea is that of saying **`const`** as part of a variable's definition. Doing so promises that we are not going to change the value of the variable for the rest of its lifetime. Strictly speaking, this program gains nothing by using `const`. However, pointing out which variables will not change can make a program much easier to understand.

Note that if we say that a variable is `const`, we must initialize it then and there, because we won't have the opportunity later. Note also that the value that we use to initialize the `const` variable need not itself be a constant. In this example, we won't know the value of `greeting` until after we have read a value into `name`, which obviously can't happen until we run the program. For this reason, we cannot say that `name` is `const`, because we change its value by reading into it.

One property of an operator that never changes is its associativity. We learned in Chapter 0 that `<<` is left-associative, so that `std::cout << s << t` means the same as `(std::cout << s) << t`. Similarly, the `+` operator (and, for that matter, the `>>` operator) is also left-associative. Accordingly, the value of `"Hello, " + name + "!"` is the result of concatenating `"Hello, "` with `name`, and concatenating the result of that concatenation with `"!"`. So, for example, if the variable `name` contains `Estragon`, then the value of `"Hello, " + name + "!"` is `Hello, Estragon!`

At this point, we have figured out what we are going to say, and saved that information in the variable named `greeting`. Our next job is to build the frame that will enclose our greeting. In order to do so, we introduce three more ideas in a single statement:

```
std::string spaces(greeting.size(), ' ');
```

When we defined `greeting`, we used an = symbol to initialize it. Here, we are following `spaces` by two expressions, which are separated by a comma and enclosed in parentheses. When we use the = symbol, we are saying explicitly what value we would like the variable to have. By using parentheses in a definition, as we do here, we tell the implementation to ***construct*** the variable—in this case, `spaces`—from the expressions, in a way that depends on the type of the variable. In other words, in order to understand this definition, we must understand what it means to construct a `string` from two expressions.

How a variable is constructed depends entirely on its type. In this particular case, we are constructing a `string` from—well, from what? Both expressions are of forms that we haven't seen before. What do they mean?

The first expression, `greeting.size()`, is an example of calling a ***member function***. In effect, the object named `greeting` has a component named `size`, which turns out to be a function, and which we can therefore call to obtain a value. The variable `greeting` has type `std::string`, which is defined so that evaluating `greeting.size()` yields an integer that represents the number of characters in `greeting`.

The second expression, `' '`, is a ***character literal***. Character literals are completely distinct from string literals. A character literal is always enclosed in single quotes; a string literal is always enclosed in double quotes. The type of a character literal is the built-in type **`char`**; the type of a string literal is much more complicated, and we shall not explain it until §10.2/176. A character literal represents a single character. The characters that have special meaning inside a string literal have the same special meaning in a character literal. Thus, if we want `'` or `\`, we must precede it by `\`. For that matter, `'\n'`, `'\t'`, `'\"'`, and related forms work analogously to the way we saw in Chapter 0 that they work for string literals.

To complete our understanding of `spaces`, we need to know that when we construct a `string` from an integer value and a `char` value, the result has as many copies of the `char` value as the value of the integer. So, for example, if we were to define

```
std::string stars(10, '*');
```

then `stars.size()` would be `10`, and `stars` itself would contain `**********`.

Thus, `spaces` contains the same number of characters as `greeting`, but all of those characters are blanks.

Understanding the definition of `second` requires no new knowledge: We concatenate `"* "`, our string of spaces, and `" *"` to obtain the second line of our framed message. The definition of `first` requires no new knowledge either; it gives `first` a value that contains as many `*` characters as the number of characters in `second`.

The rest of the program should be familiar; all it does is write strings in the same way we did in §1.1/9.

1.3 Details

Types:

`char` Built-in type that holds ordinary characters as defined by the implementation.

`wchar_t` Built-in type intended to hold "wide characters," which are big enough to hold characters for languages such as Japanese.

The `string` type is defined in the standard header `<string>`. An object of type `string` contains a sequence of zero or more characters. If `n` is an integer, `c` is a `char`, `is` is an input stream, and `os` is an output stream, then the `string` operations include

`std::string s;`

Defines `s` as a variable of type `std::string` that is initially empty.

`std::string t = s;`
: Defines `t` as a variable of type `std::string` that initially contains a copy of the characters in `s`, where `s` can be either a `string` or a string literal.

`std::string z(n, c);`
: Defines `z` as a variable of type `std::string` that initially contains `n` copies of the character `c`. Here, `c` must be a `char`, not a `string` or a string literal.

`os << s`
: Writes the characters contained in `s`, without any formatting changes, on the output stream denoted by `os`. The result of the expression is `os`.

`is >> s`
: Reads and discards characters from the stream denoted by `is` until encountering a character that is not whitespace. Then reads successive characters from `is` into `s`, overwriting whatever value `s` might have had, until the next character read would be whitespace. The result is `is`.

`s + t`
: The result of this expression is an `std::string` that contains a copy of the characters in `s` followed by a copy of the characters in `t`. Either `s` or `t`, but not both, may be a string literal or a value of type `char`.

`s.size()`
: The number of characters in `s`.

Variables can be defined in one of three ways:

```
std::string hello = "Hello";      // define the variable with an explicit initial value

std::string stars(100, '*');      // construct the variable
                                  // according to its type and the given expressions

std::string name;                 // define the variable with an implicit initialization,
                                  // which depends on its type
```

Variables defined inside a pair of curly braces are local variables, which exist only while executing the part of the program within the braces. When the implementation reaches the `}`, it destroys the variables, and returns any memory that they occupied to the system.

Defining a variable as `const` promises that the variable's value will not change during its lifetime. Such a variable must be initialized as part of its definition, because there is no way to do so later.

Input: Executing `std::cin >> v` discards any whitespace characters in the standard input stream, then reads from the standard input into variable `v`. It returns `std::cin`, which has type `istream`, in order to allow chained input operations.

Exercises

1-0. Compile, execute, and test the programs in this chapter.

1-1. Are the following definitions valid? Why or why not?

```
const std::string hello = "Hello";
const std::string message = hello + ", world" + "!";
```

1-2. Are the following definitions valid? Why or why not?

```
const std::string exclam = "!";
const std::string message = "Hello" + ", world" + exclam;
```

1-3. Is the following program valid? If so, what does it do? If not, why not?

```
#include <iostream>
#include <string>

int main()
{
    { const std::string s = "a string";
      std::cout << s << std::endl; }

    { const std::string s = "another string";
      std::cout << s << std::endl; }
    return 0;
}
```

1-4. What about this one? What if we change `}}` to `};}` in the third line from the end?

```
#include <iostream>
#include <string>

int main()
{
    { const std::string s = "a string";
      std::cout << s << std::endl;
    { const std::string s = "another string";
      std::cout << s << std::endl; }}
    return 0;
}
```

1-5. Is this program valid? If so, what does it do? If not, say why not, and rewrite it to be valid.

```
#include <iostream>
#include <string>

int main()
{
    { std::string s = "a string";
    { std::string x = s + ", really";
      std::cout << s << std::endl; }
      std::cout << x << std::endl;
    }
    return 0;
}
```

1-6. What does the following program do if, when it asks you for input, you type two names (for example, `Samuel Beckett`)? Predict the behavior before running the program, then try it.

```
#include <iostream>
#include <string>

int main()
{
    std::cout << "What is your name? ";
    std::string name;
    std::cin >> name;
    std::cout << "Hello, " << name
              << std::endl << "And what is yours? ";
    std::cin >> name;
    std::cout << "Hello, " << name
              << "; nice to meet you too!" << std::endl;
    return 0;
}
```

2

Looping and counting

In §1.2/11, we developed a program that writes a formatted frame around a greeting. In this chapter, we're going to make the program more flexible so that we can change the size of the frame without rewriting the program.

Along the way, we'll start learning about arithmetic in C++, and how C++ supports loops and conditions, and we'll explore the related idea of loop invariants.

2.1 The problem

The program in §1.2/12 wrote a greeting with a frame around it. For example, if our user gave us the name `Estragon`, our program would write

```
**********************
*                    *
* Hello, Estragon! *
*                    *
**********************
```

The program built up the output a line at a time. It defined variables named `first` and `second` to contain the first and second lines of the output, and wrote the greeting itself, surrounded by some characters, as the third line. We didn't need separate variables for the fourth or fifth output lines, because those were the same as the second and first lines respectively.

This approach has a major shortcoming: Each line of the output has a part of the program—and a variable—that corresponds to it. Therefore, even a simple change to the output format, such as removing the spaces between the greeting and the frame, would require rewriting the program. We would like to produce a more flexible form of output without having to store each line in a local variable.

We will approach this problem by generating each character of the output separately, except for the greeting itself, which we already have available as a `string`. What we shall discover is that there is no need to store the output characters in variables, because once we have written a character, we don't need it any more.

2.2 Overall structure

We'll begin by reviewing the part of the program that we don't have to rewrite:

```
#include <iostream>
#include <string>

int main()
{
    // ask for the person's name
    std::cout << "Please enter your first name: ";

    // read the name
    std::string name;
    std::cin >> name;

    // build the message that we intend to write
    const std::string greeting = "Hello, " + name + "!";

    // we have to rewrite this part...

    return 0;
}
```

As we rewrite the part of the program that the *we have to rewrite this part...* comment represents, we shall already be in a context that defines `name`, `greeting`, and the relevant names from the standard library. We will build up the new version of the program a piece at a time, and then, in §2.5.4/29, we'll put all the pieces together.

