# Chapter 2

# C++ Fundamentals

### 2.1 THE "Hello World" PROGRAM

Every C++ program must include the following code:

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

This is called the main() function. The program statements that are to be executed are placed between the braces. That section is called the ***body*** of the main () function. This simplest version has no statements, so it would do nothing. But it would compile and run.

**Warning:** Some pre-Standard C++ compilers require the statement

```
return 0;
```

within the body of the `main()` function.

**EXAMPLE 2.1 The "Hello World" Program**

Here is a simple program that prints a message:

```
#include <iostream>
using namespace std;
int main()
{
    cout << "Hello, World!\n";
}
```

The first line directs the precompiler to copy all the source code from the file `iostream` into this program, replacing the `#include` statement. This file is part of the ISO Standard C++ Library, so the precompiler knows where to find it. It defines various objects and functions, including the stream object `std: : cout` that is used on the fifth line. This precompiler directive is required in eveiy program that has input or output; *i.e.*, in eveiy useful program. The second line is needed to access commonly used names like `cout` that are defined in the `std` namespace.

The fifth line tells the system to print the message `"Hello, World!"`. The complete string of characters ends with the non-printing character `'\n'`, which simply directs the printer (or monitor) to move the cursor down to the beginning of the next line after printing the exclamation point (!). This non-printing character, which is formed from the two characters `"\"` and `"n"`, is called the *newline* character. The two-character symbol `"<<"` is called the *insertion operator*. It is used to indicate that the string should be "inserted" into the output stream cout. This output stream acts as a conduit to the computer's output device: either the printer or the monitor. So inserting a string into that stream causes it to be printed (or displayed).

**EXAMPLE 2.2 The Pre-Standard Version of the "Hello World" Program**

If you are using an older C+t compiler that does not conform to the 1998 ISO Standard, then the program in Example 2.1 should be written like this:

```
#include <iostream.h>
int main()
{
    cout << "Hello, World!\n";
}
```

The two differences are (1) write `<iostream.h>` instead of `<iostream>` and (2) omit the second line:

```
using namespace std;.
```

Henceforth, we will omit the two required lines

```
#include <iostream>
using namespace std;
```

from the programs in this book. If you are using a Standard C++ compiler, you should insert them at the beginning of your programs. If you are using a pre-Standard compiler, insert the single line

```
#include <iostream.h>
```

instead. Note that the `#include` line is a *precompiler directive* and therefore contains no semicolon, but the `using` line is a Standard C++ statement so it ends with a semicolon.

## 2.2 VARIABLES AND DECLARATIONS

Here is a simple program that uses a variable named n:

**EXAMPLE 2.3 Using a Variable**

```
int main()
{
    int n;
    n = 44;
    cout << "The value of n is " << n << '\n';
}
```

The variable `n` is declared in the second line. The keyword **`int`** means that `n` represents an integer. On the fourth line, `n` is assigned the value 44. Then when it is used on the fifth line, its value is printed, like this:

```
The value of n is 44
```

Note in the output statement the difference between the character `n` in the string literal, the variable `n`, and the character `'\n'`. In a string literal, `n` is printed as the letter `n`; as a variable, the value 44 is printed for `n`; and when '\n' is printed, it simply advances the cursor to the beginning of the next line.

All names in a C++ program must be declared. A variable like `n` in Example 2.3 is declared by specifying its name after its type, like this:

```
int n;
```

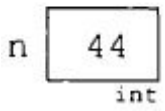
The type of a variable specifies how the variable can be used and how its value is stored in memory. To the right below is a picture of the object n. The box represents the variable itself, its name n is on its left, and its type int is below it.

After the value 44 is assigned to `n`, we can visualize it like this: The value of the variable is stored inside it, like putting a hat in a hat box.


The statement

```
n = 44
```

is called an *assignment statement*, and the equal sign = is called the *assignment operator*. Note that the action of an assignment is right-to-left: the *value* being assigned is on the right side of the assignment operator, and the name of the object to which it is assigned is on the left side. When used this way, a name is called an *lvalue*.




## 2.3 KEYWORDS AND IDENTIFIERS

A computer source code program consists of a sequence of *tokens* (predefined symbols, literals, keywords, and user-defined names) and *whitespace* (blanks, tabs, and newlines). The source code in Example 2.3 contains 14 tokens: the seven tokens `(`, `)`, `{`, `;`, `=`, `<<`, and `}` are predefined symbols; the three tokens `44`, `"The value of n is "`, and `"\n"` are literals; the token **`int`** is a keyword, and the three tokens `main`, `n`, and `cout` are user-defined names.

A *name* (also called an *identifier*) is a string of characters that identifies something. In C++ the only characters that can be used in a name are the 26 capital letters, the 26 lowercase letters, the underscore character (_), and the 10 digits. The first character in a name cannot be a digit.

**EXAMPLE 2.4 Declarations**

```
int sum;          // ok
int Sum;          // ok
int sum;          // ILLEGAL: "sum" is already declared
int _sum;         // ok
int r2d2;         // ok
int C3PO;         // ok
int 3PO;          // ILLEGAL: name must begin with a letter
int maxSize;      // ok
int max size;     // ILLEGAL: names may not include blanks
int class;        // ILLEGAL: reserved word
int O00o1111;     // ok, but not good (O looks like 0)
```

Every programming language has a special set of reserved words that have special meaning and cannot be used as names. These reserved words together with the language's predefined names are called *keywords*. ISO Stadard C++ defines 63 keywords:

| | | | | |
|---|---|---|---|---|
| asm | do | if | return | typedef |
| auto | double | inline | short | dypeid |
| bool | dynamic_cast | int | signed | typename |
| break | else | long | sizeof | union |
| case | enum | mutable | static | unsigned |
| char | explicit | namespace | static_cast | using |
| catch | export | new | struct | virtual |
| class | extern | operator | switch | void |
| const | false | private | template | volatile |
| const_cast | float | protected | this | wchar_t |
| continue | for | public | throw | while |
| default | friend | register | true | |
| delete | goto | reinterpret_cast | try | |

The compiler knows these words and interprets each according to the definition of the language. For example, `class` and `if` are reserved words, `bool` and `int` are names of predefined types, and `delete` and `new` are names of predefined operators.

