ALGORITHMS WITH C++

CRYSTAL EDUCATION – TURX, CALLG

STARTER

SOME FUNDAMENTAL C++ KNOWLEDGE

BASIC DATA TYPES

integer

DATA TYPE	SIZE (IN BYTES)	RANGE
short int	2	-32,768 to 32,767
unsigned short int	2	0 to 65,535
unsigned int	4	0 to 4,294,967,295
int	4	-2,147,483,648 to 2,147,483,647
long int	4	-2,147,483,648 to 2,147,483,647
unsigned long int	4	0 to 4,294,967,295
long long int	8	-(2^63) to (2^63)-1
unsigned long long int	8	0 to 18,446,744,073,709,551,615
signed char	1	-128 to 127
unsigned char	1	0 to 255
float	4	
double	8	
long double	12	
wchar_t	2 or 4	1 wide character

- Use sizeof(type) to get the size of a data type.
- Source: https://www.geeksforgeeks.org/c-data-types/

Dev-C++

HOW TO WRITE A SIMPLE C++ PROGRAM?

vscode

1. Edit a *.cpp file;
2. Compile it;
3. Run it.

- Example:
  - Hello, world. (hlw.cpp)

HOW TO OUTPUT A STRING?

Tip:
CRLF - Windows - \r\n
LF - Linux - \n
CR - macOS - \r

- Try
  1. int printf (const char * format, ...);
     - format: %[flags][width][.precision][length]specifier
     - http://www.cplusplus.com/reference/cstdio/printf/
     - printf(string, obj);
  2. extern ostream cout;
     - Object of class ostream that represents the standard output stream oriented to narrow characters (of type char). It corresponds to the C stream stdout.
     - http://www.cplusplus.com/reference/iostream/cout/
     - cout << obj << endl;
     - Try clog, cerr
     - Try (cout << objA) << objB;
  3. int putchar (int character);
     - http://www.cplusplus.com/reference/cstdio/putchar/

- printf Method
  - printf is a function (What is 'function'?) called manually by the code
- cout Method
  - cout is a stream
  - cout << obj << endl;
    - Write obj, endl((\r)\n) to (ostream)cout => stdout
- putchar Method
  - Parameter (int)character:
    - The int promotion of the character to be written.
    - The value is internally converted to an unsigned char when written.
  - Put (int)character to stdout

WHAT IS 'FUNCTION'?

- A function is a group of statements that together perform a task. Every C++ program has at least one function, which is main(), and all the most trivial programs can define additional functions.
- You can divide up your code into separate functions. How you divide up your code among different functions is up to you, but logically the division usually is such that each function performs a specific task.
- A function declaration tells the compiler about a function's name, return type, and parameters. A function definition provides the actual body of the function.
- The C++ standard library provides numerous built-in functions that your program can call. For example, function strcat() to concatenate two strings, function memcpy() to copy one memory location to another location and many more functions.
- A function is known with various names like a method or a sub-routine or a procedure etc.
- Source: https://www.tutorialspoint.com/cplusplus/cpp_functions.htm

HOW TO INPUT A STRING?

- Try
  1. int scanf (const char * format, ...);
     - format: %[*][width][length]specifier
     - http://www.cplusplus.com/reference/cstdio/scanf/
     - scanf(cstring, ref obj)
  2. extern istream cin;
     - Object of class istream that represents the standard input stream oriented to narrow characters (of type char). It corresponds to the C stream stdin.
     - http://www.cplusplus.com/reference/iostream/cin/
     - cin >> obj;
     - Try (cin >> objA) >> objB;
  3. int getchar ();
     - http://www.cplusplus.com/reference/cstdio/getchar/

- scanf Method
  - scanf is a function called manually by the code
  - obj is a reference (What is reference?)
- cin Method
  - cin is a stream
  - cin << obj;
    - Write obj to (istream)cin => stdin

stderr

- getchar Method
  - **Get character from stdin**
  - On success, the character read is returned (promoted to an int value).
  - The return type is int to accommodate for the special value EOF, which indicates failure:
  - If the standard input was at the end-of-file, the function returns EOF and sets the eof indicator (feof) of stdin.
  - If some other reading error happens, the function also returns EOF, but sets its error indicator (ferror) instead.

Tip:
cstring => char[] => char*

WHAT IS ‘POINTER’?

- C++ pointers are easy and fun to learn. Some C++ tasks are performed more easily with pointers, and other C++ tasks, such as dynamic memory allocation, cannot be performed without them.
- As you know every variable is a memory location and every memory location has its address defined which can be accessed using ampersand (&) operator which denotes an address in memory. Consider the following which will print the address of the variables defined – (~/cpp/ref/getPtrLoc.cpp)
- When the above code is compiled and executed, it produces the following result – (~/out/ref/getPtrLoc.out)
  - format:
    - Address of var1 variable: 0x********
    - Address of var2 variable: 0x********

```
f[char[])
f(char *x) {
    x[n];
}
```

WHAT IS ‘REFERENCE’?

int &b = a;

- A reference variable is an alias, that is, another name for an already existing variable. Once a reference is initialized with a variable, either the variable name or the reference name may be used to refer to the variable.
- Source: https://www.tutorialspoint.com/cplusplus/cpp_references.htm

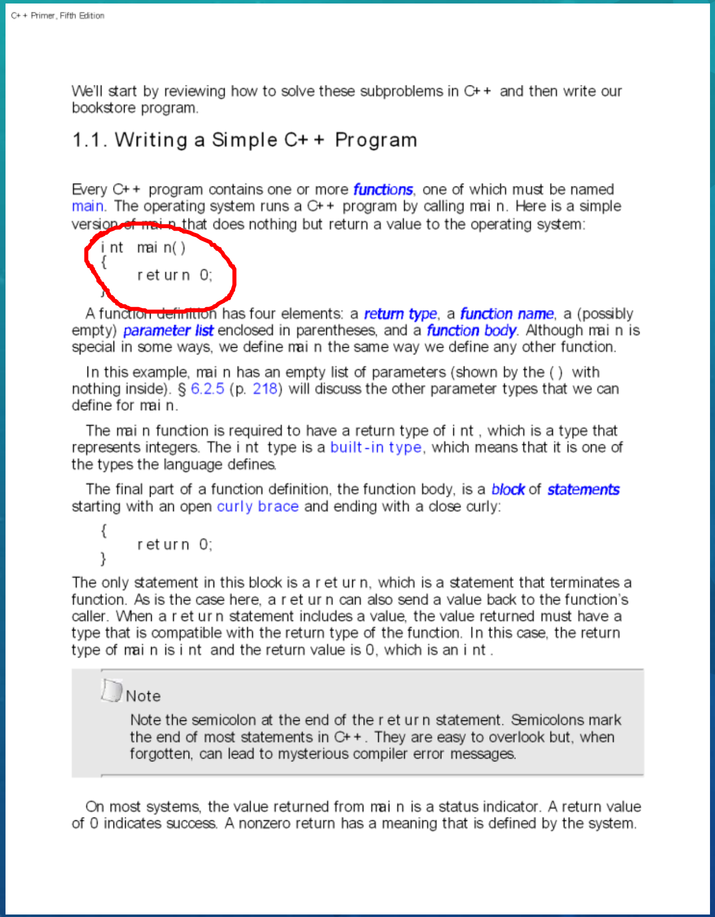
REFERENCES VS POINTERS

- References are often confused with pointers but three major differences between references and pointers are −
  - You cannot have NULL references. You must always be able to assume that a reference is connected to a legitimate piece of storage.
  - Once a reference is initialized to an object, it cannot be changed to refer to another object. Pointers can be pointed to another object at any time.
  - A reference must be initialized when it is created. Pointers can be initialized at any time.
- Source: https://www.tutorialspoint.com/cplusplus/cpp_references.htm

Algorithms with C++

Back to C++ Primer:

1.1 WRITING A SIMPLE C++ PROGRAM (P27 – P30)



C++ Primer, Fifth Edition

We'll start by reviewing how to solve these subproblems in C++ and then write our bookstore program.

1.1. Writing a Simple C++ Program

Every C++ program contains one or more ***functions***, one of which must be named main. The operating system runs a C++ program by calling main. Here is a simple version of main that does nothing but return a value to the operating system:

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

A function definition has four elements: a ***return type***, a ***function name***, a (possibly empty) ***parameter list*** enclosed in parentheses, and a ***function body***. Although main is special in some ways, we define main the same way we define any other function.

In this example, main has an empty list of parameters (shown by the () with nothing inside). § 6.2.5 (p. 218) will discuss the other parameter types that we can define for main.

The main function is required to have a return type of int, which is a type that represents integers. The int type is a built-in type, which means that it is one of the types the language defines.

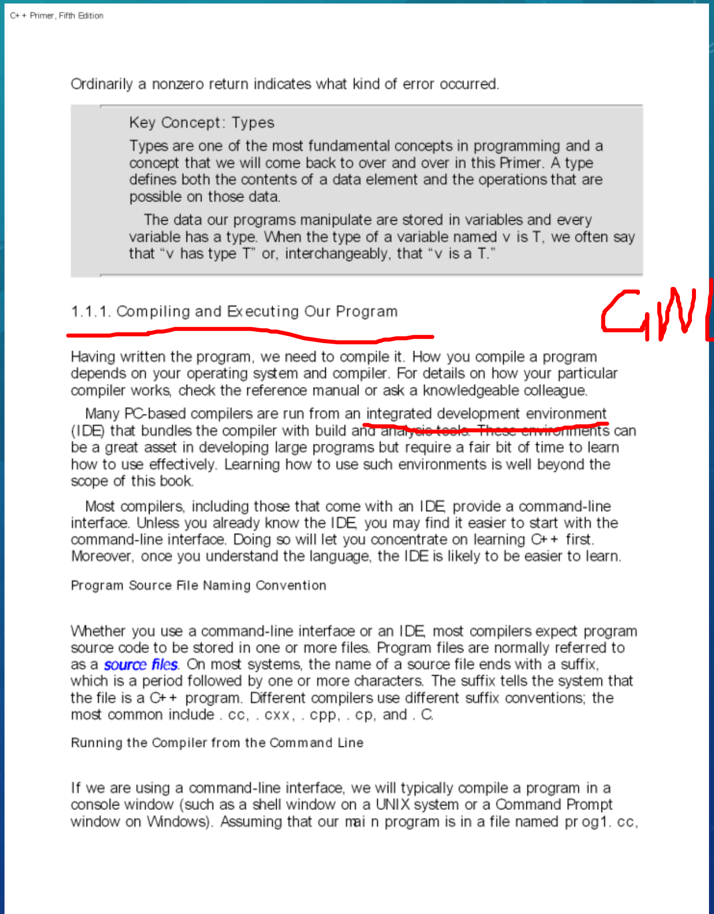
The final part of a function definition, the function body, is a ***block*** of ***statements*** starting with an open curly brace and ending with a close curly:

```
{
    return 0;
}
```

The only statement in this block is a return, which is a statement that terminates a function. As is the case here, a return can also send a value back to the function's caller. When a return statement includes a value, the value returned must have a type that is compatible with the return type of the function. In this case, the return type of main is int and the return value is 0, which is an int.

> Note
>
> Note the semicolon at the end of the return statement. Semicolons mark the end of most statements in C++. They are easy to overlook but, when forgotten, can lead to mysterious compiler error messages.

On most systems, the value returned from main is a status indicator. A return value of 0 indicates success. A nonzero return has a meaning that is defined by the system.



C++ Primer, Fifth Edition

Ordinarily a nonzero return indicates what kind of error occurred.

> Key Concept: Types
>
> Types are one of the most fundamental concepts in programming and a concept that we will come back to over and over in this Primer. A type defines both the contents of a data element and the operations that are possible on those data.
>
> The data our programs manipulate are stored in variables and every variable has a type. When the type of a variable named v is T, we often say that "v has type T" or, interchangeably, that "v is a T."

1.1.1. Compiling and Executing Our Program

Having written the program, we need to compile it. How you compile a program depends on your operating system and compiler. For details on how your particular compiler works, check the reference manual or ask a knowledgeable colleague.

Many PC-based compilers are run from an integrated development environment (IDE) that bundles the compiler with build and analysis tools. These environments can be a great asset in developing large programs but require a fair bit of time to learn how to use effectively. Learning how to use such environments is well beyond the scope of this book.

Most compilers, including those that come with an IDE, provide a command-line interface. Unless you already know the IDE, you may find it easier to start with the command-line interface. Doing so will let you concentrate on learning C++ first. Moreover, once you understand the language, the IDE is likely to be easier to learn.

Program Source File Naming Convention

Whether you use a command-line interface or an IDE, most compilers expect program source code to be stored in one or more files. Program files are normally referred to as a ***source files***. On most systems, the name of a source file ends with a suffix, which is a period followed by one or more characters. The suffix tells the system that the file is a C++ program. Different compilers use different suffix conventions; the most common include .cc, .cxx, .cpp, .cp, and .C.