2.3 Writing an unknown number of rows

We can think of our output as a rectangular array of characters, which we must write one row at a time. Although we don't know how many rows it has, we do know how to compute the number of rows.

The greeting takes up one row, as do the top and bottom rows of the frame. We've accounted for three rows so far. If we know how many blank rows we intend to leave between the greeting and the frame, we can double that number and add three to obtain the total number of rows in the output:

```
// the number of blanks surrounding the greeting
const int pad = 1;

// total number of rows to write
const int rows = pad * 2 + 3;
```

We want to make it easy to find the part of our program that defines the number of blanks, so we give that number a name. The variable called `pad` represents the amount of padding around the frame. Having defined `pad`, we use it in computing `rows`, which will control how many rows we write.

The built-in type `int` is the most natural type to use for integers, so we've chosen that type for `pad` and `rows`. We also said that both variables are `const`, which we know from §1.2/13 is a promise that we will not change the value of either `pad` or `rows`.

Looking ahead, we intend to use the same number of blanks on the left and right sides as on the top and bottom, so one variable will serve for all four sides. If we are careful to use this variable every time we want to refer to the number of blanks, changing the size of the frame will require only changing the program to give the variable a different value.

We have computed how many rows we need to write; our next problem is to do so:

```
// separate the output from the input
std::cout << std::endl;

// write rows rows of output
int r = 0;

// invariant: we have written r rows so far
while (r != rows) {
    // write a row of output (as we will describe in §2.4/22)
    std::cout << std::endl;
    ++r;
}
```

We start, as we did in §1.2/12, by writing a blank line, so that there will be some space between the input and the output. The rest of this fragment contains so many new ideas that we need to look at it closely. Once we've understood how it works, we'll think about how to write each individual row.

2.3.1 The `while` statement

Our program controls how many rows of output it writes by using a **`while`** ***statement***, which repeatedly executes a given statement as long as a given condition is true. A `while` statement has the form

```
while (condition)
        statement
```

The *statement* is often called the **`while`** ***body***.

The `while` statement begins by testing the value of the condition. If the condition is false, it does not execute the body at all. Otherwise, it executes the body once, after which it tests the condition again, and so on. The `while` alternates between testing the condition and executing the body until the condition is false, at which point execution continues after the end of the entire `while` statement.

Loosely speaking, we can think of the `while` statement in our example as saying, ''As long as the value of `r` is not equal to `rows`, do whatever is within the `{ }`.''

It is conventional to put the `while` body on a separate line and indent it, to make programs easier to read. The implementation doesn't stop us from writing

```
while (condition) statement
```

but if we do so, we should think about whether we might be making life harder for other people who might read our program.

Note that there is no semicolon after *statement* in this description. Either the *statement* is indeed just a statement, or it is a ***block***, which is a sequence of zero or more statements

enclosed in `{ }`. If the statement is just an ordinary statement, it will end with a semicolon of its own, so there's no need for another one. If it is a block, the block's `}` marks the end of the statement, so again there's no need for a semicolon. Because a block is a sequence of statements enclosed by braces, we know from §0.7/5 that a block is a scope.

The `while` begins by testing its ***condition***, which is an expression that appears in a context where a truth value is required. The expression `r != rows` is an example of a condition. This example uses the ***inequality operator***, `!=`, to compare `r` and `rows`. Such an expression has type **`bool`**, which is a built-in type that represents truth values. The two possible values of type `bool` are **`true`** and **`false`**, with the obvious meanings.

The other new facility in this program is the last statement in the `while` body, which is

```
++r;
```

The `++` is the ***increment*** operator, which has the effect of incrementing—adding `1` to—the variable `r`. We could have written

```
r = r + 1;
```

instead, but incrementing an object is so common that a special notation for doing so is useful. Moreover, as we shall see in §5.1.2/79, the idea of transforming a value into its immediate successor, in contrast with computing an arbitrary value, is so fundamental to abstract data structures that it deserves a special notation for that reason alone.

2.3.2 Designing a `while` statement

Determining exactly what condition to write in a `while` statement is sometimes difficult. Similarly, it can be hard to understand precisely what a particular `while` statement does. It is not too hard to see that the `while` statement in §2.3/19 will write a number of output rows that depends on the value of `rows`, but how can we be confident that we know exactly how many rows the program will write? For example, how do we know whether the number will be `rows`, `rows - 1`, `rows + 1`, or something else entirely? We could trace through the `while` by hand, noting the effect of each statement's execution on the state of the program, but how do we know that we haven't made a mistake along the way?

There is a useful technique for writing and understanding `while` statements that relies on two key ideas—one about the definition of a `while` statement, and the other about the behavior of programs in general.

The first idea is that when a `while` finishes, its condition must be `false`—otherwise the `while` wouldn't have finished. So, for example, when the `while` in §2.3/19 finishes, we know that `r != rows` is false and, therefore, that `r` is equal to `rows`.

The second idea is that of a ***loop invariant***, which is a property that we assert will be true about a `while` each time it is about to test its condition. We choose an invariant that we can use to convince ourselves that the program behaves as we intend, and we write the program so as to make the invariant true at the proper times. Although the invariant is not part of the program text, it is a valuable intellectual tool for designing programs. Every useful `while` statement that we can imagine has an invariant associated with it. Stating the invariant in a comment can make a `while` much easier to understand.

To make this discussion concrete, we will look again at the `while` statement in §2.3/19. The comment immediately before the `while` says what the invariant is: *We have written `r` rows of output so far*.

To determine that this invariant is correct for this program fragment, we must verify that the invariant is true each time the `while` is about to test its condition. Doing so requires us to verify that the invariant will be true at two specific points in the program.

The first point is just before the `while` tests its condition for the first time. It is easy to verify the invariant at this point in our example: Because we have written no rows of output so far, it is obvious that setting `r` to `0` makes the invariant true.

The second point is just before we reach the end of the `while` body. If the invariant is true there, it will be true the next time the `while` tests the condition. Therefore, the invariant will be true every time.

In exchange for writing our program so that it meets these two requirements—causing the invariant to be true before the `while` starts, and again at the end of the `while` body—we can be confident that the invariant is true not only each time the `while` tests the condition, but also after the `while` finishes. Otherwise, the invariant would have had to be true at the beginning of one of the iterations of the `while` body and false afterward—and we have already arranged for that to be impossible.

Here is a summary of what we know about our program fragment:

```
// invariant: we have written r rows so far
int r = 0;
// setting r to 0 makes the invariant true
while (r != rows) {
    // we can assume that the invariant is true here
    // writing a row of output makes the invariant false
    std::cout << std::endl;
    // incrementing r makes the invariant true again
    ++r;
}
// we can conclude that the invariant is true here
```

The invariant for our `while` is that we have written `r` rows of output so far. When we define `r`, we give it an initial value of `0`. At this point, we haven't written anything at all. Setting `r` to `0` obviously makes the invariant true, so we have met the first requirement.

To meet the second requirement, we must verify that whenever the invariant is true when the `while` is about to test its condition, a trip through the condition and body will leave the invariant true at the end of the body.

Writing a row of output causes the invariant to become false, because `r` is no longer the number of rows we have written. However, incrementing `r` to account for the row that was written will make the invariant true again. Doing so makes the invariant true at the end of the body, so we have met the second requirement.

Because both requirements are true, we know that after the `while` finishes, we have written `r` rows. Moreover, we have already seen that `r == rows`. Together, these two facts imply that `rows` is the total number of rows that we have written.

The strategy that we used to understand this loop will come in handy in a variety of contexts. The general idea is to find an invariant that states a relevant property of the variables that the loop involves (we have written `r` rows), and to use the condition to ensure that when the loop completes, those variables will have useful values (`r == rows`). The loop body's job is then to manipulate the relevant variables so as to arrange for the condition to be false eventually, while maintaining the truth of the invariant.

2.4 Writing a row

Now that we understand how to write a given number of rows, we can turn our attention to writing a single row. In other words, we can start filling in the part of the program represented by the *write a row of output* comment in §2.3/19.

We begin by observing that all the output lines are the same length. If we think of the output as a rectangular array, then that length is the number of columns in the array. We can compute that number by adding twice the padding to the length of the greeting, and then adding two for the asterisks at the ends:

```
const std::string::size_type cols = greeting.size() + pad * 2 + 2;
```

The easy part of reading this definition is to see that we've said that `cols` is `const`, thereby promising that the value of `cols` will not change after we have defined it. The harder part to understand is that we have defined `cols` using an unfamiliar type, namely `std::string::size_type`. We know that the first `::` is the scope operator and that the qualified name `std::string` means the name `string` from the namespace `std`. The second `::` similarly says that we want the name `size_type` from the class `string`. Like namespaces and blocks, classes define their own scopes. The `std::string` type defines `size_type` to be the name of the appropriate type for holding the number of characters in a `string`. Whenever we need a local variable to contain the size of a `string`, we should use `std::string::size_type` as the type of that variable.

The reason that we have given `cols` a type of `std::string::size_type` is to ensure that `cols` is capable of containing the number of characters in `greeting`, no matter how large that number might be. We could simply have said that `cols` has type `int`, and indeed, doing so would probably work. However, the value of `cols` depends on the size of the input to our program, and we have no control over how long that input might be. It is conceivable that someone might give our program a string so long that an `int` is insufficient to contain its length.