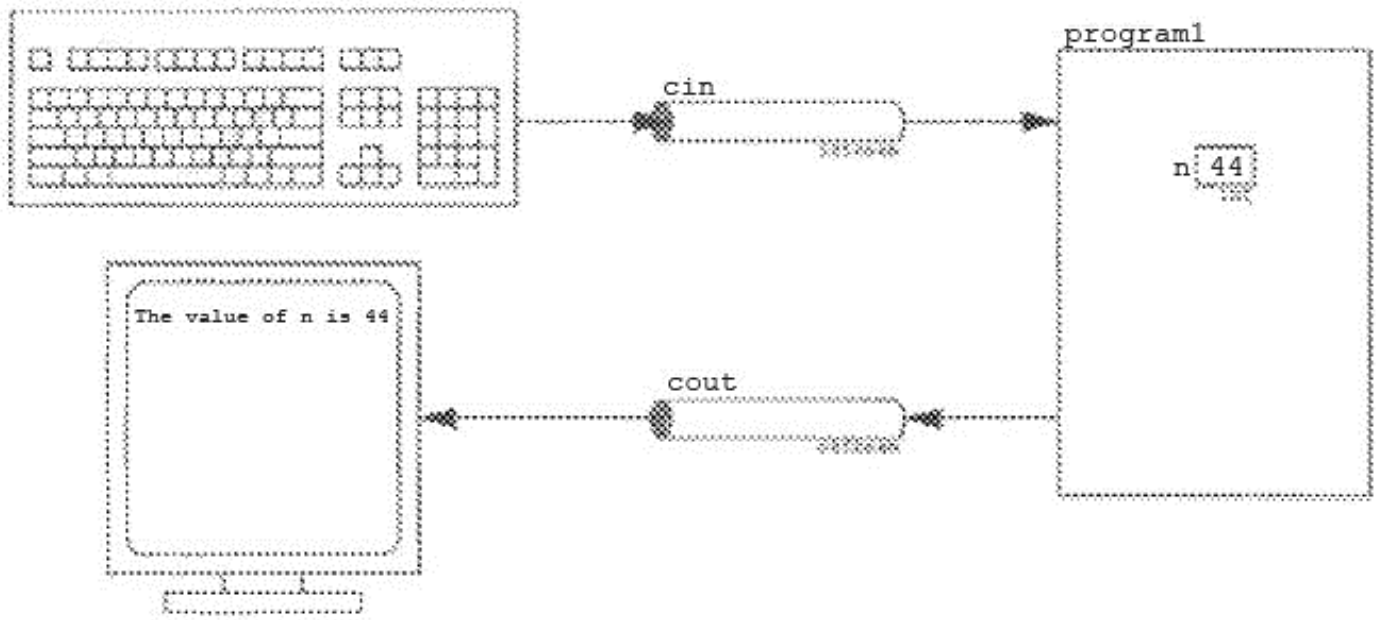
## 2.4 INPUT AND OUTPUT

The simplest way to produce output in a C++ program is through the standard output stream object named `cout`. A stream object can be visualized as a conduit between the program and the outside. For example, the results of the program in Example 2.3 can be viewed las shown in the picture below. The output flows from the program through `cout` to the output device **(**e.g., the monitor). }

Similarly, C++ defines an *input stream* named `cin` through which data flows from an input device (e.g., the keyboard). Like `cout`, the input stream object `cin` is also defined in the `<iostream>` file.

**EXAMPLE 2.5 Using the cin Object**

```
int main()
{
    int n;
    cin >> n;
    cout << "The value of n is " << n << "\n";
}
```




When this program runs, the system will wait until the user types in an integer and presses <Enter>. If she inputs 44, then the output will be the same as in Example 2.3, as indicated in the picture above.

The symbol >> is called the *extraction operator*, or simply the *input operator*. It is used to extract input from the `cin` input stream, similarly to how the insertion operator << is used to insert output into the `cout` output stream.

## 2.5 EXPRESSIONS AND OPERATORS

The insertion operator << and the extraction operator >> are two of a set of over 60 operators deftned in C++. These operators are categorized according to the types of objects upon which they operate.

The *arithmetic operators* are +, -, *, /, and %. All but the last of these are the familiar arithmetic operations that operate on integer and real numbers. The % operator, called the *modulus* operator, operates only on integers. The expression `m%n` evaluates to the remainder from the division of `m` by `n`. If `q == m/n` is the quotient and `r == m%n`, is the remainder, then `m = q*n + r`.

**EXAMPLE 2.6 Using the Modulus Operator**

```
int main()
{
    int m = 33;
    int n - 7;
    int q - m/n; // the quotient of m by n
    b r = m%n; // the remainder of m by n
    cout << m << "/" << n << " = " << q << endl;
    cout << m << "%" << n << " = " << r << endl;
    cout << q << "*" << n << " + " << r << " = " << q*n + r << endl;
}
```

The output from this program is

```
33/7 = 4
33%7 = 5
4*7 + 5 = 33
```

A *literal* is a symbol for a specific value of a variable. For example, 33 is an integer literal, and `"Hello, World! \n"` is a string literal. An *expression* is a combination of literals, variables, and operators. For example, `2*m - n%3` is an integer expression. Expressions are evaluated according to the precedence rules of their operators. For example, if the values of `m` and `n` are 8 and 4, then the previous expression is evaluated in order: (1) `2*m == 2*8 == 16;` (2) `n%3 == 4%3 == l;` `2*m - n%3 == 16 - 1 == 15.` The multiplication is done before the modulus because those two operators have the same precedence level and the multiplication is on the left. The subtraction is done last because that operator has lower precedence

than the other two. Operator precedence can be overridden with parentheses. For example, if `m` and `n` are 7 and 5, the expression `2*(m - n)%3` evaluates to 1, but the expression `2*(m - n%3)` evaluates to 10.