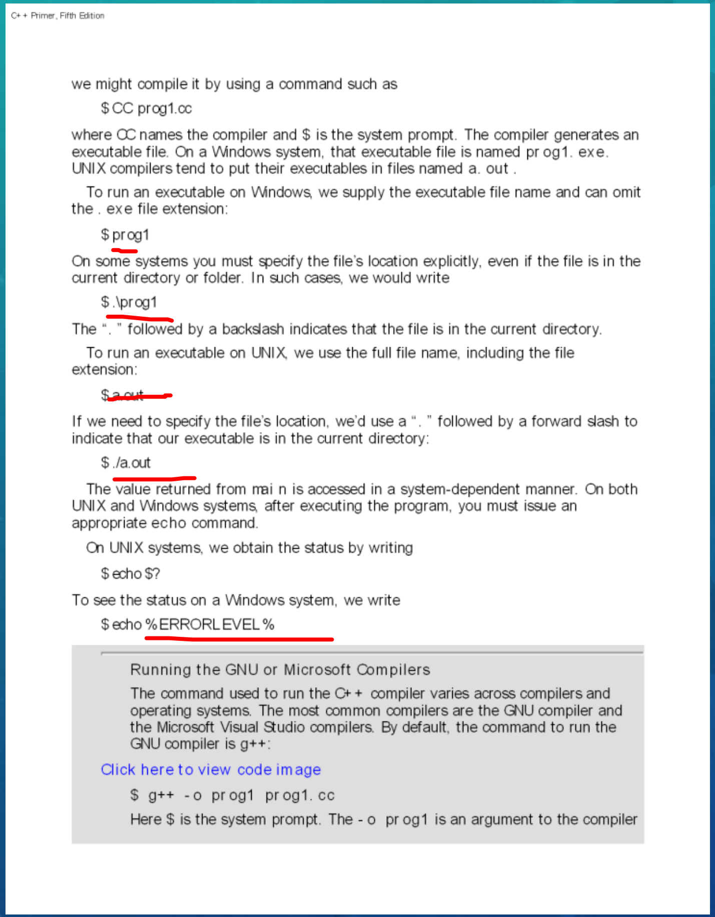
Running the Compiler from the Command Line

If we are using a command-line interface, we will typically compile a program in a console window (such as a shell window on a UNIX system or a Command Prompt window on Windows). Assuming that our main program is in a file named prog1.cc,

GNU G++

Back to C++ Primer:

1.1 WRITING A SIMPLE C++ PROGRAM (P27 – P30)



C++ Primer, Fifth Edition

we might compile it by using a command such as

```
$ CC prog1.cc
```

where CC names the compiler and $ is the system prompt. The compiler generates an executable file. On a Windows system, that executable file is named prog1.exe. UNIX compilers tend to put their executables in files named a.out.

To run an executable on Windows, we supply the executable file name and can omit the .exe file extension:

```
$ prog1
```

On some systems you must specify the file's location explicitly, even if the file is in the current directory or folder. In such cases, we would write

```
$ .\prog1
```

The "." followed by a backslash indicates that the file is in the current directory.

To run an executable on UNIX, we use the full file name, including the file extension:

$ ~~a.out~~

If we need to specify the file's location, we'd use a "." followed by a forward slash to indicate that our executable is in the current directory:

```
$ ./a.out
```

The value returned from main is accessed in a system-dependent manner. On both UNIX and Windows systems, after executing the program, you must issue an appropriate echo command.

On UNIX systems, we obtain the status by writing

```
$ echo $?
```

To see the status on a Windows system, we write

```
$ echo %ERRORLEVEL%
```

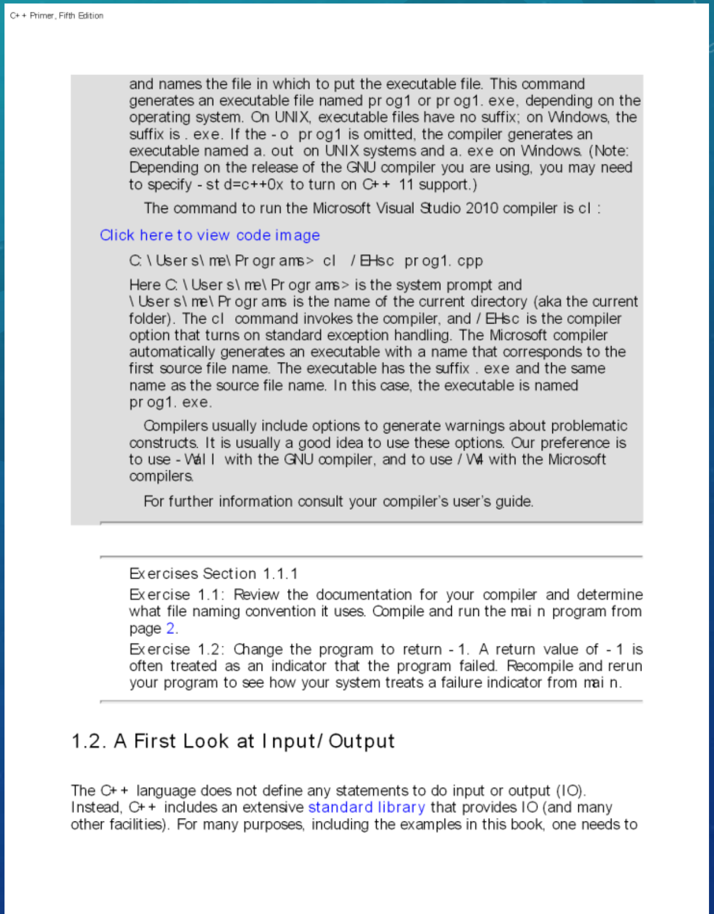
Running the GNU or Microsoft Compilers

The command used to run the C++ compiler varies across compilers and operating systems. The most common compilers are the GNU compiler and the Microsoft Visual Studio compilers. By default, the command to run the GNU compiler is g++:

Click here to view code image

```
$ g++ -o prog1 prog1.cc
```

Here $ is the system prompt. The -o prog1 is an argument to the compiler



C++ Primer, Fifth Edition

and names the file in which to put the executable file. This command generates an executable file named prog1 or prog1.exe, depending on the operating system. On UNIX, executable files have no suffix; on Windows, the suffix is .exe. If the -o prog1 is omitted, the compiler generates an executable named a.out on UNIX systems and a.exe on Windows. (Note: Depending on the release of the GNU compiler you are using, you may need to specify -std=c++0x to turn on C++ 11 support.)

The command to run the Microsoft Visual Studio 2010 compiler is cl:

Click here to view code image

```
C:\Users\me\Programs> cl /EHsc prog1.cpp
```

Here C:\Users\me\Programs> is the system prompt and \Users\me\Programs is the name of the current directory (aka the current folder). The cl command invokes the compiler, and /EHsc is the compiler option that turns on standard exception handling. The Microsoft compiler automatically generates an executable with a name that corresponds to the first source file name. The executable has the suffix .exe and the same name as the source file name. In this case, the executable is named prog1.exe.

Compilers usually include options to generate warnings about problematic constructs. It is usually a good idea to use these options. Our preference is to use -Wall with the GNU compiler, and to use /W4 with the Microsoft compilers.

For further information consult your compiler's user's guide.

Exercises Section 1.1.1

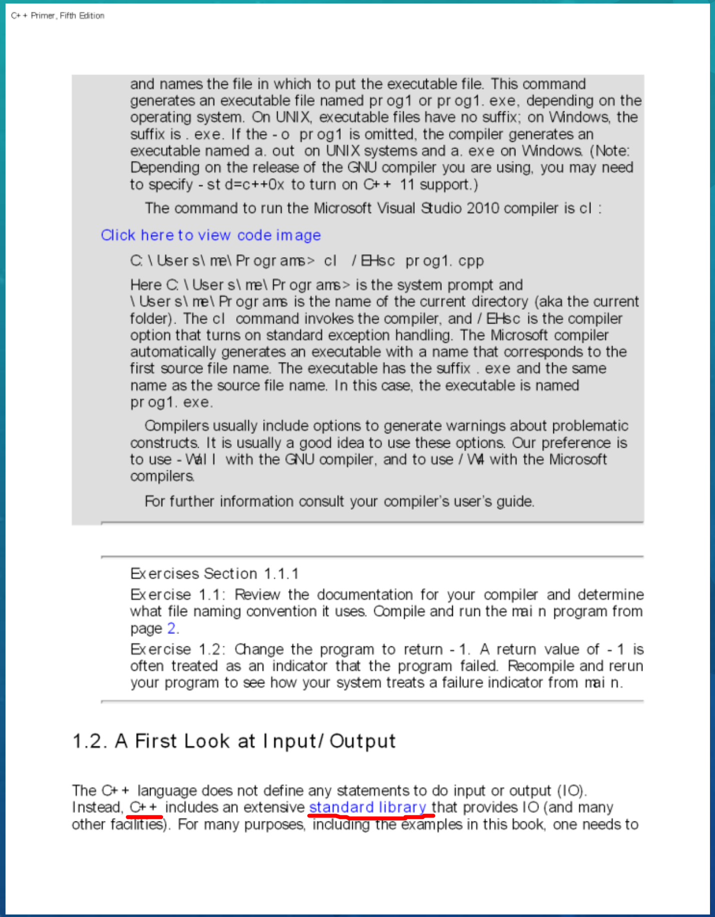
Exercise 1.1: Review the documentation for your compiler and determine what file naming convention it uses. Compile and run the main program from page 2.

Exercise 1.2: Change the program to return -1. A return value of -1 is often treated as an indicator that the program failed. Recompile and rerun your program to see how your system treats a failure indicator from main.

1.2. A First Look at Input/Output

The C++ language does not define any statements to do input or output (IO). Instead, C++ includes an extensive standard library that provides IO (and many other facilities). For many purposes, including the examples in this book, one needs to

1.2. A FIRST LOOK AT INPUT/OUTPUT (P30 – P35)



C++ Primer, Fifth Edition

and names the file in which to put the executable file. This command generates an executable file named prog1 or prog1.exe, depending on the operating system. On UNIX, executable files have no suffix; on Windows, the suffix is .exe. If the -o prog1 is omitted, the compiler generates an executable named a.out on UNIX systems and a.exe on Windows. (Note: Depending on the release of the GNU compiler you are using, you may need to specify -std=c++0x to turn on C++ 11 support.)

The command to run the Microsoft Visual Studio 2010 compiler is cl:

Click here to view code image

```
C:\Users\me\Programs> cl /EHsc prog1.cpp
```

Here C:\Users\me\Programs> is the system prompt and \Users\me\Programs is the name of the current directory (aka the current folder). The cl command invokes the compiler, and /EHsc is the compiler option that turns on standard exception handling. The Microsoft compiler automatically generates an executable with a name that corresponds to the first source file name. The executable has the suffix .exe and the same name as the source file name. In this case, the executable is named prog1.exe.

Compilers usually include options to generate warnings about problematic constructs. It is usually a good idea to use these options. Our preference is to use -Wall with the GNU compiler, and to use /W4 with the Microsoft compilers.

For further information consult your compiler's user's guide.

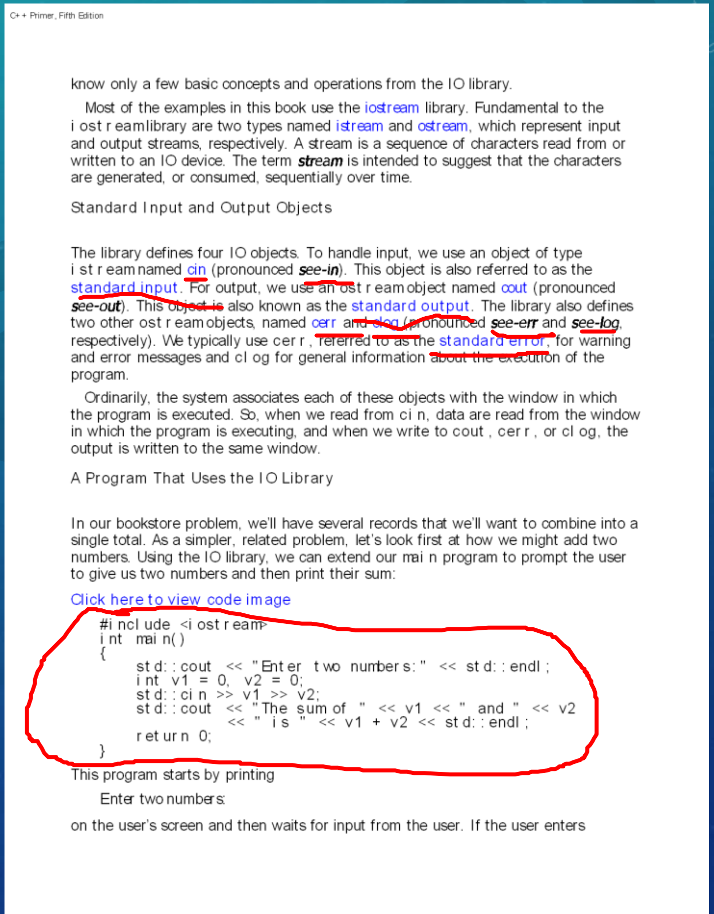
Exercises Section 1.1.1

Exercise 1.1: Review the documentation for your compiler and determine what file naming convention it uses. Compile and run the main program from page 2.

Exercise 1.2: Change the program to return -1. A return value of -1 is often treated as an indicator that the program failed. Recompile and rerun your program to see how your system treats a failure indicator from main.

1.2. A First Look at Input/Output

The C++ language does not define any statements to do input or output (IO). Instead, C++ includes an extensive standard library that provides IO (and many other facilities). For many purposes, including the examples in this book, one needs to



C++ Primer, Fifth Edition

know only a few basic concepts and operations from the IO library.

Most of the examples in this book use the iostream library. Fundamental to the iostream library are two types named istream and ostream, which represent input and output streams, respectively. A stream is a sequence of characters read from or written to an IO device. The term ***stream*** is intended to suggest that the characters are generated, or consumed, sequentially over time.

Standard Input and Output Objects

The library defines four IO objects. To handle input, we use an object of type istream named cin (pronounced ***see-in***). This object is also referred to as the standard input. For output, we use an ostream object named cout (pronounced ***see-out***). This object is also known as the standard output. The library also defines two other ostream objects, named cerr and clog (pronounced ***see-err*** and ***see-log***, respectively). We typically use cerr, referred to as the standard error, for warning and error messages and clog for general information about the execution of the program.

Ordinarily, the system associates each of these objects with the window in which the program is executed. So, when we read from cin, data are read from the window in which the program is executing, and when we write to cout, cerr, or clog, the output is written to the same window.

