البرمجة بلغة C++

1.1. C++ language:

In 1970 two programmers, Brian Kernighan and Dennis Ritchie, created a new language called **C**. (The name came about because C was preceded by the old programming language, they were using called B). **C** was designed with one goal in mind: writing operating systems. The language was extremely simple and flexible and soon was used for many different types of programs.
In 1980 Bjarne Stroustrup started working on a new language, called **"C with Classes**". This language improved on **C** by adding a number of new features, the most important of which was classes.
In 1983, the name of the language was changed from **C with Classes** to **C++**. The **++** operator in the C language is an operator for incrementing a variable, Many new features were added around this time, the most notable of which are **virtual functions**, **function overloading**, **references with the & symbol**, **the const keyword**, and **single-line comments using two forward slashes**. Figure 1-1 shows the relationship of C and C++.

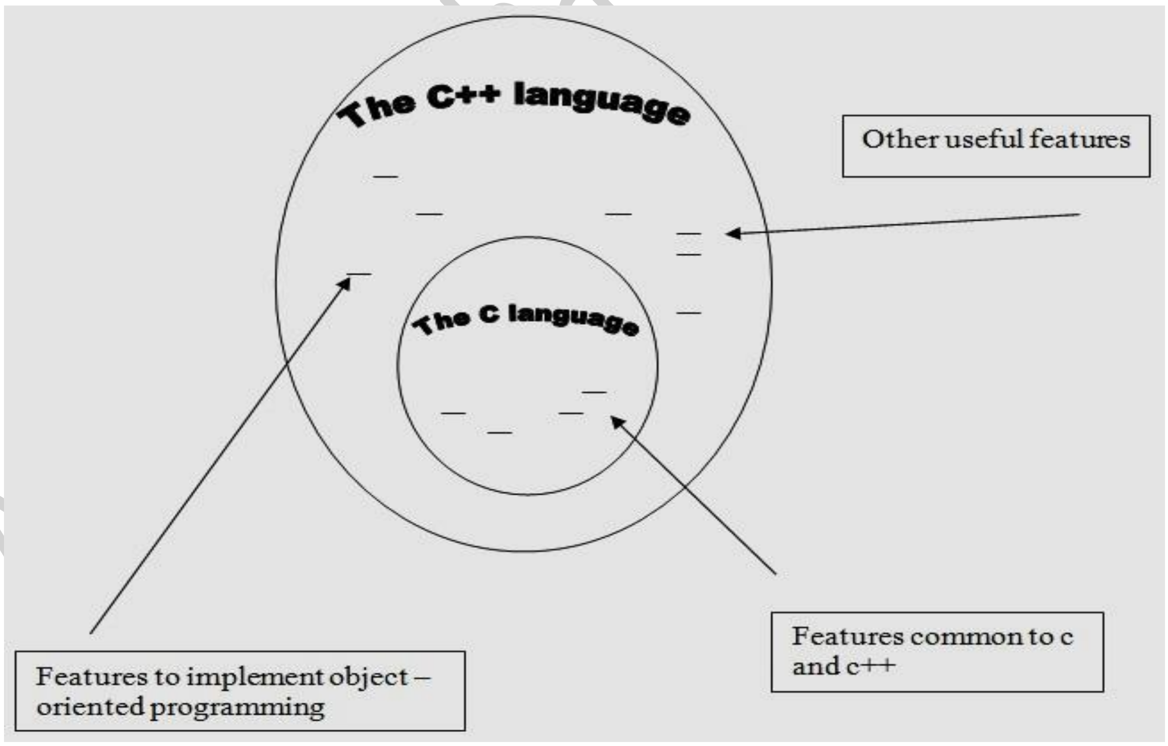

The C++ language
The C language
Other useful features
Features to implement object –oriented programming
Features common to c and c++

Figure 1-1

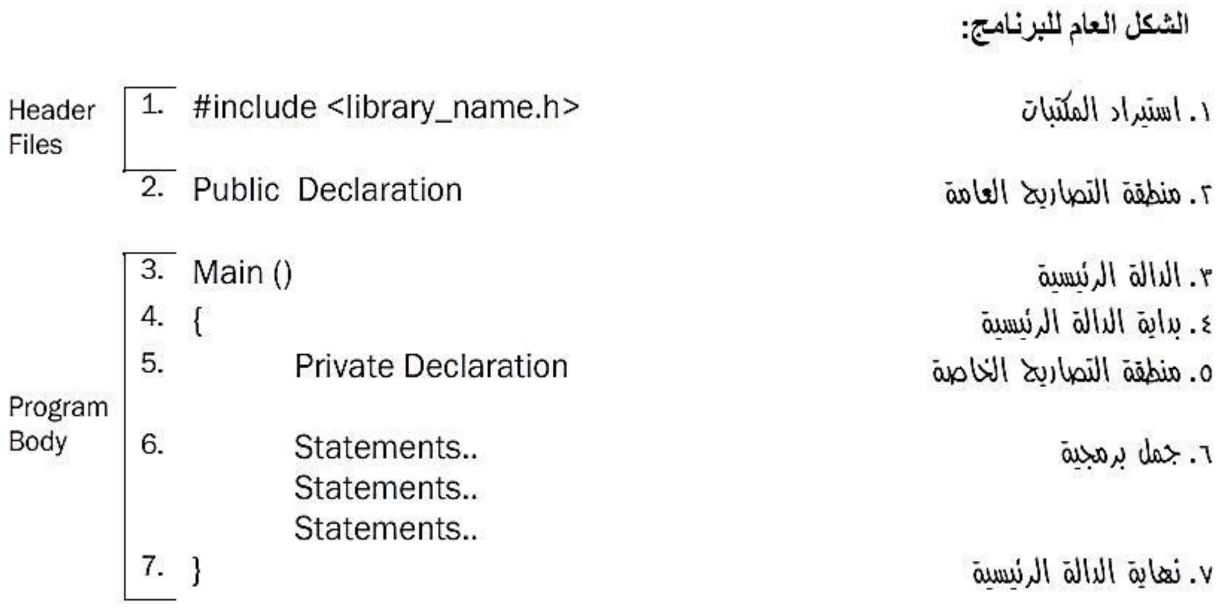
1.2. Why learn C++:

- C++ is a "middle level" language, therefore easier to learn and use than an assembly language, contains many of the low-level capabilities of an assembly language. Thus, C++ is sometimes called a "middle-level" language.
- C++ is portable. Most high-level languages are portable in the sense that a program written in a high-level language should execute, with minor modifications, on several different computer systems. C++ is one of the most portable of all the high-level languages. Properly written, a C++ program can run on several computer systems with no modifications. Portability is important because it is now common for a business to have several different types of computer systems (a mainframe, minicomputers, pcs and Macintoshes.
- C++ is small. The C++ language does not contain many of the built-in features present in other programming languages (such as visual basic has about 150 keywords on the other hand C++ has 60 keywords)
- C++ is an object-oriented extension of the c programming language.

1.3. C++ program contains:

C++ programs have parts and components that serve specific purposes. Every C++ program has an anatomy. Unlike human anatomy, the parts of C++ programs are not always in the same place. Nevertheless, the parts are there and your first step in learning C++ is to learn what they are.

1.3.1. Structure of a program in C++ is:



Probably the best way to start learning a programming language is by writing a program. Therefore, here is our first program (Program 1-1):

Program 1-1

```
// A simple C++ program
#include <iostream>


int main()
{
cout << "Programming is great fun!";
return 0;
}
```

The output of the program is shown below. This is what appears on the screen when the program runs.

```
Programming is great fun!
```

Let's examine the program line by line. Here's the first line: // A simple C++ program

The // marks the beginning of a comment. The compiler ignores everything from the **double-slash** to the end of the line. That means you can type anything you want on that line and the compiler will never complain! Although comments are not required, they are very important to programmers. Most programs are much more complicated than the example in Program 1-1, and comments help explain what's going on.

Line 2 looks like this: #include <iostream>

This line must be included in a C++ program in order to get input from the keyboard or print output to the screen. Since the **cout** statement (on line 7) will print output to the computer screen, we need to include this line. When a line begins with a **#** it indicates it is a preprocessor directive. The preprocessor reads your program before it is compiled and only executes those lines beginning with a **#** symbol. Think of the preprocessor as a program that "sets up" your source code for the compiler. The **#include** directive causes the preprocessor to include the contents of another file in the program. The word inside the brackets, **iostream**, is the name of the file that is to be included. The **iostream** file contains code that allows a C++ program to display output on the screen and read input from the keyboard. Because this program uses **cout** to display screen output, the **iostream** file must be included. Its contents are included in the program at the point the **#include** statement appears. The **iostream** file is called a header file, so it should be included at the head, or top, of the program.

Line 5 reads: int main()

This marks the beginning of a function. A function can be thought of as a group of one or more programming statements that has a name. The name of this function is **main**, and the set of parentheses that follows the name indicates that it is a function. The word **int** stands for "**integer**". It indicates that the function sends an integer value back to the operating system when it is finished executing. Although most C++ programs have more than one function, every C++ program must have a function called **main**. It is the starting point of the program. If you're ever reading someone else's program and want to find where it starts, just look for the function called **main**.

Line 6 contains a single, solitary character: {

This is called a **left-brace**, or an **opening brace**, and it is associated with the beginning of the function **main**. All the statements that make up a function are enclosed in a set of braces. If you look at the third line down from the opening brace you'll see the **closing brace**. Everything between the two braces is the contents of the function **main**.

After the **opening brace** you see the following statement in line 7:

```
cout << "Programming is great fun!";
```

To put it simply, this line displays a message on the screen. You will read more about **cout** and the << operator later. The message "**Programming is great fun!**" is printed without the **quotation marks**. In programming terms, the group of characters inside the quotation marks is called a *string literal*, a *string constant*, or *simply a string*.

Notice that line 7 ends with a **semicolon**. Just as a period marks the end of a sentence, a **semicolon** is required to mark the end of a complete statement in C++. But many C++ lines do not end with **semicolons**. Some of these include comments, preprocessor directives, and the beginning of functions. Here are some examples of when to use, and not use, semicolons.

```
// Semicolon examples // This is a comment
# include <iostream> // This is a preprocessor directive
int main() // This begins a function
cout << "Hello"; // This is a complete statement
```

As you spend more time working with C++ you will get a feel for where you should and should not use semicolons. For now don't worry about it. Just concentrate on learning the parts of a program.

Line 8 reads: `return 0;`

This sends the integer value 0 back to the operating system upon the program's completion. The value 0 usually indicates that a program executed successfully.

The last line of the program, line 9, contains the **closing brace**: }

This brace marks the end of the **main function**. Because **main** is the only function in this program, it also marks the end of the program.

In the sample program you encountered several sets of special characters. Table 1-1 provides a short summary of how they were used.

Table 1-1:

Character	**Name**	**Description**
//	Double slash	Marks the beginning of a comment.
#	Pound sign	Marks the beginning of a preprocessor directive.
< >	Opening and closing brackets	Encloses a filename when used with the **#include** directive.
( )	Opening and closing parentheses	Used in naming a function, as in **int main()**.
{ }	Opening and closing braces	Encloses a group of statements, such as the contents of a function.
" "	Opening and closing quotation marks	Encloses a string of characters, such as a message that is to be printed on the screen.
;	Semicolon	Marks the end of a complete programming statement.

البرمجة بلغة C++

2.1.Basic files in C++:

C++ code files (with **a .cpp** extension) are not the only files commonly seen in C++ programs. The other type of file is called a **header file**, sometimes known as an **include file**.

Header files contain definitions of **Functions and Variables**, which is imported or used into any C++ program by using the pre-processor **#include** statement. **Header files** usually have **a .h** extension, but you will sometimes see them with **a .hpp** extension or no extension at all. The purpose of a **header file** is to hold declarations for other files to use.

Types of **Header files**:

- **System header files:** It is comes with compiler.
- **User header files:** It is written by programmer.

Both user and system **header files** are include using the pre-processing directive **#include**. Here is simple explanation for a few basic files, that C++ program include (header files):

Table 1-2: Simple example for a few basic **System header files in C++**

Header File	**Function and Description**
<iostream>	This file defines the **cin, cout, cerr** and **clog** objects, which correspond to the standard input stream, the standard output stream, the un-buffered standard error stream and the buffered standard error stream, respectively.
<stdio.h>	This file defines the **scanf** and **printf** objects, which correspond to the standard input stream, the standard output stream.
< math.h>	This file defines the **abs(x)**, **sin(x)**, **cos(x)**, **tan(x)**, **sinh(x)**, **cosh(x)**, **tanh(x)**, **pow(x , y)**, **exp(x)**, **sqrt(x)** and … etc objects.

2.2.The main() Function:

A C++ program is a collection of functions that work together to solve a problem. The collection of functions that make up a C++ program must contain exactly one function called **main()**. A C++ program automatically begins execution at the first executable statement of the function **main()**.

The statements in **main()** must be enclosed in braces**{…}**. Each statements ends in a **semicolon ;**.

It is very important to keep in mind that C++ is case sensitive. That is, when coding C++ statements there is a difference between **uppercase** and **lowercase** letters.

The last statement executed by **main()** should be the **return** statement, we declare the function **main()** to produce an **integer value**. The **return 0;** statement does two things. First , it ends the execution of **main()**. Second , it sends the integer value 0 back to the operating system to indicate that the program ended normally. Here is the general form of **main()** function:

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

The **function header** is a capsule summary of the function's interface with the rest of the program, and the **function body** represents instructions to the computer about what the function should do. In C++ each complete instruction is called a **statement**. You must terminate each **statement** with a **semicolon**, so don't omit the **semicolons** when you type the examples.