The `int` type is sufficient for `rows` because the number of rows depends only on the value of `pad`, which we control. Every C++ implementation is required to allow every `int` variable to take on values up to at least `32767`, which is plenty. Nevertheless, whenever we define a variable that contains the size of a particular data structure, it is a good habit to use the type that the library defines as being appropriate for that specific purpose.

It is impossible for a string to contain a negative number of characters. Accordingly, `std::string::size_type` is an ***unsigned*** type—objects of that type are incapable of containing negative values. This property does not affect the programs in this chapter, but we shall see later on in §8.1.3/142 that it can be critically important.

Having figured out how many characters to write, we can use another `while` statement to write them:

```
std::string::size_type c = 0;

// invariant: we have written c characters so far in the current row
while (c != cols) {
    // write one or more characters
    // adjust the value of c to maintain the invariant
}
```

This `while` behaves analogously to the one in §2.3/19, except for one difference in the body: This time we have said *write one or more characters* instead of writing exactly one row as we did in §2.3/19. There is no reason that we have to write only a single character each time through the body. As long as we write at least one character, we will ensure progress. All we have to do is ensure that the total number of characters we write on this row is exactly `cols`.

2.4.1 Writing border characters

Our remaining problem is to figure out what characters to write. We can solve part of this problem by noting that if we are on the first or last row, or on the first or last column, then we know that we should write an asterisk. Moreover, we can use our knowledge of the loop invariants to determine whether it is time to write an asterisk.

For example, if `r` is zero, we know from the invariant that we have not yet written any rows, which means that we are writing part of the first row. Similarly, if `r` is equal to `rows - 1`, we know that we have written `rows - 1` rows already, so we must now be writing part of the last row. We can use analogous reasoning to conclude that if `c` is zero, then we are writing part of the first column, and if `c` is equal to `cols - 1`, we are writing part of the last column. Using this knowledge, we can fill in more of our program:

```
// invariant: we have written c characters so far in the current row
while (c != cols) {
    if (r == 0 || r == rows - 1 || c == 0 || c == cols - 1) {
        std::cout << "*";
        ++c;
    } else {
        // write one or more nonborder characters
        // adjust the value of c to maintain the invariant
    }
}
```

This statement introduces so many new ideas that it requires detailed explanation.

2.4.1.1 `if` statements

The `while` body consists of a block (§2.3.1/19) that contains an **`if`** ***statement***, which we use to determine whether it is time to write an asterisk. An `if` statement can take either of two forms:

```
if (condition)
        statement
```

or, as used here,

```
if (condition)
        statement1
else
        statement2
```

As with the `while` statement, the *condition* is an expression that yields a truth value. If the condition is true, then the program executes the statement that follows the `if`. In the second form of the `if` statement, if the *condition* is false, then the program executes the statement that follows the `else`.

It is worth noting, just as with our description of the form of the `while` statement, that the formatting that we use to illustrate the `if` statement is merely conventional. However, readers will find it much easier if code follows formatting conventions such as the ones that we've used in the examples in this book.

2.4.1.2 Logical operators

What about the condition itself?

```
r == 0 || r == rows - 1 || c == 0 || c == cols - 1
```

This condition is true if `r` is `0` or `rows - 1`, or if `c` is `0` or `cols - 1`. The condition uses two new operators, the `==` operator and the `||` operator. C++ programs test for ***equality*** by using the `==` symbol, to distinguish it from the assignment operator `=`. Thus, `r == 0` yields a `bool` that indicates whether the value of `r` is equal to `0`. The ***logical-or*** operator, written as `||`, yields `true` if either of its operands is `true`.

The relational operators have lower ***precedence*** than the arithmetic operators. In expressions that contain more than one operator, precedence defines how the operands group. For example,

```
r == rows - 1
```

means

```
r == (rows - 1)
```

rather than

```
(r == rows) - 1
```

because the arithmetic operator `-` has higher precedence than the relational operator `==`. In other words, we are subtracting `1` from `rows` and comparing the result with `r`, which, in this program, is what we wanted.

We can override precedence by enclosing in parentheses a subexpression that we want to use as a single operand. For example, `(r == rows) - 1`, with its parentheses, compares `r` with `rows` to yield a `bool` that indicates whether `r` is equal to `rows`, and subtracts `1`

from that `bool`. A `bool` used as a number behaves as `1` if it is `true` and `0` if it is `false`. Therefore, `(r == rows) - 1` yields `0` if `r` is equal to `rows` and `-1` otherwise.

The logical-or operator tests whether *either* of its operands is `true`. Its form is

condition1 `||` *condition2*

where, as usual, *condition1* and *condition2* are conditions—expressions that yield truth values. The `||` expression yields a `bool`, which is `true` if either of the conditions is `true`.

The `||` operator has lower precedence than the relational operators, and, like most C++ binary operators, is left-associative. Moreover, it has a property that most other C++ operators do not share: If a program finds that the left operand of `||` is true, it does not evaluate the right operand at all. This property is often called ***short-circuit evaluation***, and as we shall see in §5.6/89, it can have a crucial effect on how we write our programs.

Because `||` is left-associative, and because of the relative precedence of `||`, `==`, and `-`,

```
r == 0 || r == rows - 1 || c == 0 || c == cols - 1
```

means the same as it would if we were to place all of its subexpressions in parentheses:

```
((r == 0 || r == (rows - 1)) || c == 0) || c == (cols - 1)
```

In order to evaluate this latter expression using the short-circuit strategy, the program first evaluates the left operand of the outermost `||`, which is

```
(r == 0 || r == (rows - 1)) || c == 0
```

To do so, it must first evaluate the left operand of this inner `||`, which is

```
r == 0 || r == (rows - 1)
```

which, in turn, means evaluating

```
r == 0
```

If `r` is equal to `0`, then each of the expressions

```
r == 0 || r == (rows - 1)
(r == 0 || r == (rows - 1)) || c == 0
((r == 0 || r == (rows - 1)) || c == 0) || c == (cols - 1)
```

must be true. If `r` is nonzero, the next step is to compare `r` with `rows - 1`. If that test fails, then the program will compare `c` with zero, and if that fails, it will compare `c` with `cols - 1` to determine the final result.

In other words, when we write a series of conditions separated by `||` operators, we are asking the program to test each of these conditions in turn. If any of the inner conditions is `true`, then the whole condition is `true`; otherwise, the whole condition is `false`. Each `||` operator stops as soon as it can determine its result, so if any of the inner conditions is `true`, the subsequent conditions go untested. If we step back from the details, we should be able to see that these four equality tests are checking whether we are in the first row, the last row, the first column, or the last column, and, therefore, that the `if` statement writes an asterisk if we're in the top or bottom row, or if we're in the first or last column. Otherwise, it does something else, which we must now define.

2.4.2 Writing nonborder characters

It is now time to write the statements that correspond to the comments that say

```
// write one or more nonborder characters
// adjust the value of c to maintain the invariant
```

in the program fragment in §2.4.1/23. These statements must deal with the characters that are not part of the border. It should be easy to see that each of these characters is either a space or part of the greeting. The only problem is figuring out which one it is, and what to do about it.

We begin by testing whether we are about to write the first character of the greeting, which we do by finding if we're in the correct row and on the correct column within that row. The row we seek is the one after we've written the initial row of asterisks, followed by `pad` additional rows. The appropriate column comes after we have written the initial asterisk on this row, followed by `pad` spaces. Our knowledge of the invariants tells us that we're on the right row when `r` is equal to `pad + 1`, and be at the appropriate column when `c` is equal to `pad + 1`.

In other words, to determine whether we are about to write the first character of the greeting, we must check whether `r` and `c` are both equal to `pad + 1`. If we've reached the right place to write the greeting, we'll do so; otherwise, we'll write a space instead. In both cases, we have to remember to update `c` appropriately:

```
if (r == pad + 1 && c == pad + 1) {
    std::cout << greeting;
    c += greeting.size();
} else {
    std::cout << " ";
    ++c;
}
```

The condition inside the `if` statement uses the ***logical-and*** operator. As with the `||` operator, the `&&` operator tests two conditions and yields a truth value. It is left-associative and uses a short-circuit evaluation strategy. Unlike the `||` operator, the `&&` operator yields `true` only if *both* conditions are `true`. If either condition is `false`, the result of `&&` is `false`. The second condition will be tested if and only if the first condition is `true`.

If the test succeeds, then it's time to write the greeting. In doing so, we falsify our invariant, because `c` is no longer equal to the number of characters we have written on this row. We make our invariant true again by adjusting the value of `c` to account for the characters that we have written. The expression that updates `c` uses another new operator, called the ***compound-assignment*** operator, to adjust `c` to account for the number of characters in the name when we wrote it. Such a compound assignment is a shorthand way of adding the right- and left-hand sides together and storing the result in the left-hand side. In other words, if we write `c += greeting.size()`, that statement has the same effect as if we had written `c = c + greeting.size()`.

The remaining possibility is that we're not on the border, and we're not about to write the greeting. In that case, we need to write a space and increment `c` to make the invariant true again, which we do in the `else` branch of the `if` statement.

2.5 The complete framing program

At this point, we have revised the entire program, but the code is scattered enough to be hard to find. Therefore, we shall show the whole program again. However, before we do so, we want to shorten the program in three ways.

The first abbreviation will be a kind of declaration that lets us say once and for all that a given name comes from the standard library. Doing so will allow us to avoid saying `std::` in so many places. The second abbreviation is a shorthand way of writing a particularly common kind of `while` statement. Finally, we can shorten the program slightly by incrementing `c` in one place instead of two.