The expressions that an operator operates on are called its *operands.* For example, `2*m` and `n%3` are the operands for the `-` operator in the expression `2*m - n%3`. A *binary operator* is an operator that takes two operands. All the operators described above are binary operators. Besides binary operators, C++ also has several *unary operators* (operating on a single operand) and one *ternary operator* (operating on three operands). For example, the minus symbol `-` is used for both the binary operation of subtraction and the unary operation of negation.

The most widely used operator is the assignment operator `=` introduced in Section 2.2. In C++, the assignment operator can be combined with many other operators to produce combination assignment operators. For most binary operators `op=` whose operands have the same type, the combination operator `op=` can be used as

```
variable op= expression
```

to perform the combined operations

```
variable = variable op expression;
```

**EXAMPLE 2.7 Using Combination Assignment Operators**

```
int main()
{
    int n = 33;
    n += 5;  // same as n = n + 5; makes n == 33 + 5 == 38
    n %= 8;  // same as n = n%8; makes n == 38%8 == 6
    n *= n;  // same as n = n*n; makes n == 6*6 == 36
}
```

The double symbol ++ defines two unary operators in C++, called the *prefix increment* operator and the *postfix increment* operator, or, more simply, the *pre increment* and *post increment* operators. When applied to an integer variable, each of these increases its value by 1. If the variable being incremented is part of a larger expression, then the pre-increment operator will increment the variable before using its value in the expression, whereas the post-increment operator will use the value before incrementing the variable.

**EXAMPLE 2.8 Using the Pre-increment and the Post-Increment Operators**

```
int main()
{
    int n = 44;
    cout << n++ << endl; // prints 44 and then increments n to 45
    cout << ++n << b; // increments n to 46 and then prints 46
}
```

Two decrement operators are similar. The pre-decrement operator `--n` reduces the value of `n` by 1 first and then uses that reduced value, whereas the post-decrement operator `n--` uses the current value of `n` and then decreases it by 1.

When Bjarne Stroustrup chose the name C++ for his enhancement of the C language in 1983, he obviously had in mind the effect of the post-increment operator.

## 2.6 INITIALIZATIONS AND CONSTANTS

Ordinary variables defined in functions like `main()` are not automatically given initial values.

**EXAMPLE 2.9 Local Variables are Not Initialized by Default**

```
int main()
{
    int n;
    cout << n << endl; // unpredictable output!
}
```

The output from this program when run on a UNIX workstation was

```
302025904
```

This is an example of what is technically known as *garbage.* It is the result of the system trying to interpret a string of 32 random bits as an integer.

Fortunately, it is easy to initialize variables. This is done with an *initializer*, which is an expression of the form `= constant` appended to the declaration of the variable. Note that, in an initializer, the = symbol is not the assignment operator; an initialization is not the same as an assignment.

**EXAMPLE 2.10 Initializing a Variable**

```
int main ()
{
    int n = 44;   // n is initialized with the value 44
    cout << n << endl; // predictable output
}
```

The output from this program is, of course

```
44
```

A *constant* is an object whose value cannot be changed. An object is designated to be constant when it declared by preceding its type with the keyword `const.` All constants must be initialized.

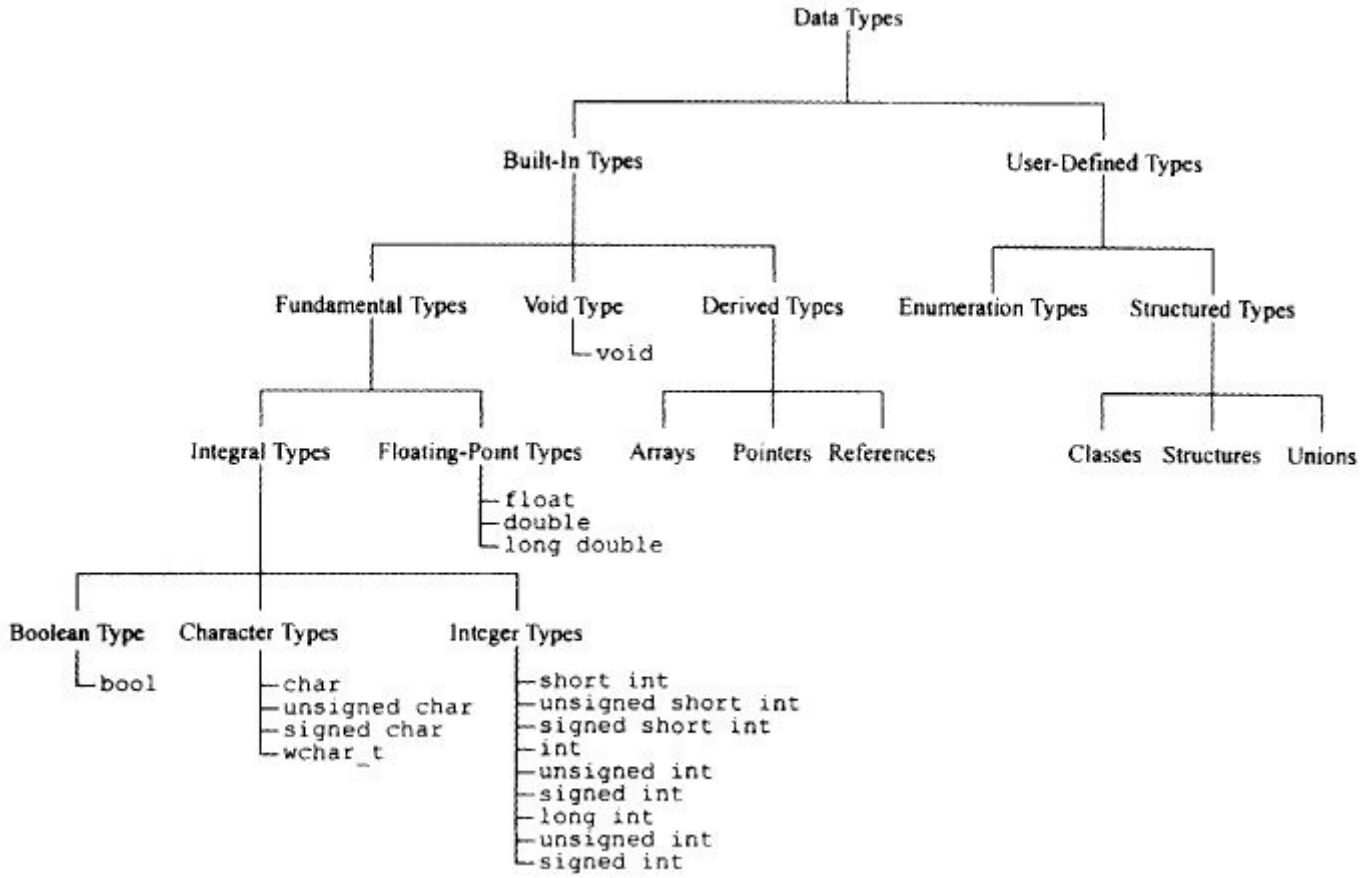
**EXAMPLE 2.11 Declaring a constant**

```
int main()
{
    const int n = 44; // n is a constant integer with value 44
    cout << n << endl; // predictable output
}
```