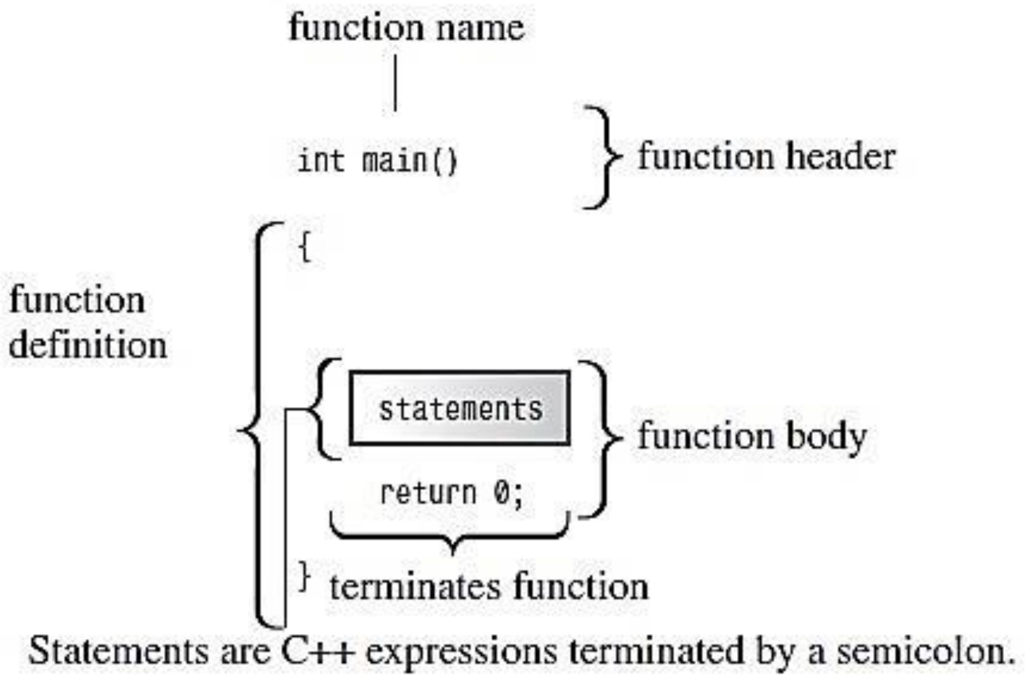


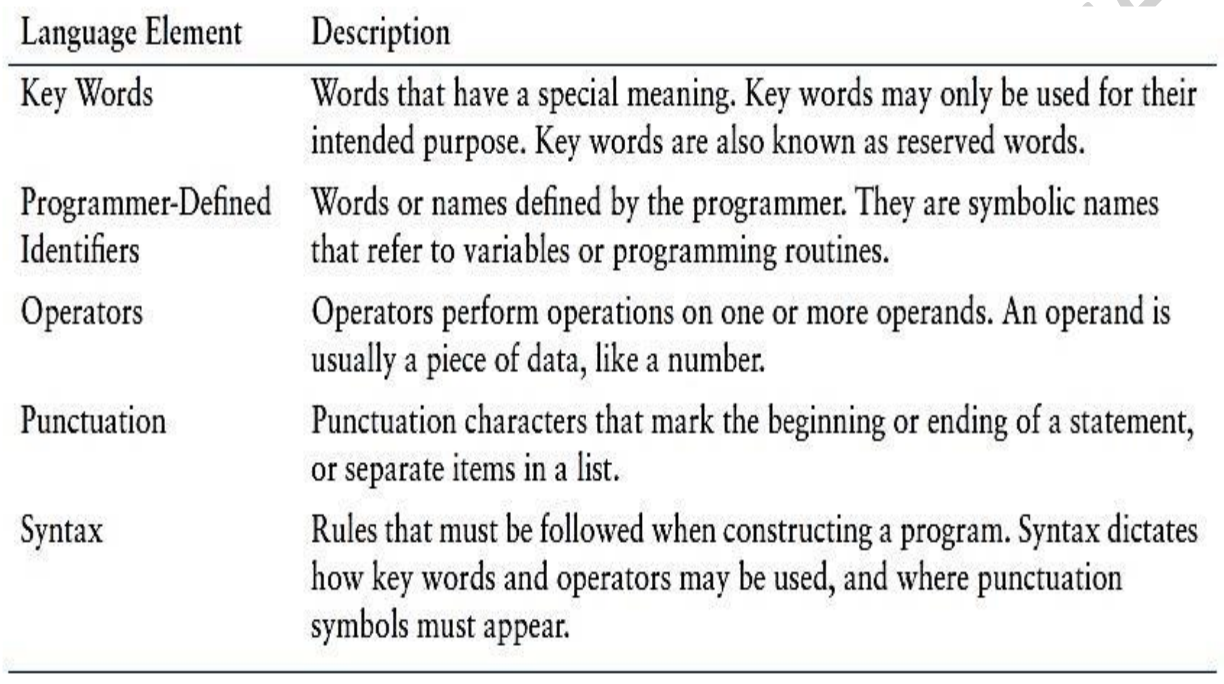
Figure 1-2

البرمجة بلغة ++C

3.1. Basic element and tools of C++ language:

There are certain elements(Language Elements) that are common to all programming languages. All programming languages have a few things in common. Table 2-1 lists the common elements found in almost every language.

Table 2-1:



Language Element	Description
Key Words	Words that have a special meaning. Key words may only be used for their intended purpose. Key words are also known as reserved words.
Programmer-Defined Identifiers	Words or names defined by the programmer. They are symbolic names that refer to variables or programming routines.
Operators	Operators perform operations on one or more operands. An operand is usually a piece of data, like a number.
Punctuation	Punctuation characters that mark the beginning or ending of a statement, or separate items in a list.
Syntax	Rules that must be followed when constructing a program. Syntax dictates how key words and operators may be used, and where punctuation symbols must appear.

3.1.1. Key Words in C++ language:

Are words always written in **lowercase**, each have a special meaning in C++ and can only be used for their intended purposes. As you will see, the programmer is allowed to make up his or her own names for certain things in a program. **Key words**, however, are reserved and cannot be used for anything other than their designated purposes. Part of learning a programming language is learning what the key words are, what they mean, and how to use them. Here is a list of **reserved words**(keywords) in C++:

asm, bool, break, case, catch, char, class, compl, const, continue, default, delete, do, double, else, false, float, for, goto, if, inline, int, long, namespace, new, not, nullptr, or, private, public, register, return, short, signed, sizeof, static, struct, switch, true, unsigned, void, wchar_t, while, xor and xor_eq.

3.1.2. Predefined Identifiers:

Beginning C++ programmers are sometimes confused by the difference between the two terms ***reserved word*** and ***predefined identifier***, and some potential for confusion. One of the difficulties is that some keywords that one might "expect" to be reserved words are not. The keyword **main** is a prime example, and others include things like the **endl** manipulator and other keywords from the vast collection of C++ libraries.

For example, you could declare a variable called **main** inside your **main function**, initialize it, and then print out its value (but ONLY do that to verify that you can!). On the other hand, you could not do this with a variable named **else**. The difference is that **else** is a ***reserved word***, while **main** is "only" a ***predefined identifier***. Here is a short list of some **predefined identifiers**:

cin, endl, INT_MIN, iomanip, main, npos, std, cout, include, INT_MAX, iostream, MAX_RAND NULL and string.

3.1.3. Programmer-Defined(Identifiers):

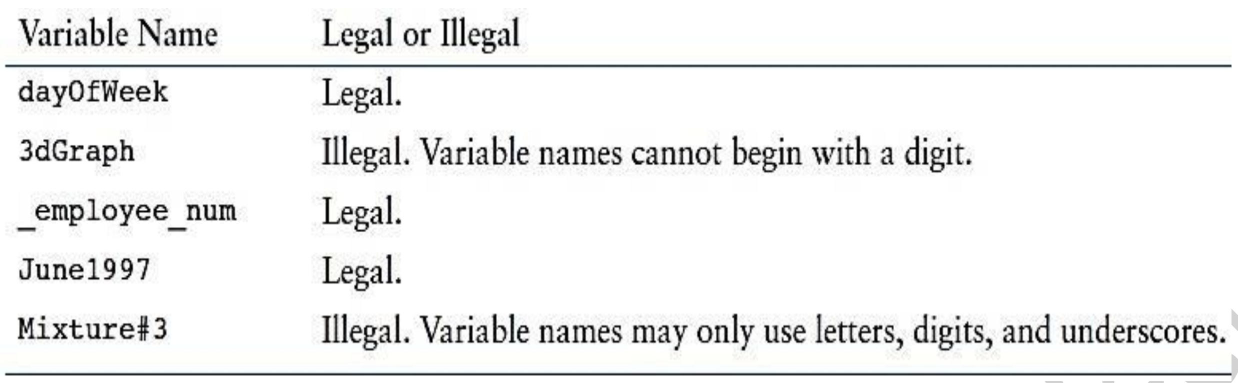
They are not part of the C++ language but rather are names made up by the programmer. They are the names of **variables**. **variables** are the names of memory locations that may hold data.

- **Legal Identifiers:**

  Regardless of which style you adopt, be consistent and make your **variable names** as sensible as possible. Here are some specific rules that must be followed with all C++ identifiers:

1- The first character must be one of the letters **a** through **z**, **A** through **Z**, or an underscore character (_).
2- After the first character you may use the letters **a** through **z** or **A** through **Z**, the digits **0** through **9**, or underscores.
3- **Uppercase** and **lowercase** characters are distinct. This means **ItemsOrdered** is not the same as **itemsordered**. Table 2-2 lists **variable names** and indicates whether each is legal or illegal in C++.

Table 2-2:



Variable Name	Legal or Illegal
dayOfWeek	Legal.
3dGraph	Illegal. Variable names cannot begin with a digit.
_employee_num	Legal.
June1997	Legal.
Mixture#3	Illegal. Variable names may only use letters, digits, and underscores.

3.1.4. Operator:

An **operator** is a symbol that tells the compiler to perform specific mathematical or logical manipulations. C++ is rich in **built-in operators** and provide the following types of **operators**:

- Arithmetic Operators **(+, -, *, /, %, ++** and **--)**.
- Relational Operators **(==, !=, >, <, >=** and **<=)**.
- Logical Operators **(||, &&** and **!)**.
- Bitwise Operators **(&, |, ^, ~, <<** and **>>)**.
- Assignment Operators **(=, +=, -=, *=, /=** and **%=)**.
- Misc Operators.

Later will examine the arithmetic, relational, logical, bitwise, assignment and other operators one by one.

3.1.5. Punctuation:

They are characters(symbols) have special meanings in C++ like: **[**, **]**, **(**, **)**, **{**, **}**, *****, **,**, **:**, **=**, **;** and **#.**

3.1.6. Syntax:

Syntax are legal uses of key words, operators, punctuation, and other language elements. If the program is free of **syntax errors**, the compiler stores the translated **machine language** instructions, which are called **object code**, in an **object file**.

البرمجة بلغة C++

4.1.Constant represent:

Constants are data items whose values cannot change while the program is running. By symbolic names for constants can suggest what the constant represents. Also, if the program uses the constant in several places and you need to change the value, you can just change the single symbol definition. We can use simple way in C++ to **define constants** by **const** keyword.

The **general form** to **declaration** and **initialization** constant in this way is:

const data-type CONSTANT_NAME= value**;**

Now you can use MONTHS in a program instead of 12. After you initialize a constant such as MONTHS, its value is set. The compiler does not let you subsequently change the value MONTHS.

const int MONTHS = 12**;** // Months is symbolic constant for 12

A common practice is to use all **uppercase** for the name to help remind yourself that MONTHS is a constant.

4.2.Variables represent:

Variables represent **storage locations**(**reference**) in the computer's memory. The concept of a **variable** in computer programming is somewhat different from the concept of a variable in mathematics. In programming, a **variable** is a named storage location for holding data. **Variables** allow you to store and work with data in the computer's memory. They provide an "interface" to RAM. Part of the job of programming is **to determine how many variables a program will need** and **what type of information each will hold**. To store an item of information in a computer, the program must keep track of three fundamental properties:

1. Where the information is stored.
2. What value is kept there.
3. What kind of information is stored.

Here is the **general form** to **declaration** a variable in C++:

data-type variable _name**;**

The **data type** used in the **declaration** describes the kind of information, and the **variable name** represents the value symbolically(memory location label). For example, suppose we have the following statements:

int braincount**;** // declaration variable **braincount**
braincount **=** 5**;** // initialization variable **braincount** by give it value 5

These statements tell the program that it is storing an integer and that the name **braincount** represents the **integer's value**, **5** in this case.
And here is the **general form** to **declaration** and **initialization** a variable at the same time in C++:

data-type variable _name **=** value**;**

For example:

int braincount**=5;** // declaration and initialization variable **braincount**

4.3. Data types in C++, and they represent methods in memory:

Data used by your program is stored in memory and manipulated by various data structure techniques, depending on the nature of your program. Let's take a close look at main memory and how data is stored in memory before exploring how to manipulate data using data structures.

Memory is a bunch of electronic switches called ***transistors*** that can be placed in one of two states: **on** or **off**. The state of a switch is meaningless unless you assign a value to each state, which you do using the **binary numbering system**.

The ***binary numbering system*** consists of two digits called ***binary digits*** (**bits**): **zero** and **one**. A switch in the **off** state represents **zero**, and a switch in the **on** state represents **one**. This means that one transistor can represent one of two digits.

Memory is organized into groups of **eight bits** called a ***byte***, enabling **256** combinations of **zeros** and **ones** that can store numbers from **0 through 255**.

Although a unit of memory holds a **byte**, data used in a program can be larger than a **byte** and require 2, 4, or 8 **bytes** to be stored in memory. Before any data can be stored in memory, you must tell the computer how much space to reserve for data by using a **data type**.

Memory is reserved by using a data type in a **declaration statement**. The form of a **declaration statement** to **variables** in C++ you learned about it in the previous week (**2.4. Variables represent**). There are many different types of data. Variables are classified according to their data type, which determines the kind of information that may be stored in them.

Although C++ offers many data types, in the very broadest sense there are only two: **numeric** and **character**. **Numeric** data types are broken into two additional categories: **integer** and **floating-point**, as shown in Figure 2-1.

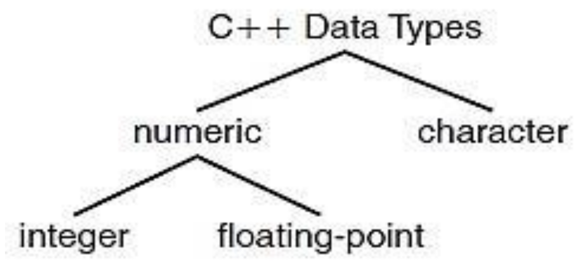


Figure 2-1

C++ offers the programmer a rich assortment of **built-in** as well as user defined data types. Following table lists down seven basic C++ data types:

Table 2-3:

Data Type	**Keyword in C++**	**Data Type Size in bits (Bytes)**
Boolean	bool	1 B
Characters	char	16 bit =(2 Byte)
Integers	int	32 b =(4 B)

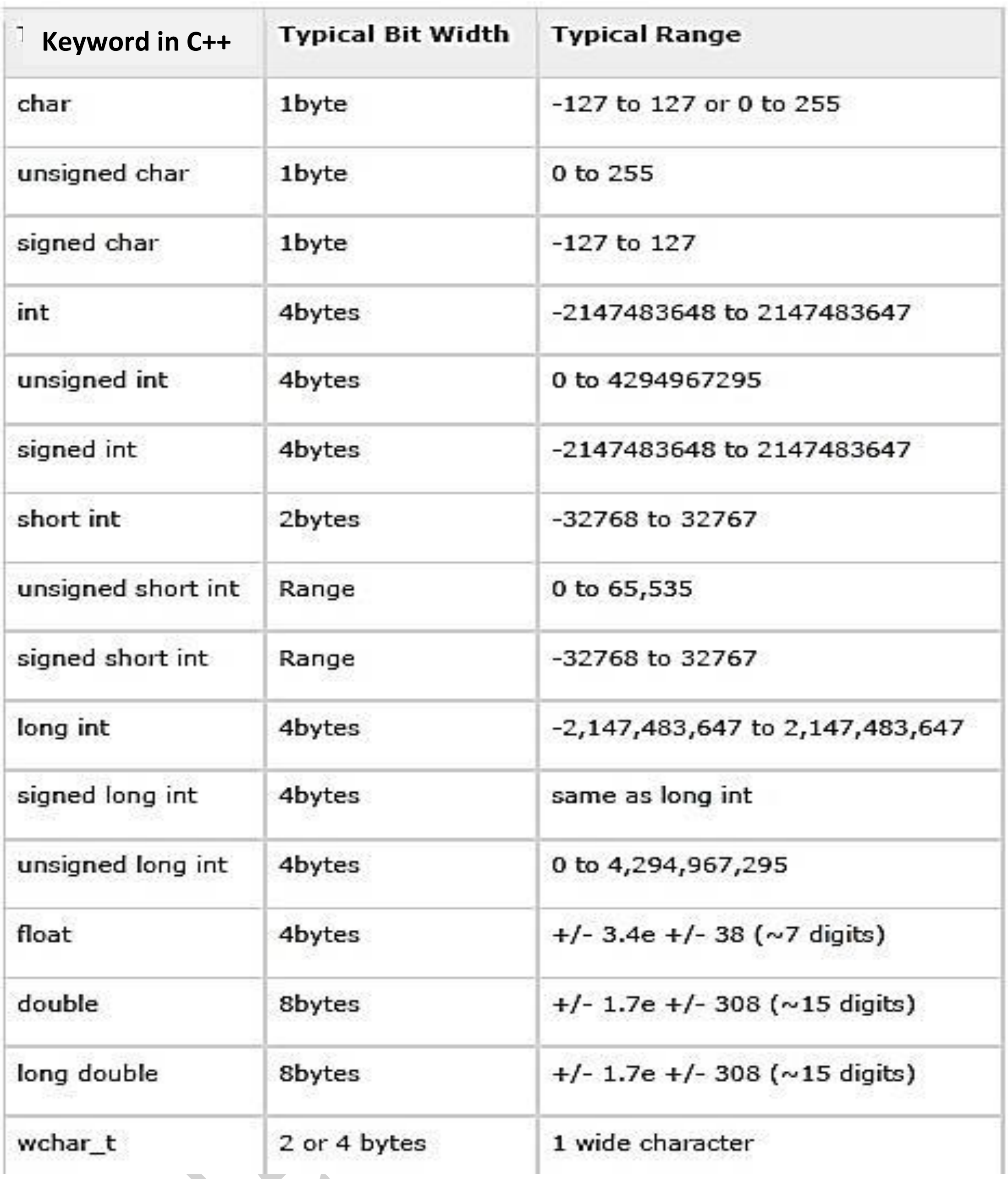
Data Type	**Keyword in C++**	**Data Type Size in bits (Bytes)**
Floating-point	float	32 b =(4 B)
Double floating point	double	64 b =(8 B)

Several of the basic types can be modified using one or more of these type modifiers:

- signed
- unsigned
- short
- long

The following table shows the variable type, how much memory it takes to store the value in memory, and what is maximum and minimum value which can be stored in such type of variables.