2.5.1 Abbreviating repeated uses of `std::`

By now, you are probably tired of seeing—and writing—`std::` in front of every name from the standard library. Saying `std::` explicitly was a good way of reminding you which names came from the standard library, but you should have a pretty good idea of what they are at this point.

C++ offers a way of saying that a particular name should always be interpreted as coming from a particular namespace. For example, by writing

```
using std::cout;
```

we can say that we intend to use the name `cout` to mean `std::cout` exclusively, and that we do not intend to define anything named `cout` ourselves. Once we have done so, we can say `cout` instead of `std::cout`.

Logically enough, such a declaration is called a **`using`-*declaration***. The name that it mentions behaves similarly to other names. For example, if a `using`-declaration appears within braces, the name that it defines has its given meaning from where it is defined to the closing brace.

From now on, we'll write `using`-declarations to shorten our programs.

2.5.2 Using `for` statements for compactness

Let's look again at the control structures that we used in the program in §2.3/19. If we look only at the program's outermost structure, we see

```
int r = 0;

while (r != rows) {
    // stuff that doesn't change the value of r
    ++r;
}
```

This particular form of `while` appears frequently. Before it starts, we define and initialize a local variable, which we test in the condition. The `while` body adjusts the value of the variable so that eventually the condition will fail. Because this kind of control structure is so common, the language provides a shorthand way of writing it:

```
for (int r = 0; r != rows; ++r) {
    // stuff that doesn't change the value of r
}
```

In the body of each of these loops, `r` will take on a sequence of values, the first of which is `0`, and the last of which is `rows - 1`. We can think of `0` as being the beginning of a range, and `rows` as being the ***off-the-end value*** for the range. Such a range is called a ***half-open range***, and is often written as `[`*begin*`, `*off-the-end*`)`. The deliberately unbalanced brackets `[ )` remind the reader that the range is asymmetric. So, for example, the range `[1, 4)` contains `1`, `2`, and `3`, but not `4`. Similarly, we say that `r` takes on the values in `[0, rows)`.

A **`for`** ***statement*** has the following form:

```
for (init-statement   condition; expression)
        statement
```

The first line is often known as the ***for header***. It controls the statement that follows, which is often called the ***for body***. The *init-statement* must be either a definition (§1.1/10) or an expression statement (§0.8/6). Because each of these kinds of statement ends with its own semicolon, there is no additional semicolon between the *init-statement* and the *condition*.

A `for` statement begins by executing the *init-statement* part of the `for` header, which it does only once, at the beginning of the `for`. Typically, the *init-statement* defines and initializes the loop control variable, which will be tested as part of the *condition*. If a variable is defined in the *init-statement*, it is destroyed on exit from the `for`, so it is inaccessible to code that follows the `for` statement.

On every trip through the loop, including the first, the program evaluates the *condition*. If the condition yields `true`, then it executes the `for` body. Having done so, it executes the *expression*. It then repeats the test, continuing to execute the `for` body followed by the *expression* in the `for` header until the test condition fails.

More generally, the meaning of a `for` statement is

```
{
    init-statement
    while (condition) {
        statement
        expression;
    }
}
```

where we have been careful to enclose the *init-statement* and the `while` in extra braces, thereby limiting the lifetime of any variables declared in the *init-statement*. Note particularly the presence and absence of semicolons. We do not write a semicolon after the *init-statement* or *statement* because they are statements, with their own semicolons if they need them. We do include a semicolon after *expression* in order to turn it into a statement.

2.5.3 Collapsing tests

We can divide the code associated with the *write one or more characters* comment in §2.4/23 into three cases: We are writing a single asterisk, a space, or the entire greeting. As our program stands, we adjust `c` to maintain our invariant after we write an asterisk, and we

adjust it again after we write a space. There's nothing wrong with doing so, but it is often possible to change the order of tests in a program so as to make it possible to merge two or more identical statements into one.

Because our three cases are mutually exclusive, we can test them in any order. If we begin by first testing whether we are about to write the greeting, then we know that in the other two cases, incrementing `c` suffices to maintain the invariant, so we can collapse the two increments into one:

```
if (we are about to write the greeting) {
    cout << greeting;
    c += greeting.size();
} else {
    if (we are in the border)
        cout << "*";
    else
        cout << " ";
    ++c;
}
```

After collapsing the increments, we also find that two of our blocks are just single statements, so we can drop two pairs of braces. Notice how the different indentation of `++c;` draws attention to the fact that it is executed regardless of whether we are in the border.

2.5.4 The complete framing program

If we put all the pieces together and use these three abbreviation techniques, we get the following program:

```
#include <iostream>
#include <string>

// say what standard-library names we use
using std::cin;          using std::endl;
using std::cout;         using std::string;

int main()
{
    // ask for the person's name
    cout << "Please enter your first name: ";

    // read the name
    string name;
    cin >> name;

    // build the message that we intend to write
    const string greeting = "Hello, " + name + "!";

    // the number of blanks surrounding the greeting
    const int pad = 1;

    // the number of rows and columns to write
    const int rows = pad * 2 + 3;
    const string::size_type cols = greeting.size() + pad * 2 + 2;
```

```
    // write a blank line to separate the output from the input
    cout << endl;

    // write rows rows of output
    // invariant: we have written r rows so far
    for (int r = 0; r != rows; ++r) {

        string::size_type c = 0;

        // invariant: we have written c characters so far in the current row
        while (c != cols) {

            // is it time to write the greeting?
            if (r == pad + 1 && c == pad + 1) {
                cout << greeting;
                c += greeting.size();
            } else {

                // are we on the border?
                if (r == 0 || r == rows - 1 ||
                    c == 0 || c == cols - 1)
                    cout << "*";
                else
                    cout << " ";
                ++c;
            }
        }

        cout << endl;
    }

    return 0;
}
```

2.6 Counting

Most experienced C++ programmers have a habit that may seem weird at first: Their programs invariably begin counting from `0` rather than from `1`. For example, if we reduce the outer `for` loop of the program above to its essentials, we get

```
for (int r = 0; r != rows; ++r) {
    // write a row
}
```

We could have written this loop as

```
for (int r = 1; r <= rows; ++r) {
    // write a row
}
```

One version counts from `0` and uses `!=` as its comparison; the other counts from `1` and uses `<=` as its comparison. The number of iterations is the same in each case. Is there any reason to prefer one form over the other?

One reason to count from `0` is that doing so encourages us to use asymmetric ranges to express intervals. For example, it is natural to use the range `[0, rows)` to describe the first `for` statement, as it is to use the range `[1, rows]` to describe the second one.

Asymmetric ranges are usually easier to use than symmetric ones because of an important property: A range of the form `[m, n)` has `n - m` elements, and a range of the form `[m, n]` has `n - m + 1` elements. So, for example, the number of elements in `[0, rows)` is obvious (i.e., `rows - 0`, or `rows`) but the number in `[1, rows]` is less so.

This behavioral difference between asymmetric and symmetric ranges is particularly evident in the case of empty ranges: If we use asymmetric ranges, we can express an empty range as `[n, n)`, in contrast to `[n, n-1]` for symmetric ranges. The possibility that the end of a range could ever be less than the beginning can cause no end of trouble in designing programs.

Another reason to count from `0` is that doing so makes loop invariants easier to express. In our example, counting from `0` makes the invariant straightforward: We have written `r` rows of output so far. What would be the invariant if we counted from `1`?

One would be tempted to say that the invariant is that we are about to write the `r`th row, but that statement does not qualify as an invariant. The reason is that the last time the `while` tests its condition, `r` is equal to `rows + 1`, and we intend to write only `rows` rows. Therefore, we are *not* about to write the `r`th row, so the invariant is not true!

Our invariant could be that we have written `r - 1` rows so far. However, if that's our invariant, why not simplify it by starting `r` at `0`?

Another reason to count from `0` is that we have the option of using `!=` as our comparison instead of `<=`. This distinction may seem trivial, but it affects what we know about the state of the program when a loop finishes. For example, if the condition is `r != rows`, then when the loop finishes, we know that `r == rows`. Because the invariant says that we have written `r` rows of output, we know that we have written exactly `rows` rows all told. On the other hand, if the condition is `r <= rows`, then all we can prove is that we have written *at least* `rows` rows of output. For all we know, we might have written more.

If we count from `0`, then we can use `r != rows` as a condition when we want to ensure that there are exactly `rows` iterations, or we can use `r < rows` if we care only that the number of iterations is `rows` or more. If we count from `1`, we can use `r <= rows` if we want at least `rows` iterations—but what if we want to ensure that `rows` is the exact number? Then we must test a more complicated condition, such as `r == rows + 1`. This extra complexity offers no compensating advantage.

2.7 Details

Expressions: C++ inherits a rich set of operators from C, several of which we have already used. In addition, as we've already seen with the input and output operators, C++ programs can extend the core language by defining what it means to apply built-in operators to objects of class type. Correctly understanding complicated expressions is a fundamental prerequisite to effective programming in C++. Understanding such expressions requires understanding

- How the operands group, which is controlled by the precedence and associativity of the operators used in the expression
- How the operands will be converted to other types, if at all
- The order in which the operands are evaluated

Different operators have different precedence. Most operators are left-associative, but the assignment and conditional operators, and the operators that take a single argument, are right-associative. We list the most common operators here—regardless of whether we've used them in this chapter. We've ordered them by precedence from highest to lowest, with a double line separating groupings with the same precedence.

