Chapter Two

2. C++ Basics

2.1. Structure of C++ Program

A C++ program has the following structure

[Comments]

[Preprocessor directives]

[Global variable declarations]

[Prototypes of functions]

[Definitions of functions]

2.2. C++ IDE

The complete development cycle in C++ is: Write the program, compile the source code, link the program, and run it.

Writing a Program

To write a source code, your compiler may have its own built-in text editor, or you may be using a commercial text editor or word processor that can produce text files. The important thing is that whatever you write your program in, it must save simple, plain-text files, with no word processing commands embedded in the text. Examples of safe editors include Windows Notepad, the DOS Edit command, EMACS, and vi. Many commercial word processors, such as WordPerfect, Word, and dozens of others, also offer a method for saving simple text files.

The files you create with your editor are called source files, and for C++ they typically are named with the extension .CPP.

Compiling

Your source code file can't be executed, or run, as a program can. To turn your source code into a program, you use a compiler. How you invoke your compiler, and how you tell it where to find your source code, will vary from compiler to compiler; check your documentation. In Borland's Turbo C++ you pick the RUN menu command or type

```
tc <filename>
```

from the command line, where <filename> is the name of your source code file (for example, `test.cpp`). Other compilers may do things slightly differently. After your source code is compiled, an object file is produced. This file is often named with the extension `.OBJ`. This is still not an executable program, however. To turn this into an executable program, you must run your linker.

Linking

C++ programs are typically created by linking together one or more OBJ files with one or more libraries. A library is a collection of linkable files that were supplied with your compiler, that you purchased separately, or that you created and compiled. All C++ compilers come with a library of useful functions (or procedures) and classes that you can include in your program. A function is a block of code that performs a service, such as adding two numbers or printing to the screen. A class is a collection of data and related functions.

Summary

The steps to create an executable file are

1. Create a source code file, with a .CPP extension.
2. Compile the source code into a file with the .OBJ extension.
3. Link your OBJ file with any needed libraries to produce an executable program.

2.3. Showing Sample program

Any meaningful program written in C++ has to contain a number of components: the *`main`* function; some variable declarations; and some executable statements. For example, the following is a very basic C++ program:

```
1: #include <iostream.h>
2:
3:  int main()
4: {
5:    cout << "Hello World!\n";
6:      return 0;
7: }
```

On line 1, the file *iostream.h* is included in the file. The first character is the # symbol, which is a signal to the preprocessor. Each time you start your compiler, the preprocessor is run. The preprocessor reads through your source code, looking for lines that begin with the pound symbol (#), and acts on those lines before the compiler runs.

include is a preprocessor instruction that says, "What follows is a filename. Find that file and read it in right here." The angle brackets around the filename tell the preprocessor to look in all the usual places for this file. If your compiler is set up correctly, the angle brackets will cause the preprocessor to look for the file *iostream.h* in the directory that holds all the *H files* for your compiler. The file *iostream.h* (Input-Output-Stream) is used by *cout*, which assists with writing to the screen. The effect of line 1 is to include the file iostream.h into this program as if you had typed it in yourself.

The preprocessor runs before your compiler each time the compiler is invoked. The preprocessor translates any line that begins with a pound symbol (#) into a special command, getting your code file ready for the compiler.

Line 3 begins the actual program with a function named *main*(). Every C++ program has a *main()* function. In general, a function is a block of code that performs one or more actions. Usually functions are invoked or called by other functions, but *main*() is special. When your program starts, *main*() is called automatically.

main(), like all functions, must state what kind of value it will return. The return value type for *main*() in HELLO.CPP is int, which means that this function will return an integer value.

All functions begin with an opening brace ({) and end with a closing brace (}). The braces for the *main*() function are on lines 4 and 7. Everything between the opening and closing braces is considered a part of the function.

The meat and potatoes of this program is on line 5. The object cout is used to print a message to the screen. cout is used in C++ to print strings and values to the screen. A string is just a set of characters.

Here's how cout is used: type the word cout, followed by the output redirection operator *(<<).* Whatever follows the output redirection operator is written to the screen. If you want a string of characters written, be sure to enclose them in double quotes ("), as shown on line 5. A text string is a series of printable characters.