Table 2-4:



Keyword in C++	Typical Bit Width	Typical Range
char	1byte	-127 to 127 or 0 to 255
unsigned char	1byte	0 to 255
signed char	1byte	-127 to 127
int	4bytes	-2147483648 to 2147483647
unsigned int	4bytes	0 to 4294967295
signed int	4bytes	-2147483648 to 2147483647
short int	2bytes	-32768 to 32767
unsigned short int	Range	0 to 65,535
signed short int	Range	-32768 to 32767
long int	4bytes	-2,147,483,647 to 2,147,483,647
signed long int	4bytes	same as long int
unsigned long int	4bytes	0 to 4,294,967,295
float	4bytes	+/- 3.4e +/- 38 (~7 digits)
double	8bytes	+/- 1.7e +/- 308 (~15 digits)
long double	8bytes	+/- 1.7e +/- 308 (~15 digits)
wchar_t	2 or 4 bytes	1 wide character

4.3.1. Char data type:

A variable of the char data type holds only a single character.

You might be wondering why there isn't a 1-byte integer data type. Actually there is. It is called the **char** data type, which gets its name from the word “**character**". A variable defined as a **char** can hold a single character, but strictly speaking, it is an integer data type.

The reason an integer data type is used to store characters is because **characters are internally represented by numbers**. The most commonly used method for encoding characters is **ASCII**, which stands for the **American Standard Code for Information Interchange**.

When a character is stored in memory, it is actually the numeric code that is stored. When the computer is instructed to print the value on the screen, it displays the character that corresponds with the numeric code. **Notice** that the number 65 is the code for A, 66 is the code for B, and so on.

Figure 2-2 illustrates that when you think of characters, such as A, B, and C, being stored in memory, it is really the numbers 65, 66, and 67 that are stored.

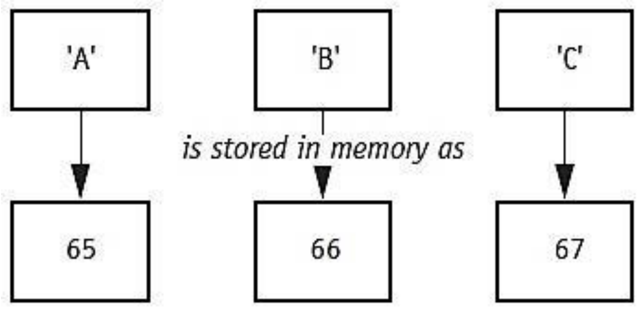


Figure 2-2

Here is the **general form** to **declaration** a variable of character type in C++:

char variable _name**;** // declaration variable of character type

char variable _name=Value**;** // declaration and initialization variable of character type

Characters are enclosed in single quotation marks **'**. 'A', 'a' and ' ! ' are character constants.

For example: **Char** letter**;** // valid in C++

Char letter**1=**'A'**;** // valid in C++

Character constants can only hold a single character. To store a series of characters in a constant we need a string constant. In the following example, 'H' is a character constant and "Hello" is a string constant. Notice that a character constant is enclosed in single quotation marks whereas a string constant is enclosed in double quotation marks.

cout << 'H'**;**

cout << "Hello"**;**

4.3.2. Integer data type:

Integer variables can only hold whole numbers.

Your primary considerations for selecting the best data type for a numeric variable are the following:

- whether the variable needs to hold integers or floating-point values,
- the largest and smallest numbers that the variable needs to be able to store,
- whether the variable needs to hold signed (both positive and negative) or only unsigned (just zero and positive) numbers, and
- the number of decimal places of precision needed for values stored in the variable.

Later in lap lecture you will learn to use the **sizeof** operator to determine how large all the data types are on your computer.
Here is the **general form** to **declaration** a variable of integer type in C++:

int variable _name**;** // declaration variable of integer type

int variable _name=Value**;** // declaration and initialization variable of integer type

For example: **int** num**;** // valid in C++

int num**=10;** // valid in C++

4.3.3. Real data type (floating point numbers):

Real numbers are numbers that have a **fractional part**. Because of the way they are stored internally, real numbers are also known as ***floating point numbers***. The numbers 5.5, 8.3, and -12.6 are all floating point numbers. C++ uses the decimal point to distinguish between **floating point numbers** and **integers**, so a number such as 5.0 is a floating point number while 5 is an integer. **Floating point numbers** must contain a decimal point. Numbers such as 3.14159, 0.5, 1.0, and 8.88 are floating point numbers.

Here is the **general form** to **declaration** a variable of floating point type in C++:

float variable _name**;** // declaration variable of floating point type

float variable _name=Value**;** // declaration and initialization variable of floating point type

For example: **float** num**;** // valid in C++

float num=**5.3;** // valid in C++

4.3.4. Boolean (logical) data type:

Boolean variables are set to either true or false.

Expressions that have a true or false value are called **Boolean** expressions, named in honor of English mathematician George Boole (1815–1864). The **bool** data type allows you to create variables that hold true or false values. It is actually an integer variable that stores 0 for false and 1 for true.

Here is the **general form** to **declaration** a variable of boolean type in C++:

bool variable _name**;** // declaration variable of boolean type

bool variable _name=Value**;** // declaration and initialization variable of boolean type

For example: **bool** flag**;** // valid in C++

bool flag**=False;** // valid in C++

البرمجة بلغة ++C

5. A statement

a **statement** is a part of your program that can be executed. That is, a **statement** specifies an action. C and C++ categorize **statements** into these groups:

1. Declaration
2. Expression
3. Selection
4. Iteration
5. Jump
6. Label
7. Block
8. Try block

Included in the **declaration statements** are declaration variables, constants, functions and so on. **Expression statements** are statements composed of a valid expression. Selection statements are **if** and **switch**. (The term ***conditional statement*** is often used in place of "selection statement.") The **iteration statements** are **while**, **for**, and **do-while**. These are also commonly called ***loop statements***. The **jump statements** are **break**, **continue**, **goto**, and **return**. The **label statements** include the **case** and **default** statements (discussed along with the **switch** statement) and the label statement (discussed with **goto**). **Block statements** are simply blocks of code. (A block begins with a { and ends with a }.) Block statements are also referred to as ***compound statements***.

declaration statements already we learned about it in the previous week 1 and 2. Here we will discuss **expressions statements**.

5.1. Expressions types in C++ language:

An **expression** in a programming language is a combination of one or more
explicit values, constants, variables, operators, and functions that the programming language interprets (according to its particular rules of precedence and of association) and computes to produce another value.

An **expression** in C++ is any valid combination of its elements. Because most expressions tend to follow the general rules of algebra, they are often taken for granted. However, a few aspects of expressions relate specifically to C++. And we will discuss some of these expressions like:

1- Arithmetic expression.
2- Relational expression.
3- Logical expression.
4- Compound expression.

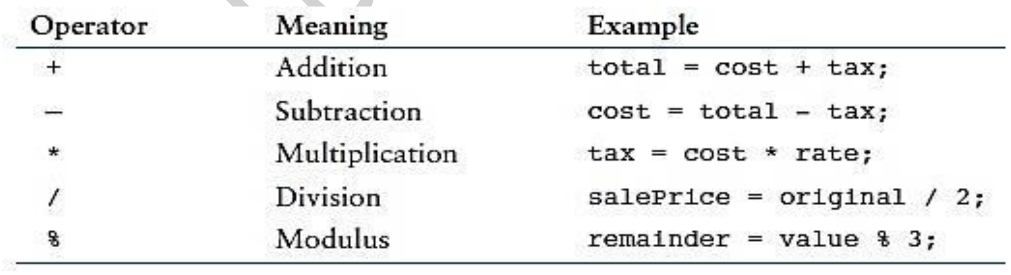
1. Arithmetic expression:

An arithmetic expression is a syntactically correct combination of numbers, operators, parenthesis, and variables. And basically we need to know what are the arithmetic operation and its priorities.
C++ provides many operators for manipulating data. Generally, there are three types of operators: **unary**, **binary**, and **ternary**. These terms reflect the number of operands an operator requires.

- **Unary** operators only require a single operand. For example, consider the following expression: −5 Of course, we understand this represents the value negative five. The constant 5 is preceded by the minus sign.
- **Binary** operators work with two operands.
- **Ternary** operators, as you may have guessed, require three operands. C++ only has one **ternary** operator, which will be discussed later.

Arithmetic operations occur frequently in programming. Table 3-1 shows the common arithmetic operators in C++. All are **binary** operators.

Table 3-1:



Operator	Meaning	Example
+	Addition	`total = cost + tax;`
–	Subtraction	`cost = total - tax;`
*	Multiplication	`tax = cost * rate;`
/	Division	`salePrice = original / 2;`
%	Modulus	`remainder = value % 3;`

Here is an example of how each of these operators works.

The ***addition operator*** returns the sum of its two operands.

total = 4 + 8; // total is assigned the value 12

The ***subtraction operator*** returns the value of its right operand subtracted from its left operand.

candyBars = 8 - 3; // candyBars is assigned the value 5

The ***multiplication operator*** returns the product of its two operands.

points = 3 * 7 // points is assigned the value 21

The ***division operator*** works differently depending on whether its operands are integer or floating point numbers. When both numbers are integers, the division operator performs ***integer division***. This means that the result is always an integer. If there is any remainder, it is discarded.

fullBoxes = 26 / 8; // fullBoxes is assigned the value 3

The variable fullBoxes is assigned the value 3 because 8 goes into 26 three whole times with a remainder of 2. The remainder is discarded. If you want the division operator to perform regular division, you must make sure at least one of the operands is a floating point number.

boxes = 26.0 / 8; // boxes is assigned the value 3.25

The ***modulus operator*** computes the ***remainder*** of doing an integer divide.

leftOver = 26 % 8; // leftOver is assigned the value 2

Figure 3-1 illustrates the use of the integer divide and modulus operations.

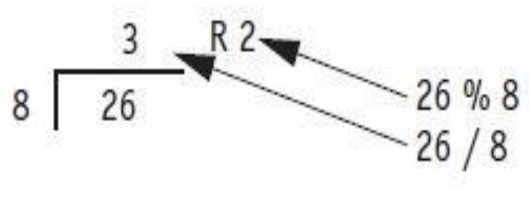


Figure 3-1

- **Operator Precedence(priorities):** It is possible to build mathematical expressions with several operators. The following statement assigns the sum of 17, ***x***, 21, and ***y*** to the variable **answer**.

  answer = 17 + x + 21 + y;

Some expressions are not that straightforward, however. Consider the following statement:

outcome = 12 + 6 / 3;

What value will be stored in **outcome**? It could be assigned either 6 or 14, depending on whether the addition operation or the division operation takes place first. The answer is 14 because the division operator has higher **priorities** than the addition operator.

So the example statement works like this:

A) 6 is divided by 3, yielding a result of 2

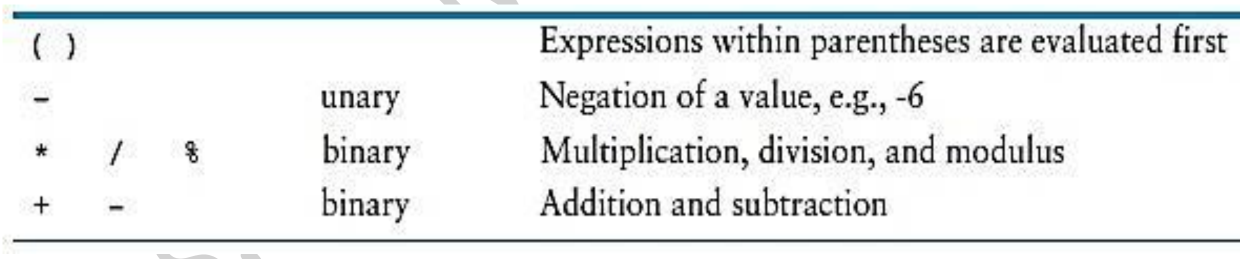
B) 12 is added to 2, yielding a result of 14

It could be diagrammed in the following way:

$$\begin{tikzcd}12 + 6 / 3 \arrow[d] \\ 12 + 2 \arrow[d] \\ 14\end{tikzcd}$$

Table 3-2 shows the **priorities** of the arithmetic operators. The operators at the top of the table have higher **priorities** than the ones below it.

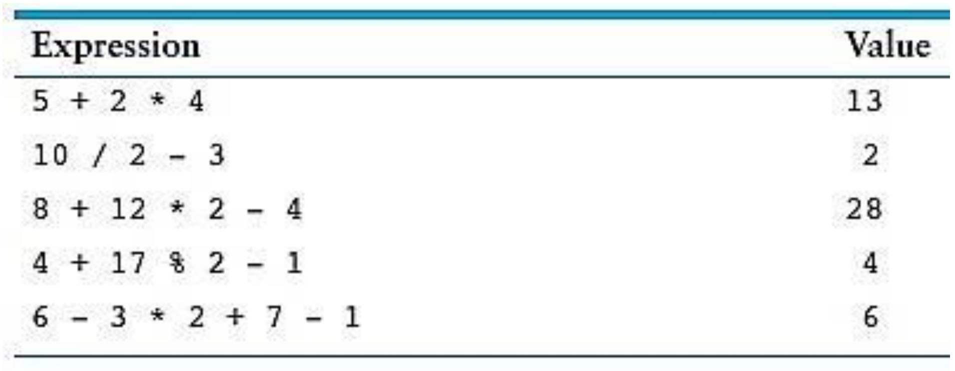
Table 3-2:



( )				Expressions within parentheses are evaluated first
-			unary	Negation of a value, e.g., -6
*	/	%	binary	Multiplication, division, and modulus
+	-		binary	Addition and subtraction

The multiplication, division, and modulus operators have the same precedence. This is also true of the addition and subtraction operators. Table 3-3 shows some expressions with their values.