`x.y`	The member `y` of object `x`
`x[y]`	The element in object `x` indexed by `y`
`x++`	Increments `x`, returning the original value of `x`
`x--`	Decrements `x`, returning the original value of `x`
`++x`	Increments `x`, returning the incremented value
`--x`	Decrements `x`, returning the decremented value
`!x`	Logical negation. If `x` is `true` then `!x` is `false`.
`x * y`	Product of `x` and `y`
`x / y`	Quotient of `x` and `y`. If both operands are integral, so is the result; the implementation chooses whether to round toward zero or $-\infty$.
`x % y`	Remainder; equivalent to `x - ((x / y) * y)`. `x` and `y` must be integral.
`x + y`	Sum of `x` and `y`
`x - y`	Result of subtracting `y` from `x`
`x >> y`	For integral `x` and `y`, `x` shifted right by `y` bits; `y` must be non-negative. If `x` is an `istream`, reads from `x` into `y`.
`x << y`	For integral `x` and `y`, `x` shifted left by `y` bits; `y` must be non-negative. If `x` is an `ostream`, writes `y` onto `x`.
`x` *relop* `y`	Relational operators yield a `bool` indicating the truth of the relation. The operators (<, >, <=, and >=) have their obvious meanings.
`x == y`	Yields a `bool` indicating whether `x` equals `y`
`x != y`	Yields a `bool` indicating whether `x` is not equal to `y`
`x && y`	Yields a `bool` indicating whether both `x` and `y` are `true`. Evaluates `y` only if `x` is `true`.
`x || y`	Yields a `bool` indicating whether either `x` or `y` is `true`. Evaluates `y` only if `x` is `false`.
`x = y`	Assign the value of `y` to `x`, yielding `x` as its result.
`x` *op*`= y`	Compound assignment operators; equivalent to `x = x` *op* `y`, where *op* is one of `+ - * / % & | ^ << >>`
`x ? y : z`	Yields `y` if `x` is `true`; `z` otherwise. Evaluates only one of `y` and `z`.

There is usually no guarantee as to the order in which an expression's operands are evaluated. Accordingly, it is important to avoid writing a single expression in which one operand depends on the value of another operand. We'll see an example in §4.1.5/60.

Operands will be converted to the appropriate type when possible. Numeric operands in expressions or relational expressions are converted by the ***usual arithmetic conversions*** described in detail in §A.2.4.4/304. Basically, the usual arithmetic conversions attempt to preserve precision. Smaller types are converted to larger types, and signed types are converted to unsigned. Arithmetic values may be converted to `bool`: A value of `0` is considered `false`; any other value is `true`. Operands of class type are converted as specified by the type. We'll see in Chapter 12 how to control such conversions.

Types:

`bool` Built-in type representing truth values; may be either `true` or `false`

`unsigned` Integral type that contains only non-negative values

`short` Integral type that must hold at least 16 bits

`long` Integral type that must hold at least 32 bits

`size_t` Unsigned integral type (from `<cstddef>`) that can hold any object's size

`string::size_type` Unsigned integral type that can hold the size of any `string`

Half-open ranges include one but not both of their endpoints. For example, `[1, 3)` includes `1` and `2`, but not `3`.

Condition: An expression that yields a truth value. Arithmetic values used in conditions are converted to `bool`: Nonzero values convert to `true`; zero values convert to `false`.

Statements:

`using` *namespace-name*`::`*name*`;`
: Defines *name* as a synonym for *namespace-name*`::`*name*.

type-name name`;`
: Defines *name* with type *type-name*.

type-name name `=` *value*`;`
: Defines *name* with type *type-name* initialized as a copy of *value*.

type-name name`(`*args*`);`
: Defines *name* with type *type-name* constructed as appropriate for the given arguments in *args*.

expression`;`
: Executes *expression* for its side effects.

`{` *statement(s)* `}`
: Called a block. Executes the sequence of zero or more *statement(s)* in order. May be used wherever a *statement* is expected. Variables defined inside the braces have scope limited to the block.

`while (`*condition*`)` *statement*
: If *condition* is `false`, do nothing; otherwise, execute *statement* and then repeat the entire `while`.

`for (`*init-statement condition*`;` *expression*`)` *statement*
: Equivalent to `{` *init-statement* `while (`*condition*`) {` *statement expression*`; } }` (unless the *statement* contains—or is—a `continue` statement (§A.4/309)).

`if (`*condition*`)` *statement*
: Executes *statement* if *condition* is `true`.

`if (`*condition*`)` *statement* `else` *statement2*
: Executes *statement* if *condition* is `true`, otherwise executes *statement2*.
Each `else` is associated with the nearest matching `if`.

`return` *val*`;`
: Exits the function and returns `val` to its caller.

Exercises

2-0. Compile and run the program presented in this chapter.

2-1. Change the framing program so that it writes its greeting with no separation from the frame.

2-2. Change the framing program so that it uses a different amount of space to separate the sides from the greeting than it uses to separate the top and bottom borders from the greeting.

2-3. Rewrite the framing program to ask the user to supply the amount of spacing to leave between the frame and the greeting.

2-4. The framing program writes the mostly blank lines that separate the borders from the greeting one character at a time. Change the program so that it writes all the spaces needed in a single output expression.

2-5. Write a set of ''*'' characters so that they form a square, a rectangle, and a triangle.

2-6. What does the following code do?

```
int i = 0;
while (i < 10) {
    i += 1;
    std::cout << i << std::endl;
}
```

2-7. Write a program to count down from `10` to `-5`.

2-8. Write a program to generate the product of the numbers in the range `[1, 10)`.

2-9. Write a program that asks the user to enter two numbers and tells the user which number is larger than the other.

2-10. Explain each of the uses of `std::` in the following program:

```
int main()
{
    int k = 0;
    while (k != 10) {              // invariant: we have written k asterisks so far
        using std::cout;
        cout << "*";
        ++k;
    }
    std::cout << std::endl;      // std:: is required here
    return 0;
}
```

3

Working with batches of data

The programs that we explored in Chapters 1 and 2 did little more than read a single string and write it again, sometimes with decoration. Most problems are more complicated than such simple programs can solve. Among the most common sources of complexity in programs is the need to handle multiple pieces of similar data.

Our programs have already started doing so, in the sense that a `string` comprises multiple characters. Indeed, it is exactly the ability to put an unknown number of characters into a single object—a `string`—that makes these programs easy to write.

In this chapter, we'll look at more ways of dealing with batches of data, by writing a program that reads a student's exam and homework grades and computes a final grade. Along the way, we'll learn how to store all the grades, even if we don't know in advance how many grades there are.

3.1 Computing student grades

Imagine a course in which each student's final exam counts for 40% of the final grade, the midterm exam counts for 20%, and the average homework grade makes up the remaining 40%. Here is our first try at a program that helps students compute their final grades:

```
#include <iomanip>
#include <ios>
#include <iostream>
#include <string>

using std::cin;                    using std::setprecision;
using std::cout;                   using std::string;
using std::endl;                   using std::streamsize;

int main()
{
    // ask for and read the student's name
    cout << "Please enter your first name: ";
    string name;
    cin >> name;
    cout << "Hello, " << name << "!" << endl;
```

```
    // ask for and read the midterm and final grades
    cout << "Please enter your midterm and final exam grades: ";
    double midterm, final;
    cin >> midterm >> final;

    // ask for the homework grades
    cout << "Enter all your homework grades, "
            "followed by end-of-file: ";

    // the number and sum of grades read so far
    int count = 0;
    double sum = 0;

    // a variable into which to read
    double x;

    // invariant:
    //   we have read count grades so far, and
    //   sum is the sum of the first count grades
    while (cin >> x) {
        ++count;
        sum += x;
    }

    // write the result
    streamsize prec = cout.precision();
    cout << "Your final grade is " << setprecision(3)
         << 0.2 * midterm + 0.4 * final + 0.4 * sum / count
         << setprecision(prec) << endl;

    return 0;
}
```

As usual, we begin with `#include` directives and `using`-declarations for the library facilities that we intend to use. These facilities include `<iomanip>` and `<ios>`, which we have not yet seen. The `<ios>` header defines `streamsize`, which is the type that the input–output library uses to represent sizes. The `<iomanip>` header defines the manipulator `setprecision`, which lets us say how many significant digits we want our output to contain.

When we used `endl`, which is also a manipulator, we did not have to include the `<iomanip>` header. The `endl` manipulator is used so often that its definition appears in `<iostream>`, rather than in `<iomanip>`.

The program begins by asking for and reading the student's name, and the midterm and final grades. Next, it asks for the student's homework grades, which it continues to read until it encounters an end-of-file signal. Different C++ implementations offer their users different ways of sending such a signal to a program, the most common way being to begin a new line of input, hold down the *control* key, and press `z` (for computers running Microsoft Windows) or `d` (for computers running the Unix or Linux systems).

While reading the grades, the program uses `count` to keep track of how many grades were entered, and stores in `sum` a running total of the grades. Once the program has read all the grades, it writes a greeting message, and reports the student's final grade. In doing so, it uses `count` and `sum` to compute the average homework grade.

Much of this program should already be familiar, but there are several new usages, which we will explain.