The final two characters, \n, tell cout to put a new line after the words Hello World! All ANSI-compliant programs declare main() to return an int. This value is "returned" to the operating system when your program completes. Some programmers signal an error by returning the value 1.

The main() function ends on line 7 with the closing brace.

2.4. Basic Elements

2.4.1. Keywords (reserved words)

Reserved/Key words have a unique meaning within a C++ program. These symbols, the reserved words, must not be used for any other purposes. All reserved words are in lower-case letters. The following are some of the reserved words of C++.

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

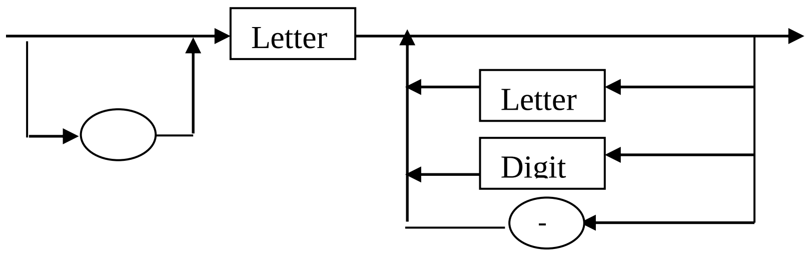
Notice that main is not a reserved word. However, this is a fairly technical distinction, and for practical purposes you are advised to treat main, cin, and cout as if they were reserved as well.

2.4.2. Identifiers

An identifier is name associated with a function or data object and used to refer to that function or data object. An identifier must:

- Start with a letter or underscore
- Consist only of letters, the digits 0-9, or the underscore symbol _
- Not be a reserved word

Syntax of an identifier



For the purposes of C++ identifiers, the underscore symbol, _, is considered to be a letter. Its use as the first character in an identifier is not recommended though, because many library functions in C++ use such identifiers. Similarly, the use of two consecutive underscore symbols, _ _, is forbidden.

The following are valid identifiers

Length	days_in_year	DataSet1	Profit95
Int	_Pressure	first_one	first_1

Although using _Pressure is not recommended.

The following are invalid:

days-in-year	1data	int	first.val
throw	my__best	No##	bestWish!

Although it may be easier to type a program consisting of single character identifiers, modifying or correcting the program becomes more and more difficult. The minor typing effort of using meaningful identifiers will repay itself many fold in the avoidance of simple programming errors when the program is modified.

At this stage it is worth noting that C++ is case-sensitive. That is lower-case letters are treated as distinct from upper-case letters. Thus the word NUM different from the word num or the word Num. Identifiers can be used to identify variable or constants or functions. Function identifier is an identifier that is used to name a function.

2.4.3. Literals

Literals are constant values which can be a number, a character of a string. For example the number 129.005, the character ‘A’ and the string “hello world” are all literals. There is no identifier that identifies them.

2.4.4. Comments

A comment is a piece of descriptive text which explains some aspect of a program. Program comments are totally ignored by the compiler and are only intended for human readers. C++ provides two types of comment delimiters:

- Anything after // (until the end of the line on which it appears) is considered a comment.

- Anything enclosed by the pair /* and */ is considered a comment.

2.5. Data Types, Variables, and Constants

2.5.1. Variables

A variable is a symbolic name for a memory location in which data can be stored and subsequently recalled. Variables are used for holding data values so that they can be utilized in various computations in a program. All variables have two important attributes:

- A type, which is, established when the variable is defined (e.g., integer, float, character). Once defined, the type of a C++ variable cannot be changed.
- A value, which can be changed by assigning a new value to the variable. The kind of values a variable can assume depends on its type. For example, an integer variable can only take integer values (e.g., 2, 100, -12) not real numbers like 0.123.

Variable Declaration

Declaring a variable means defining (creating) a variable. You create or define a variable by stating its type, followed by one or more spaces, followed by the variable name and a semicolon. The variable name can be virtually any combination of letters, but cannot contain spaces and the first character must be a letter or an underscore. Variable names cannot also be the same as keywords used by C++. Legal variable names include x, J23qrsnf, and myAge. Good variable names tell you what the variables are for; using good names makes it easier to understand the flow of your program. The following statement defines an integer variable called myAge:

int myAge;

IMPORTANT- Variables must be declared before used!