Table 3-3:



Expression	Value
`5 + 2 * 4`	13
`10 / 2 - 3`	2
`8 + 12 * 2 - 4`	28
`4 + 17 % 2 - 1`	4
`6 - 3 * 2 + 7 - 1`	6

Note: Without parentheses, the computer evaluates the arithmetic operators in an expression according to the following:

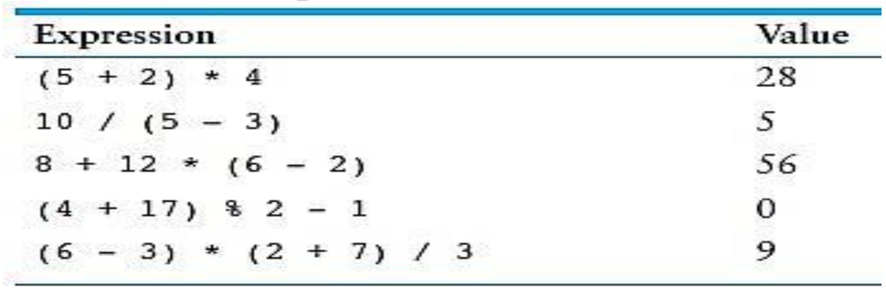
1- Do multiplications, divisions, and remainder operations first. If there are more than one such operation, do them in order from left to right.
2- Next, do additions and subtractions. If there are more than one such operation, do them in order from left to right.

- **Grouping with Parentheses**: Parts of a mathematical expression may be grouped with parentheses to force some operations to be performed before others. In the following statement, the sum of a plus b is divided by 4.

average = (a + b) / 4;

Without the parentheses b would be divided by 4 **before** adding a to the result. Table 3-4 shows more expressions and their values.

Table 3-4:



Expression	Value
`(5 + 2) * 4`	28
`10 / (5 - 3)`	5
`8 + 12 * (6 - 2)`	56
`(4 + 17) % 2 - 1`	0
`(6 - 3) * (2 + 7) / 3`	9

- **Converting Algebraic Expressions to Programming Statements:** In algebra it is not always necessary to use an operator for multiplication. C++, however, requires an operator for any mathematical operation. Table 3-5 shows some algebraic expressions that perform multiplication and the equivalent C++ expressions.

Table 3-5:

Algebraic Expression	Operation	C++ Equivalent
6B	6 times B	`6 * B`
(3)(12)	3 times 12	`3 * 12`
4xy	4 times x times y	`4 * x * y`

When converting some algebraic expressions to C++, you may have to insert parentheses that do not appear in the algebraic expression. For example, look at the following expression:

$$x = \frac{a + b}{c}$$

To convert this to a C++ statement, *a* + *b* will have to be enclosed in parentheses:

```
x = (a + b) / c;
```

Table 3-6 shows more algebraic expressions and their C++ equivalents.

Table 3-6:

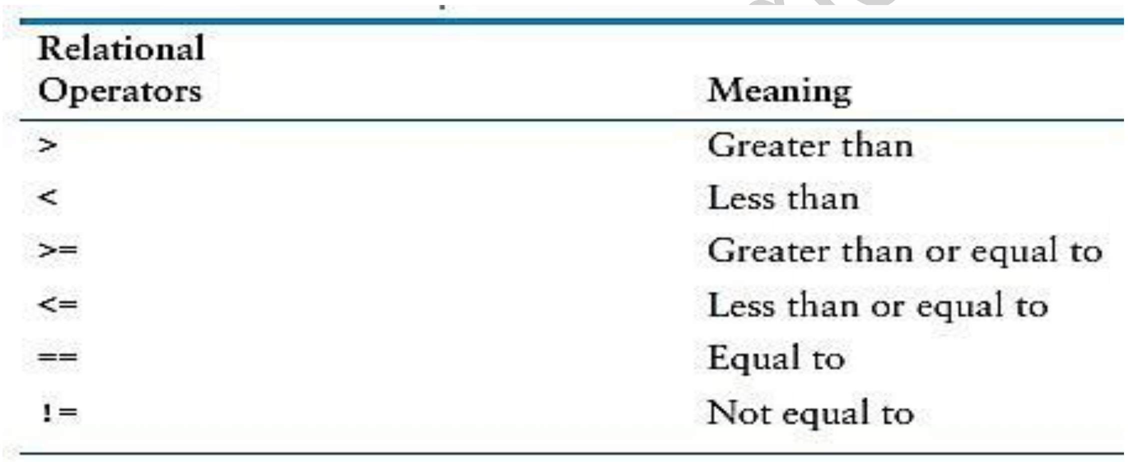
Algebraic Expression	C++ Expression
$y = 3\frac{x}{2}$	`y = x / 2 * 3;`
$z = 3bc + 4$	`z = 3 * b * c + 4;`
$a = \frac{3x + 2}{4a - 1}$	`a = (3 * x + 2) / (4 * a - 1)`

البرمجة بلغة ++C

6.1. Relational Expressions:

Computers are more than relentless number crunchers. They have the capability to compare values, and this capability is the foundation of computer **decision making**. In C++ **relational operators** embody this ability. C++ provides **six relational operators** to compare numbers. Because characters are represented by their ASCII codes, you can use these operators with characters, too.
All expressions have a value. **Relational expressions** are ***Boolean expressions***, which means their value can only be ***true*** or ***false***. Each **relational expression** reduces to the **bool value true** if the comparison is true and to the **bool value false** if the comparison is false. (Older implementations evaluate **true relational expressions** to **1** and **false relational expressions** to **0**.) the next table summarizes these operators.

Table 4-1: **Relational Operators**



Relational Operators	Meaning
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to
==	Equal to
!=	Not equal to

All of the relational operators are binary. This means they use two operands. Here is an example of an expression using the greater-than operator:

$$x > y$$

This expression is called a ***relational expression***. It is used to **determine whether x is greater than y**. The following expression determines whether **x** is less than **y**:

$$x < y$$

If x is greater than y, the expression x > y will be true, while the expression y == x will be false.

The **== operator determines whether the operand on its left is equal to the operand on its right**. If both operands have the same value, the expression is true.

Assuming that **a** is 4, the following expression is true: **a** == 4

But the following is false: a == 2

A couple of the relational operators actually test for two relationships. **The >= operator determines whether the operand on its left is greater than *or* equal to the operand on the right**.

Assuming that **a** is 4, **b** is 6, and **c** is 4, both of the following expressions are true:

```
b >= a
a >= c
```

But the following is false:

```
a >= 5
```

The **<= operator determines whether the operand on its left is less than *or* equal to the operand on its right**. Once again, assuming that **a** is 4, **b** is 6, and **c** is 4, both of the following expressions are true:

```
a <= c
b <= 10
```

But the following is false:

```
b <= a
```

The last relational operator is **!=**, which is the **not-equal operator**. **It determines whether the operand on its left is not equal to the operand on its right**, which is the opposite of the == operator. As before, assuming **a** is 4, **b** is 6, and **c** is 4, both of the following expressions are true:

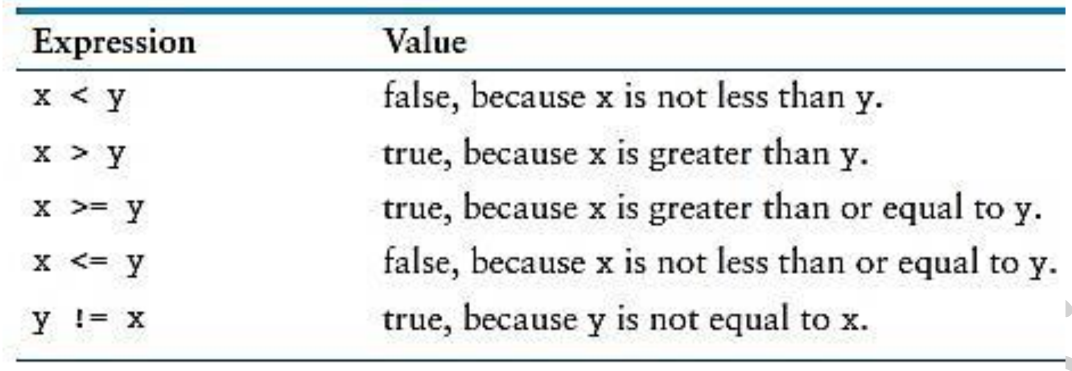
```
a != b
b != c
```

These expressions are true because **a** is ***not* equal** to **b** and **b** is ***not* equal** to **c**. But the following expression is false because **a** ***is* equal** to **c**:

```
a != c
```

Table 4-2 shows Example Relational Expressions (Assume **x** is 10 and **y** is 7.).

Table 4-2**:**

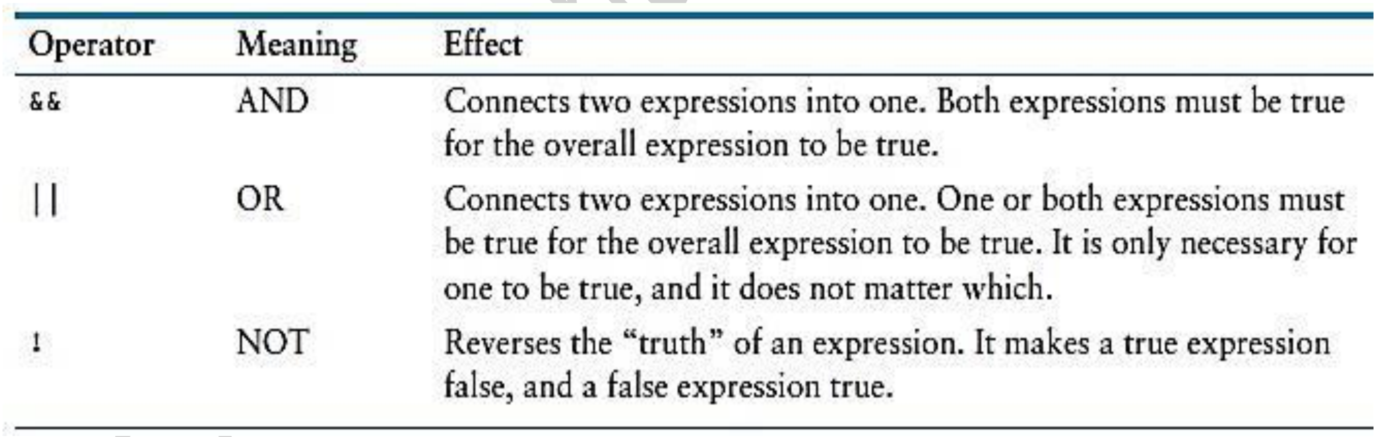


Expression	Value
x < y	false, because x is not less than y.
x > y	true, because x is greater than y.
x >= y	true, because x is greater than or equal to y.
x <= y	false, because x is not less than or equal to y.
y != x	true, because y is not equal to x.

6.2. Logical Expressions:

Often you must test for more than one condition. For example, for a character to be a lowercase letter, its value must be **greater than or equal to** 'a' **and less than or equal to** 'z'. Or, if you ask a user to respond with a **y** or an **n**, you want to accept uppercase (**Y** and **N**) as well as lowercase. To meet this kind of need, C++ provides **three logical operators** to combine or modify existing expressions. The next table shows these operators:

Table 4-3: **Logical Operators**



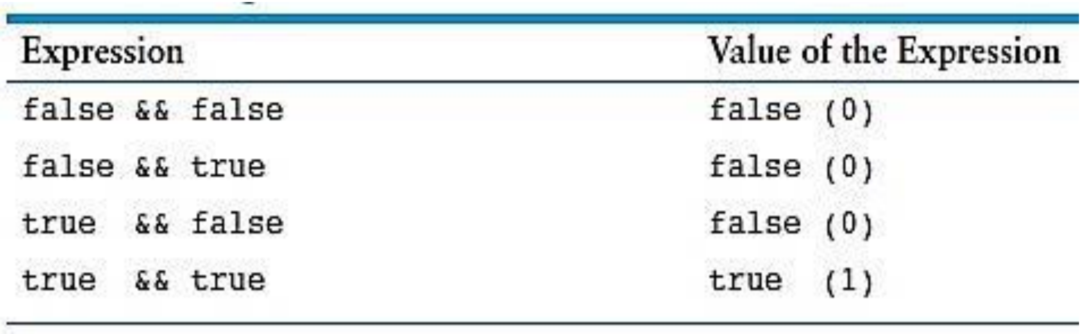
Operator	Meaning	Effect
&&	AND	Connects two expressions into one. Both expressions must be true for the overall expression to be true.
||	OR	Connects two expressions into one. One or both expressions must be true for the overall expression to be true. It is only necessary for one to be true, and it does not matter which.
!	NOT	Reverses the "truth" of an expression. It makes a true expression false, and a false expression true.

- **The && Operator:**

The **&&** operator is known as the logical **AND** operator. It takes two expressions as operands and creates an expression that is true only when both sub-expressions are true.

Table 4-4 shows a **truth table** for the **&&** operator.

Table 4-4:



Expression	Value of the Expression
`false && false`	`false (0)`
`false && true`	`false (0)`
`true && false`	`false (0)`
`true && true`	`true (1)`

Here are some examples:

5 == 5 && 4 == 4 // true because both expressions are true

5 == 3 && 4 == 4 // false because first expression is false

5 > 3 && 5 > 10 // false because second expression is false

5 > 8 && 5 < 10 // false because first expression is false

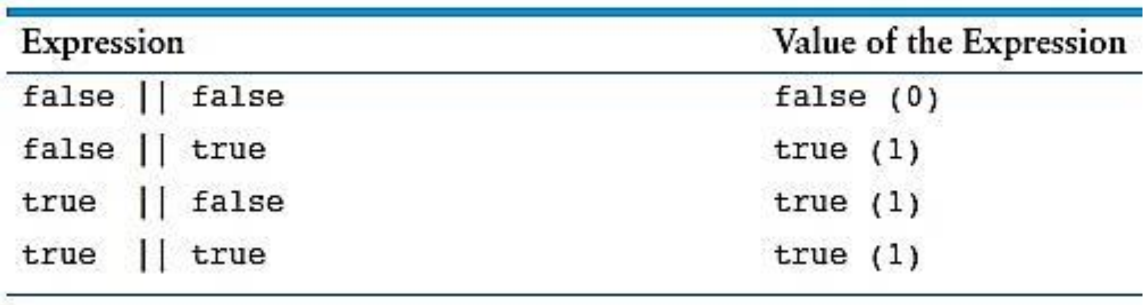
5 < 8 && 5 > 2 // true because both expressions are true

5 > 8 && 5 < 2 // false because both expressions are false

Because the **&&** has a lower precedence than the **relational operators**, you don't need to use parentheses in these expressions.

- **The || Operator:**

The || operator is known as the logical **OR** operator. It takes two expressions as operands and creates an expression that is true when either of the sub-expressions or both are true.
In English, the word ***or*** can indicate when one or both of two conditions satisfy a requirement.
Table 4-5 shows a truth table for the || operator.
Table 4-5:



Expression	Value of the Expression
`false || false`	`false (0)`
`false || true`	`true (1)`
`true || false`	`true (1)`
`true || true`	`true (1)`

All it takes for an OR expression to be true is for one of the sub-expressions to be true. It doesn't matter if the other sub-expression is false or true.