Using constants wherever appropriate is considered good "defensive programming" because it gives the compiler more opportunities to find your mistakes for you. It also makes your programs easier to maintain.

## 2.7 STANDARD C++ DATA TYPES

Data types in Standard C++ are classified as shown in the diagram below. This includes the new Boolean type `bool` whose values are either `false` or `true,` and the new character type `wchar t,` which usually represents the international 16-bit Unicode character set.



The 17 *fundamental types* and the `void` type have keyword names such as `signed long int`. The multi-word names for the integer types can be abbreviated by omitting the word `int`. For example, `unsigned short` means `unsigned short int`. The only difference between the `short`, `int`, and `long` types is the number of bytes (and therefore the range of values) used to store the objects. Typically (although not necessarily), `short` uses 2 bytes and `long` uses 4 bytes. Similarly, `float` usually uses 4 bytes and `double` uses 8. The `unsigned` types are used for bit strings.

Normally the `bool` values `false` and `true` are printed as 0 or 1. However, that can be overridden with the `boolalpha` flag that is defined in the `<iomanip>` file.

**EXAMPLE 2.12 Printing bool Values**

Here is a complete Standard C++ program:

```
#include <iomanip>           // use <iomanip.h> in pre-Standard C++
#include <iostream>         // use <iostream.h> in pre-Standard C++
using namespace std;                      // omit in pre-Standard C++
int main()
{
    cout << false << " " << true << " "
         << boolalpha << false << " " << true << endl;
}
```

Its output is

```
0 1 false true
```

The character types store integers (determined by their character codes) but print characters. In addition, the `char` type reads characters and can be assigned character literals delimited by apostrophes.

**EXAMPLE 2.13 Using `char` Type**

```
int main()
{
    char ch = 'A';    // ch is stored in one byte as the integer 65
    cout << ch << " " << int(ch) << endl;
}
```

The output is

```
A 65
```

Here, the `int` type was used to cast the value of `ch` from `char` to `int`.

The `void` type is used only with functions and pointers. (See Chapters `4` and `8`.)
*Derived types* are constructed from other types using the special symbols `{}`, `*`, and `&`.

**EXAMPLE 2.14 Derived Types**

```
float x = 666.66;
float y[8] = {0};  // y is an array of 8 floats, all initialized to 0
float* p = &x;     // p is a pointer to the float x
float& r = x;      // y is a reference to the float x
```

Here, `x` has the fundamental type **`float`**, and `y`, `p`, and `r` have types derived from **`float`** type.

An *array* is a sequence of elements, all of the same type, that can be accessed with the same name using an integer subscript. Arrays are described in Chapter 6. A *pointer* is a memory address that can be used to access an object stored at that address. A *reference* is a synonym for an existing object. Pointers and references are described in Chapter 9. The fundamental types, the **`void`** type, and the derived types are called *built-in types* because they can be used without any special definitions.

## 2.8 ENUMERATION TYPES

An *enumeration type* is one that is defined by the user simply by listing its set of possible values.

**EXAMPLE 2.15 An Enumeration Type**

```
enum Direction {NORTH, EAST, SOUTH, WEST};
```

This defines the type `Direction` and the four constants `NORTH, EAST, SOUTH,` and `WEST` as its possible values. Then objects can be defined and assigned in the normal way:

```
Direction current = EAST;
current = SOUTH;
```

Like character values, enumeration values are stored as integers. Normally the defined constant values are assigned the integers in order beginning with 0. Thus in Example 2.15, the assignment of the value `SOUTH` to `current` actually stores the integer 2 in that variable.

**EXAMPLE 2.16 Specifying Enumeration Constant Values**

```
enum Roman {1=1, V=5, X=10, L=50, C=100, D=500, M=1000};
enum Rank {DEUCE=2, TREY, FOUR, FIVE, SIX, SEVEN, EIGHT, NINE, TEN,
           JACK, QUEEN, KING, ACE};
enum Color {WHITE, BLUE=0xff, GREEN=0xff00, RED=0xff0000};
enum {SIZE=100}; // an anonymous enumeration type
```

The definition of type `Rank` initializes the constants `DEUCE` to 2, `TREY` to 3,..., `KING` to 13, and ACE to 14. The definition of the type `Color` uses the standard (hexadecimal) integer values for the basic colors. The anonymous enumeration type illustrates an alternative way to define integer constants. This is equivalent to

```
const int SIZE = 100;
```

## 2.9 THE STANDARD LIBRARY

The C++ language includes a set of files which is called the *Standard C++ Library*. These files define various extensions to the language. They included by means of the `#include` precompiler directive. For example, in Example 2.12 we included the `<iomanip>` file in order to use the `boolalpha` flag, and we included the `<iostream>` file in order to use the `cout` stream object and the `endl` inserter.