As a general programming practice, avoid such horrific names as J23qrsnf, and restrict single-letter variable names (such as x or i) to variables that are used only very briefly. Try to use expressive names such as myAge or howMany.

A point worth mentioning again here is that C++ is case-sensitive. In other words, uppercase and lowercase letters are considered to be different. A variable named age is different from Age, which is different from AGE.

Creating More Than One Variable at a Time

You can create more than one variable of the same type in one statement by writing the type and then the variable names, separated by commas. For example:

```
int myAge, myWeight;   // two int variables
long area, width, length;       // three longs
```

As you can see, myAge and myWeight are each declared as integer variables. The second line declares three individual long variables named area, width, and length. However keep in mind that you cannot mix types in one definition statement.

Assigning Values to Your Variables

You assign a value to a variable by using the assignment operator (=). Thus, you would assign 5 to Width by writing

```
int  Width;
Width = 5;
```

You can combine these steps and initialize `Width` when you define it by writing

```
int Width = 5;
```

Initialization looks very much like assignment, and with integer variables, the difference is minor. The essential difference is that initialization takes place at the moment you create the variable.

Just as you can define more than one variable at a time, you can initialize more than one variable at creation. For example:

```
// create two int variables and initialize them
 int width = 5, length = 7;
```

This example initializes the integer variable width to the value 5 and the length variable to the value 7. It is possible to even mix definitions and initializations:

```
int myAge = 39, yourAge, hisAge = 40;
```

This example creates three type int variables, and it initializes the first and third.

2.5.2. Basic Data Types

When you define a variable in C++, you must tell the compiler what kind of variable it is: an integer, a character, and so forth. This information tells the compiler how much room to set aside and what kind of value you want to store in your variable.

Several data types are built into C++. The varieties of data types allow programmers to select the type appropriate to the needs of the applications being developed. The data types supported by C++ can be classified as basic (fundamental) data types, user defined data types, derived data types and empty data types. However, the discussion here will focus only on the basic data types.

Basic (fundamental) data types in C++ can be conveniently divided into numeric and character types. Numeric variables can further be divided into integer variables and floating-point variables. Integer variables will hold only integers whereas floating number variables can accommodate real numbers.

Both the numeric data types offer modifiers that are used to vary the nature of the data to be stored. The modifiers used can be short, **long**, **signed** and **unsigned.**

The data types used in C++ programs are described in Table 1.1. This table shows the variable type, how much room it takes in memory, and what kinds of values can be stored in these variables. The values that can be stored are determined by the size of the variable types.

Type	Size	Values
`unsigned short int`	2 bytes	0 to 65,535
`short int(signed short int)`	2 bytes	-32,768 to 32,767
`unsigned long int`	4 bytes	0 to 4,294,967,295
`long int(signed long int)`	4 bytes	-2,147,483,648 to 2,147,483,647
`int`	2 bytes	-32,768 to 32,767
`unsigned int`	2 bytes	0 to 65,535
`signed int`	2 bytes	-32,768 to 32,767
`char`	1 byte	256 character values
`float`	4 bytes	3.4e-38 to 3.4e38
`double`	8 bytes	1.7e-308 to 1.7e308
`long double`	10 bytes	1.2e-4932 to 1.2e4932

Table C++ data types and their ranges

2.5.3. Signed and Unsigned

As shown above, integer types come in two varieties: signed and unsigned. The idea here is that sometimes you need negative numbers, and sometimes you don't. Integers (short and long) without the word "unsigned" are assumed to be signed. signed integers are either negative or positive. Unsigned integers are always positive.

Because you have the same number of bytes for both signed and unsigned integers, the largest number you can store in an unsigned integer is twice as big as the largest positive number you can store in a signed integer. An unsigned short integer can handle numbers from 0 to 65,535. Half the numbers represented by a signed short are negative, thus a signed short can only represent numbers from -32,768 to 32,767.