The first new idea occurs in the section that reads the student's exam grades:

```
cout << "Enter your midterm and final exam grades: ";
double midterm, final;
cin >> midterm >> final;
```

The first of these statements should be familiar: It writes a message, which, in this case, tells the student what to do next. The next statement defines `midterm` and `final` as having type `double`, which is the built-in type for double-precision floating-point numbers. There is also a single-precision floating-point type, called `float`. Even though it might seem that `float` is the appropriate type, it is almost always right to use `double` for floating-point computations.

These types' names date back to when memory was much more expensive than it is today. The shorter floating-point type, called `float`, is permitted to offer as little precision as six significant (decimal) digits or so, which is not even enough to represent the price of a house to the nearest penny. The `double` type is guaranteed to offer at least ten significant digits, and we know of no implementation that does not offer at least 15 significant digits. On modern computers, `double` is usually much more accurate than `float`, and not much slower. Sometimes, `double` is even faster.

Now that we have defined the `midterm` and `final` variables, we read values into them. Like the output operator (§0.7/4), the input operator returns its left operand as its result. So, we can chain input operations just as we chain output operations, so

```
cin >> midterm >> final;
```

has the same effect as

```
cin >> midterm;
cin >> final;
```

Either form reads a number from the standard input into `midterm`, and the next number into `final`.

The next statement asks the student to enter homework grades:

```
cout << "Enter all your homework grades, "
        "followed by end-of-file: ";
```

A careful reading will reveal that this statement contains only a single `<<`, even though it seems to be writing two string literals. We can get away with this because two or more string literals in a program, separated only by whitespace, are automatically concatenated. Therefore, this statement has exactly the same effect as

```
cout << "Enter all your homework grades, followed by end-of-file: ";
```

By breaking the string literal in two, we avoid lines in our programs that are too long to read conveniently.

The next section of the code defines the variables that we'll use to hold the information that we intend to read. Of these, the interesting part is

```
int count = 0;
double sum = 0;
```

Note that we give the initial value `0` to both `sum` and `count`. The value `0` has type `int`, which means that the implementation must convert it to type `double` in order to use it to initialize `sum`. We could have avoided this conversion by initializing `sum` to `0.0` instead of `0`, but it makes no practical difference in this context: Any competent implementation will do the conversion during compilation, so there is no run-time overhead, and the result will be exactly the same.

In this case, what's more important than the conversion is that we give these variables an initial value at all. When we do not specify an initial value for a variable we are implicitly relying on ***default-initialization***. The initialization that happens by default depends on the type of the variable. For objects of class type, the class itself says what initializer to use if there is not one specified. For example, we noted in §1.1/10 that if we do not explicitly initialize a `string`, then the `string` is implicitly initialized to be empty. No such implicit initialization happens for local variables of built-in type.

Local variables of built-in type that are not explicitly initialized are ***undefined***, which means that the variable's value consists of whatever random garbage already happens to occupy the memory in which the variable is created. It is illegal to do anything to an undefined value except to overwrite it with a valid value. Many implementations do not detect the violations of this rule, and allow access to undefined values. The result is almost always a crash or a wrong result, because whatever is in memory by happenstance is almost never a correct value; often it is a value that is invalid for the type.

Had we not given either `sum` or `count` an initial value, our program most likely would have failed. The reason is that the first thing the program does with these variables is to use their values: The program reads `count` in order to increment it, and it reads `sum` in order to add its value to the one we just read. By the same token, we do not bother to give an initial value to `x`, because the first thing we do with it is read into it, thereby obliterating any value we might have given it.

The only new aspect of the `while` statement is the form of its condition:

```
// invariant:
//    we have read count grades so far, and
//    sum is the sum of the first count grades
while (cin >> x) {
    ++count;
    sum += x;
}
```

We already know that the `while` loop executes so long as the condition `cin >> x` succeeds. We'll explore the details of what it means to treat `cin >> x` as a condition in §3.1.1/39, but for now, what's important to know is that this condition succeeds if the most recent input request (i.e., `cin >> x`) succeeded.

Inside the `while`, we use the increment and compound-assignment operators, both of which we used in Chapter 2. From the discussion there we know that `++count` adds `1` to `count`, and that `sum += x` adds `x` to `sum`.

All that is left to explain is how the program does its output:

```
streamsize prec = cout.precision();
cout << "Your final grade is " << setprecision(3)
     << 0.2 * midterm + 0.4 * final + 0.4 * sum / count
     << setprecision(prec) << endl;
```

Our goal is to write the final grade with three significant digits, which we do by using `setprecision`. Like `endl`, `setprecision` is a manipulator. It manipulates the stream by causing subsequent output on that stream to appear with the given number of significant digits. By writing `setprecision(3)`, we ask the implementation to write grades with three significant digits, generally two before the decimal point and one after.

By using `setprecision`, we change the precision of any subsequent output that might appear on `cout`. Because this statement is at the end of the program, we know that there is no such output. Nevertheless, we believe that it is wise to reset `cout`'s precision to what it was before we changed it. We do so by calling a member function (§1.2/14) of `cout` named `precision`. This function tells us the precision that a stream uses for floating-point output. We use `setprecision` to set the precision to `3`, write the final grade, and then set the precision back to the value that `precision` gave us. The expression that computes the grade uses several of the arithmetic operators: `*` for multiplication, `/` for division, and `+` for addition, each of which has the obvious meaning.

We could have used the `precision` member function to set the precision, by writing

```
// set precision to 3, return previous value
streamsize prec = cout.precision(3);

cout << "Your final grade is "
     << 0.2 * midterm + 0.4 * final + 0.4 * sum / count << endl;

// reset precision to its original value
cout.precision(prec);
```

However, we prefer to use the `setprecision` manipulator, because by doing so, we can minimize the part of the program in which the precision is set to an unusual value.

3.1.1 Testing for end of input

Conceptually, the only really new part of this program is the condition in the `while` statement. That condition implicitly uses an `istream` as the subject of the `while` condition:

```
while (cin >> x) { /* ... */ }
```

The effect of this statement is to attempt to read from `cin`. If the read succeeds, `x` will hold the value that we just read, and the `while` test also succeeds. If the read fails (either because we have run out of input or because we encountered input that was invalid for the type of `x`), then the `while` test fails, and we should not rely on the value of `x`.

Understanding how this code works is a bit subtle. We can start by remembering that the `>>` operator returns its left operand, so that asking for the value of `cin >> x` is equivalent to executing `cin >> x` and then asking for the value of `cin`. For example, we can read a single value into `x`, and test whether we were successful in doing so, by executing

```
if (cin >> x) { /* ... */ }
```

This statement has the same meaning as

```
cin >> x;
if (cin) { /* ... */ }
```

When we use `cin >> x` as a condition, we aren't just testing the condition; we are also reading a value into `x` as a side effect. Now, all we need to do is figure out what it means to use `cin` as a condition in a `while` statement.

Because `cin` has type `istream`, which is part of the standard library, we must look to the definition of `istream` for the meaning of `if (cin)` or `while (cin)`. The details of that definition turn out to be complicated enough that we won't discuss it in detail until §12.5/222. However, even without these details, we can already understand a useful amount of what is happening.

The conditions that we used in Chapter 2 all involved relational operators that directly yield values of type `bool`. In addition, we can use expressions that yield values of arithmetic type as conditions. When used in a condition, the arithmetic value is converted to a `bool`: Nonzero values convert to `true`; zero values convert to `false`. For now, what we need to know is that similarly, the `istream` class provides a conversion that can be used to convert `cin` into a value that can be used in a condition. We don't yet know what type that value has, but we do know that the value can be converted to `bool`. Accordingly, we know that the value can be used in a condition. The value that this conversion yields depends on the internal state of the `istream` object, which will remember whether the last attempt to read worked. Thus, using `cin` as a condition is equivalent to testing whether the last attempt to read from `cin` was successful.

There are several ways in which trying to read from a stream can be unsuccessful:

- We might have reached the end of the input file.
- We might have encountered input that is incompatible with the type of the variable that we are trying to read, such as might happen if we try to read an `int` and find something that isn't a number.
- The system might have detected a hardware failure on the input device.

In any of these cases, the effect is the same: Using this input stream as a condition will indicate that the condition is false. Moreover, once we have failed to read from a stream, all further attempts to read from that stream will fail until we reset the stream, which we'll learn how to do in §4.1.3/57.

3.1.2 The loop invariant

Understanding the invariant (§2.3.2/20) for this loop requires special care, because the condition in the `while` has side effects. Those side effects affect the truth of the invariant: Successfully executing `cin >> x` makes the first part of the invariant—the part that says that we have read `count` grades—false. Accordingly, we must change our analysis to account for the effect that the condition itself might have on the invariant.

We know that the invariant was true before evaluating the condition, so we know that we have already read `count` grades. If `cin >> x` succeeds, then we have now read `count + 1` grades. We can make this part of the invariant true again by incrementing `count`. However, doing so falsifies the second part of the invariant—the part that says

that `sum` is the sum of the first `count` grades—because after we have incremented `count`, `sum` is now the sum of the first `count - 1` grades, not the first `count` grades. Fortunately, we can make the second part of the invariant true by executing `sum += x;` so that the entire invariant will be true on subsequent trips through the `while`.

If the condition is false, it means that our attempt at input failed, so we didn't get any more data, and so the invariant is still true. As a result, we do not have to account for the condition's side effects after the `while` finishes.

3.2 Using medians instead of averages

The program that we've written so far has a design shortcoming: It throws away each homework grade as soon as it has read it. Doing so is fine for computing averages, but what if we wanted to use the median homework grade instead of the average?