**EXAMPLE 2.17 Using the <iomanip> File**

This complete C++program illustrates the `<iomanip>` file `(ciomanip. h>` in pre-Standard C++)

```
#include <iomanip>                // use <iomanip.h> in pre-Standard C++
#include <iostream>               // use <iostream.h> in pre-Standard C++
using namespace std;                          // omit in pre-Standard C++
int main()
{
    int n = 75;
    cout << '\t' << oct << n << '\t' << hex << n
         << '\t' << dec << n << endl;
    cout << '\t' << setw(16) << n << endl;
    cout << '\t' << set fi11('.') << setw(16) << n << endl;
    cout << '\t' << set fi11('*') << setw(16) << n << endl;
    cout.setf(ios::left, ios::adjustfield); // left justify output
    cout << '\t' << setfill ('<') << setw(16) << n << endl;
    cout << setfill('.') << setw(40) << "Chapter 13";
    cout.setf(ios::right, ios::adjustfield);
    cout << setw(30) << "415" << endl;
    const double PI = 3.14159265358979323846;
    cout << PI << endl;
}
```

Here is the output:

```
        113     4b     75
                    75
        .............75
        *************75
        75<<<<<<<<<<<<<
    Chapter 13.....................................................415
```

```
3.14159
3.141592654
3.14159265359
3.1415926535898
3.141592653589793
3.14159265358979312
```

The first `cout` statement prints `n` in octal, hexadecimal, and decimal on the same line. The next four lines of output print `n` (in decimal) in a 16-column field using various fill characters. The first three of these use the default right-justification within the field; the last uses left-justification. The next three lines of code print a 70- character line using the dot ' . ' as a fill character as you would in a table of contents. The last six lines of output show how to obtain a number of significant digits of precision when printing a floating-point number. Note that the number is rounded to the specified number of digits. Also note that 16 is the maximum number of accurate digits obtainable for a `double` on this computer.

**EXAMPLE 2.18 Using the `<cmath>` File**

Here is a complete C++ program that illustrates some of the useful functions defined in the `<cmath>` file (`<math.h>` in pre-Standard C++):

```
#include <cmath>           // use <math.h>
#include <iostream>    // use <iostream.h>
using namespace std;               // omit
int main()
{
    double x = 2.718281828459045;
    cout << "ceil(x)    = " << ceil(x) << endl;
    cout << "floor(x)   = " << floor(x);
    cout << "sqrt(x)    = " << sqrt(x);
    cout << "pow(x, 0.5)= " << pow(x, 0.5);
    cout << "log(x)     = " << log(x);
    cout << "log10(x)   = " << log10(x);
    cout << "exp(x)     = " << exp(x);
    cout << "sin(x)     = " << sin(x);
}
```

Here is its output:

```
ceil(x)         = 3
floor(x)        = 2
sqrt(x)         = 1.64872
pow(x, 0.5)     = 1.64872
log(x)          = 1
log10(x)        = 0.434294
exp(x)          = 15.1543
sin(x)          = 0.410781
```

The `ceil (x)` (for "ceiling") and `floor(x)` functions return the integers that bracket `x.` The `sqrt(x)` function returns the square root of `x`. The `pow(x, y)` function returns the value of `x` raised to the power `y;` *i.e.*, The `log(x)` function returns the natural logarithm (base e) of `x.` The `logl0(x)` function returns the common logarithm (base 10) of `x.` The `exp(x)` function returns the exponential (base e) of `x;` *i.e.*, $e^x$. And `sin(x)` is the trigonometric sine function.

## 2.10 ERRORS

The art of defensive programming is based upon a faith in *Murphy's Law.* "If something can go wrong, it will." A good programmer tries to anticipate what could go wrong in order to prevent it. This skill requires an understanding of the kinds of errors that can occur.

The simplest kind of computer programming errors are *syntax errors.* These are usually easy to discover and fix because the compiler usually tells you exactly where they are.

**EXAMPLE 2.19 Syntax Errors**

Here's an incorrect version of the program in Example 2.1:

```
int main()
{   cout << "Hello, World!\n;
}
```

When compiled, the compiler printed the following diagnostic error messages:

```
ex0219.cc:8: unterminated string or character constant
ex0219.cc:8: possible real start of unterminated constant
```

The first line refers to line 8 of the source code file named `ex0219.cc`. That is the line

```
{ cout << "Hello, World!\n
```

The phrase "unterminated string" simply means that it is missing its right quotation mark. The second error message means that the beginning of this "unterminated string" is also on line 8.

The compiler will also locate other kinds of errors; *e.g.*, names that have not been declared.

**EXAMPLE 2.20 Other Compile-Time Errors**

Here's another incorrect version of the program in Example 2.1:

```
int main()
{ count << "Hello, World!\n";
}
```

When compiled, the compiler printed the following diagnostic error messages:

```
ex0220.cc: In function 'int main()':
ex0220.cc:8: 'count' undeclared (first use this function)
ex0220.cc:8: (Each undeclared identifier is reported only once
ex0220.cc:8: for each function it appears in.)
```

All this is simply reporting that the name `count` is undeclared. The programmer obviously meant to write `cout` instead of `count`.

After the compiler compiles your source code, it links it with other source code needed from the Standard Library. If you neglect to include the necessary files, the linkage will fail.