Example: A demonstration of the use of variables.

```
2:   #include <iostream.h>
3:
4:   int main()
5:   {
6:     unsigned short int Width = 5, Length;
7:     Length = 10;
8:
9:     // create  an unsigned short and initialize with result
10:       // of multiplying Width by Length
11:     unsigned short int Area  = Width * Length;
12:
13:     cout << "Width:" << Width << "\n";
14:     cout << "Length: "  << Length << endl;
15:     cout << "Area: " << Area << endl;
16:        return 0;
17: }
```

Output: Width:5
Length: 10
Area: 50

Line 2 includes the required include statement for the iostream's library so that cout will work. Line 4 begins the program.

On line 6, Width is defined as an unsigned short integer, and its value is initialized to 5. Another unsigned short integer, Length, is also defined, but it is not initialized. On line 7, the value 10 is assigned to Length.

On line 11, an unsigned short integer, Area, is defined, and it is initialized with the value obtained by multiplying Width times Length. On lines 13-15, the values of the variables are printed to the screen. Note that the special word endl creates a new line.

Wrapping around integer values

The fact that unsigned long integers have a limit to the values they can hold is only rarely a problem, but what happens if you do run out of room? When an unsigned integer reaches its maximum value, it wraps around and starts over, much as a car odometer might. The following example shows what happens if you try to put too large a value into a short integer.

```
Example: A demonstration of putting too large a value in a variable
#include <iostream.h>
int main()
{
   unsigned short int smallNumber;
   smallNumber = 65535;
   cout << "small number:" << smallNumber << endl;
   smallNumber++;
   cout << "small number:" << smallNumber << endl;
   smallNumber++;
   cout << "small number:" << smallNumber << endl;
       return 0;
}
Output: small number:65535
small number:0
small number:1
```

A signed integer is different from an unsigned integer, in that half of the values you can represent are negative. Instead of picturing a traditional car odometer, you might picture one that rotates up for positive numbers and down for negative numbers. One mile from 0 is either 1 or -1. When you run out of positive numbers, you run right into the largest negative numbers and then count back down to 0. The whole idea here is putting a number that is above the range of the variable can create unpredictable problem.

```
Example: A demonstration of adding too large a number to a signed
integer.
#include <iostream.h>
int main()
{
   short int smallNumber;
   smallNumber = 32767;
   cout << "small number:" << smallNumber << endl;
   smallNumber++;
   cout << "small number:" << smallNumber << endl;
   smallNumber++;
   cout << "small number:" << smallNumber << endl;
       return 0;
}
Output: small number:32767
```

```
small number:-32768
small number:-32767
```

IMPORTANT – To any variable, do not assign a value that is beyond its range!

2.5.4. Characters

Character variables (type char) are typically 1 byte, enough to hold 256 values. A char can be interpreted as a small number (0-255) or as a member of the ASCII set. ASCII stands for the American Standard Code for Information Interchange. The ASCII character set and its ISO (International Standards Organization) equivalent are a way to encode all the letters, numerals, and punctuation marks.

In the ASCII code, the lowercase letter "a" is assigned the value 97. All the lower- and uppercase letters, all the numerals, and all the punctuation marks are assigned values between 1 and 128. Another 128 marks and symbols are reserved for use by the computer maker, although the IBM extended character set has become something of a standard.

2.5.5. Characters and Numbers

When you put a character, for example, \`a', into a char variable, what is really there is just a number between 0 and 255. The compiler knows, however, how to translate back and forth between characters (represented by a single quotation mark and then a letter, numeral, or punctuation mark, followed by a closing single quotation mark) and one of the ASCII values.

The value/letter relationship is arbitrary; there is no particular reason that the lowercase "a" is assigned the value 97. As long as everyone (your keyboard, compiler, and screen) agrees, there is no problem. It is important to realize, however, that there is a big difference between the value 5 and the character \`5'. The latter is actually valued at 53, much as the letter \`a' is valued at 97.

2.6. Operators

C++ provides operators for composing arithmetic, relational, logical, bitwise, and conditional expressions. It also provides operators which produce useful side-effects, such as assignment, increment, and decrement. We will look at each category of operators in turn. We will also discuss the precedence rules which govern the order of operator evaluation in a multi-operator expression.

2.6.1. Assignment Operators

The assignment operator is used for storing a value at some memory location (typically denoted by a variable). Its left operand should be an lvalue, and its right operand may be an arbitrary expression. The latter is evaluated and the outcome is stored in the location denoted by the lvalue.