A Program That Uses the IO Library

In our bookstore problem, we'll have several records that we'll want to combine into a single total. As a simpler, related problem, let's look first at how we might add two numbers. Using the IO library, we can extend our main program to prompt the user to give us two numbers and then print their sum:

Click here to view code image

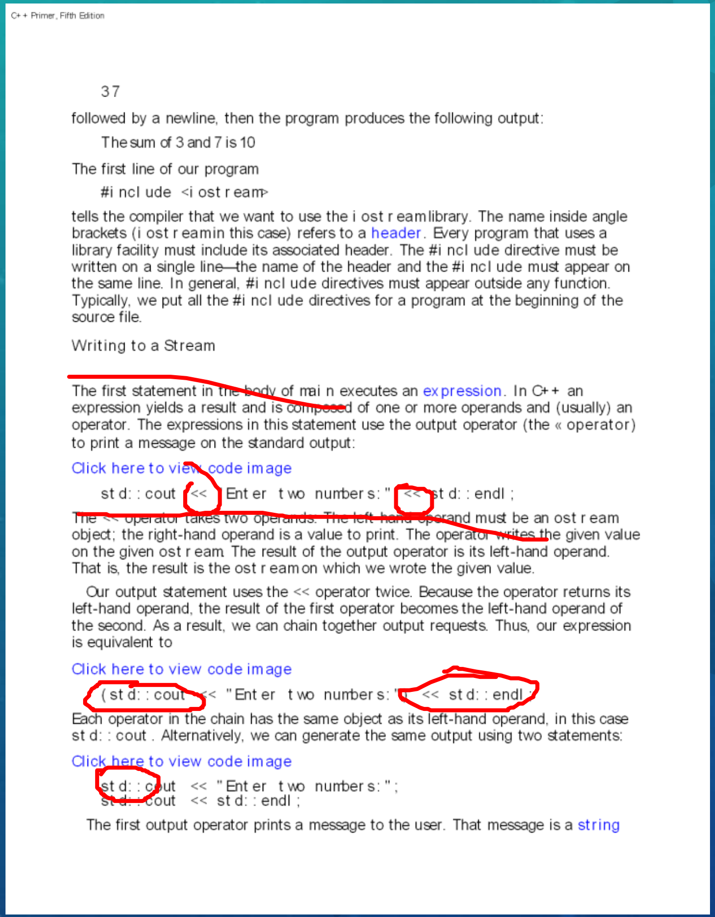
```
#include <iostream>
int main()
{
    std::cout << "Enter two numbers:" << std::endl;
    int v1 = 0, v2 = 0;
    std::cin >> v1 >> v2;
    std::cout << "The sum of " << v1 << " and " << v2
              << " is " << v1 + v2 << std::endl;
    return 0;
}
```

This program starts by printing

```
Enter two numbers:
```

on the user's screen and then waits for input from the user. If the user enters

1.2. A FIRST LOOK AT INPUT/OUTPUT (P30 – 35)



```
37
```

followed by a newline, then the program produces the following output:

```
The sum of 3 and 7 is 10
```

The first line of our program

```cpp
#include <iostream>
```

tells the compiler that we want to use the iostream library. The name inside angle brackets (iostream in this case) refers to a header. Every program that uses a library facility must include its associated header. The #include directive must be written on a single line—the name of the header and the #include must appear on the same line. In general, #include directives must appear outside any function. Typically, we put all the #include directives for a program at the beginning of the source file.

Writing to a Stream

The first statement in the body of main executes an expression. In C++ an expression yields a result and is composed of one or more operands and (usually) an operator. The expressions in this statement use the output operator (the « operator) to print a message on the standard output:

Click here to view code image

```cpp
std::cout << "Enter two numbers:" << std::endl;
```

The << operator takes two operands: The left-hand operand must be an ostream object; the right-hand operand is a value to print. The operator writes the given value on the given ostream. The result of the output operator is its left-hand operand. That is, the result is the ostream on which we wrote the given value.

Our output statement uses the << operator twice. Because the operator returns its left-hand operand, the result of the first operator becomes the left-hand operand of the second. As a result, we can chain together output requests. Thus, our expression is equivalent to

Click here to view code image

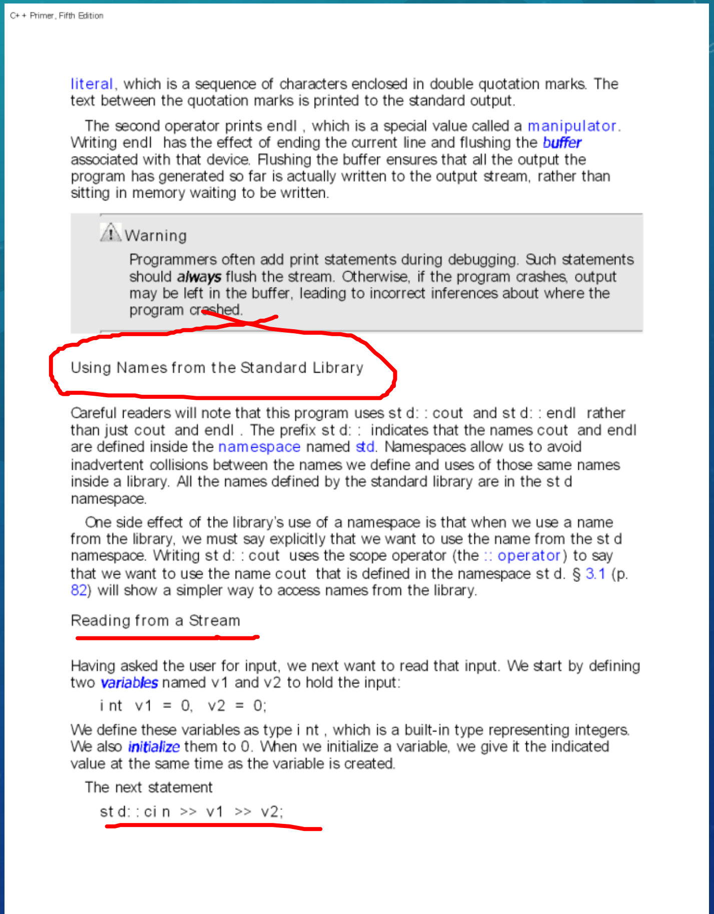
```cpp
(std::cout << "Enter two numbers:") << std::endl;
```

Each operator in the chain has the same object as its left-hand operand, in this case std::cout. Alternatively, we can generate the same output using two statements:

Click here to view code image

```cpp
std::cout << "Enter two numbers:";
std::cout << std::endl;
```

The first output operator prints a message to the user. That message is a string



literal, which is a sequence of characters enclosed in double quotation marks. The text between the quotation marks is printed to the standard output.

The second operator prints endl, which is a special value called a manipulator. Writing endl has the effect of ending the current line and flushing the ***buffer*** associated with that device. Flushing the buffer ensures that all the output the program has generated so far is actually written to the output stream, rather than sitting in memory waiting to be written.

> ⚠ Warning
>
> Programmers often add print statements during debugging. Such statements should ***always*** flush the stream. Otherwise, if the program crashes, output may be left in the buffer, leading to incorrect inferences about where the program crashed.

Using Names from the Standard Library

Careful readers will note that this program uses std::cout and std::endl rather than just cout and endl. The prefix std:: indicates that the names cout and endl are defined inside the namespace named std. Namespaces allow us to avoid inadvertent collisions between the names we define and uses of those same names inside a library. All the names defined by the standard library are in the std namespace.

One side effect of the library's use of a namespace is that when we use a name from the library, we must say explicitly that we want to use the name from the std namespace. Writing std::cout uses the scope operator (the :: operator) to say that we want to use the name cout that is defined in the namespace std. § 3.1 (p. 82) will show a simpler way to access names from the library.

Reading from a Stream

Having asked the user for input, we next want to read that input. We start by defining two ***variables*** named v1 and v2 to hold the input:

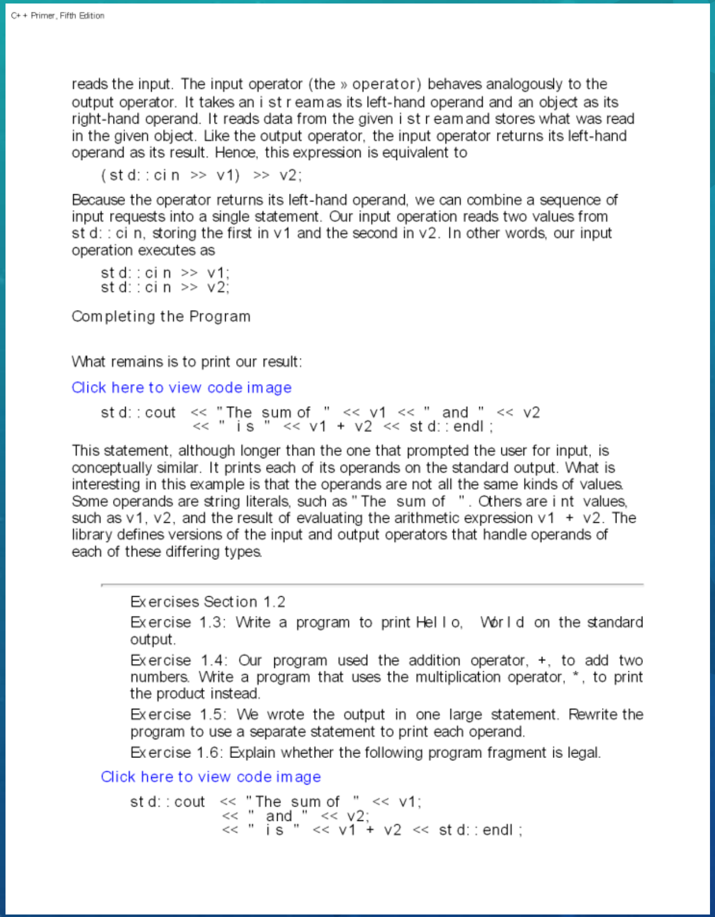
```cpp
int v1 = 0, v2 = 0;
```

We define these variables as type int, which is a built-in type representing integers. We also ***initialize*** them to 0. When we initialize a variable, we give it the indicated value at the same time as the variable is created.

The next statement

```cpp
std::cin >> v1 >> v2;
```

1.2. A FIRST LOOK AT INPUT/OUTPUT (P30 – 35)



C++ Primer, Fifth Edition

reads the input. The input operator (the » operator) behaves analogously to the output operator. It takes an istream as its left-hand operand and an object as its right-hand operand. It reads data from the given istream and stores what was read in the given object. Like the output operator, the input operator returns its left-hand operand as its result. Hence, this expression is equivalent to

```
(std::cin >> v1) >> v2;
```

Because the operator returns its left-hand operand, we can combine a sequence of input requests into a single statement. Our input operation reads two values from std::cin, storing the first in v1 and the second in v2. In other words, our input operation executes as

```
std::cin >> v1;
std::cin >> v2;
```

Completing the Program

What remains is to print our result:

Click here to view code image

```
std::cout << "The sum of " << v1 << " and " << v2
          << " is " << v1 + v2 << std::endl;
```

This statement, although longer than the one that prompted the user for input, is conceptually similar. It prints each of its operands on the standard output. What is interesting in this example is that the operands are not all the same kinds of values. Some operands are string literals, such as "The sum of ". Others are int values, such as v1, v2, and the result of evaluating the arithmetic expression v1 + v2. The library defines versions of the input and output operators that handle operands of each of these differing types.

Exercises Section 1.2

Exercise 1.3: Write a program to print Hello, World on the standard output.

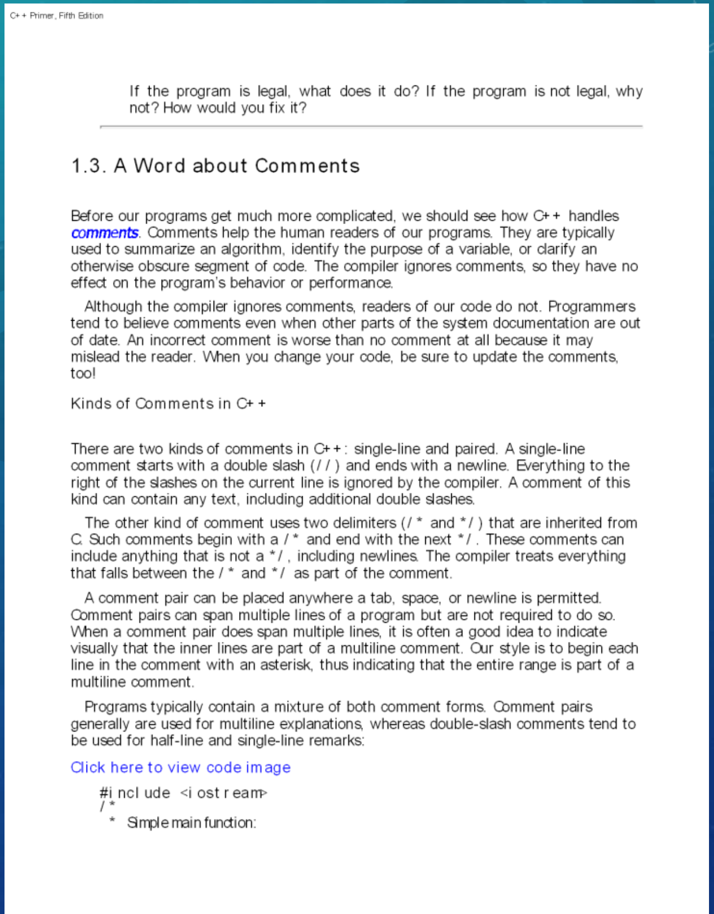
Exercise 1.4: Our program used the addition operator, +, to add two numbers. Write a program that uses the multiplication operator, *, to print the product instead.

Exercise 1.5: We wrote the output in one large statement. Rewrite the program to use a separate statement to print each operand.

Exercise 1.6: Explain whether the following program fragment is legal.

Click here to view code image

```
std::cout << "The sum of " << v1;
          << " and " << v2;
          << " is " << v1 + v2 << std::endl;
```



C++ Primer, Fifth Edition

If the program is legal, what does it do? If the program is not legal, why not? How would you fix it?

1.3. A Word about Comments

Before our programs get much more complicated, we should see how C++ handles ***comments***. Comments help the human readers of our programs. They are typically used to summarize an algorithm, identify the purpose of a variable, or clarify an otherwise obscure segment of code. The compiler ignores comments, so they have no effect on the program's behavior or performance.

Although the compiler ignores comments, readers of our code do not. Programmers tend to believe comments even when other parts of the system documentation are out of date. An incorrect comment is worse than no comment at all because it may mislead the reader. When you change your code, be sure to update the comments, too!

Kinds of Comments in C++

There are two kinds of comments in C++: single-line and paired. A single-line comment starts with a double slash (//) and ends with a newline. Everything to the right of the slashes on the current line is ignored by the compiler. A comment of this kind can contain any text, including additional double slashes.