**EXAMPLE 2.21 Link-Time Errors**

Here is the correct version of the program in Example 2.1 but without the necessary `#include <iostream>` precompiler directive:

```
int main()
{   cout << "Hello, World!\n";
}
```

After compiling, the compiler printed the following diagnostic error messages:

```
eex0221.cc: In function 'int main()':
ex0221.cc:7: 'cout' undeclared (first use this function)
ex0221.cc:7: (Each undeclared identifier is reported only once
ex0221.cc:7: for each function it appears in.)
```

Although this looks like the same kind of error as in Example 2.20, it is not a compile-time error. There is nothing wrong with this source code. It is simply incomplete. Just add the missing two lines shown in Example 2.1 (or for pre-Standard C++ compilers, the equivalent one line shown in Example 2.2).

Programs can fail even after they have been compiled and linked successfully. There are many situations where the operating system may be unable to execute the program's instructions. These are called *run-time errors.*

**EXAMPLE 2.22 A Run-Time Error**

Here is a complete C++ program that compiles, links, and runs:

```
#include <iostream>
#include <math>
using namespace std;
int main()
{
    int n = 2000;
    cout << n << endl;
    n *= n;
    cout << n << endl;
    n * = n;
    cout << n << endl;
}
```

But the output is

```
2000
4000000
1246822400
```

The second number is correct: 2000*2000 = 4,000,000. But the third number should then be 4,000,000*4,000,000 = 16,000,000,000,000. That number is too large for an int type. This error is called *integer overflow.*

A program will give incorrect answers if its numeric values exceed the bounds of their types. This is called *overflow*. That can happen with integer types and with floating-point types. Other kinds of numeric run-time errors include *floating-point underflow* (values are too small), *roundoff errors,* attempted division by zero, and function arguments that are out of range.

**EXAMPLE 2.23 Round-Off Error**

This program implements the quadratic formula for solving the quadratic equation $ax^2 + \text{bx} + c = 0$:

```
#include <iostream>
#include <cmath>
int main()
{
    float a = 1e0; // == 1.0
    float b = -1e10; // == -10,000,000,000.0
    float c = 1e0; // == 1.0
    float d = b*b - 4.0*a*c;
    float xl = (-b + sqrt(d))/{2.0*a);
    float x2 = (-b - sqrt(d))/(2.0*a);
    cout << xl << '\t' << x2 << endl;
}
```

for the case where $a = 1$, $b = -10^{10}$, and $c = 1$. The output is

```
le+10 -50.1022
```

These two solutions are incorrect: $ax_1^2 + bx_1 + c = (1)(10^{10})^2 + (-10^{10})(10^{10}) + (1) = 1 \neq 0$, and $ax_2^2 + bx_2 + c = (1)(-50.1022)^2 + (-10^{10})(-50.1022) + (1) = 501{,}022{,}002{,}511.230445 \neq 0$. The first solution is close, considering the size of the numbers. But the second solution is way off. This is the result of roundoff error.

Run-time errors are more difficult to repair than compile-time errors because the operating system may not be able to pinpoint the problem. It can tell you why your program failed, but it may not be able to tell you where the failure occurred in your program.

Usually the worst kind of errors programmers have to handle are logical errors because there may be no easy way to detect their existence. A *logical error* is an error in the algorithm itself. The computer compiles and executes the program without signalling any problems. The problem is that the programmer gave the wrong instructions to the computer.

**EXAMPLE 2.24 A Logical Error**

Here is an implementation of the quadratic formula to solve the equation $3x^2 - 3\text{x} - 6 = 0$:

```
#include <iostream>
#include <cmath>
int main()
{
    float a = 3.0;
    float b = -3.0;
    float c = -6.0;
    float d = b*b - 4.0 * a * c;
    float xl = (-b + sqrt(d))/2.0*a;
    float x2 = (-b - sqrt(d))/2.0*a;
    cout << xl << '\t' << x2 << endl;
}
```

The output is

```
18     -9
```

But these solutions are wrong: $\text{ax}_1^2 + \text{bx}_1 + \text{c} = 3(18)^2 - 3(18) - 6 = 912 \neq 0$;
$\text{ax}_2^2 + \text{bx}_2 + \text{c} = 3(-9)^2 - 3(-9) - 6 = 264 \neq 0$. And there is no hint from the computer about where the problem is.

The logical errors here are at the ends of the lines that assign values to `xl` and `x2`. The quadratic formula requires the expression $2a$ to be in the denominator. But `num/2.0*a` means `(num/2.0)*a` in C++.

Most C++ compilers come with a *debugger*, which is a separate program that allows you to step through your program one instruction at a time, checking the values of the variables at each step. This is often the best way to debug your programs.

**Review Questions**

2.1 What is the purpose of the two lines:

```
#include <iostream>
using namespace std;
```

2.2 If your C++ compiler does not conform to the new C++ Standard, how should your precompiler directives differ from those illustrated in this book?

2.3 Which of the following declarations are illegal:

```
int the_cat's_pyjamas;
int the_second_largest_numer_in_the_list_of_minimum_values;
int union;
int 44;
int last?;
int NextNumber;
int round-up;
```

2.4 Why isn't `main` a keyword in C++?

2.5 What is a literal?

2.6 What is an operator?

2.7 What is wrong with the following code fragment?

```
cout >> "Enter x :";
cin << x;
```

2.8 How do the following two statements differ?

```
char ch = 'A';
char ch = 65;
```

2.9 What code could you execute to find the character whose ASCII code is 100?

2.10 What is wrong with the following code fragment?

```
enum Season { SPRING, SUMMER, FALL, WINTER };
enum Semester { FALL, SPRING, SUMMER };
```

## Programming Problems

2.11 Write a complete C++ program that prints your name and address.

2.12 Write a complete C++ program that inputs two integers and then prints the sum, difference, product, quotient, and remainder of the two integers.