An lvalue (standing for left value) is anything that denotes a memory location in which a value may be stored. The only kind of lvalue we have seen so far is a variable. Other kinds of lvalues (based on pointers and references) will be described later. The assignment operator has a number of variants, obtained by combining it with the arithmetic and bitwise operators.

Operator	Example	Equivalent To
=	n = 25	
+=	n += 25	n = n + 25
-=	n -= 25	n = n - 25
*=	n *= 25	n = n * 25
/=	n /= 25	n = n / 25
%=	n %= 25	n = n % 25
&=	n &= 0xF2F2	n = n & 0xF2F2
|=	n |= 0xF2F2	n = n | 0xF2F2
^=	n ^= 0xF2F2	n = n ^ 0xF2F2
<<=	n <<= 4	n = n << 4
>>=	n >>= 4	n = n >> 4

An assignment operation is itself an expression whose value is the value stored in its left operand. An assignment operation can therefore be used as the right operand of another assignment operation. Any number of assignments can be concatenated in this fashion to form one expression. For example:

```
int m, n, p;
m = n = p = 100;                // means: n = (m = (p = 100));
m = (n = p = 100) + 2;      // means: m = (n = (p = 100)) + 2;
```

This is equally applicable to other forms of assignment. For example:

```
m = 100;
m += n = p = 10;                    // means: m = m + (n = p = 10);
```

2.6.2. Arithmetic Operators

C++ provides five basic arithmetic operators. These are summarized in table below

Operator	Name	Example
+	Addition	12 + 4.9 // gives 16.9
-	Subtraction	3.98 - 4 // gives -0.02
*	Multiplication	2 * 3.4 // gives 6.8
/	Division	9 / 2.0 // gives 4.5
%	Remainder	13 % 3 //gives 1

Arithmetic operators.

Except for remainder (%) all other arithmetic operators can accept a mix of integer and real operands. Generally, if both operands are integers then the result will be an integer. However, if one or both of the operands are reals then the result will be a real (or double to be exact).

When both operands of the division operator (/) are integers then the division is performed as an integer division and not the normal division we are used to. Integer division always results in an integer outcome (i.e., the result is always rounded down). For example:

```
9 / 2       // gives 4, not 4.5!
-9 / 2      // gives -5, not -4!
```

Unintended integer divisions are a common source of programming errors. To obtain a real division when both operands are integers, you should cast one of the operands to be real:

```
int     cost = 100;
int     volume = 80;
double  unitPrice = cost / (double) volume;     // gives 1.25
```

The remainder operator (%) expects integers for both of its operands. It returns the remainder of integer-dividing the operands. For example 13%3 is calculated by integer dividing 13 by 3 to give an outcome of 4 and a remainder of 1; the result is therefore 1.

It is possible for the outcome of an arithmetic operation to be too large for storing in a designated variable. This situation is called an overflow. The outcome of an overflow is machine-dependent and therefore undefined. For example:

```
unsigned char   k = 10 * 92;    // overflow: 920 > 255
```

It is illegal to divide a number by zero. This results in a run-time *division-by-zero* failure, which typically causes the program to terminate.

There are also a number of predefined library functions, which perform arithmetic operations. As with input & output statements, if you want to use these you must put a `#include` statement at the start of your program. Some of the more common library functions are summarised below.

Header File	***Function***	**Parameter Type(s)**	***Result Type***	***Result***
<stdlib.h>	abs(i)	int	int	Absolute value of *i*
<math.h>	cos(x)	float	float	Cosine of *x* (*x* is in radians)
<math.h>	fabs(x)	float	float	Absolute value of *x*
<math.h>	pow(x, y)	float	float	*x* raised to the power of *y*
<math.h>	sin(x)	float	float	Sine of *x* (*x* is in radians)
<math.h>	sqrt(x)	float	float	Square root of *x*
<math.h>	tan(x)	float	float	Tangent of *x*

2.6.3. Relational Operators

C++ provides six relational operators for comparing numeric quantities. These are summarized in table below. Relational operators evaluate to 1 (representing the true outcome) or 0 (representing the false outcome).