The other kind of comment uses two delimiters (/* and */) that are inherited from C. Such comments begin with a /* and end with the next */. These comments can include anything that is not a */, including newlines. The compiler treats everything that falls between the /* and */ as part of the comment.

A comment pair can be placed anywhere a tab, space, or newline is permitted. Comment pairs can span multiple lines of a program but are not required to do so. When a comment pair does span multiple lines, it is often a good idea to indicate visually that the inner lines are part of a multiline comment. Our style is to begin each line in the comment with an asterisk, thus indicating that the entire range is part of a multiline comment.

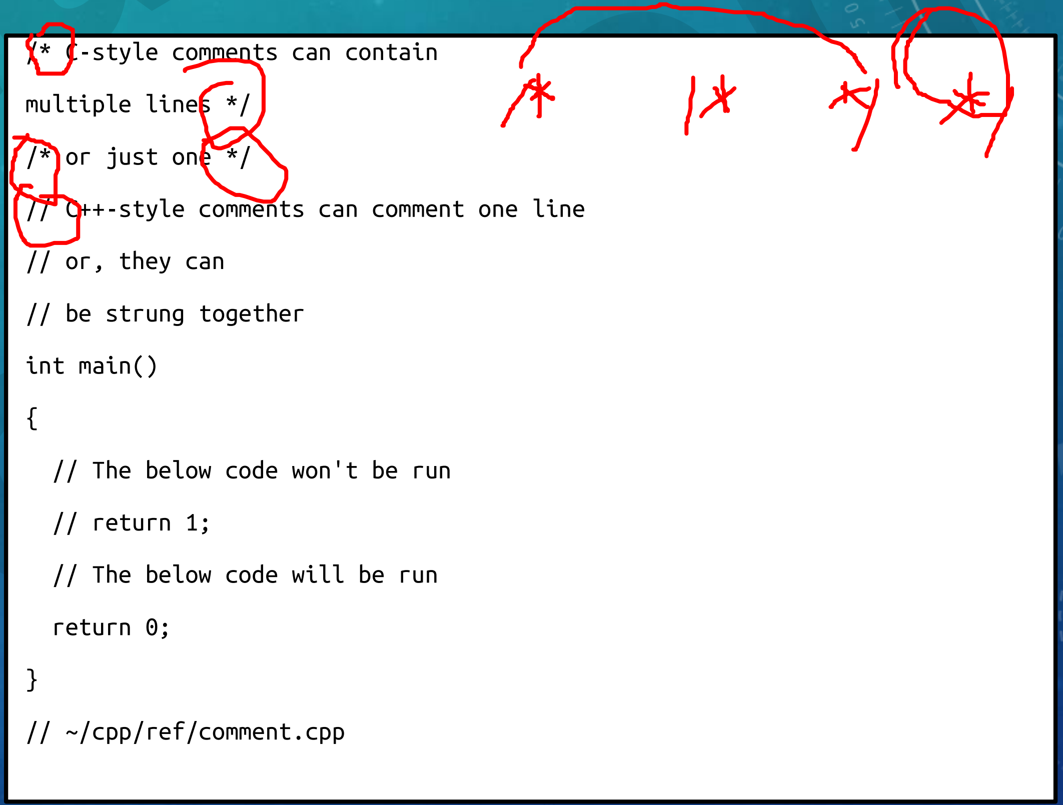
Programs typically contain a mixture of both comment forms. Comment pairs generally are used for multiline explanations, whereas double-slash comments tend to be used for half-line and single-line remarks:

Click here to view code image

```
#include <iostream>
/*
 * Simple main function:
```

COMMENTS

- C-style
  - C-style comments are usually used to comment large blocks of text, however, they can be used to comment single lines. To insert a C-style comment, simply surround text with /* and */; this will cause the contents of the comment to be ignored by the compiler. Although it is not part of the C++ standard, /** and */ are often used to indicate documentation blocks; this is legal because the second asterisk is simply treated as part of the comment. C-style comments cannot be nested.
- C++-style
  - C++-style comments are usually used to comment single lines, however, multiple C++-style comments can be placed together to form multi-line comments. C++-style comments tell the compiler to ignore all content between // and a new line.
- Source: https://en.cppreference.com/w/cpp/comment



```
/* C-style comments can contain
multiple lines */
/* or just one */
// C++-style comments can comment one line
// or, they can
// be strung together
int main()
{
  // The below code won't be run
  // return 1;
  // The below code will be run
  return 0;
}
// ~/cpp/ref/comment.cpp
```

COMMENTS: TIP

- Because comments are removed before the preprocessor stage, a macro cannot be used to form a comment and an unterminated C-style comment doesn't spill over from an #include'd file.
- Besides commenting out, other mechanisms used for source code exclusion are

```
#if 0
    std::cout << "this will not be executed or even compiled\n";
#endif
```

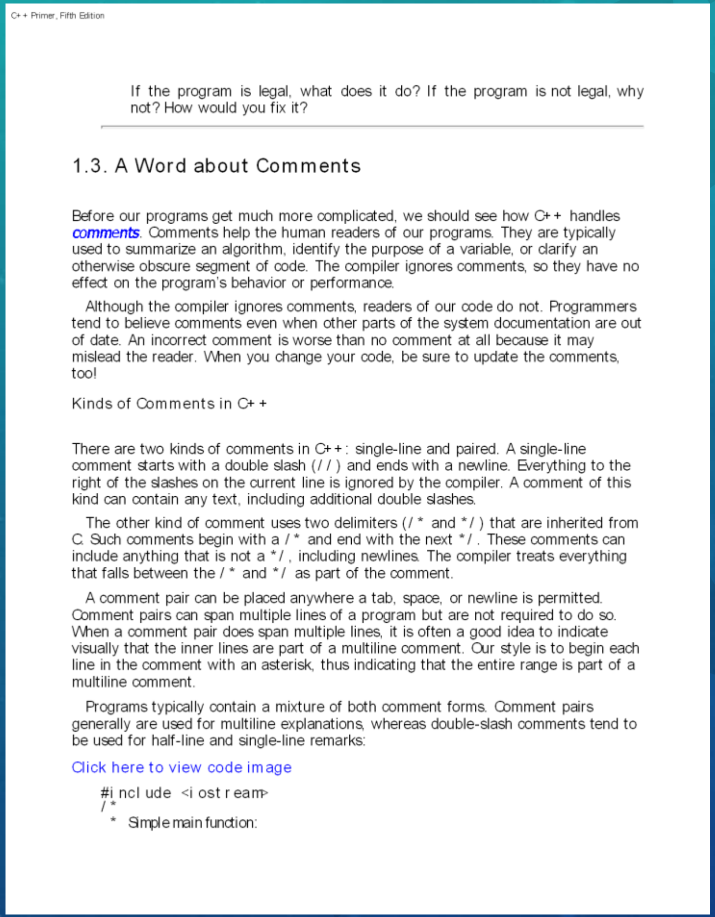
- and

```
if(false) {
    std::cout << "this will not be executed\n"
}
```

#ifdef

Back to C++ Primer:

1.3. A WORD ABOUT COMMENTS (P35 – P37)



C++ Primer, Fifth Edition

If the program is legal, what does it do? If the program is not legal, why not? How would you fix it?

1.3. A Word about Comments

Before our programs get much more complicated, we should see how C++ handles *comments*. Comments help the human readers of our programs. They are typically used to summarize an algorithm, identify the purpose of a variable, or clarify an otherwise obscure segment of code. The compiler ignores comments, so they have no effect on the program's behavior or performance.

Although the compiler ignores comments, readers of our code do not. Programmers tend to believe comments even when other parts of the system documentation are out of date. An incorrect comment is worse than no comment at all because it may mislead the reader. When you change your code, be sure to update the comments, too!

Kinds of Comments in C++

There are two kinds of comments in C++: single-line and paired. A single-line comment starts with a double slash (//) and ends with a newline. Everything to the right of the slashes on the current line is ignored by the compiler. A comment of this kind can contain any text, including additional double slashes.

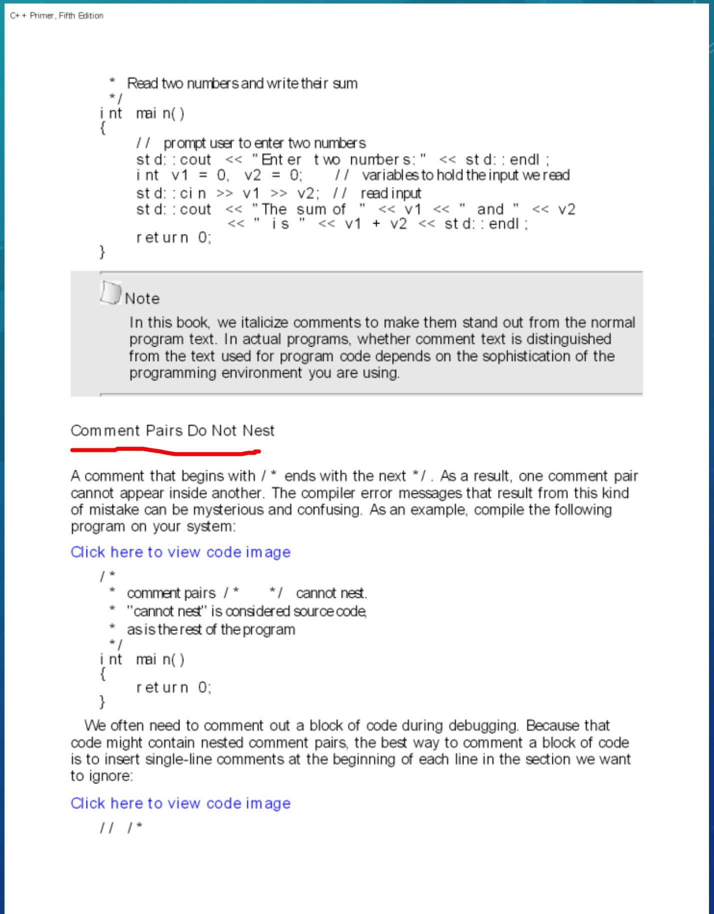
The other kind of comment uses two delimiters (/* and */) that are inherited from C. Such comments begin with a /* and end with the next */. These comments can include anything that is not a */, including newlines. The compiler treats everything that falls between the /* and */ as part of the comment.

A comment pair can be placed anywhere a tab, space, or newline is permitted. Comment pairs can span multiple lines of a program but are not required to do so. When a comment pair does span multiple lines, it is often a good idea to indicate visually that the inner lines are part of a multiline comment. Our style is to begin each line in the comment with an asterisk, thus indicating that the entire range is part of a multiline comment.

Programs typically contain a mixture of both comment forms. Comment pairs generally are used for multiline explanations, whereas double-slash comments tend to be used for half-line and single-line remarks:

Click here to view code image

```
#include <iostream>
/*
 * Simple main function:
```



C++ Primer, Fifth Edition

```
 * Read two numbers and write their sum
 */
int main()
{
    // prompt user to enter two numbers
    std::cout << "Enter two numbers:" << std::endl;
    int v1 = 0, v2 = 0;   // variables to hold the input we read
    std::cin >> v1 >> v2; // read input
    std::cout << "The sum of " << v1 << " and " << v2
              << " is " << v1 + v2 << std::endl;
    return 0;
}
```

Note

In this book, we italicize comments to make them stand out from the normal program text. In actual programs, whether comment text is distinguished from the text used for program code depends on the sophistication of the programming environment you are using.

Comment Pairs Do Not Nest

A comment that begins with /* ends with the next */. As a result, one comment pair cannot appear inside another. The compiler error messages that result from this kind of mistake can be mysterious and confusing. As an example, compile the following program on your system:

Click here to view code image

```
/*
 * comment pairs /*   */ cannot nest.
 * ''cannot nest'' is considered source code,
 * as is the rest of the program
 */
int main()
{
    return 0;
}
```

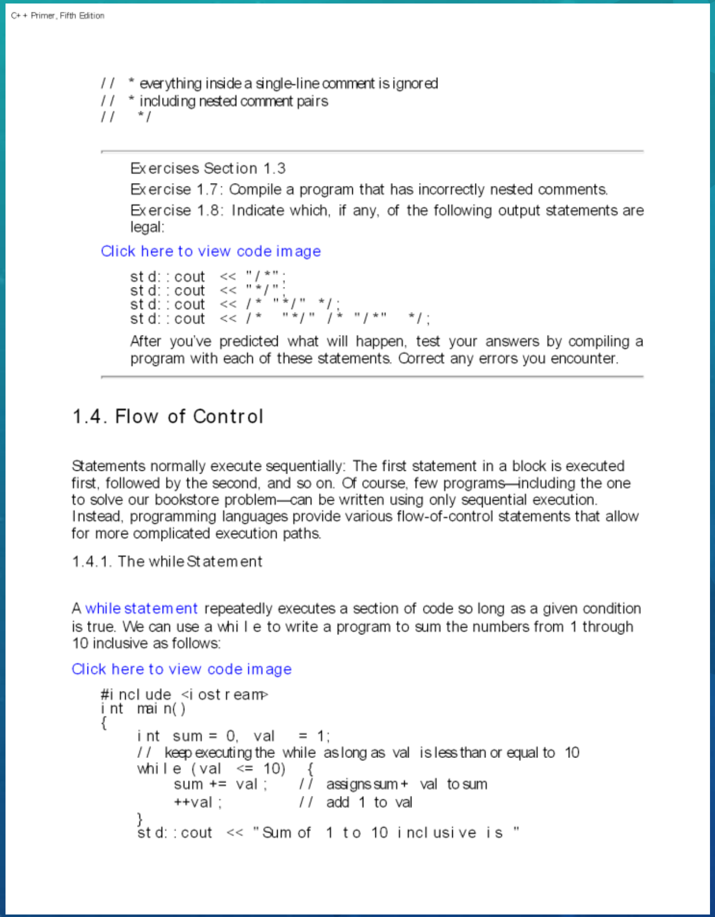
We often need to comment out a block of code during debugging. Because that code might contain nested comment pairs, the best way to comment a block of code is to insert single-line comments at the beginning of each line in the section we want to ignore:

Click here to view code image

```
// /*
```

Back to C++ Primer:

1.3. A WORD ABOUT COMMENTS (P35 – P37)



C++ Primer, Fifth Edition

```
// * everything inside a single-line comment is ignored
// * including nested comment pairs
//   */
```

Exercises Section 1.3

Exercise 1.7: Compile a program that has incorrectly nested comments.