The most straightforward way to find the median of a collection of values is to sort the values into increasing (or decreasing) order and pick the middle one—or, if the number of values is even, take the average of the two values nearest the middle. Medians are often more useful than averages, because although they still account correctly for consistent performance, they won't cause a few lousy grades to blow the whole course.

A bit of thinking should convince us that to compute medians, we must change our program fundamentally. In order to find the median of an unknown number of values, we must store every value until we have read them all. To find the average, we were able to store only the count and running total of the items we'd read. The average was just the total divided by the count.

3.2.1 Storing a collection of data in a `vector`

To compute the median, we must read and store all the homework grades, then sort them, and finally pick the middle one (or two). To do this computation conveniently and efficiently, we need a way to

- Store a number of values that we will read one at a time, without knowing in advance how many values there are
- Sort the values after we have read them all
- Get at the middle value(s) efficiently

The standard library provides a type, named **`vector`**, that we can use to solve all these problems easily. A `vector` holds a sequence of values of a given type, grows as needed to accommodate additional values, and lets us get at each individual value efficiently.

Let's start rewriting our grading program by making it put the grades into a `vector`, instead of computing the sum and throwing the grades away. The original version of that code looked like

```
// original program (excerpt):
int count = 0;
double sum = 0;
double x;
```

```
// invariant:
//    we have read count grades so far, and
//    sum is the sum of the first count grades
while (cin >> x) {
    ++count;
    sum += x;
}
```

This loop kept track of how many grades it read, and kept a running total of their value. The need to keep both these variables in step with the values as we read them made the loop invariant relatively complicated. In contrast, using a `vector` to store values as we read them is much simpler:

```
// revised version of the excerpt:
double x;
vector<double> homework;

// invariant: homework contains all the homework grades read so far
while (cin >> x)
    homework.push_back(x);
```

We haven't changed the basic structure of our code: It still reads values one at a time into `x` until it encounters end-of-file or invalid input. What's different is what we do with those values.

Let's start with `homework`, which we define as having type `vector<double>`. A `vector` is a ***container*** that holds a collection of values. All of the values in an individual `vector` are the same type, but different `vector`s can hold objects of different types. Whenever we define a `vector`, we must specify the type of the values that the `vector` will hold. Our definition of `homework` says that it is a `vector`, which will hold values of type `double`.

The `vector` type is defined using a language feature called ***template classes***. We'll see how to define a template class in Chapter 11. For now, what's important is to realize that we can separate what it means to be a `vector` from the particular type of the objects that the `vector` holds. We specify the type of the objects inside angle brackets. For example, objects of type `vector<double>` are `vector`s that hold objects of type `double`, objects of type `vector<string>` hold `string`s, and so on.

The `while` loop operates by reading values from the standard input and storing them in the `vector`. As before, we read into `x` until we hit end-of-file or encounter input that is not a `double`. What is new is

```
homework.push_back(x);
```

As with `greeting.size()` in §1.2/14, we can see that `push_back` is a member function, which is defined as part of the `vector` type, and that we are asking that function to act on behalf of the object named `homework`. We call that function, passing it `x`. What `push_back` does is append a new element to the end of the `vector`. It gives that new element the value that we passed as the argument to `push_back`. Thus, `push_back` *pushes* its argument onto the *back* of a `vector`. As a side effect, it increases the size of the `vector` by one.

Because the `push_back` function is such a good match for what we're trying to do, it is trivial to see that calling it will maintain our loop invariant. Therefore, it is clear that when we drop out of the `while`, we will have read all the homework grades and stored them in `homework`, which is what we wanted.

Next, we have to think about the output.

3.2.2 Generating the output

In the original version of the program from §3.1/35, we calculated the student's grade within the output expression itself:

```
streamsize prec = cout.precision();
cout << "Your final grade is " << setprecision(3)
     << 0.2 * midterm + 0.4 * final + 0.4 * sum / count
     << setprecision(prec) << endl;
```

where `final` and `midterm` held the exam grades, and `sum` and `count` contained the sum of all the homework grades and the count of how many grades were entered.

As we remarked in §3.2.1/41, the easiest way to calculate the median is to sort our data and then find the middle value, or the average of the two middle values if we have an even number of elements. We can make the computation easier to understand if we separate the computation of the median from the code that writes the output.

In order to find the median, we begin by noting that we are going to need to know the size of the `homework` vector at least twice: once to check whether the size is zero, and again to compute the location of the middle element(s). To avoid having to ask for the size twice, we will store the size in a local variable:

```
typedef vector<double>::size_type vec_sz;
vec_sz size = homework.size();
```

The `vector` type defines a type named `vector<double>::size_type`, and a function named `size`. These members operate analogously to the ones in `string`: The type defined by `size_type` is an `unsigned` type guaranteed sufficient to hold the size of the largest possible `vector`, and `size()` returns a `size_type` value that represents the number of elements in the `vector`.

Because we need to know the size in two places, we will remember the value in a local variable. Different implementations use different types to represent sizes, so we cannot write the appropriate type directly and remain implementation-independent. For that reason, it is good programming practice to use the `size_type` that the library defines to represent container sizes, which we do in naming the type of `size`.

In this example, that type is unwieldy to write—and to read. To simplify our program, we have used a language facility that we haven't encountered before, called a **`typedef`**. When we include the word `typedef` as part of a definition, we are saying that we want the name that we define to be a synonym for the given type, rather than a variable of that type. Thus, because our definition includes `typedef`, it defines the name `vec_sz` as a synonym for `vector<double>::size_type`. Names defined via `typedef` have the

same scope as any other names. That is, we can use the name `vec_sz` as a synonym for the `size_type` until the end of the current scope.

Once we know how to name the type of the value that `homework.size()` returns, we can store that value in a local variable, named `size`, of the same type. It is worth noting that even though we are using the name `size` for two different purposes, there is no conflict or ambiguity. The only way to ask for the size of a `vector` is by putting a call to the `size` function on the right-hand side of a dot, with a `vector` on the left-hand side. In other words, the `size` that is defined as a local variable is in a different scope than the one that is an operation on `vectors`. Because these names are in different scopes, the compiler (and the programmer) can know which `size` is intended.

Because it is meaningless to find the median of an empty dataset, our next job is to verify that we have some data:

```
if (size == 0) {
    cout << endl << "You must enter your grades.  "
                    "Please try again." << endl;
    return 1;
}
```

We can detect this state of affairs by checking whether `size` is zero. If it is, the most sensible action is to complain and stop the program. We do so by returning `1` to indicate failure. As discussed in Chapter 0, the system presumes that if `main` returns `0`, the program succeeded. Returning anything else has an implementation-defined meaning, but most implementations treat any nonzero value as failure.

Now that we have verified that we have data, we can begin computing the median. The first part of doing so is to sort the data, which we do by calling a library function:

```
sort(homework.begin(), homework.end());
```

The `sort` function, defined in the `<algorithm>` header, rearranges the values in a container so that they are in nondecreasing order. We say nondecreasing instead of increasing because some elements might be equal to one another.

The arguments to `sort` specify the elements to be sorted. The `vector` class has member functions named `begin` and `end` for this purpose, so that `homework.begin()` denotes the first element in the `vector` named `homework`, and `homework.end()` denotes (one past) the last element in `homework`. We'll have much more to say in §5.2.2/81 about `begin` and `end`, and in §8.2.7/149 about the importance of ''(one past).''

The `sort` function does its work in place: It moves the values of the container's elements around rather than creating a new container to hold the results.

Once we have sorted `homework`, we need to find the middle element or elements:

```
vec_sz mid = size/2;
double median;
median = size % 2 == 0 ? (homework[mid] + homework[mid-1]) / 2
                       : homework[mid];
```

We begin by dividing `size` by `2` in order to locate the middle of the `vector`. If the number of elements is even, this division is exact. If the number is odd, then the result is the next lower integer.

Exactly how we compute the median depends on whether the number of elements is even or odd. If it is even, the median is the average of the two elements closest to the middle. Otherwise, there is an element in the middle, the value of which is the median.

The expression that assigns a value to `median` uses two new operators: the ***remainder*** operator, `%`, and the ***conditional*** operator, often called the `? :` operator.

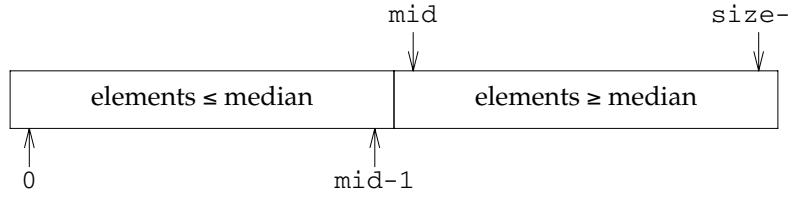
The remainder operator, `%`, returns the remainder that results from dividing its left operand by its right operand. If the remainder after dividing by `2` is `0`, then the program has read an even number of elements.

The conditional operator is shorthand for a simple if-then-else expression. First, it evaluates the expression, `size % 2 == 0`, that precedes the `?` part of the operator as a condition to obtain a `bool` value. If the condition yields `true`, then the result is the value of the expression between the `?` and the `:` that follows; otherwise, the result is the value of the expression after the `:`. So, if we read an even number of elements, we'll set `median` to the average of the two middle elements. If we read an odd number, then we'll set `median` to `homework[mid]`. Analogous to `&&` and `||`, the `? :` operator first evaluates its leftmost operand. Based on the resulting value, it then evaluates exactly one of its other operands.