2.13 Each of the following programs has an error. Locate the error, classify it as either a syntax error, a (non-syntax) compile-time error, a link-time error, a run-time error, or a logical error, and then correct it.:

***a.***
```
#include <iostream>
using namespace std
int main()
{   int n = 22;
    cout << n << endl;
}
```

***b.***
```
int main()
{   float x = 100.0;
}
```

***c.***
```
#include <iostream>
using namespace std;
int main()
{   int n += 22;
    cout << n << endl;
}
```

***d.***
```
#include <iostream>
using namespace std;
int main()
{   int n = 0;
    n /= n;
    cout << n << endl;
}
```

***e.***
```
#include <iostream>
using namespace std;
int main()
{   float x = 1e20;
    x *= x;
    cout << x << endl;
}
```

***f.***
```
#include <iostream>
using namespace std;
int main()
{   float x = sqrt(1.01);
    cout << x << endl;
}
```

***g.***
```
#include <iostream>
#include <math>
using namespace std;
int main()
{   float x = sqrt(-1.01) ;
    cout << x << endl;
}
```

***h.***
```
#include <iostream>
#include <math>
using namespace std;
int main()
{   float x = 100.0;
    cout << pow(x, -x) << endl;
}
```

2.14 The number `h` of different 7-card hands that can be dealt from an ordinary deck of 52 playing cards is $(52 \cdot 51 \cdot 50 \cdot 49 \cdot 48 \cdot 47 \cdot 46)/(7 \cdot 6 \cdot 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1)$.

***a.*** Run a program that performs this calculation directly, using only 4-byte integers.

***b.*** Rewrite the formula so that your program, still using only 4-byte integers, gives the correct value (133,784,560) for the integer `h`.

2.15 Rewrite and then rerun the program in Example 2.23 to minimize the effects of round-off error. Use the following algebraic identity to reformulate the assignment to `x2`:

$$\frac{-b - \sqrt{b^2 - 4ac}}{2a} = \frac{2c}{-b + \sqrt{b^2 - 4ac}}$$

2.16 Then explain why the output from this version is so much more accurate.

2.17 Write and run a program that causes floating-point overflow.

2.18 Write and run a program that causes floating-point underflow.

2.19 Write a program that converts a given number of inches to centimeters. (1 inch = 2.54 cm.)

2.20 Write a program that returns the Celsius value for a given temperature measured in Fahrenheit. For example, the input 68 would output 20. Use the conversion formula $5(F - 32) = 9C$.

2.21 Write a program that returns the Fahrenheit value for a given temperature measured in Celsius. For example, the input 20 would output 68. This is the inverse of the function in Problem 2.19.

2.22 Write a program that inputs a number of hours and outputs the equivalent number of weeks, days, and hours. For example, an input of 4000 would output 23 weeks, 5 days, and 16 hours.

2.23 Write a program that inputs a number of cents (from 0 to 99) and outputs the minimal number of pennies, nickels, dimes, and quarters with the same value. For example, 94 cents is the same as 3 quarters, 1 dime, 1 nickel, and 4 pennies.

2.24 Write a program that inputs a 6-digit positive integer (*i.e.*, in the range 100,000 – 999,999) and then constructs and outputs the integer whose digits are the reverse of the input. For example, if 289405 is input, then the integer 504982 is constructed and printed.

2.25 Write a program that rounds a real number to a given number of digits.

**Solutions**

**2.1** The directive `#include <iostream>` tells the precompiler to insert the contents of the Standard C++ library file `iostream` which contains the definitions necessary for doing stream input and output. The directive `using namespace std;` tells the compiler that it should look in the *namespace* `std` (defined in `iostream`) for the definitions of `cin` and `cout` (and any other unresolved references).

**2.2** If you are using a nonstandard C++ compiler, you should use

```
#include ciostream.h> instead of
#include <iostream>
 using namespace std;
```

You will also have to write your own `bool`, `string`, and `vector` classes unless your compiler provides its own version of these standard classes.

**2.3** The following declarations are illegal for the reasons indicated in the comments:

```
int the_cat's_pyjamas; // identifiers cannot contain apostrophes
int union;                // union is a keyword in C++
int 44;                   // identifiers cannot begin with a digit
int last?;              // identifiers cannot contain question marks
int round-up;             // identifiers cannot contain hyphens
```

**2.4** The word `main` is not a keyword because it is a name. It is the name of a function that is defined in the program.

**2.5** A literal is a specific anonymous constant value. For example, 88 is an integer literal, 3.14159 is a floating-point literal, `'G'` is a character literal, `"Hello, World\n"` is a string literal.

**2.6** An operator is a function that has a special symbol for a name. For example, `+` is the addition operator.

**2.7** In this code, the input and output operators are reversed. The output stream object `cout` uses the output operator `<<`, and the input stream object `cin` uses the input operator `>>`. If you imagine the input and output objects as being external conduits that lie to the left of the program, then it is easier to remember that output flows out to the left and input flows in from the left.

**2.8** Both statements have the same effect: declare `ch` to be a `char` and initialize it with the value `65`. Since this is the ASCII code for `'A'`, that character constant can also be used to initialize ch to `65`.

**2.9** One way to print the character whose ASCII code is 100 would be to create the correct `char` object and then print it:

```
char ch=100;
cout << ch << endl;
```

A simpler (but equivalent) way would be to cast the integer 100 as a character and then print that:

```
cout << char(100) << endl;
```

**2.10** The identifiers listed for the values of an enumeration type are constants and therefore must be unique within the same program scope. For example, the identifier `FALL` cannot be used in two different enumeration types.