Exercise 1.8: Indicate which, if any, of the following output statements are legal:

Click here to view code image

```
std::cout << "/*";
std::cout << "*/";
std::cout << /* "*/" */;
std::cout << /*  "*/" /* "/*"  */;
```

After you've predicted what will happen, test your answers by compiling a program with each of these statements. Correct any errors you encounter.

1.4. Flow of Control

Statements normally execute sequentially: The first statement in a block is executed first, followed by the second, and so on. Of course, few programs—including the one to solve our bookstore problem—can be written using only sequential execution. Instead, programming languages provide various flow-of-control statements that allow for more complicated execution paths.

1.4.1. The while Statement

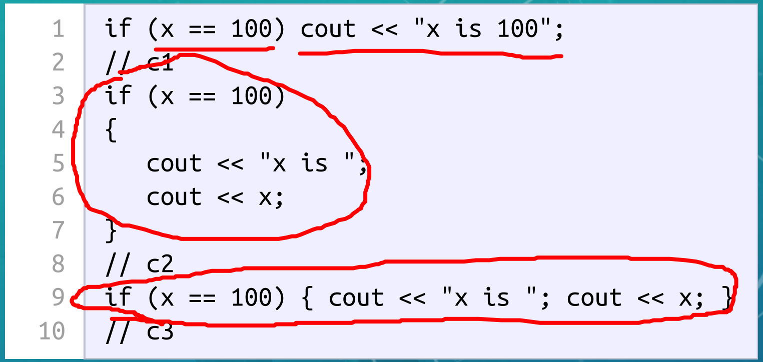
A while statement repeatedly executes a section of code so long as a given condition is true. We can use a while to write a program to sum the numbers from 1 through 10 inclusive as follows:

Click here to view code image

```
#include <iostream>
int main()
{
    int sum = 0, val = 1;
    // keep executing the while as long as val is less than or equal to 10
    while (val <= 10) {
        sum += val;     // assigns sum + val to sum
        ++val;          // add 1 to val
    }
    std::cout << "Sum of 1 to 10 inclusive is "
```

STATEMENTS AND FLOW CONTROL

- A simple C++ statement is each of the individual instructions of a program, like the variable declarations and expressions seen in previous sections. They always end with a semicolon (;), and are executed in the same order in which they appear in a program.
- But programs are not limited to a linear sequence of statements. During its process, a program may repeat segments of code, or take decisions and bifurcate. For that purpose, C++ provides flow control statements that serve to specify what has to be done by our program, when, and under which circumstances.
- Many of the flow control statements explained in this section require a generic (sub)statement as part of its syntax. This statement may either be a simple C++ statement, -such as a single instruction, terminated with a semicolon (;) - or a compound statement. A compound statement is a group of statements (each of them terminated by its own semicolon), but all grouped together in a block, enclosed in curly braces: {}:
- `{ statement1; statement2; statement3; }`
- The entire block is considered a single statement (composed itself of multiple substatements). Whenever a generic statement is part of the syntax of a flow control statement, this can either be a simple statement or a compound statement.

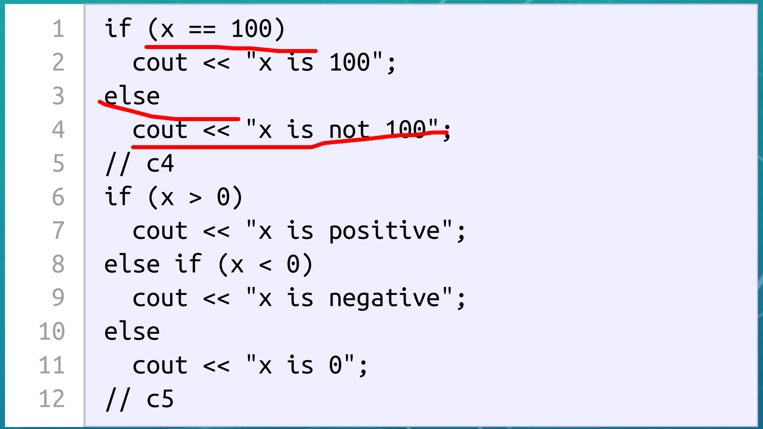


```
if (x == 100) cout << "x is 100";
// c1
if (x == 100)
{
   cout << "x is ";
   cout << x;
}
// c2
if (x == 100) { cout << "x is "; cout << x; }
// c3
```

SELECTION STATEMENTS: IF AND ELSE

- The if keyword is used to execute a statement or block, if, and only if, a condition is fulfilled. Its syntax is:
- `if (condition) statement`
- Here, condition is the expression that is being evaluated. If this condition is true, statement is executed. If it is false, statement is not executed (it is simply ignored), and the program continues right after the entire selection statement.
- For example, the following code fragment prints the message (x is 100), only if the value stored in the x variable is indeed 100: (c1)
- If x is not exactly 100, this statement is ignored, and nothing is printed.
- If you want to include more than a single statement to be executed when the condition is fulfilled, these statements shall be enclosed in braces ({}), forming a block: (c2)
- As usual, indentation and line breaks in the code have no effect, so the above code is equivalent to: (c3)

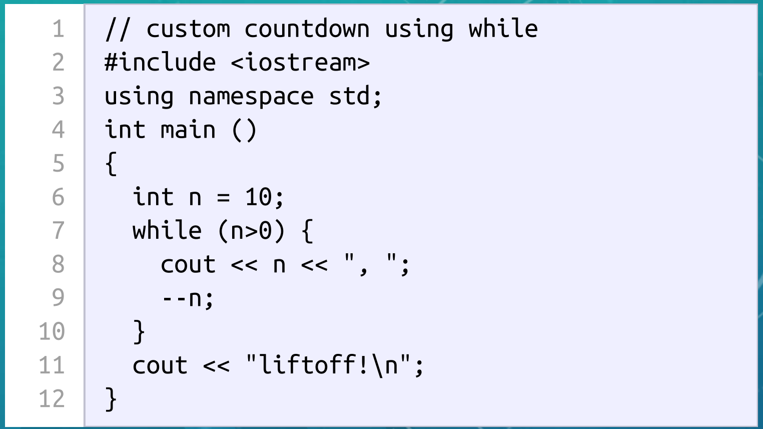
SELECTION STATEMENTS: IF AND ELSE



```
if (x == 100)
  cout << "x is 100";
else
  cout << "x is not 100";
// c4
if (x > 0)
  cout << "x is positive";
else if (x < 0)
  cout << "x is negative";
else
  cout << "x is 0";
// c5
```

- Selection statements with if can also specify what happens when the condition is not fulfilled, by using the else keyword to introduce an alternative statement. Its syntax is:
- `if (condition) statement1 else statement2`
- where statement1 is executed in case condition is true, and in case it is not, statement2 is executed.
- For example: (c4)
- This prints x is 100, if indeed x has a value of 100, but if it does not, and only if it does not, it prints x is not 100 instead.
- Several if + else structures can be concatenated with the intention of checking a range of values. For example: (c5)
- This prints whether x is positive, negative, or zero by concatenating two if-else structures. Again, it would have also been possible to execute more than a single statement per case by grouping them into blocks enclosed in braces: {}.
- All the codes: ~/cpp/ref/if.cpp

ITERATION STATEMENTS (LOOPS)

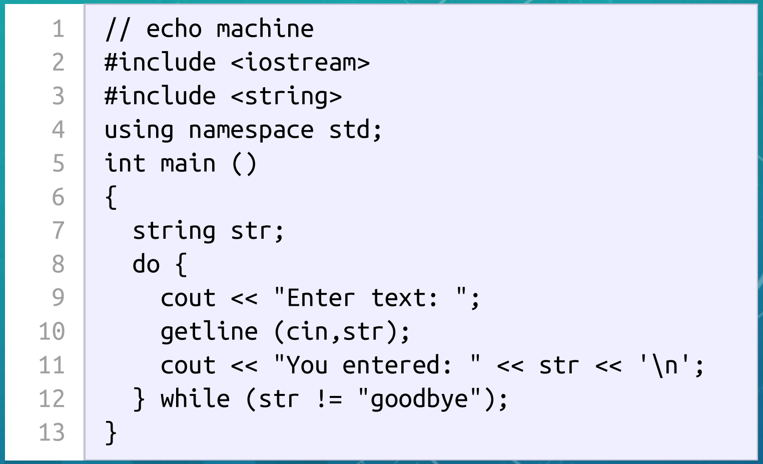


```
// custom countdown using while
#include <iostream>
using namespace std;
int main ()
{
  int n = 10;
  while (n>0) {
    cout << n << ", ";
    --n;
  }
  cout << "liftoff!\n";
}
```

- Loops repeat a statement a certain number of times, or while a condition is fulfilled. They are introduced by the keywords while, do, and for.
- The while loop
  - The simplest kind of loop is the while-loop. Its syntax is:
  - while (expression) statement
  - The while-loop simply repeats statement while expression is true. If, after any execution of statement, expression is no longer true, the loop ends, and the program continues right after the loop. For example, let's have a look at a countdown using a while-loop: (~/cpp/ref/while.cpp)
  - The first statement in main sets n to a value of 10. This is the first number in the countdown. Then the while-loop begins: if this value fulfills the condition n>0 (that n is greater than zero), then the block that follows the condition is executed, and repeated for as long as the condition (n>0) remains being true.

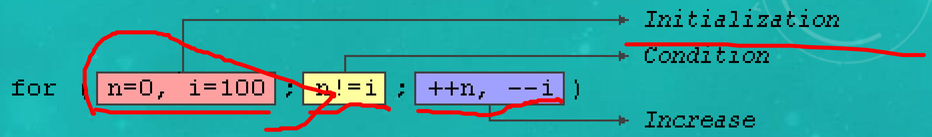
while (true)

ITERATION STATEMENTS (LOOPS)



```
// echo machine
#include <iostream>
#include <string>
using namespace std;
int main ()
{
  string str;
  do {
    cout << "Enter text: ";
    getline (cin,str);
    cout << "You entered: " << str << '\n';
  } while (str != "goodbye");
}
```

- The do-while loop
  - A very similar loop is the do-while loop, whose syntax is:
  - do statement while (condition);
  - It behaves like a while-loop, except that condition is evaluated after the execution of statement instead of before, guaranteeing at least one execution of statement, even if condition is never fulfilled. For example, the following example program echoes any text the user introduces until the user enters goodbye: (~/cpp/ref/do-while.cpp)
  - The do-while loop is usually preferred over a while-loop when the statement needs to be executed at least once, such as when the condition that is checked to end of the loop is determined within the loop statement itself. In the previous example, the user input within the block is what will determine if the loop ends. And thus, even if the user wants to end the loop as soon as possible by entering goodbye, the block in the loop needs to be executed at least once to prompt for input, and the condition can, in fact, only be determined after it is executed.



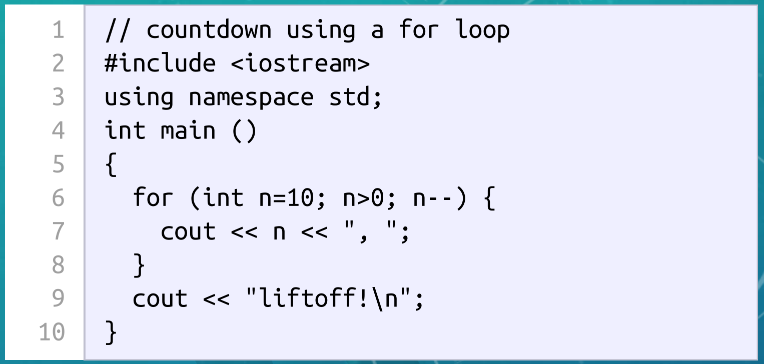
for (n=0, i=100 ; n!=i ; ++n, --i)

Initialization

Condition

Increase

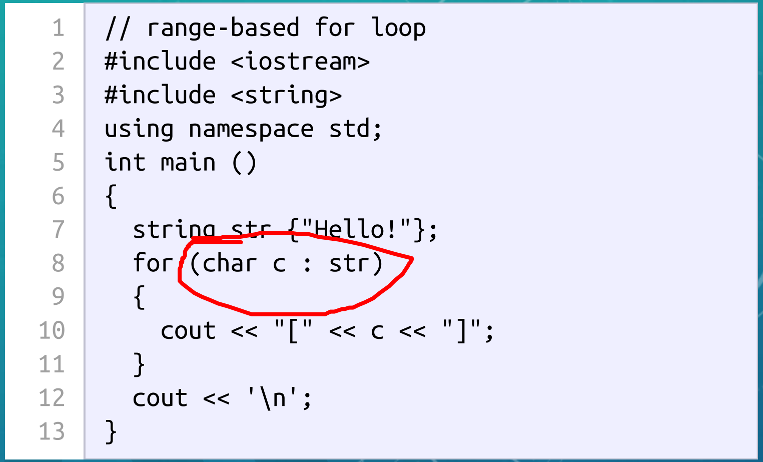
ITERATION STATEMENTS (LOOPS)



```
// countdown using a for loop
#include <iostream>
using namespace std;
int main ()
{
  for (int n=10; n>0; n--) {
    cout << n << ", ";
  }
  cout << "liftoff!\n";
}
```