Operator	Name	Example
==	Equality	5 == 5 // gives 1
!=	Inequality	5 != 5 // gives 0
<	Less Than	5 < 5.5 // gives 1
<=	Less Than or Equal	5 <= 5 // gives 1
>	Greater Than	5 > 5.5 // gives 0
>=	Greater Than or Equal	6.3 >= 5 // gives 1

Relational operators

Note that the <= and >= operators are only supported in the form shown. In particular, =< and => are both invalid and do not mean anything.

The operands of a relational operator must evaluate to a number. Characters are valid operands since they are represented by numeric values. For example (assuming ASCII coding):

```
'A' < 'F'                    // gives 1 (is like 65 < 70)
```

The relational operators should not be used for comparing strings, because this will result in the string addresses being compared, not the string contents. For example, the expression "HELLO" < "BYE" causes the address of "HELLO" to be compared to the address of "BYE". As these addresses are determined by the compiler (in a machine-dependent manner), the outcome may be 0 or 1, and is therefore undefined. C++ provides library functions (e.g., strcmp) for the lexicographic comparison of string.

2.6.4. Logical Operators

C++ provides three logical operators for combining logical expression. These are summarized in the table below. Like the relational operators, logical operators evaluate to 1 or 0.

Operator	Name	Example
!	Logical Negation	!(5 == 5) // gives 0
&&	Logical And	5 < 6 && 6 < 6 // gives 1
||	Logical Or	5 < 6 || 6 < 5 // gives 1

Logical operators

Logical negation is a unary operator, which negates the logical value of its single operand. If its operand is nonzero it produces 0, and if it is 0 it produces 1.

Logical and produces 0 if one or both of its operands evaluate to 0. Otherwise, it produces 1. Logical or produces 0 if both of its operands evaluate to 0. Otherwise, it produces 1.

Note that here we talk of zero and nonzero operands (not zero and 1). In general, any nonzero value can be used to represent the logical true, whereas only zero represents the logical false. The following are, therefore, all valid logical expressions:

```
!20                     // gives 0
10 && 5                 // gives 1
10 || 5.5               // gives 1
10 && 0                 // gives 0
```

C++ does not have a built-in boolean type. It is customary to use the type int for this purpose instead. For example:

```
int     sorted = 0;     // false
int     balanced = 1; // true
```

2.6.5. Bitwise Operators

C++ provides six bitwise operators for manipulating the individual bits in an integer quantity. These are summarized in the table below.

Operator	Name	Example
~	Bitwise Negation	~'\011' // gives '\366'
&	Bitwise And	'\011' & '\027' // gives '\001'
|	Bitwise Or	'\011' | '\027' // gives '\037'
^	Bitwise Exclusive Or	'\011' ^ '\027' // gives '\036'
<<	Bitwise Left Shift	'\011' << 2 // gives '\044'
>>	Bitwise Right Shift	'\011' >> 2 // gives '\002'

Bitwise operators

Bitwise operators expect their operands to be integer quantities and treat them as bit sequences. Bitwise negation is a unary operator which reverses the bits in its operands. Bitwise and compares the corresponding bits of its operands and produces a 1 when both bits are 1, and 0 otherwise. Bitwise or compares the corresponding bits of its operands and produces a 0 when both bits are 0, and 1 otherwise. Bitwise exclusive or compares the corresponding bits of its operands and produces a 0 when both bits are 1 or both bits are 0, and 1 otherwise.

Bitwise left shift operator and bitwise right shift operator both take a bit sequence as their left operand and a positive integer quantity n as their right operand. The former produces a bit sequence equal to the left operand but which has been shifted n bit positions to the left. The latter produces a bit sequence equal to the left operand but which has been shifted n bit positions to the right. Vacated bits at either end are set to 0.

Table 2.1 illustrates bit sequences for the sample operands and results in Table 2.**Error! Bookmark not defined.**. To avoid worrying about the sign bit (which is machine dependent), it is common to declare a bit sequence as an unsigned quantity:

```
unsigned char x = '\011';
unsigned char y = '\027';
```

Table 2.1 How the bits are calculated.