Here are some examples:

5 ==5 || 5 == 9 // true because first expression is true

5 > 3 || 5 > 10 // true because first expression is true

5 > 8 || 5 < 10 // true because second expression is true

5 < 8 || 5 > 2 // true because both expressions are true

5 > 8 || 5 < 2 // false because both expressions are false

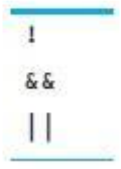
Because the || has a lower precedence than the relational operators, you don't need to use parentheses in these expressions.

- **The ! Operator:**

The **!** operator performs a logical **NOT** operation. It takes an operand and reverses its truth or falsehood. In other words, if the expression is true, the ! operator returns false, and if the expression is false, it returns true.

Table 4-6 shows the precedence of C++'s logical operators, from highest to lowest.

Table 4-6:



!
&&
||

As mentioned earlier, the C++ **logical OR** and **logical AND** operators **have a lower precedence than relational operators**. This means that an expression such as this

x > 5 && x < 10

is read this way:

(x > 5) && (x < 10)

The **!** operator, on the other hand, has a higher precedence than any of the relational or arithmetic operators. Therefore, to negate an expression, you should enclose the expression in parentheses, like this:

!(x > 5) // is it false that x is greater than 5

!x > 5 // is !x greater than 5

Incidentally, the second expression here is always false because **!x** can have only the values true or false, which get converted to 1 or 0.

The **logical AND** operator has a higher precedence than the **logical OR** operator. Thus this expression:

```
age > 30 && age < 45 || weight > 300
```

means the following:

```
(age > 30 && age < 45) || weight > 300
```

That is, one condition is that **age** be in the range 31–44, and the second condition is that **weight** be greater than 300. The entire expression is true if one the other or both of these conditions are true.
You can, of course, use parentheses to tell the program the interpretation you want. For example, suppose you want to use **&&** to combine the condition that **age** be greater than 50 or **weight** be greater than 300 with the condition that donation be greater than 1,000. You have to enclose the OR part within parentheses:

```
(age > 50 || weight > 300) && donation > 1000
```

Otherwise, the compiler combines the **weight** condition with the donation condition instead of with the **age** condition.
Although the C++ operator precedence rules often make it possible to write compound comparisons without using parentheses, the simplest course of action is to use parentheses to group the tests, whether or not the parentheses are needed. It makes the code easier to read, it doesn't force someone else to look up some of the less commonly used precedence rules, and it reduces the chance of making errors because you don't quite remember the exact rule that applies.
C++ guarantees that when a program evaluates a logical expression, it evaluates it from left to right and stops evaluation as soon as it knows what the answer is. Suppose, for example, that you have this condition:

```
x != 0 && 1.0 / x > 100.0
```

If the first condition is false, then the whole expression must be false. That's because for this expression to be true, each individual condition must be true. Knowing the first condition is false, the program doesn't bother evaluating the second condition. That's fortunate in this example because evaluating the second condition would result in dividing by zero, which is not in a computer's repertoire of possible actions.

البرمجة بلغة ++C

7.1. Comments:

Comments are notes of explanation that document lines or sections of a program.

It may surprise you that one of the most important parts of a program has absolutely no impact on the way it runs. We are speaking, of course, of the comments. Comments are part of the program, but the compiler ignores them. They are intended for people who may be reading the source code.

1. Single Line Comments

You have already seen one way to place comments in a C++ program. You simply place two forward slashes (//) where you want the comment to begin. The compiler ignores everything from that point to the end of the line.

2. Multi-Line Comments

The second type of comment in C++ is the **multi-line comment**. **Multi-line comments** start with **/*** (a forward slash followed by an asterisk) and end with ***/** (an asterisk followed by a forward slash). Everything between these markers is ignored. The **Program 4-1** illustrates the use of both a **multi-line comment** and **single line comments**. The multi-line comment starts on line 1 with the /* symbol, and ends on line 6 with the */ symbol.

Program 4-1

```
1 /*
2 PROGRAM: PAYROLL.CPP
3 Written by Herbert Dorfmann
4 This program calculates company payroll
5 Last modified: 8/20/2006
6 */
7 #include <iostream>
8 using namespace std;
9
10 int main()
11 {
12 int employeeID; // Employee ID number
13 double payRate; // Employees hourly pay rate
14 double hours; // Hours employee worked this week
```

(The remainder of this program is left out.)

Notice that unlike a comment started with //, a multi-line comment can span several lines. This makes it more convenient to write large blocks of comments because you do not have to mark every line. On the other hand, the multi-line comment is inconvenient for writing single line comments because you must type both a beginning and ending comment symbol.

When using multi-line comments:

• Be careful not to reverse the beginning symbol with the ending symbol.

• Be sure not to forget the ending symbol.

Both of these mistakes can be difficult to track down, and will prevent the program from compiling correctly.

7.2. Variable Assignments and Initialization:

An assignment operation assigns, or copies, a value into a variable. When a value is assigned to a variable as part of the variable's definition, it is called an initialization.

As you have already seen in several examples, a value is stored in a variable with an **assignment statement**. For example, the following statement copies the value 12 into the variable unitsSold.

```
unitsSold = 12;
```

The = symbol, as you recall, is called the **assignment operator**. **Operators** perform operations on data. The data that **operators** work with are called **operands**. The **assignment operator** has two **operands**. In the previous statement, the operands are **unitsSold** and **12**. It is important to remember that in an **assignment statement**, C++ requires the name of the variable receiving the assignment to appear on the left side of the operator. The following statement is incorrect.

```
12 = unitsSold; // Incorrect!
```

You have also seen that it is possible to assign values to variables when they are defined. This is called initialization. When multiple variables are defined in the same statement, it is possible to initialize some of them without having to initialize all of them. The **program 4-2** illustrates this.

Program 4-2

```
// This program shows variable initialization.
#include <iostream>

using namespace std;

int main()
{

int year, // year is not initialized
days = 360; // days is initialized to 360

year = 2007; // Now year is assigned a value

cout << "In " << year << " "
<< " had " << days << " days";

return 0;
 }
```

7.3. Multiple and Combined Assignment:

1- Multiple Assignment:

Multiple assignment means to assign the same value to several variables with one statement.

C++ allows you to assign a value to **multiple variables** at once. If a program has several variables, such as **a, b, c,** and **d,** and each variable needs to be assigned a value, such as 12, the following statement may be constructed:

a = b = c = d = 12;

The value 12 will be assigned to each variable listed in the statement. This works because the assignment operations are carried out from right to left. First 12 is assigned to d. Then d's value, now a 12, is assigned to c. Then c's value is assigned to b, and finally b's value is assigned to a.

2- **Combined Assignment operators:**

Quite often programs have assignment statements of the following form:

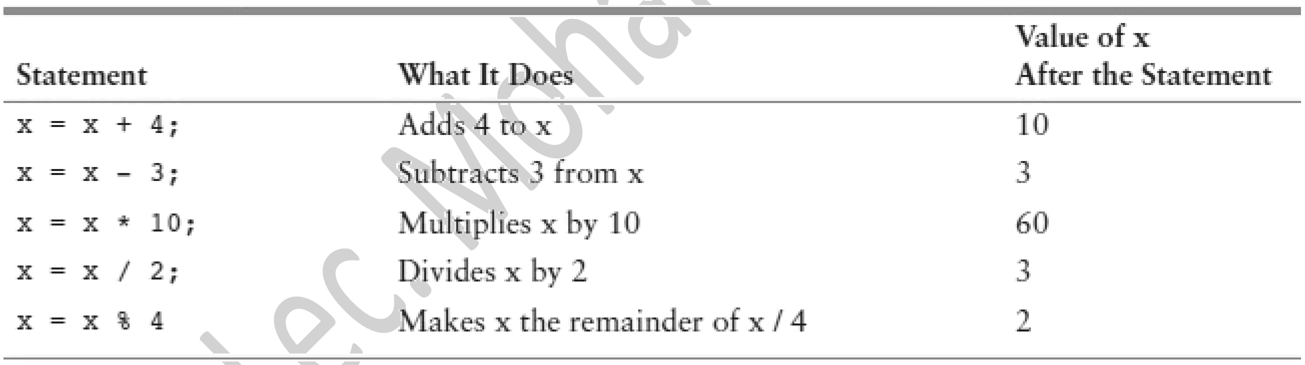
```
num = num + 1;
```

The expression on the right side of the assignment operator gives the value of **num** plus 1. The result is then assigned to **num**, replacing the value that was previously stored there. Effectively, this statement adds 1 to **num**. In a similar fashion, the following statement subtracts 5 from **num**.

```
num = num – 5;
```

If you have never seen this type of statement before, it might cause some initial confusion because the same variable name appears on both sides of the assignment operator.
Table 4-7 shows other examples of statements written this way.

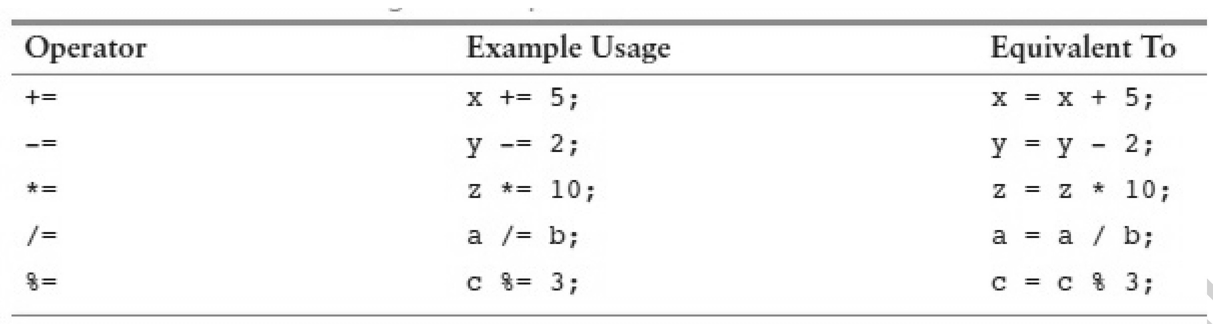
Table 4-7: Assignment Statements that Change a Variable's Value (Assume x = 6)



Statement	What It Does	Value of x After the Statement
`x = x + 4;`	Adds 4 to x	10
`x = x - 3;`	Subtracts 3 from x	3
`x = x * 10;`	Multiplies x by 10	60
`x = x / 2;`	Divides x by 2	3
`x = x % 4`	Makes x the remainder of x / 4	2

Because these types of operations are so common in programming, C++ offers a special set of operators designed specifically for these jobs. Table 4-8 shows the **combined assignment operators**, also known as **compound operators** or **arithmetic assignment operators**.

Table 4-8: Combined Assignment Operators



Operator	Example Usage	Equivalent To
`+=`	`x += 5;`	`x = x + 5;`
`-=`	`y -= 2;`	`y = y - 2;`
`*=`	`z *= 10;`	`z = z * 10;`
`/=`	`a /= b;`	`a = a / b;`
`%=`	`c %= 3;`	`c = c % 3;`

As you can see, the **combined assignment operators** do not require the programmer to type the variable name twice. Also, they give a clear indication of what is happening in the statement.

More elaborate statements may be expressed with the combined assignment operators.
Here is an example:

```
result *= a + 5;
```

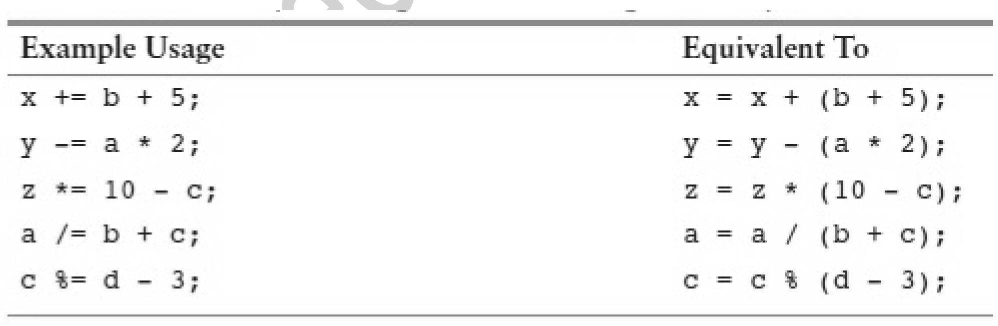
In this statement, **result** is multiplied by the **sum** of a + 5. Notice that the precedence of the combined assignment operators is lower than that of the regular arithmetic operators.
The above statement is equivalent to

```
result = result * (a + 5);
```

which is different from

```
result = result * a + 5;
```

Table 4-9 shows additional examples using combined assignment operators.
Table 4-9**:** Examples Using Combined Assignment Operators and Arithmetic Operators



Example Usage	Equivalent To
`x += b + 5;`	`x = x + (b + 5);`
`y -= a * 2;`	`y = y - (a * 2);`
`z *= 10 - c;`	`z = z * (10 - c);`
`a /= b + c;`	`a = a / (b + c);`
`c %= d - 3;`	`c = c % (d - 3);`

البرمجة بلغة C++

8.1. The Increment and Decrement Operators:

C++ provides a pair of unary operators for incrementing and decrementing variables.

To **increment** a value means to increase it, and to **decrement** a value means to decrease it. In the example below, **qtyOrdered** is incremented by 10 and **numSold** is decremented by 3.

```
qtyOrdered = qtyOrdered + 10;
numSold = numSold -3;
```

Although the values stored in variables can be increased or decreased by any amount, it is particularly common to increment them or decrement them by 1. C++ provides a pair of unary operators to do this. The **++ operator** increases its operand's value by 1. The **-- operator** decreases its operand's value by 1. Here are three different ways to increment the value of the variable **num** by 1.

```
num = num + 1;   //normal
num += 1;   //combine assignment
num++;  //increment
```

And here are three different ways to decrement it by 1:

```
num = num - 1;
num -= 1;
num--;
```

Our examples so far show the increment and decrement operators used in **postfix mode**, which means the operator is placed after the variable. The operators also work in **prefix mode**, where the operator is placed before the variable name:

```
++num;
--num;
```

In both **prefix** and **postfix mode**, these operators add 1 to, or subtract 1 from, their operand. The following example illustrates the use of these operators in both **prefix** and **postfix mode**.

Notice that there is no space between the name of the variable and the **++** or **--** preceding it or following it.

```
num = 4;
num++; // now num has the value 5
```

```
++num; // now num has the value 6
num--; // now num has the value 5 again
--num; // now num has the value 4 again
```

Next program includes these 5 lines of code along with **cout** statements to further illustrate how they work.

Program 5-1

```
// This program demonstrates the ++ and -- operators.
#include <iostream>
using namespace std;

int main()
{
int num = 4; // num starts out with 4

// Display the value in num
cout << "The variable num is " << num << endl;
cout << "I will now increment num.\n\n";

// Use postfix ++ to increment num
num++;
cout << "Now the variable num is " << num << endl;
cout << "I will increment num again.\n\n";

// Use prefix ++ to increment num
++num;
cout << "Now the variable num is " << num << endl;
cout << "I will now decrement num.\n\n";

// Use postfix -- to decrement num
num--;
cout << "Now the variable num is " << num << endl;
cout << "I will decrement num again.\n\n";

// Use prefix -- to increment num
--num;
cout << "Now the variable num is " << num << endl;
return 0;
}
```

Program Output

```
The variable num is 4
I will now increment num.
Now the variable num is 5
I will increment num again.
Now the variable num is 6
I will now decrement num.
Now the variable num is 5
I will decrement num again.
Now the variable num is 4
```

8.2. The Difference Between Postfix and Prefix Modes:

In the simple statements used in Program 5-1, it doesn’t matter if the increment or decrement operator is used in postfix or prefix mode. The difference is important, however, when these operators are used in statements that do more than just increment or decrement.