**2.11**
```
#include <iostream>
using namespace std;
int main()
{
    cout << "\tSara Somers\n";
    cout << "\12401 Wadebridge Road\n";
    cout << "\tMidlothian, VA 23113-3841\n";
}
```

**2.12**
```
#include <iostream>
using namespace std;
int main()
{
    int m, n;
    cout << "Enter two integers: ";
    cin >> m >> n;
}
```

**2.13**

***a.*** Syntax error: the required semicolon at the end of the `using namespace` line is missing.

***b.*** Logical error; this program has no output, so it is useless.

***c.*** Compile-time error: the assignment operator += cannot be used in an initialization.

***d.*** Run-time error: since `n` is initialized to be 0, the assignment `n /= n` would divide n by 0;

***e.*** Run-time error: since `x` is initialized to be $10^{20}$, the assignment `x *= x` will overflow.

***f.*** Link-time error: the `sqrt()` function is defined in `<math>`, so that file must be `#included`.

***g.*** Run-time error: the call `sqrt(-1.01)` will fail because `-1.01` is negative.

***h***. Run-time error: since `x` is initialized to be 100, the call `pow(x, -x)` will underflow.

**2.14** ***a.***
```
int main ()
{
    cout << (52*51*50*49*48*47*46)/(7*6*5*4*3*2*1) << endl;
}
```

***b.***
```
int main ()
{   int h = 1;
    int c = 52;
    int n = 1;
    h *= c;
    h /= n;
    cout << c-- << '\t' << n++ << '\t' << h << endl;
    h *= c;
    h /= n;
    cout << c-- << '\t' << n++ << '\t' << h << endl;
    h *= c;
    h /= n;
    cout << c-- << '\t' << n++ << '\t' << h << endl;
    h *= c;
    h /= n;
    cout << c-- << '\t' << n++ << '\t' << h << endl;
    h *= c;
    h /= n;
```

```
        cout << c-- << '\t' << n++ << '\t' << h << endl;
        h *= c;
        h /= n;
        cout << c-- << '\t' << n++ << '\t' << h << endl;
        h *= c;
        h /= n;
        cout << c-- << '\t' << n++ << '\t' << h << endl;
    }
```

**2.15**
```
#include <iostream>
#include <cmath>
int main()
{
    float a = le0;
    float b = -1e10;
    float c = le0;
    float d = b*b - 4 * a * c;
    float xl = (-b + sqrt(d))/(2*a);
    float x2 = (2*c)/(-b + sqrt(d));
    cout << xl << '\t' << x2 << endl;
}
```

The output from this version of the program is

```
le+10 le-10
```

The first solution is $10^{10}$ which is the same as in Example 2.23. But the second solution ($10^{-10}$ = 0.0000000001) is now very close to satisfying the equation:

$$ax^2 + bx + c = x^2 - 10^{10}x + 1 = (10^{-10})^2 - 10^{10}(10^{-10}) + 1 = 10^{-20} - 1 + 1 = 10^{-20} \approx 0.$$

This dramatic improvement is a result avoiding the near cancellation that occurred in the expression $-\mathrm{b}-\sqrt{d}$. Since 4*ac* is very much less than $b^2$, $\mathrm{d} = \mathrm{b}^2 - 4\mathrm{ac} \approx \mathrm{b}^2$, making $\sqrt{\mathrm{d}} \approx \mathrm{b} = 10^{10}$. The worst kind of round-off error occurs when one very large number is subtracted from another nearly equal number.

**2.16** This program crashes due to floating-point overflow:

```
int main()
{
    float x = 1.0;
    cout << x << endl;
    x *= 1e10;
    cout << x << endl;
    x *= 1e10;
    cout << x << endl;
    x *= 1e10;
    cout << x << endl;
    x *= 1e10;
    cout << x << endl;
}
```

**2.17** This program crashes due to floating-point underflow:

```
int main()
{
    float x = 1.0;
    cout << x << endl;
    x /= 1e10;
    cout << x << endl;
    x /= 1e10;
    cout << x << endl;
    x /= le10;
    cout << x << endl;
    x /= 1e10;
    cout << x << endl;
    x /= 1e10;
    cout << x << endl;
}
```

**2.18**
```
int main ()
{   float inches, cm;
    cout << "Input inches: ";
    cin >> inches;
    cm = 2.54*inches;
    cout << inches << " inches = " << cm << " centimeters\r." ;
}
```

**2.19**
```
int main()
{   float far, cel;
    cout << "Input temperature in degrees Farenheit: ";
    cin >> far;
    cel = 5.0* (far - 32.0)/9.0;
    cout << far << " degrees Farenheit = " << cel << " degrees Celsius\n";
}
```

**2.20**
```
int main ()
{   float cel, far;
    cout << "Input temperature in degrees Celsius: ";
    cin >> cel;
    far = 1.8*cel + 32.0;
    cout << cel << " degrees Celsius = "
         << far << " degrees Farenheit\n";
}
```

**2.21**
```
int main()
{   int hours, days, weeks;
    cout << "Enter number of hours: ";
    cin >> hours;
    cout << hours << days = hours/24;
    hours %= 24;
    weeks = days/7;
    days %= 7;
    cout << weeks << " weeks, " << days << " days, and " << hours << " hours.\n";
}
```

**2.22**
```
int main()
{   int pennies, nickels, dimes, quarters;
    cout << "Enter number of cents: ";
    cin >> pennies;
    cout << pennies << " cents = ";
    quarters = pennies/25;
    pennies %= 25;
    dimes = pennies/10;
    pennies %= 10;
    nickels = pennies/5;
    pennies %= 5;
    cout << quarters << " quarters, " << dimes << "dimes, "
         << nickels << " nickels, and " << pennies << " pennies.\n";
}
```

**2.23**
```
int main()
{   int n, d0, dl, d2, d3, d4, d5;
    cout << "Enter a 6-digit positive integer: ";
    cin >> n;
    d0 = n%10;
    n /= 10;
    d1 = n%10;
```

```
    d3 = n%10;
    n /= 10;
    d4 = n%10;
    n /= 10;
    d5 = n%10;
    n = ((((d0*10 + d1)*10 + d2)*10 + d3)*10 + d4)*10 + d5;
    cout << n << endl;
}
```

**2.24**

```
int main()
{   double x = 3.1415926535897932;
    cout << setprecision(16) << x << endl;
    int n;
    cout << "Enter number of digits: ";
    cin >> n;
    x *= pow(10,n);
    int round = int(x + 0.5);
    x = double(round)/pow(10,n);
    cout << x << endl;
}
```

}