Example	Octal Value	Bit Sequence							
x	011	0	0	0	0	1	0	0	1
y	027	0	0	0	1	0	1	1	1
~x	366	1	1	1	1	0	1	1	0
x & y	001	0	0	0	0	0	0	0	1
x | y	037	0	0	0	1	1	1	1	1
x ^ y	036	0	0	0	1	1	1	1	0
x << 2	044	0	0	1	0	0	1	0	0
x >> 2	002	0	0	0	0	0	0	1	0

2.6.6. Increment/decrement Operators

The auto increment (++) and auto decrement (--) operators provide a convenient way of, respectively, adding and subtracting 1 from a numeric variable. These are summarized in the following table. The examples assume the following variable definition:

```
int k = 5;
```

Operator	Name	Example
++	Auto Increment (prefix)	++k + 10 // gives 16
++	Auto Increment (postfix)	k++ + 10 // gives 15
--	Auto Decrement (prefix)	--k + 10 // gives 14
--	Auto Decrement (postfix)	k-- + 10 // gives 15

Increment and decrement operators

Both operators can be used in prefix and postfix form. The difference is significant. When used in prefix form, the operator is first applied and the outcome is then used in the expression. When used in the postfix form, the expression is evaluated first and then the operator applied. Both operators may be applied to integer as well as real variables, although in practice real variables are rarely useful in this form.

2.7. Precedence of Operators

The order in which operators are evaluated in an expression is significant and is determined by precedence rules. These rules divide the C++ operators into a number of precedence levels. Operators in higher levels take precedence over operators in lower levels.

Level	Operator						Kind	Order
Highest	::						Unary	Both
	()	[]	->	.			Binary	Left to Right
	+ -	++ --	! ~	* &	new delete	sizeof()	Unary	Right to Left
	->*	.*					Binary	Left to Right
	*	/	%				Binary	Left to Right
	+	-					Binary	Left to Right
	<<	>>					Binary	Left to Right
	<	<=	>	>=			Binary	Left to Right
	==	!=					Binary	Left to Right
	&						Binary	Left to Right
	^						Binary	Left to Right
	|						Binary	Left to Right
	& &						Binary	Left to Right
	||						Binary	Left to Right
	? :						Ternary	Left to Right
	=	+= -=	*= /=	^= %=	&= =	<<= >>=	Binary	Right to Left
Lowest	,						Binary	Left to Right

For example, in

a == b + c * d

c * d is evaluated first because * has a higher precedence than + and ==. The result is then added to b because + has a higher precedence than ==, and then == is evaluated. Precedence rules can be overridden using brackets. For example, rewriting the above expression as

```
a == (b + c) * d
```

causes + to be evaluated before *.

Operators with the same precedence level are evaluated in the order specified by the last column of Table 2.7. For example, in

```
a = b += c
```

the evaluation order is right to left, so first b += c is evaluated, followed by a = b.

2.8. Simple Type Conversion

A value in any of the built-in types we have see so far can be converted (type-cast) to any of the other types. For example:

```
(int) 3.14       // converts 3.14 to an int to give 3
(long) 3.14      // converts 3.14 to a long to give 3L
(double) 2       // converts 2 to a double to give 2.0
(char) 122       // converts 122 to a char whose code is 122
(unsigned short) 3.14    // gives 3 as an unsigned short
```

As shown by these examples, the built-in type identifiers can be used as type operators. Type operators are unary (i.e., take one operand) and appear inside brackets to the left of their operand. This is called explicit type conversion. When the type name is just one word, an alternate notation may be used in which the brackets appear around the operand:

```
int(3.14)        // same as: (int) 3.14
```

In some cases, C++ also performs implicit type conversion. This happens when values of different types are mixed in an expression. For example:

```
double  d = 1;           // d receives 1.0
int     i = 10.5;        // i receives 10
i = i + d;               // means: i = int(double(i) + d)
```

In the last example, i + d involves mismatching types, so i is first converted to double (promoted) and then added to d. The result is a double which does not match the type of i on the left side of the assignment, so it is converted to int (demoted) before being assigned to i.

The above rules represent some simple but common cases for type conversion.

2.9. Statements

This chapter introduces the various forms of C++ statements for composing programs. Statements represent the lowest-level building blocks of a program. Roughly speaking, each statement represents a computational step which has a certain side-effect. (A side-effect can be thought of as a change in the program state, such as the value of a variable changing because of an assignment.) Statements are useful because of the side-effects they cause, the combination of which enables the program to serve a specific purpose (e.g., sort a list of names).