For example, look at the following lines:

```
num = 4;
cout << num++;
```

This **cout** statement is doing two things:

1) displaying the value of **num**, and

2) incrementing **num**.

But which happens first? **cout** will display a different value if **num** is incremented first than if **num** is incremented last. The answer depends on the mode of the increment operator. **Postfix** mode causes the increment to happen after the value of the variable is used in the expression. In the example, **cout** will display 4, then **num** will be incremented to 5. **Prefix** mode, however, causes the increment to happen first. In the following statements, **num** will first be incremented to 5, then **cout** will display 5:

```
num = 4;
cout << ++num;
```

Program 5-2 illustrates these dynamics further by placing increment and decrement operators in **cout** statements. This makes it easy to see the difference between using them in **prefix** and **postfix** mode. However, this should not normally be done. That is, in actual programming applications it is not recommended to place increment or decrement operators in **cout** statements. So you should not write code like this.

Program 5-2

```
// This program demonstrates the postfix and prefix
// modes of the increment and decrement operators.
#include <iostream>
using namespace std;

int main()
{
int num = 4;

// Illustrate postfix and prefix ++ operator
cout << num << " "; // Displays 4
cout << num++ << " "; // Displays 4, then adds 1 to num
cout << num << " "; // Displays 5
cout << ++num << "\n\n"; // Adds 1 to num, then displays 6

// Illustrate postfix and prefix -- operator
cout << num << " "; // Displays 6
cout << num-- << " "; // Displays 6, then subtracts 1 from num
cout << num << " "; // Displays 5
cout << --num << "\n\n"; // Subtracts 1 from num, then displays 4

return 0;
}
```

Program Output

```
4 4 5 6
6 6 5 4
```

For another example, look at the following code:

```
int x = 1;
int y;
y = x++; // Postfix increment
```

The first statement defines the variable x (initialized with the value 1) and the second statement defines the variable y. The third statement does two things:

• It assigns the value of x to the variable y.

• The variable x is incremented.

The value that will be stored in y depends on when the increment takes place. Because the ++ operator is used in postfix mode, the old value of x (which is 1) is assigned to y before x is incremented. After the statement executes, y will contain 1, and x will contain 2. Let's look at the same code, but with the ++ operator used in prefix mode:

```
int x = 1;
int y;
y = ++x; // Prefix increment
```

In the third statement, the ++ operator is used in prefix mode, causing variable x to be incremented before the assignment takes place. So, this code will store 2 in y. After the code has executed, x and y will both contain 2.

البرمجة بلغة ++C

9.1. Using ++ and -- in Mathematical Expressions:

The increment and decrement operators can also be used on variables in mathematical expressions. Consider the following program segment:

```
a = 2;
b = 5;
c = a * b++;
cout << a << " " << b << " " << c;
```

In the statement c = a * b++, c is assigned the value of a times b, which is 10. The variable b is then incremented. The **cout** statement will display

```
2 6 10
```

If the statement were changed to read

```
c = a * ++b;
```

the variable b would be incremented before it was multiplied by a. In this case c would be assigned the value of 2 times 6, so the **cout** statement would display

```
2 6 12
```

You can pack a lot of action into a single statement using the increment and decrement operators, but don't get too tricky with them. You might be tempted to try something like the following, thinking that c will be assigned 11:

```
a = 2;
b = 5;
c = ++(a * b); // Error!
```

9.2. Formatted Output and Input functions:

The three primary activities of a program are input, processing, and output.
Computer programs typically perform a three-step process of gathering **input**, **performing some process** on the information gathered, and then **producing output**.

Input is information a program collects from the outside world. It can be sent to the program by the user, who is entering data at the keyboard or using the mouse. It can also be read from disk files or hardware devices connected to the computer.

Once information is gathered from the outside world, a program usually **processes** it in some manner.

Output is information that a program sends to the outside world. It can be words or graphics displayed on a screen, a report sent to the printer, data stored in a file, or information sent to any device connected to the computer.

9.2.1. The cin function:

cin can be used to read data typed at the keyboard.

For example, a program that calculates the area of a circle might ask the user to enter the circle's radius. When the circle area has been computed and printed, the program could be run again and a different radius could be entered. Program 5-3 shows **cin** being used to read values input by the user.

Program 5-3

```
// This program calculates and displays the area of a rectangle.
#include <iostream>
using namespace std;

int main()
{
int length, width, area;

cout << "This program calculates the area of a rectangle.\n";

// Have the user input the rectangle's length and width
cout << "What is the length of the rectangle? ";
cin >> length;
cout << "What is the width of the rectangle? ";
cin >> width;
16
17 // Compute and display the area
18 area = length * width;
19 cout << "The area of the rectangle is " << area << endl;
20 return 0;
21 }
```

Program Output with Example Input Shown in Bold

This program calculates the area of a rectangle.
What is the length of the rectangle? **10[Enter]**
What is the width of the rectangle? **20[Enter]**
The area of the rectangle is 200.

Instead of calculating the area of one rectangle, this program can be used to compute the area of any rectangle. The values that are stored in the length and width variables are entered by the user when the program is running. Look at lines 12 and 13.

```
cout << "What is the length of the rectangle? ";
cin >> length;
```

In line 12 **cout** is used to display the question "What is the length of the rectangle?" This is called a **prompt**. It lets the user know that an input is expected and prompts them as to what must be entered. When **cin** will be used to get input from the user, it should always be preceded by a prompt.
Gathering input from the user is normally a two-step process:

1. Use **cout** to display a prompt on the screen.
2. Use **cin** to read a value from the keyboard.

The prompt should ask the user a question, or tell the user to enter a specific value.

Notice that the << and >> operators appear to point in the direction that data is flowing. It may help to think of them as arrows. In a statement that uses **cout**, the << operator always points toward **cout**, as shown here. This indicates that data is flowing from a variable or a literal to the **cout** object.

```
cout << "What is the length of the rectangle? ";
cout ← "What is the length of the rectangle? ";
```

In a statement that uses **cin**, the >> operator always points toward the variable receiving the value. This indicates that data is flowing from the **cin** object to a variable.

```
cin >> length;
cin → length;
```

The **cin** object causes a program to wait until data is typed at the keyboard and the **[Enter]** key is pressed. No other lines will be executed until **cin** gets its input.

- **Entering Multiple Values**: You can use **cin** to input multiple values at once. Look at Program 5-4, which is a modified version of Program 5-3.

Program 5-4

```
1 // This program calculates and displays the area of a rectangle.
2 #include <iostream>
3 using namespace std;
4
5 int main()
6 {
7 int length, width, area;
8
9 cout << "This program calculates the area of a rectangle.\n";
10
11 // Have the user input the rectangle's length and width
12 cout << "Enter the length and width of the rectangle ";
13 cout << "separated by a space.\n";
14 cin >> length >> width;
15
16 // Compute and display the area
17 area = length * width;
18 cout << "The area of the rectangle is " << area << endl;
19 return 0;
20 }
```

Program Output with Example Input Shown in Bold

```
This program calculates the area of a rectangle.
Enter the length and width of the rectangle separated by a space.
10 20[Enter]
The area of the rectangle is 200
```

Line 14 waits for the user to enter two values. The first is assigned to length and the second to width.

```
cin >> length >> width;
```

In the example output, the user entered 10 and 20, so 10 is stored in length and 20 is stored in width. **Notice** the user separates the numbers by spaces as they are entered. This is how **cin** knows where each number begins and ends. It doesn't matter how many spaces are entered between the individual numbers. For example, the user could have entered

```
10   20
```

NOTE: The **[Enter]** key is pressed after the last number is entered. **cin** also can read multiple values of different data types.

9.2.2. The cout function:

cout is used to display information on the computer's screen.

To print a message on the screen, you send a stream of characters to **cout**. Look to the next line:

```
cout << "Programming is great fun!";
```

The item immediately to the right of the operator is sent to **cout** and then displayed on the screen.

Program 5-5 shows another way to write the same string “Programming is great fun!".

Program 5-5

```
// A simple C++ program
#include <iostream>
using namespace std;

int main()
{
cout << "Programming is " << "great fun!";
return 0;
}
```

Program Output

```
Programming is great fun!
```

The next program shows yet another way to accomplish the same thing.

Program 5-6

```
// A simple C++ program
#include <iostream>
using namespace std;

int main()
{
cout << "Programming is ";
cout << "great fun!";
return 0;
}
```

Program Output

```
Programming is great fun!
```

An important concept to understand about Program 5-6 is that although the output is broken into two programming statements, this program will still display the message on a single line. Unless you specify otherwise, the information you send to **cout** is displayed in a continuous stream. Sometimes this can produce less-than-desirable results. Program 5-7 illustrates this.

Program 5-7

```
// An unruly printing program
#include <iostream>
using namespace std;

int main()
{
cout << "The following items were top sellers";
cout << "during the month of June:";
cout << "Computer games";
cout << "Coffee";
cout << "Aspirin";
return 0;
}
```

Program Output

```
The following items were top sellersduring the month of June:Computer
gamesCoffeeAspirin
```

The layout of the actual output looks nothing like the arrangement of the strings in the source code.

First, notice there is no space displayed between the words “sellers” and “during,” or between “June:” and “Computer.” **cout** displays messages exactly as they are sent. If spaces are to be displayed, they must appear in the strings.

Second, even though the output is broken into five lines in the source code, it comes out as one long line of output. Because the output is too long to fit on one line of the screen, it wraps around to a second line when displayed. The reason the output comes out as one long line is that **cout** does not start a new line unless told to do so.

There are two ways to instruct **cout** to start a new line.

The first is to send **cout** a **stream manipulator** called **endl** (pronounced “end-line” or “end-L”).

Another way to cause subsequent output to begin on a new line is to insert a **\n** in the string that is being output. Program 5-8 does this.

Program 5-8

```
// Another well-adjusted printing program
#include <iostream>
using namespace std;

int main()
{
cout << "The following items were top sellers\n";
cout << "during the month of June:\n";
cout << "Computer games\nCoffee";
cout << "\nAspirin\n";
return 0;}
```

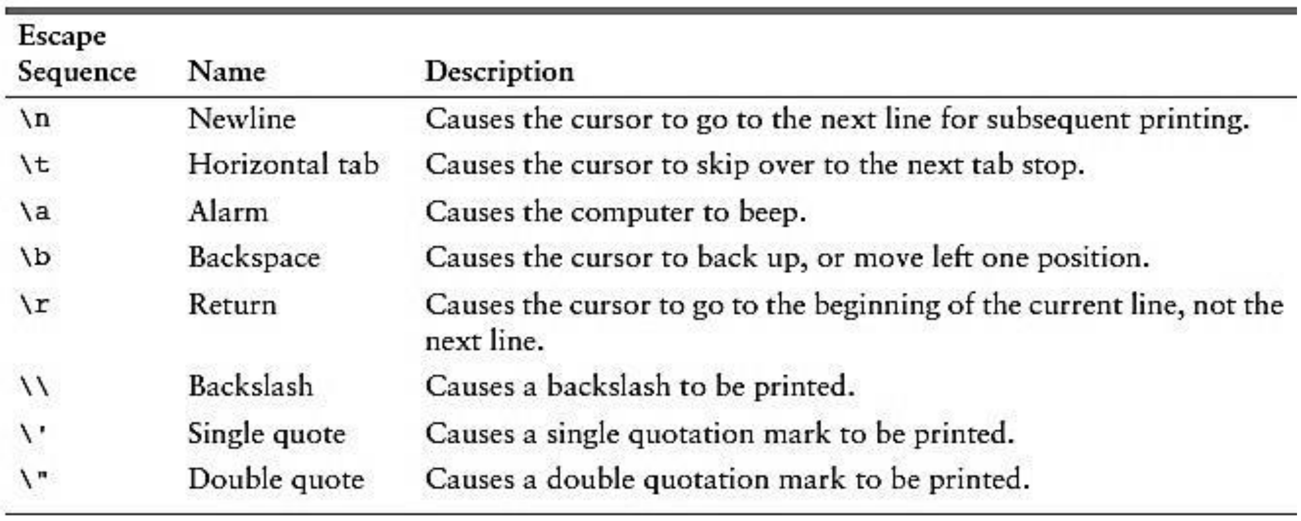
Program Output

```
The following items were top sellers
during the month of June:
Computer games
Coffee
Aspirin
```

\n is an example of an **escape sequence**. **Escape sequences** are written as a backslash character (\) followed by one or more control characters and are used to control the way output is displayed. There are many escape sequences in C++. The newline escape sequence (**\n**) is just one of them.

Escape sequences give you the ability to exercise greater control over the way information is output by your program. Table 5-1 lists a few of them.

Table 5-1: Common Escape Sequences



Escape Sequence	Name	Description
`\n`	Newline	Causes the cursor to go to the next line for subsequent printing.
`\t`	Horizontal tab	Causes the cursor to skip over to the next tab stop.
`\a`	Alarm	Causes the computer to beep.
`\b`	Backspace	Causes the cursor to back up, or move left one position.
`\r`	Return	Causes the cursor to go to the beginning of the current line, not the next line.
`\\`	Backslash	Causes a backslash to be printed.
`\'`	Single quote	Causes a single quotation mark to be printed.
`\"`	Double quote	Causes a double quotation mark to be printed.

البرمجة بلغة C++

10.Condition Statements

10.1.The if Statement

The **if statement** can cause other statements to execute only under certain conditions. Programs often need more than one path of execution, however. Many algorithms require a program to execute some statements only under certain circumstances. This can be accomplished with a **decision structure**.

In a **decision structure' s** simplest form a specific action, or set of actions, is taken only when a specific condition exists. If the condition does not exist, the actions are not performed. The flowchart in Figure 6-1 shows the logic of a decision structure with syntax. The diamond symbol represents a **yes**/**no** question or a **true**/**false** condition. If the answer to the question is **yes** (or if the condition is **true**), the program flow follows one path which leads to the actions being performed. If the answer to the question is no (or the condition is false), the program flow follows another path which skips the actions.