- The for loop
  - The for loop is designed to iterate a number of times. Its syntax is:
  - for (initialization; condition; increase) statement;
  - Like the while-loop, this loop repeats statement while condition is true. But, in addition, the for loop provides specific locations to contain an initialization and an increase expression, executed before the loop begins the first time, and after each iteration, respectively. Therefore, it is especially useful to use counter variables as condition.
  - It works in the following way:
    1. initialization is executed. Generally, this declares a counter variable, and sets it to some initial value. This is executed a single time, at the beginning of the loop.
    2. condition is checked. If it is true, the loop continues; otherwise, the loop ends, and statement is skipped, going directly to step 5.
    3. statement is executed. As usual, it can be either a single statement or a block enclosed in curly braces { }.
    4. increase is executed, and the loop gets back to step 2.
    5. the loop ends: execution continues by the next statement after it.
  - Here is the countdown example using a for loop: (~/cpp/ref/for.cpp)
  - The three fields in a for-loop are optional. They can be left empty, but in all cases the semicolon signs between them are required. For example, for (;n<10;) is a loop without initialization or increase (equivalent to a while-loop); and for (;n<10;++n) is a loop with increase, but no initialization (maybe because the variable was already initialized before the loop). A loop with no condition is equivalent to a loop with true as condition (i.e., an infinite loop).
  - This loop will execute 50 times if neither n or i are modified within the lon starts with a value of 0, and i with 100, the condition is n!=i (i.e., that n is not equal to i). Because n is increased by one, and i decreased by one on each iteration, the loop's condition will become false after the 50th iteration, when both n and i are equal to 50.op: (Image above)

ITERATION STATEMENTS (LOOPS)

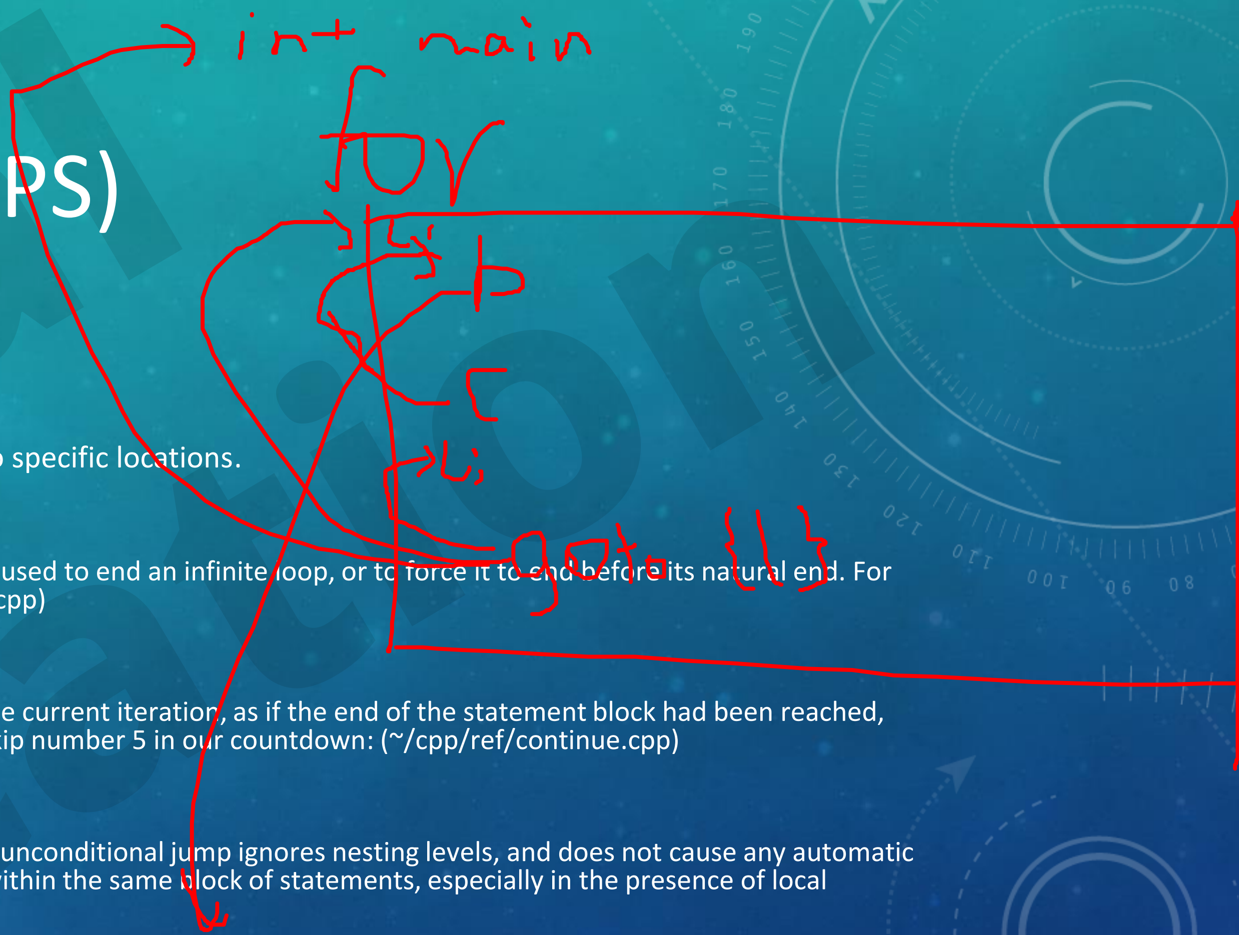


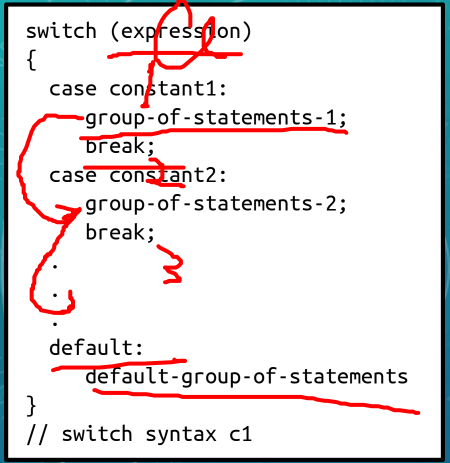
```
// range-based for loop
#include <iostream>
#include <string>
using namespace std;
int main ()
{
  string str {"Hello!"};
  for (char c : str)
  {
    cout << "[" << c << "]";
  }
  cout << '\n';
}
```

- Range-based for loop
  - The for-loop has another syntax, which is used exclusively with ranges:
  - `for ( declaration : range ) statement;`
  - This kind of for loop iterates over all the elements in range, where declaration declares some variable able to take the value of an element in this range. Ranges are sequences of elements, including arrays, containers, and any other type supporting the functions begin and end; Most of these types have not yet been introduced in this tutorial, but we are already acquainted with at least one kind of range: strings, which are sequences of characters.
  - An example of range-based for loop using strings: (~/cpp/ref/ranged-for.cpp)

ITERATION STATEMENTS (LOOPS)

- Jump statements
  - Jump statements allow altering the flow of a program by performing jumps to specific locations.
  - The break statement
    - break leaves a loop, even if the condition for its end is not fulfilled. It can be used to end an infinite loop, or to force it to end before its natural end. For example, let's stop the countdown before its natural end: (~/cpp/ref/break.cpp)
  - The continue statement
    - The continue statement causes the program to skip the rest of the loop in the current iteration, as if the end of the statement block had been reached, causing it to jump to the start of the following iteration. For example, let's skip number 5 in our countdown: (~/cpp/ref/continue.cpp)
  - The goto statement
    - goto allows to make an absolute jump to another point in the program. This unconditional jump ignores nesting levels, and does not cause any automatic stack unwinding. Therefore, it is a feature to use with care, and preferably within the same block of statements, especially in the presence of local variables.
    - The destination point is identified by a label, which is then used as an argument for the goto statement. A label is made of a valid identifier followed by a colon (:).
    - goto is generally deemed a low-level feature, with no particular use cases in modern higher-level programming paradigms generally used with C++. But, just as an example, here is a version of our countdown loop using goto: (~/cpp/ref/goto.cpp)





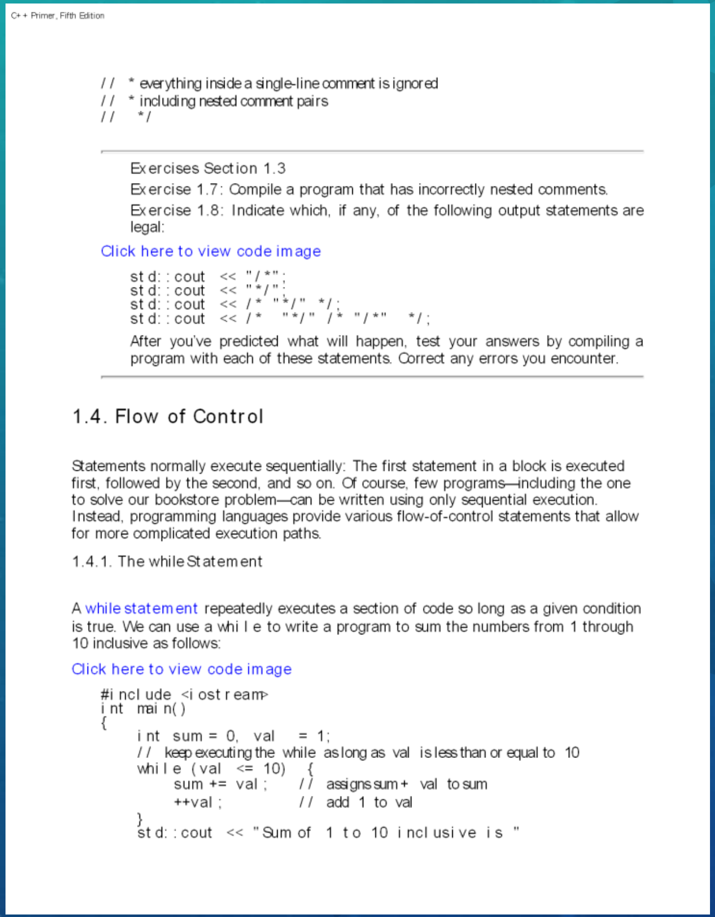
```
switch (expression)
{
  case constant1:
     group-of-statements-1;
     break;
  case constant2:
     group-of-statements-2;
     break;
  .
  .
  .
  default:
     default-group-of-statements
}
// switch syntax c1
```

ITERATION STATEMENTS (LOOPS)

- Another selection statement: switch.
  - The syntax of the switch statement is a bit peculiar. Its purpose is to check for a value among a number of possible constant expressions. It is something similar to concatenating if-else statements, but limited to constant expressions. Its most typical syntax is: (c1)
  - It works in the following way: switch evaluates expression and checks if it is equivalent to constant1; if it is, it executes group-of-statements-1 until it finds the break statement. When it finds this break statement, the program jumps to the end of the entire switch statement (the closing brace).
  - If expression was not equal to constant1, it is then checked against constant2. If it is equal to this, it executes group-of-statements-2 until a break is found, when it jumps to the end of the switch.
  - Finally, if the value of expression did not match any of the previously specified constants (there may be any number of these), the program executes the statements included after the default: label, if it exists (since it is optional).
- Both of the following code fragments have the same behavior, demonstrating the if-else equivalent of a switch statement: (Table below)
- The switch statement has a somewhat peculiar syntax inherited from the early times of the first C compilers, because it uses labels instead of blocks. In the most typical use (shown above), this means that break statements are needed after each group of statements for a particular label. If break is not included, all statements following the case (including those under any other labels) are also executed, until the end of the switch block or a jump statement (such as break) is reached.

switch example	if-else equivalent
`switch (x) { case 1: cout << "x is 1"; break; case 2: cout << "x is 2"; break; default: cout << "value of x unknown"; }`	`if (x == 1) { cout << "x is 1"; } else if (x == 2) { cout << "x is 2"; } else { cout << "value of x unknown"; }`

1.4 FLOW TO CONTROL (P37 – P46)



C++ Primer, Fifth Edition

```
//  * everything inside a single-line comment is ignored
//  * including nested comment pairs
//    */
```

Exercises Section 1.3

Exercise 1.7: Compile a program that has incorrectly nested comments.

Exercise 1.8: Indicate which, if any, of the following output statements are legal:

Click here to view code image

```
std::cout << "/*";
std::cout << "*/";
std::cout << /* "*/" */;
std::cout << /*  "*/" /* "/*"  */;
```

After you've predicted what will happen, test your answers by compiling a program with each of these statements. Correct any errors you encounter.

1.4. Flow of Control

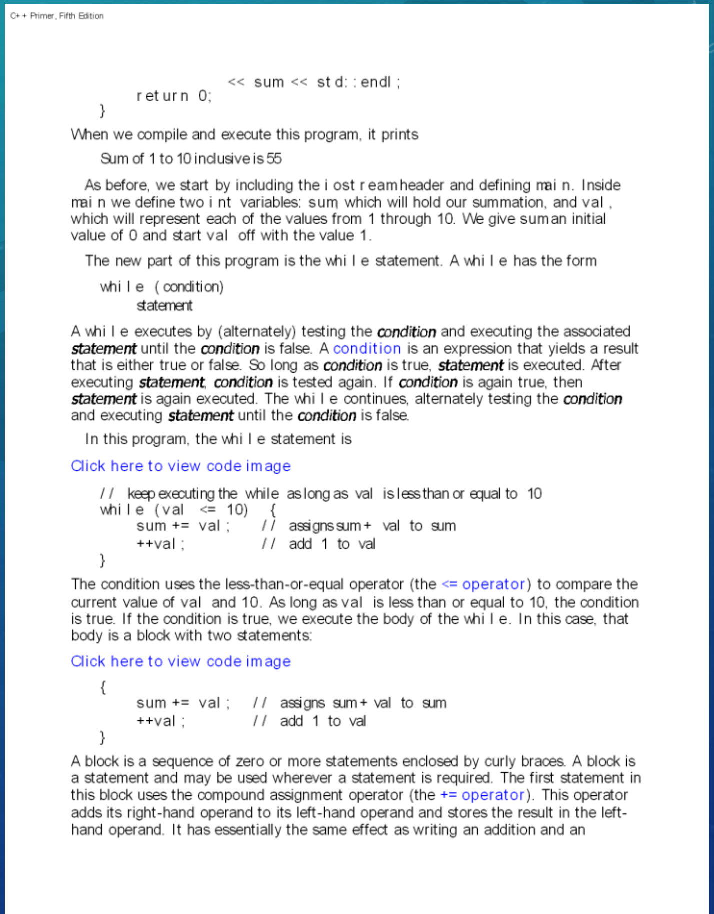
Statements normally execute sequentially: The first statement in a block is executed first, followed by the second, and so on. Of course, few programs—including the one to solve our bookstore problem—can be written using only sequential execution. Instead, programming languages provide various flow-of-control statements that allow for more complicated execution paths.