A running program spends all of its time executing statements. The order in which statements are executed is called flow control (or control flow). This term reflect the fact that the currently executing statement has the control of the CPU, which when completed will be handed over (flow) to another statement. Flow control in a program is typically sequential, from one statement to the next, but may be diverted to other paths by branch statements. Flow control is an important consideration because it determines what is executed during a run and what is not, therefore affecting the overall outcome of the program.

Like many other procedural languages, C++ provides different forms of statements for different purposes. Declaration statements are used for defining variables. Assignment-like statements are used for simple, algebraic computations. Branching statements are used for specifying alternate paths of execution, depending on the outcome of a logical condition. Loop statements are used for specifying computations which need to be repeated until a certain logical condition is satisfied. Flow control statements are used to divert the execution path to another part of the program. We will discuss these in turn.

2.9.1. Input/Output Statements

The most common way in which a program communicates with the outside world is through simple, character-oriented Input/Output (IO) operations. C++ provides two useful operators for this purpose: >> for input and << for output. We have already seen examples of output using <<. Example 2.1 also illustrates the use of >> for input.

```
Example
```

```
#include <iostream.h>
int main (void)
{
        int             workDays = 5;
        float   workHours = 7.5;
        float   payRate, weeklyPay;

        cout << "What is the hourly pay rate? ";
        cin >> payRate;

        weeklyPay = workDays * workHours * payRate;
        cout << "Weekly Pay = ";
        cout << weeklyPay;
        cout << '\n';
   }
```

Analysis

This line outputs the prompt 'What is the hourly pay rate? ' to seek user input.

This line reads the input value typed by the user and copies it to payRate. The input operator >> takes an input stream as its left operand (cin is the standard C++ input stream which corresponds to data entered via the keyboard) and a variable (to which the input data is copied) as its right operand.

When run, the program will produce the following output (user input appears in **bold**):

```
What is the hourly pay rate? 33.55
Weekly Pay = 1258.125
```

Both << and >> return their left operand as their result, enabling multiple input or multiple output operations to be combined into one statement. This is illustrated by example below which now allows the input of both the daily work hours and the hourly pay rate.

```
Example
#include <iostream.h>

int main (void)
{
        int             workDays = 5;
        float   workHours, payRate, weeklyPay;

        cout << "What are the work hours and the hourly pay rate? ";
        cin >> workHours >> payRate;

        weeklyPay = workDays * workHours * payRate;
        cout << "Weekly Pay = " << weeklyPay << '\n';
}
```

Analysis

This line reads two input values typed by the user and copies them to workHours and payRate, respectively. The two values should be separated by white space (i.e., one or more space or tab characters). This statement is equivalent to:

(cin >> workHours) >> payRate;

Because the result of >> is its left operand, (cin >> workHours) evaluates to cin which is then used as the left operand of the next >> operator.

This line is the result of combining lines 10-12 from example 2.1. It outputs "Weekly Pay = ", followed by the value of weeklyPay, followed by a newline character. This statement is equivalent to:

((cout << "Weekly Pay = ") << weeklyPay) << '\n';

Because the result of << is its left operand, (cout << "Weekly Pay = ") evaluates to cout which is then used as the left operand of the next << operator, etc.

When run, the program will produce the following output:

```
What are the work hours and the hourly pay rate? 7.5  33.55
Weekly Pay = 1258.125
```

2.9.2. **Null** *statement*

Syntax:
;
Description: Do nothing

2.9.3. The block statement

Syntax:
{
[<Declarations>].
<List of statements/statement block>.
}

Any place you can put a single statement, you can put a compound statement, also called a block. A block begins with an opening brace ({) and ends with a closing brace (}).

Although every statement in the block must end with a semicolon, the block itself does not end with a semicolon. For example

```
{
    temp = a;
    a = b;
    b = temp;
}
```

This block of code acts as one statement and swaps the values in the variables a and b.

2.9.4. The **Assignment** *statement.*

Syntax:

```
<Variable Identifier> = < expression>;
```

Description:

The <expression> is evaluated and the resulting value is stored in the memory space reserved for <variable identifier>.

```
Eg: - int x,y ;
          x=5;
          y=x+3;
          x=y*y;
```