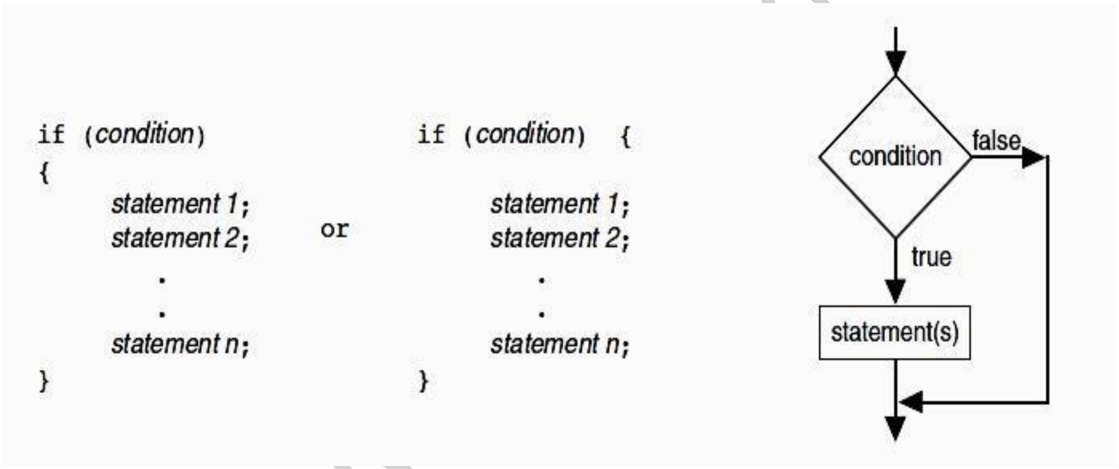

```
if (condition)                   if (condition)  {
{
      statement 1;                     statement 1;
      statement 2;          or         statement 2;
          .                                .
          .                                .
      statement n;                     statement n;
}                                }
```

Figure 6-1

Program 6-1 illustrates the use of an **if statement**. The user enters three test scores and the program calculates their average. If the average equals 100, the program congratulates the user on earning a perfect score.

Program 6-1

```
1 // This program correctly averages 3 test scores.
2 #include <iostream>
3 #include <iomanip>
4 using namespace std;
5
6 int main()
7 {
8 int score1, score2, score3;
9 double average;
10
12 // Get the three test scores
13 cout << "Enter 3 test scores and I will average them: ";
13 cin >> score1 >> score2 >> score3;
14
15 // Calculate and display the average score
16 average = (score1 + score2 + score3) / 3.0;
17 cout << fixed << showpoint << setprecision(1);
18 cout << "Your average is " << average << endl;
19
20 // If the average equals 100, congratulate the user
21 if (average == 100)
22 { cout << "Congratulations! ";
23 cout << "That's a perfect score!\n";
24 }
25 return 0;
26 }
```

Program Output with Example Input Shown in Bold

```
Enter 3 test scores and I will average them: 80 90 70[Enter]
Your average is 80.0
```

Program Output with Other Example Input Shown in Bold

```
Enter 3 test scores and I will average them: 100 100 100[Enter]
Your average is 100.0
Congratulations! That's a perfect score!
```

Let' s look more closely at lines 21-24 of Program 6-1, which cause the **congratulatory** message to be printed.

```
if (average == 100)
{ cout << "Congratulations! ";
  cout << "That's a perfect score!\n";
}
```

There are four important things to notice.
First, the word if, which begins the statement, is a C++ key word and must be written in lowercase.
Second, the condition to be tested (average == 100) must be enclosed inside parentheses.
Third, there is no semi-colon after the test condition, even though there is a semi-colon after each action associated with the if construct. We will explain why shortly.
And finally, the block of statements to be conditionally executed is surrounded by curly braces. This is required whenever **two or more actions are associated with an if statement**.

If the block of statements to be conditionally executed contains only one statement, the braces can be omitted.

Table 6.1 Example if Statements

Statements	Outcome

`if (hours > 40)`
`{   overTime = true;`
`    payRate *= 2;`
`}`
	Assigns `true` to Boolean variable `overTime` and doubles `payRate` only when `hours` is greater than 40. Because there is more than one statement in the conditionally executed block, braces `{}` are required.

`if (temperature > 32)`
`    freezing = false;`
	Assigns `false` to Boolean variable `freezing` only when `temperature` is greater than 32. Because there is only one statement in the conditionally executed block, braces `{}` are optional.

10.1.1. Programming Style and the if Statement

Even though **if statements** usually span more than one line, they are technically one long statement. For instance, the following if statements are identical except in style:

```
if (a >= 100)
   cout << "The number is out of range.\n";

if (a >= 100)  cout << "The number is out of range.\n";
```

Here are two important style rules you should adopt for writing if statements:
• The conditionally executed statement(s) should begin on the line after the if statement.
• The conditionally executed statement(s) should be indented one “level” from the if statement.

10.1.2. Three Common Errors to Watch Out For

When writing if statements, there are three common errors you must watch out for.
1. Misplaced semicolons
2. Missing braces
3. Confusing = with ==

10.2. The if/else Statement

The if/else statement will execute one set of statements when the if condition is true, and another set when the condition is false.
The **if/else statement** is an expansion of the if statement. Figure 6-2 shows the general format of this statement and a flowchart visually depicting how it works.

```
if (condition)
{
      statement set 1;
}
else
{
      statement set 2;
}
```

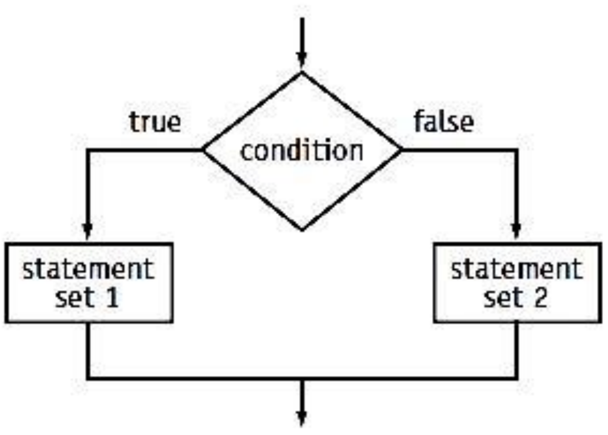


Figure 6-2

As with the if statement, a condition is tested. If the condition is true, a block containing one or more statements is executed. If the condition is false, however, a different group of statements is executed. Program **6-2** uses the **if/else statement** along with the modulus operator to determine if a number is **odd** or **even**.

Program 6-2

```
// This program uses the modulus operator to determine
// if a number is odd or even. If the number is evenly divisible
// by 2, it is an even number. A remainder indicates it is odd.
#include <iostream>
using namespace std;

int main()
{
int number;

cout << "Enter an integer and I will tell you if it\n";
cout << "is odd or even. ";
cin >> number;

if (number % 2 == 0)
cout << number << " is even.\n";
else
cout << number << " is odd.\n";
return 0;}
```

Program Output with Example Input Shown in Bold

```
Enter an integer and I will tell you if it
is odd or even. 17[Enter]
17 is odd.
```

As with the if part, if you don’t use braces the else part controls a single statement. If you wish to execute more than one statement with the else part, place these statements inside a set of braces. Program 6-3 illustrates this. It also illustrates a way to handle a classic programming problem: **division by zero.**

Division by zero is mathematically impossible to perform and it normally causes a program to crash. This means the program will prematurely stop running, sometimes with an error message.

Program 6-3 shows a way to test the value of a divisor before the division takes place. On line 15 the value of **num2** is tested. If the user enters anything other than zero, the lines controlled by the if are executed, allowing the division to be performed and the result to be displayed. But if the user enters a zero for **num2**, the lines controlled by the else are executed instead, causing an error message to be displayed.

Program 6-3

```
1 // This program makes sure that the divisor is not
2 // equal to 0 before it performs a divide operation.
3 #include <iostream>
4 using namespace std;
5
6 int main()
7 {
8 double num1, num2, quotient;
9
10 // Get the two numbers
11 cout << "Enter two numbers: ";
12 cin >> num1 >> num2;
13
```

Program Output with Example Input Shown in Bold

```
Enter two numbers: 10 0[Enter]
Division by zero is not possible.
Please run the program again and enter a number other than zero.
```

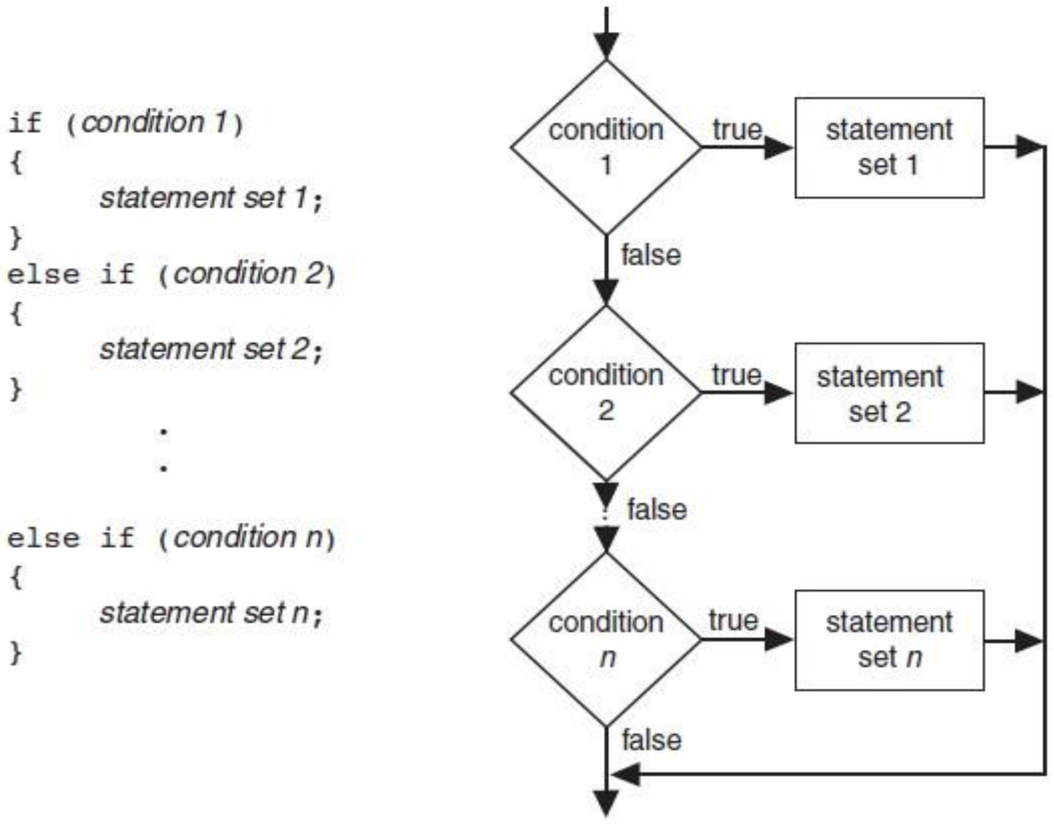
```
// If num2 is not zero, perform the division.
if (num2 != 0)
{
quotient = num1 / num2;
cout << "The quotient of " << num1 << " divided by "
<< num2 << " is " << quotient << ".\n";
}
else
{
cout << "Division by zero is not possible.\n"
<< "Please run the program again and enter "
<< "a number other than zero.\n";
}
return 0;
}
```

البرمجة بلغة C++

11.1. The if/else if Statement

The **if/else if statement** is a chain of if statements. They perform their tests, one after the other, until one of them is found to be true.
We make certain mental decisions by using sets of different but related rules.

This type of decision making is also very common in programming. In C++ it can be accomplished through the if/else if statement. Figure 6-3 shows its format and a flowchart visually depicting how it works.



```
if (condition 1)
{
      statement set 1;
}
else if (condition 2)
{
      statement set 2;
}
            .
            .
else if (condition n)
{
      statement set n;
}
```

Figure 6-3

This construction is like a chain of if/else statements. The else part of one statement is linked to the if part of another. When put together this way, the chain of **if/elses** becomes one long statement. Program 6-4 shows an example. The user is asked to enter a numeric test score and the program displays the letter grade earned.

Program 6-4

```
// This program uses an if/else if statement to assign a
// letter grade (A, B, C, D, or F) to a numeric test score.
#include <iostream>
using namespace std;

```

```
int main()
{
int testScore; // Holds a numeric test score
char grade; // Holds a letter grade

// Get the numeric score
cout << "Enter your numeric test score and I will\n";
cout << "tell you the letter grade you earned: ";
cin >> testScore;

// Determine the letter grade
if (testScore < 60)
grade = 'F';
else if (testScore < 70)
grade = 'D';
else if (testScore < 80)
grade = 'C';
else if (testScore < 90)
grade = 'B';
else if (testScore <= 100)
grade = 'A';

// Display the letter grade
cout << "Your grade is " << grade << ".\n";
return 0;
}
```

Program Output with Example Input Shown in Bold

```
Enter your numeric test score and I will
tell you the letter grade you earned: 88[Enter]
Your grade is B.
```

Figure 6-4 shows the paths that may be taken by the **if/else if statement** in previous program.

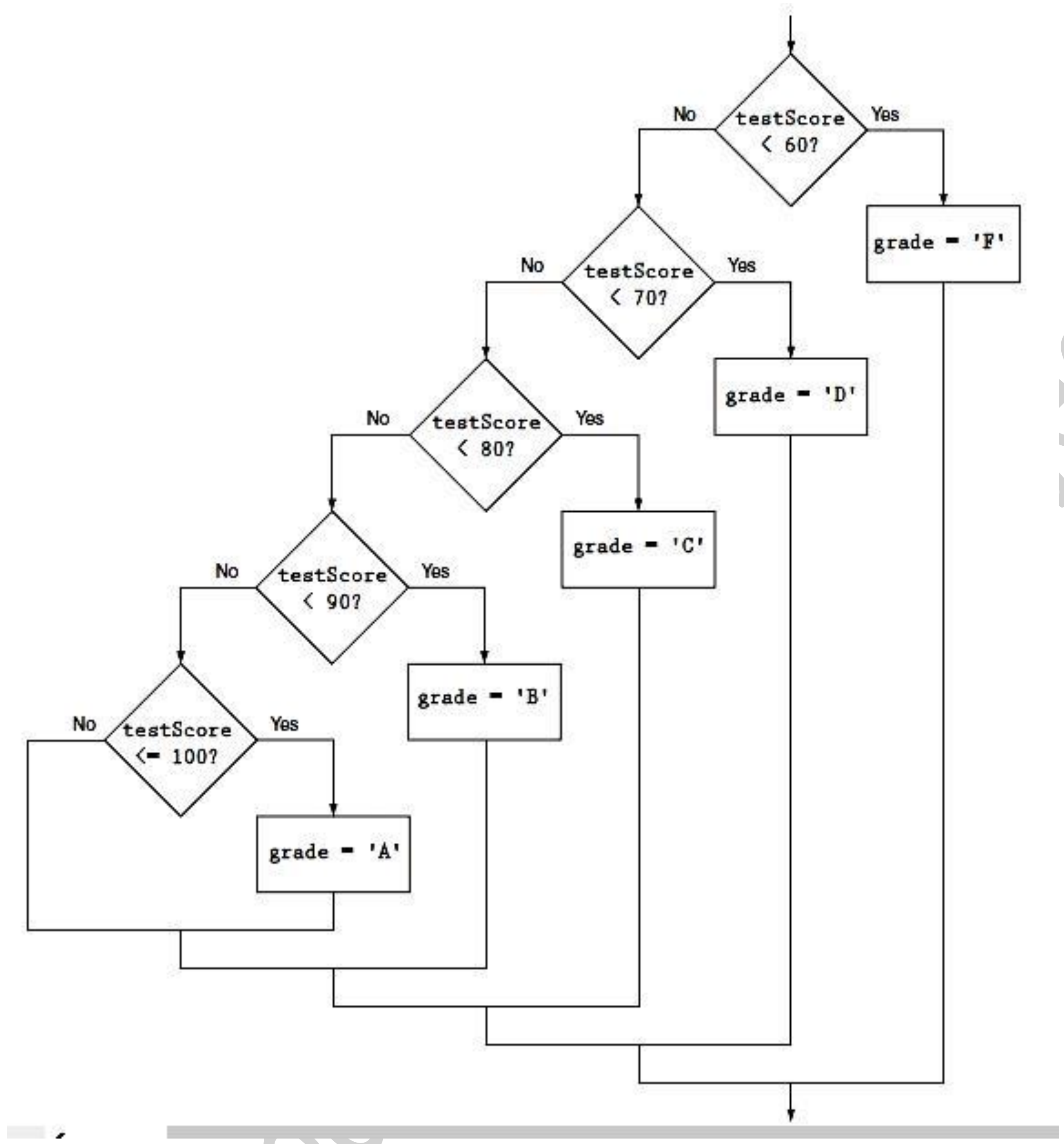


Figure 6-4

11.2. Nested if Statements

To test more than one condition, an if statement can be nested inside another if statement.
It is possible for one **if statement** or **if/else statement** to be placed inside another one.
This construct, called a ***nested if***, allows you to test more than one condition to determine which block of code should be executed. For example, consider a banking program that determines whether a bank customer qualifies for a special, low interest rate on a loan. To qualify, two conditions must exist:
1. The customer must be currently employed.
2. The customer must have recently graduated from college (in the past two years).
Figure 6-5 shows a flowchart for an algorithm that could be used in such a program.