The references to `homework[mid]` and `homework[mid-1]` show one way to access an element of a `vector`. Every element of every `vector` has an integer, called its ***index***, associated with it. So, for example, `homework[mid]` is the element of `homework` with index `mid`. As you might suspect from §2.6/30, the first element of the `vector` named `homework` is `homework[0]`, and the last element is `homework[size - 1]`.

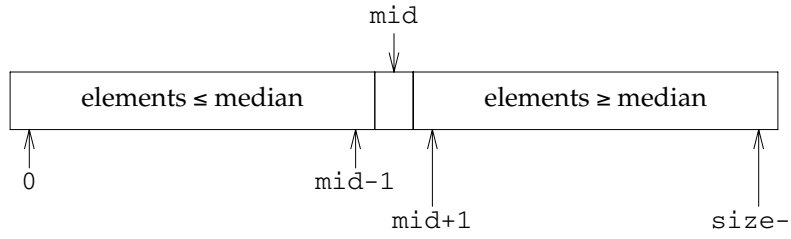
Each element is itself an (unnamed) object of the type stored in the container. So, `homework[mid]` is an object of type `double`, on which we can invoke any of the operations that type `double` supports. In particular, we can add two elements, and we can divide the resulting sum by `2` to get the average value of the two objects.

Once we know how to access the elements of `homework`, we can see how the median computation works. Assume first that `size` is even, so that `mid` is `size / 2`. Then there must be exactly `mid` elements of `homework` on each side of the middle:



Because we know that each half of `homework` has exactly `mid` elements, it should be easy to see that the indices of the two elements nearest the middle are `mid - 1` and `mid`; the median is the average of these elements.

If the number of elements is odd, then `mid` is really `(size - 1) / 2`, because of truncation. In that case, we can think of our sorted `homework` vector as two segments with `mid` elements each, separated by a single element in the middle. That element is the median:



In either case, our median computation relies on the ability to access a `vector` element knowing only its index.

Once we have computed the median, we need only compute and write the final grade:

```
streamsize prec = cout.precision();
cout << "Your final grade is " << setprecision(3)
     << 0.2 * midterm + 0.4 * final + 0.4 * median
     << setprecision(prec) << endl;
```

The final program isn't much more complicated than the program in §3.1/35, even though it does much more work. In particular, even though our `homework` vector will grow as needed to accommodate grades for as many homework assignments as our students can tolerate, our program doesn't need to worry about obtaining the memory to store all those grades. The standard library does all that work for us.

Here is the entire program. The only parts that we have not already mentioned are the `#include` directives, the corresponding `using`-declarations, and a few more comments:

```
#include <algorithm>
#include <iomanip>
#include <ios>
#include <iostream>
#include <string>
#include <vector>

using std::cin;              using std::sort;
using std::cout;             using std::streamsize;
using std::endl;             using std::string;
using std::setprecision;     using std::vector;

int main()
{
    // ask for and read the student's name
    cout << "Please enter your first name: ";
    string name;
    cin >> name;
    cout << "Hello, " << name << "!" << endl;

    // ask for and read the midterm and final grades
    cout << "Please enter your midterm and final exam grades: ";
    double midterm, final;
    cin >> midterm >> final;

    // ask for and read the homework grades
    cout << "Enter all your homework grades, "
            "followed by end-of-file: ";
```

```
    vector<double> homework;
    double x;
    // invariant: homework contains all the homework grades read so far
    while (cin >> x)
        homework.push_back(x);

    // check that the student entered some homework grades
    typedef vector<double>::size_type vec_sz;
    vec_sz size = homework.size();
    if (size == 0) {
        cout << endl << "You must enter your grades.  "
                        "Please try again." << endl;
        return 1;
    }

    // sort the grades
    sort(homework.begin(), homework.end());

    // compute the median homework grade
    vec_sz mid = size/2;
    double median;
    median = size % 2 == 0 ? (homework[mid] + homework[mid-1]) / 2
                           : homework[mid];

    // compute and write the final grade
    streamsize prec = cout.precision();
    cout << "Your final grade is " << setprecision(3)
         << 0.2 * midterm + 0.4 * final + 0.4 * median
         << setprecision(prec) << endl;

    return 0;
}
```

3.2.3 Some additional observations

This code contains some points that deserve particular attention. First, there's a bit more to know about why we exit the program if `homework` is empty. Logically, taking the median of an empty collection of values is undefined—we have no idea what it might mean. Therefore, exiting does the right thing: If we don't know what to do, we might as well quit. But it is important to know what would happen if we had continued execution. If the input were empty, and we neglected to test that we had read at least one number, the code to compute the median would fail. Why?

If we had read no elements, then `homework.size()`, and therefore `size`, would be `0`. Likewise, `mid` would be `0`. When we executed `homework[mid]`, we would be looking at the first element (the one indexed by `0`) in `homework`. But there are no elements in `homework`! When we execute `homework[0]`, all bets are off as to what we get. `vectors` do not check whether the index is in range. Such checking is up to the user.

The next important observation is that `vector<double>::size_type`, like all standard-library size types, is an ***unsigned integral type***. Such types are incapable of storing negative values at all; instead, they store values modulo 2^n, where n depends on the implementation. So, for example, there would never be any point in checking whether `homework.size() < 0`, because that comparison would always yield `false`.

Moreover, whenever ordinary integers and unsigned integers combine in an expression, the ordinary integer is converted to unsigned. In consequence, expressions such as `homework.size() - 100` yield unsigned results, which means that they, too, cannot be less than zero—even if `homework.size() < 100`.

Finally, it is also worth noting that the execution performance of our program is actually quite good, even though the `vector<double>` object grows as needed to accommodate its input, rather than being allocated with the right size immediately.

We can be confident about the program's performance because the C++ standard imposes performance requirements on the library's implementation. Not only must the library meet the specifications for behavior, but it must also achieve well-specified performance goals. Every standard-conforming C++ implementation must

- Implement `vector` so that appending a large number of elements to a `vector` is no worse than proportional to the number of elements
- Implement `sort` to be no slower on average than $n\log(n)$, where n is the number of elements being sorted

Therefore, the whole program will normally run in $n\log(n)$ time or better on any standard-conforming implementation. In fact, the standard library was designed with a ruthless attention to performance. C++ is designed for use in performance-critical applications, and the emphasis on speed pervades the library as well.

3.3 Details

Local variables are default-initialized if they are defined without an explicit initializer. Default-initialization of a built-in type means that the value is undefined. Undefined values may be used only as the left-hand side of an assignment.

Type definitions:

`typedef` *type name*; Defines *name* as a synonym for *type*.

The `vector` type, defined in `<vector>`, is a library type that is a container that holds a sequence of values of a specified type. `vectors` grow dynamically. Some important operations are:

`vector<T>::size_type`	A type guaranteed to be able to hold the number of elements in the largest possible `vector`.
`v.begin()`	Returns a value that denotes the first element in `v`.
`v.end()`	Returns a value that denotes (one past) the last element in `v`.
`vector<T> v;`	Creates an empty `vector` that can hold elements of type `T`.
`v.push_back(e)`	Grows the `vector` by one element initialized to `e`.
`v[i]`	Returns the value stored in position `i`.
`v.size()`	Returns the number of elements in `v`.

Other library facilities

`sort(b, e)`	Rearranges the elements defined by the range `[b, e)` into nondecreasing order. Defined in `<algorithm>`.
`max(e1, e2)`	Returns the larger of the expressions `e1` and `e2`; `e1` and `e2` must have exactly the same type. Defined in `<algorithm>`.
`while (cin >> x)`	Reads a value of an appropriate type into `x` and tests the state of the stream. If the stream is in an error state, the test fails; otherwise, the test succeeds, and the body of the `while` is executed.
`s.precision(n)`	Sets the precision of stream `s` to `n` for future output (or leaves it unchanged if `n` is omitted). Returns the previous precision.
`setprecision(n)`	Returns a value that, when written on an output stream `s`, has the effect of calling `s.precision(n)`. Defined in `<iomanip>`.
`streamsize`	The type of the value expected by `setprecision` and returned by `precision`. Defined in `<ios>`.

Exercises

3-0. Compile, execute, and test the programs in this chapter.

3-1. Suppose we wish to find the median of a collection of values. Assume that we have read some of the values so far, and that we have no idea how many values remain to be read. Prove that we cannot afford to discard any of the values that we have read. *Hint:* One proof strategy is to assume that we can discard a value, and then find values for the unread—and therefore unknown—part of our collection that would cause the median to be the value that we discarded.

3-2. Write a program to compute and print the quartiles (that is, the quarter of the numbers with the largest values, the next highest quarter, and so on) of a set of integers.

3-3. Write a program to count how many times each distinct word appears in its input.

3-4. Write a program to report the length of the longest and shortest `string` in its input.

3-5. Write a program that will keep track of grades for several students at once. The program could keep two `vectors` in sync: The first should hold the student's names, and the second the final grades that can be computed as input is read. For now, you should assume a fixed number of homework grades. We'll see in §4.1.3/56 how to handle a variable number of grades intermixed with student names.

3-6. The average-grade computation in §3.1/36 might divide by zero if the student didn't enter any grades. Division by zero is undefined in C++, which means that the implementation is permitted to do anything it likes. What does your C++ implementation do in this case? Rewrite the program so that its behavior does not depend on how the implementation treats division by zero.

This page intentionally left blank