1.4.1. The while Statement

A while statement repeatedly executes a section of code so long as a given condition is true. We can use a while to write a program to sum the numbers from 1 through 10 inclusive as follows:

Click here to view code image

```
#include <iostream>
int main()
{
    int sum = 0, val = 1;
    // keep executing the while as long as val is less than or equal to 10
    while (val <= 10) {
        sum += val;    // assigns sum + val to sum
        ++val;         // add 1 to val
    }
    std::cout << "Sum of 1 to 10 inclusive is "
```



C++ Primer, Fifth Edition

```
                 << sum << std::endl;
    return 0;
}
```

When we compile and execute this program, it prints

```
Sum of 1 to 10 inclusive is 55
```

As before, we start by including the iostream header and defining main. Inside main we define two int variables: sum, which will hold our summation, and val, which will represent each of the values from 1 through 10. We give sum an initial value of 0 and start val off with the value 1.

The new part of this program is the while statement. A while has the form

```
while (condition)
    statement
```

A while executes by (alternately) testing the ***condition*** and executing the associated ***statement*** until the ***condition*** is false. A condition is an expression that yields a result that is either true or false. So long as ***condition*** is true, ***statement*** is executed. After executing ***statement***, ***condition*** is tested again. If ***condition*** is again true, then ***statement*** is again executed. The while continues, alternately testing the ***condition*** and executing ***statement*** until the ***condition*** is false.

In this program, the while statement is

Click here to view code image

```
// keep executing the while as long as val is less than or equal to 10
while (val <= 10) {
    sum += val;    // assigns sum + val to sum
    ++val;         // add 1 to val
}
```

The condition uses the less-than-or-equal operator (the <= operator) to compare the current value of val and 10. As long as val is less than or equal to 10, the condition is true. If the condition is true, we execute the body of the while. In this case, that body is a block with two statements:

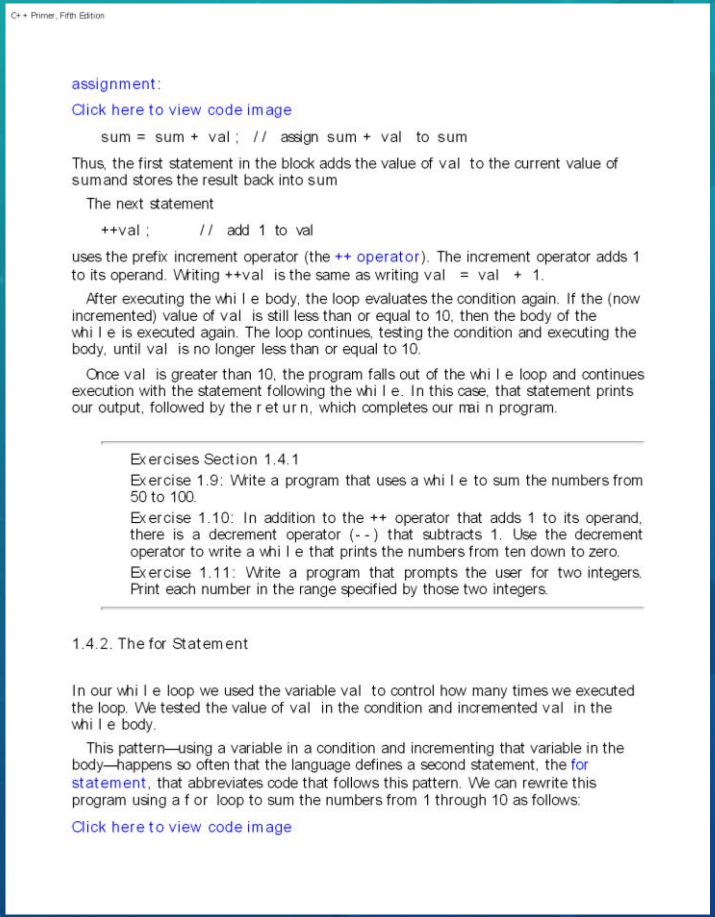
Click here to view code image

```
{
    sum += val;   // assigns sum + val to sum
    ++val;        // add 1 to val
}
```

A block is a sequence of zero or more statements enclosed by curly braces. A block is a statement and may be used wherever a statement is required. The first statement in this block uses the compound assignment operator (the += operator). This operator adds its right-hand operand to its left-hand operand and stores the result in the left-hand operand. It has essentially the same effect as writing an addition and an

Back to C++ Primer:

1.4 FLOW TO CONTROL (P37 – P46)



C++ Primer, Fifth Edition

assignment:

Click here to view code image

```
sum = sum + val;  // assign sum + val to sum
```

Thus, the first statement in the block adds the value of val to the current value of sum and stores the result back into sum

The next statement

```
++val;        // add 1 to val
```

uses the prefix increment operator (the ++ operator). The increment operator adds 1 to its operand. Writing ++val is the same as writing val = val + 1.

After executing the while body, the loop evaluates the condition again. If the (now incremented) value of val is still less than or equal to 10, then the body of the while is executed again. The loop continues, testing the condition and executing the body, until val is no longer less than or equal to 10.

Once val is greater than 10, the program falls out of the while loop and continues execution with the statement following the while. In this case, that statement prints our output, followed by the return, which completes our main program.

Exercises Section 1.4.1

Exercise 1.9: Write a program that uses a while to sum the numbers from 50 to 100.

Exercise 1.10: In addition to the ++ operator that adds 1 to its operand, there is a decrement operator (--) that subtracts 1. Use the decrement operator to write a while that prints the numbers from ten down to zero.

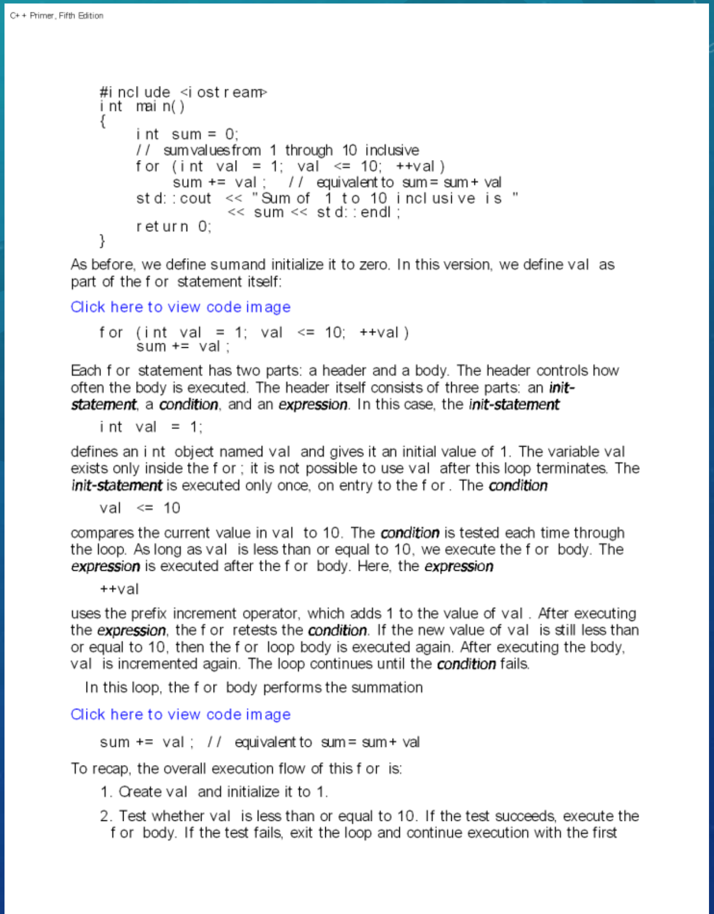
Exercise 1.11: Write a program that prompts the user for two integers. Print each number in the range specified by those two integers.

1.4.2. The for Statement

In our while loop we used the variable val to control how many times we executed the loop. We tested the value of val in the condition and incremented val in the while body.

This pattern—using a variable in a condition and incrementing that variable in the body—happens so often that the language defines a second statement, the for statement, that abbreviates code that follows this pattern. We can rewrite this program using a for loop to sum the numbers from 1 through 10 as follows:

Click here to view code image



C++ Primer, Fifth Edition

```
#include <iostream>
int main()
{
    int sum = 0;
    // sum values from 1 through 10 inclusive
    for (int val = 1; val <= 10; ++val)
        sum += val;  // equivalent to sum = sum + val
    std::cout << "Sum of 1 to 10 inclusive is "
              << sum << std::endl;
    return 0;
}
```

As before, we define sum and initialize it to zero. In this version, we define val as part of the for statement itself:

Click here to view code image

```
for (int val = 1; val <= 10; ++val)
    sum += val;
```

Each for statement has two parts: a header and a body. The header controls how often the body is executed. The header itself consists of three parts: an ***init-statement***, a ***condition***, and an ***expression***. In this case, the ***init-statement***

```
int val = 1;
```

defines an int object named val and gives it an initial value of 1. The variable val exists only inside the for; it is not possible to use val after this loop terminates. The ***init-statement*** is executed only once, on entry to the for. The ***condition***

```
val <= 10
```

compares the current value in val to 10. The ***condition*** is tested each time through the loop. As long as val is less than or equal to 10, we execute the for body. The ***expression*** is executed after the for body. Here, the ***expression***

```
++val
```

uses the prefix increment operator, which adds 1 to the value of val. After executing the ***expression***, the for retests the ***condition***. If the new value of val is still less than or equal to 10, then the for loop body is executed again. After executing the body, val is incremented again. The loop continues until the ***condition*** fails.

In this loop, the for body performs the summation

Click here to view code image

```
sum += val;  // equivalent to sum = sum + val
```

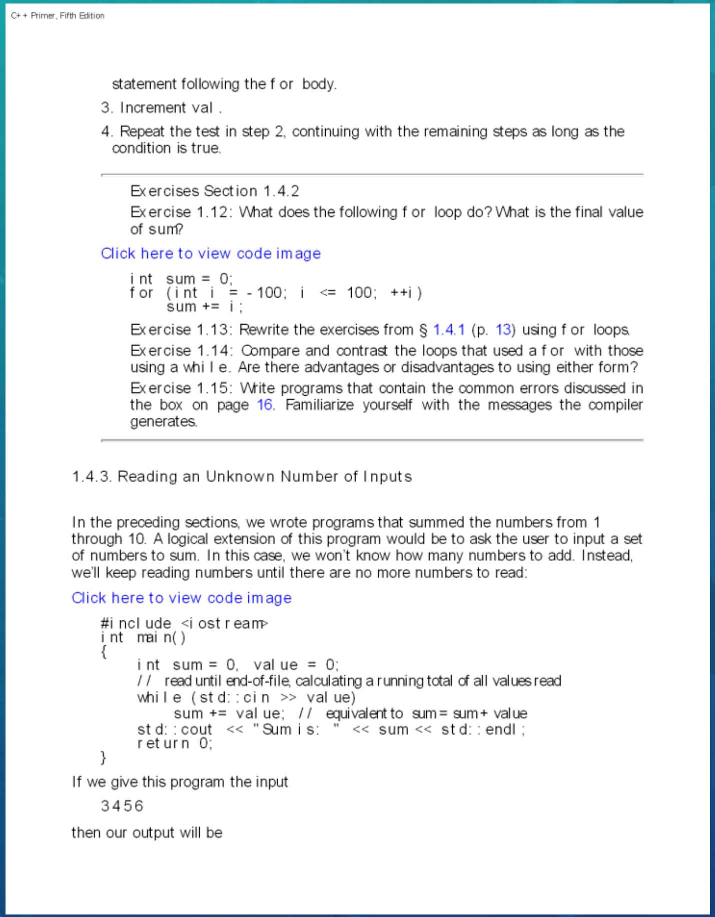
To recap, the overall execution flow of this for is:

1. Create val and initialize it to 1.

2. Test whether val is less than or equal to 10. If the test succeeds, execute the for body. If the test fails, exit the loop and continue execution with the first

Back to C++ Primer:

1.4 FLOW TO CONTROL (P37 – P46)



C++ Primer, Fifth Edition

statement following the for body.

3. Increment val.

4. Repeat the test in step 2, continuing with the remaining steps as long as the condition is true.

Exercises Section 1.4.2

Exercise 1.12: What does the following for loop do? What is the final value of sum?

Click here to view code image

```
int sum = 0;
for (int i = -100; i <= 100; ++i)
    sum += i;
```

Exercise 1.13: Rewrite the exercises from § 1.4.1 (p. 13) using for loops.

Exercise 1.14: Compare and contrast the loops that used a for with those using a while. Are there advantages or disadvantages to using either form?

Exercise 1.15: Write programs that contain the common errors discussed in the box on page 16. Familiarize yourself with the messages the compiler generates.

1.4.3. Reading an Unknown Number of Inputs

In the preceding sections, we wrote programs that summed the numbers from 1 through 10. A logical extension of this program would be to ask the user to input a set of numbers to sum. In this case, we won't know how many numbers to add. Instead, we'll keep reading numbers until there are no more numbers to read:

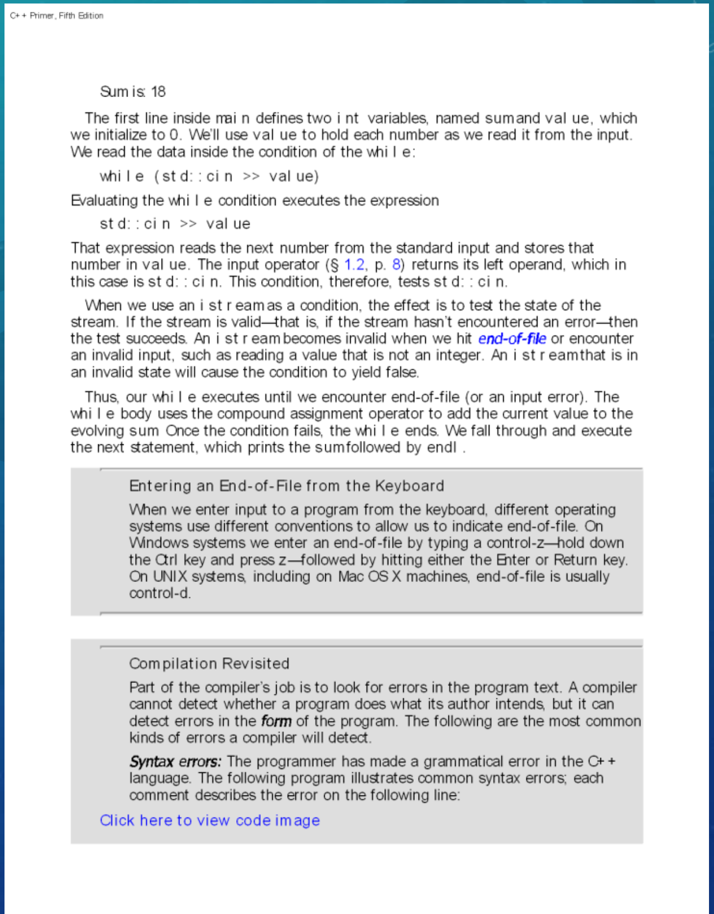
Click here to view code image

```
#include <iostream>
int main()
{
    int sum = 0, value = 0;
    // read until end-of-file, calculating a running total of all values read
    while (std::cin >> value)
        sum += value; // equivalent to sum = sum + value
    std::cout << "Sum is: " << sum << std::endl;
    return 0;
}
```

If we give this program the input

```
3456
```

then our output will be



C++ Primer, Fifth Edition

```
Sum is: 18
```

The first line inside main defines two int variables, named sum and value, which we initialize to 0. We'll use value to hold each number as we read it from the input. We read the data inside the condition of the while:

```
while (std::cin >> value)
```

Evaluating the while condition executes the expression

```
std::cin >> value
```

That expression reads the next number from the standard input and stores that number in value. The input operator (§ 1.2, p. 8) returns its left operand, which in this case is std::cin. This condition, therefore, tests std::cin.

When we use an istream as a condition, the effect is to test the state of the stream. If the stream is valid—that is, if the stream hasn't encountered an error—then the test succeeds. An istream becomes invalid when we hit ***end-of-file*** or encounter an invalid input, such as reading a value that is not an integer. An istream that is in an invalid state will cause the condition to yield false.

Thus, our while executes until we encounter end-of-file (or an input error). The while body uses the compound assignment operator to add the current value to the evolving sum. Once the condition fails, the while ends. We fall through and execute the next statement, which prints the sum followed by endl.

Entering an End-of-File from the Keyboard

When we enter input to a program from the keyboard, different operating systems use different conventions to allow us to indicate end-of-file. On Windows systems we enter an end-of-file by typing a control-z—hold down the Ctrl key and press z—followed by hitting either the Enter or Return key. On UNIX systems, including on Mac OS X machines, end-of-file is usually control-d.

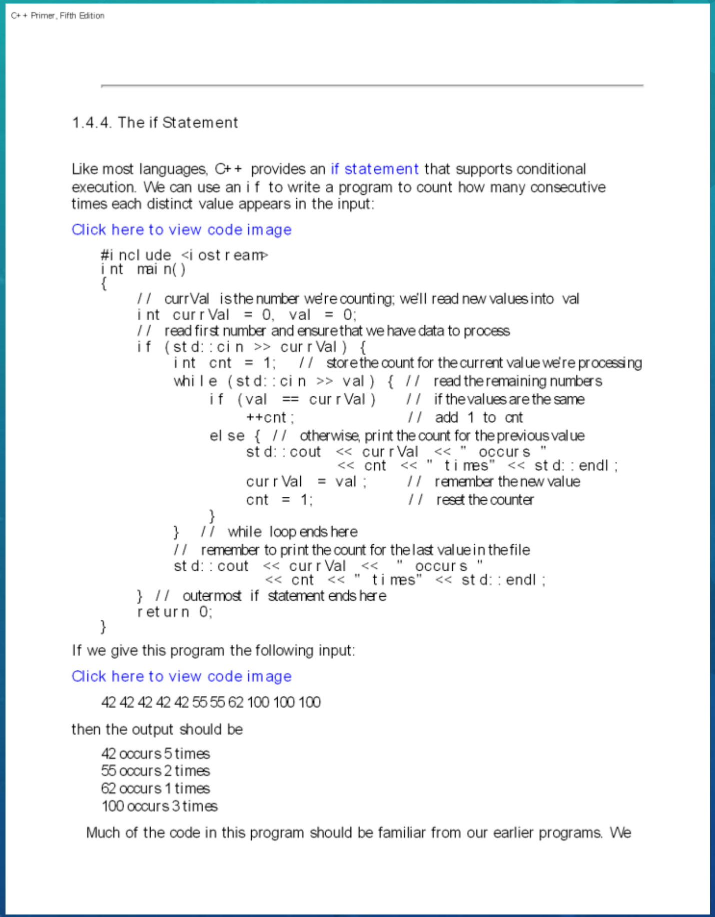
Compilation Revisited

Part of the compiler's job is to look for errors in the program text. A compiler cannot detect whether a program does what its author intends, but it can detect errors in the ***form*** of the program. The following are the most common kinds of errors a compiler will detect.

Syntax errors: The programmer has made a grammatical error in the C++ language. The following program illustrates common syntax errors; each comment describes the error on the following line:

Click here to view code image

1.4 FLOW TO CONTROL (P37 – P46)



C++ Primer, Fifth Edition

1.4.4. The if Statement

Like most languages, C++ provides an if statement that supports conditional execution. We can use an if to write a program to count how many consecutive times each distinct value appears in the input:

Click here to view code image

```
#include <iostream>
int main()
{
    // currVal is the number we're counting; we'll read new values into val
    int currVal = 0, val = 0;
    // read first number and ensure that we have data to process
    if (std::cin >> currVal) {
        int cnt = 1;  // store the count for the current value we're processing
        while (std::cin >> val) { // read the remaining numbers
            if (val == currVal)   // if the values are the same
                ++cnt;            // add 1 to cnt
            else { // otherwise, print the count for the previous value
                std::cout << currVal << " occurs "
                          << cnt << " times" << std::endl;
                currVal = val;    // remember the new value
                cnt = 1;          // reset the counter
            }
        }  // while loop ends here
        // remember to print the count for the last value in the file
        std::cout << currVal << " occurs "
                  << cnt << " times" << std::endl;
    } // outermost if statement ends here
    return 0;
}
```

If we give this program the following input:

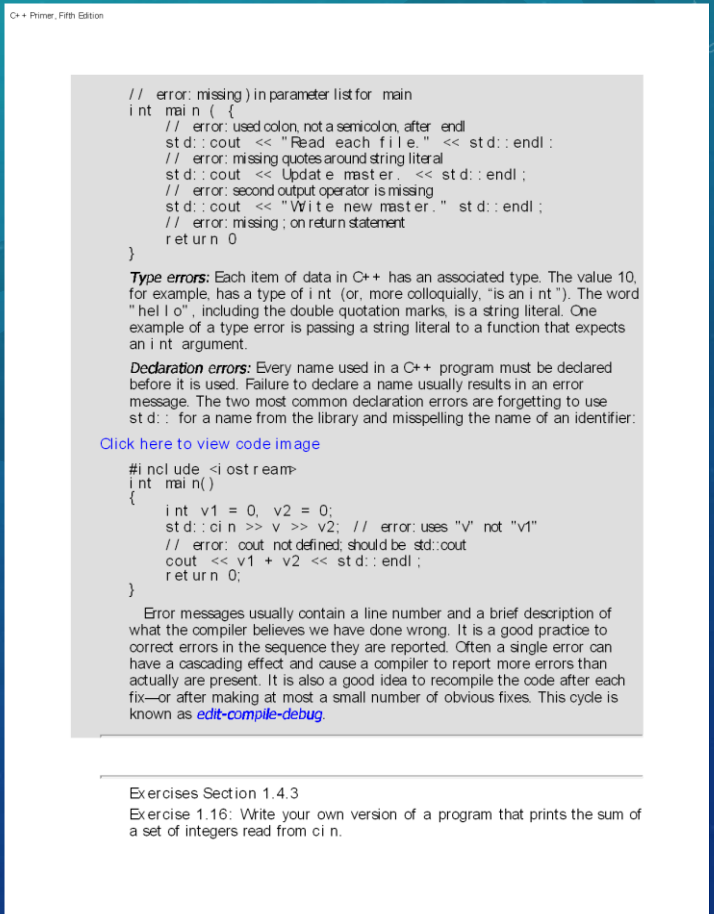
Click here to view code image

```
42 42 42 42 42 55 55 62 100 100 100
```

then the output should be

```
42 occurs 5 times
55 occurs 2 times
62 occurs 1 times
100 occurs 3 times
```

Much of the code in this program should be familiar from our earlier programs. We



C++ Primer, Fifth Edition

```
// error: missing ) in parameter list for main
int main ( {
    // error: used colon, not a semicolon, after endl
    std::cout << "Read each file." << std::endl:
    // error: missing quotes around string literal
    std::cout << Update master. << std::endl;
    // error: second output operator is missing
    std::cout << "Write new master." std::endl;
    // error: missing ; on return statement
    return 0
}
```

Type errors: Each item of data in C++ has an associated type. The value 10, for example, has a type of int (or, more colloquially, “is an int”). The word "hello", including the double quotation marks, is a string literal. One example of a type error is passing a string literal to a function that expects an int argument.

Declaration errors: Every name used in a C++ program must be declared before it is used. Failure to declare a name usually results in an error message. The two most common declaration errors are forgetting to use std:: for a name from the library and misspelling the name of an identifier:

Click here to view code image

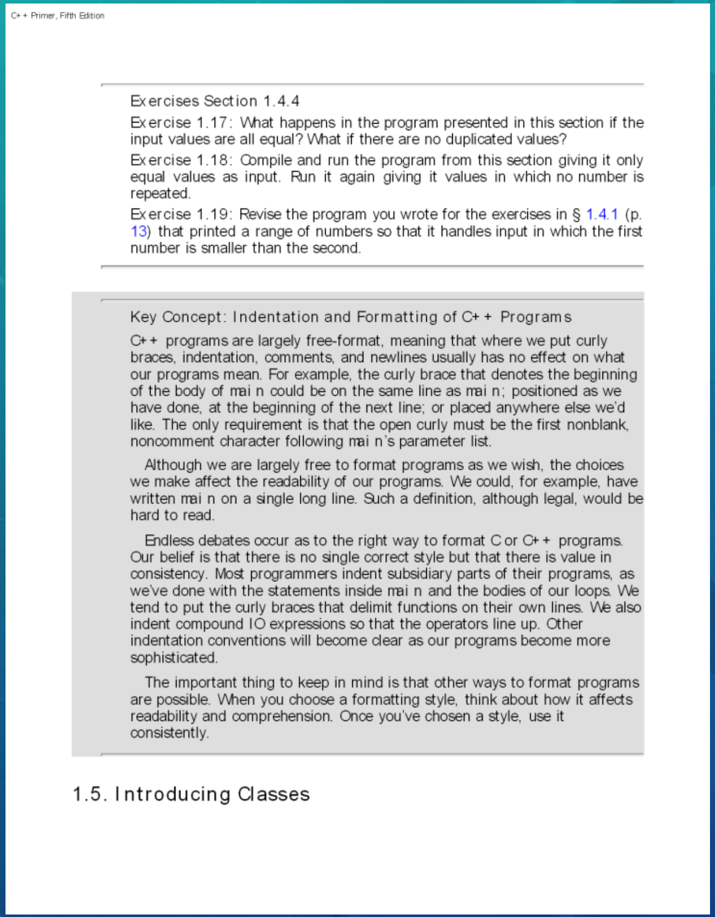
```
#include <iostream>
int main()
{
    int v1 = 0, v2 = 0;
    std::cin >> v >> v2; // error: uses "v" not "v1"
    // error: cout not defined; should be std::cout
    cout << v1 + v2 << std::endl;
    return 0;
}
```

Error messages usually contain a line number and a brief description of what the compiler believes we have done wrong. It is a good practice to correct errors in the sequence they are reported. Often a single error can have a cascading effect and cause a compiler to report more errors than actually are present. It is also a good idea to recompile the code after each fix—or after making at most a small number of obvious fixes. This cycle is known as *edit-compile-debug*.

Exercises Section 1.4.3

Exercise 1.16: Write your own version of a program that prints the sum of a set of integers read from cin.

1.4 FLOW TO CONTROL (P37 – P46)



C++ Primer, Fifth Edition

Exercises Section 1.4.4

Exercise 1.17: What happens in the program presented in this section if the input values are all equal? What if there are no duplicated values?

Exercise 1.18: Compile and run the program from this section giving it only equal values as input. Run it again giving it values in which no number is repeated.

Exercise 1.19: Revise the program you wrote for the exercises in § 1.4.1 (p. 13) that printed a range of numbers so that it handles input in which the first number is smaller than the second.

Key Concept: Indentation and Formatting of C++ Programs

C++ programs are largely free-format, meaning that where we put curly braces, indentation, comments, and newlines usually has no effect on what our programs mean. For example, the curly brace that denotes the beginning of the body of main could be on the same line as main; positioned as we have done, at the beginning of the next line; or placed anywhere else we'd like. The only requirement is that the open curly must be the first nonblank, noncomment character following main's parameter list.

Although we are largely free to format programs as we wish, the choices we make affect the readability of our programs. We could, for example, have written main on a single long line. Such a definition, although legal, would be hard to read.

Endless debates occur as to the right way to format C or C++ programs. Our belief is that there is no single correct style but that there is value in consistency. Most programmers indent subsidiary parts of their programs, as we've done with the statements inside main and the bodies of our loops. We tend to put the curly braces that delimit functions on their own lines. We also indent compound IO expressions so that the operators line up. Other indentation conventions will become clear as our programs become more sophisticated.

The important thing to keep in mind is that other ways to format programs are possible. When you choose a formatting style, think about how it affects readability and comprehension. Once you've chosen a style, use it consistently.

1.5. Introducing Classes

HOMEWORK:

- Read following chapters of *C++ Primer*:
  - 2.1 – 2.5 EXCEPT 2.5.3 → 不看
  - 3.1 – 3.6 EXCEPT 3.4
  - 4
  - 5 EXCEPT 5.6
  - 6 EXCEPT 6.1.3, 6.5

- Know how to input / output from file, use `freopen` in `<cstdio>`