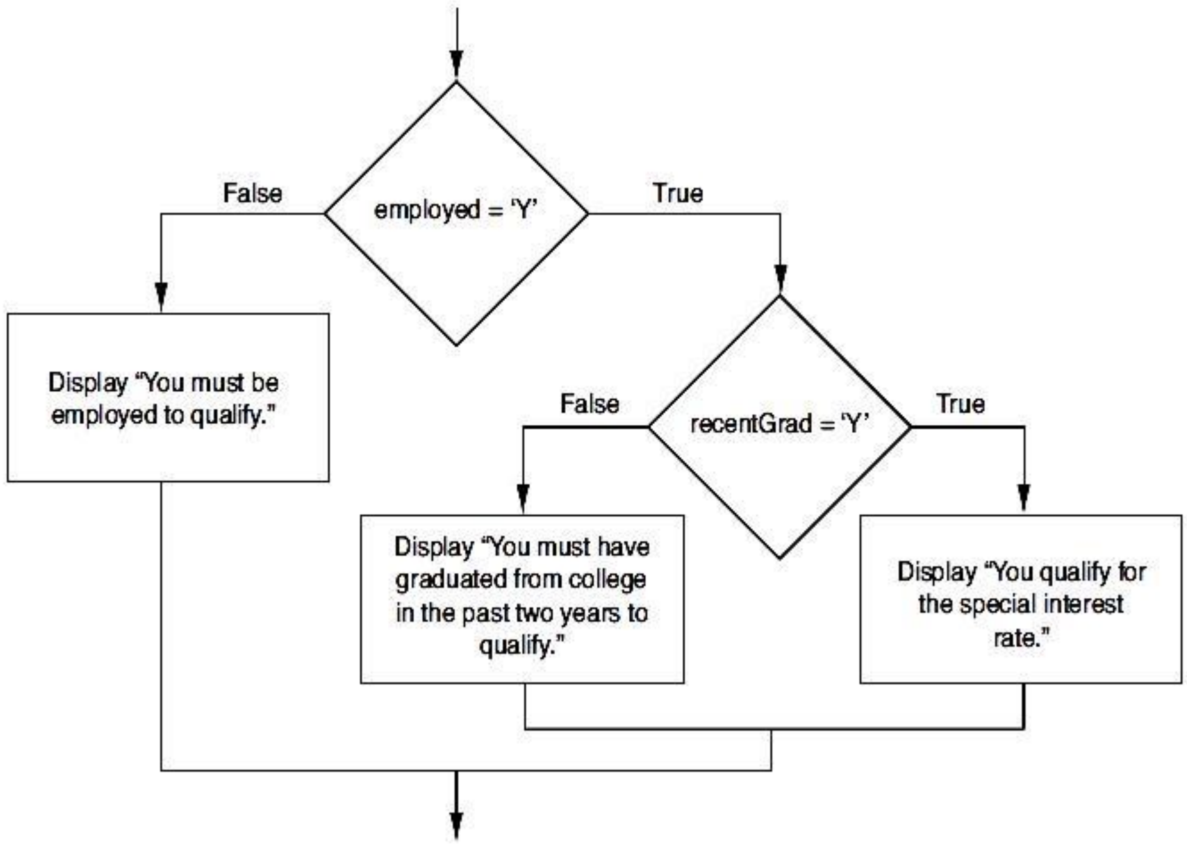

employed = 'Y'
False
True
Display "You must be employed to qualify."
recentGrad = 'Y'
False
True
Display "You must have graduated from college in the past two years to qualify."
Display "You qualify for the special interest rate."

Figure 6-5

Program 6-5 shows the code that corresponds to the logic of the flowchart. It nests one **if/else statement** inside another one.

Program 6-5

```
// This program determines whether a loan applicant qualifies for
// a special loan interest rate. It uses nested if/else statements.
#include <iostream>
using namespace std;

int main()
{
char employed, // Currently employed? (Y or N)
recentGrad; // Recent college graduate? (Y or N)

// Is the applicant employed and a recent college graduate?
cout << "Answer the following questions\n";
cout << "with either Y for Yes or N for No.\n";

cout << "Are you employed? ";
cin >> employed;
cout << "Have you graduated from college in the past two years? ";
cin >> recentGrad;

// Determine the applicant's loan qualifications
if (employed == 'Y')
{
if (recentGrad == 'Y') // Employed and a recent grad
{
cout << "You qualify for the special interest rate.\n";
}
else // Employed but not a recent grad
{
cout << "You must have graduated from college in the past\n";
cout << "two years to qualify for the special interest rate.\n";
}
}
else  // Not employed
{
cout << "You must be employed to qualify for the "
<< "special interest rate. \n";
}
return 0;
}
```

Program Output with Example Input Shown in Bold

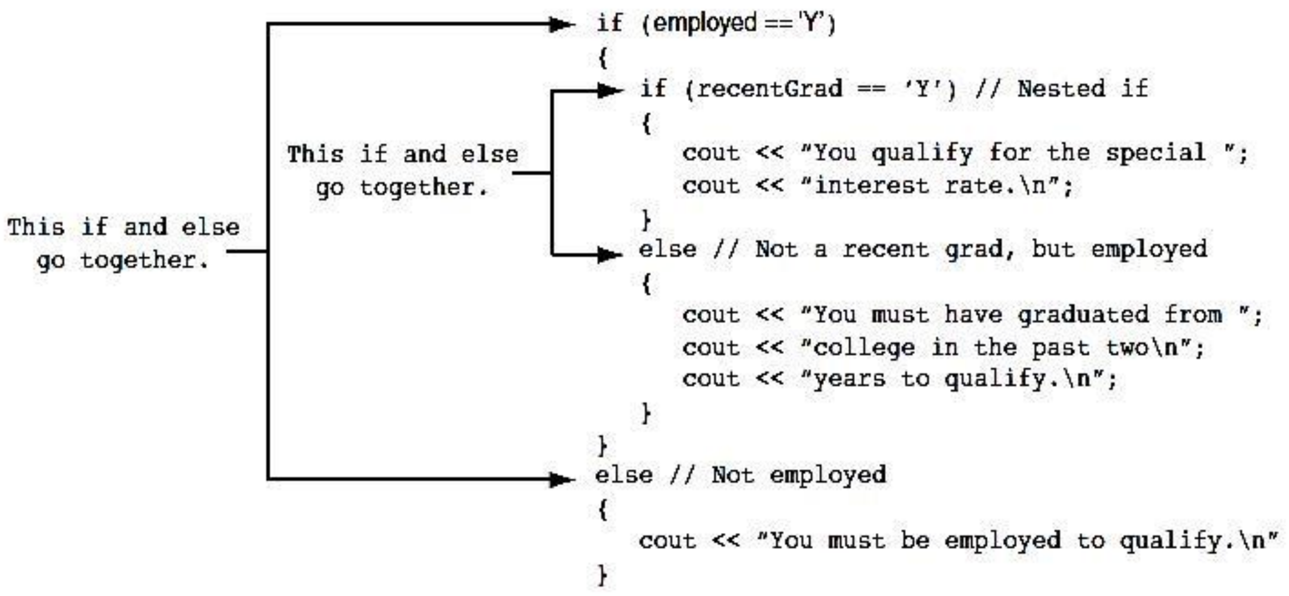
```
Answer the following questions
with either Y for Yes or N for No.
Are you employed? N[Enter]
Have you graduated from college in the past two years? Y[Enter]
You must be employed to qualify for the special interest rate.
```

Program Output with Other Example Input Shown in Bold

```
Answer the following questions
with either Y for Yes or N for No.
Are you employed? Y[Enter]
Have you graduated from college in the past two years? N[Enter]
You must have graduated from college in the past
two years to qualify for the special interest rate.
```

Program Output with Other Example Input Shown in Bold

```
Answer the following questions
with either Y for Yes or N for No.
Are you employed? Y[Enter]
Have you graduated from college in the past two years? Y[Enter]
You qualify for the special interest rate.
```



```
if (employed == 'Y')
{
   if (recentGrad == 'Y') // Nested if
   {
      cout << "You qualify for the special ";
      cout << "interest rate.\n";
   }
   else // Not a recent grad, but employed
   {
      cout << "You must have graduated from ";
      cout << "college in the past two\n";
      cout << "years to qualify.\n";
   }
}
else // Not employed
{
   cout << "You must be employed to qualify.\n"
}
```

Figure 6-6

11.3. The switch Statement

The switch statement lets the value of a variable or expression determine where the program will branch to.

A branch occurs when one part of a program causes another part to execute. The **if/else if statement** allows your program to branch into one of several possible paths. It performs a series of tests (usually relational) and branches when one of these tests is true. The **switch statement** is a similar mechanism. It, however, tests the value of an integer expression and then uses that value to determine which set of statements to branch to. **Here is the format of the switch statement**:

switch (*IntegerExpression*)
{
case *ConstantExpression:* // Place one or more
// statements here
case *ConstantExpression:* // Place one or more
// statements here
// case statements may be repeated
// as many times as necessary
default: // Place one or more
// statements here
}

The first line of the statement starts with the word switch, followed by an integer expression inside parentheses. This can be either of the following:

• A variable of any of the integer data types (including char).
• An expression whose value is of any of the integer data types.

On the next line is the beginning of a block containing several case statements. **Each case statement is formatted in the following manner**:

case *ConstantExpression:* // Place one or more
// statements here

After the word case is a constant expression (which must be of an integer type such as an **int** or **char**), followed by a colon. The constant expression can be either an integer literal or an integer named constant. **The expression cannot be a variable and it cannot be a Boolean expression such as x < 22 or n == 25**. The case statement marks the beginning of a section of statements. These statements are branched to if the value of the switch expression matches that of the case expression.

Program 6-6 shows how a simple switch statement works.

Program 6-6

```
// This program demonstrates the use of a switch statement.
// The program simply tells the user what character they entered.
#include <iostream>
using namespace std;

int main()
{
char choice;

cout << "Enter A, B, or C: ";
cin >> choice;

switch (choice)
{
case 'A':cout << "You entered A.\n";
break;
case 'B':cout << "You entered B.\n";
break;
case 'C':cout << "You entered C.\n";
break;
default: cout << "You did not enter A, B, or C!\n";
}
return 0;
}
```

Program Output with Example Input Shown in Bold

```
Enter A, B, or C: B[Enter]
You entered B.
```

Program Output with Different Example Input Shown in Bold

```
Enter A, B, or C: F[Enter]
You did not enter A, B, or C!
```

البرمجة بلغة ++C

Repetition Statements

12.1. The for Loop

The for loop is a pretest loop that combines the initialization, testing, and updating of a loop control variable in a single loop header.
In general, there are two categories of loops: **conditional loops** and **count-controlled loops**.
A ***conditional loop*** executes as long as a particular condition exists. For example, an input validation loop executes as long as the input value is invalid. When you write a conditional loop, you have no way of knowing the number of times it will iterate.
Sometimes you know the exact number of iterations that a loop must perform. A loop that repeats a specific number of times is known as a ***count-controlled loop***. For example, if a loop asks the user to enter the sales amounts for each month in the year, it will iterate twelve times. In essence, the loop counts to twelve and asks the user to enter a sales amount each time it makes a count. A count-controlled loop must possess three elements:
1. It must initialize a counter variable to a starting value.
2. It must test the counter variable by comparing it to a final value. When the counter variable reaches its final value, the loop terminates.
3. It must update the counter variable during each iteration. This is usually done by incrementing the variable.
Count-controlled loops are so common that C++ provides a type of loop specifically for them. It is known as the for loop. The for loop is specifically designed to initialize, test, and update a counter variable. **Here is the format of the for loop**.

```
for (initialization; test; update)
{
statement;
statement;
// Place as many statements
// here as necessary.
}
```

if there is only one statement in the loop body, the braces may be omitted.
The first line of the for loop is the loop header. After the key word **for**, there are three expressions inside the parentheses, separated by **semicolons**. (Notice there is no **semicolon** after the third expression.) The **first expression** is the **initialization expression**. It is typically used to initialize a counter to its starting value. This is the first action performed by the loop and it is only done once.
The second expression is the **test expression**. It tests a condition in the same way the test expression in the while and do-while loop does, and controls the execution of the loop. As long as this condition is true, the body of the for loop will repeat. The for loop is a pretest loop, so it evaluates the test expression before each iteration.
The third expression is the **update expression**. It executes at the end of each iteration.
Typically, this is a statement that increments the loop' s counter variable.

Here is an example of a simple for loop that prints “Hello” five times:

```
for (count = 1; count <= 5; count++)
    cout << "Hello" << endl;
```

Figure 7-1 illustrates the sequence of events that take place during the loop’s execution. **Notice** that Steps 2 through 4 are repeated as long as the test expression is true.

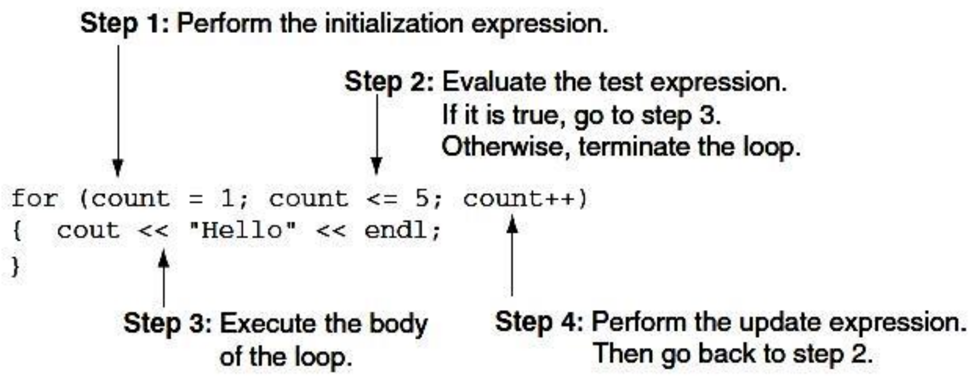

Step 1: Perform the initialization expression.
Step 2: Evaluate the test expression. If it is true, go to step 3. Otherwise, terminate the loop.
for (count = 1; count <= 5; count++)
{ cout << "Hello" << endl;
}
Step 3: Execute the body of the loop.
Step 4: Perform the update expression. Then go back to step 2.

Figure 7-1

Figure 7-2 shows the loop’s logic in the form of a flowchart.

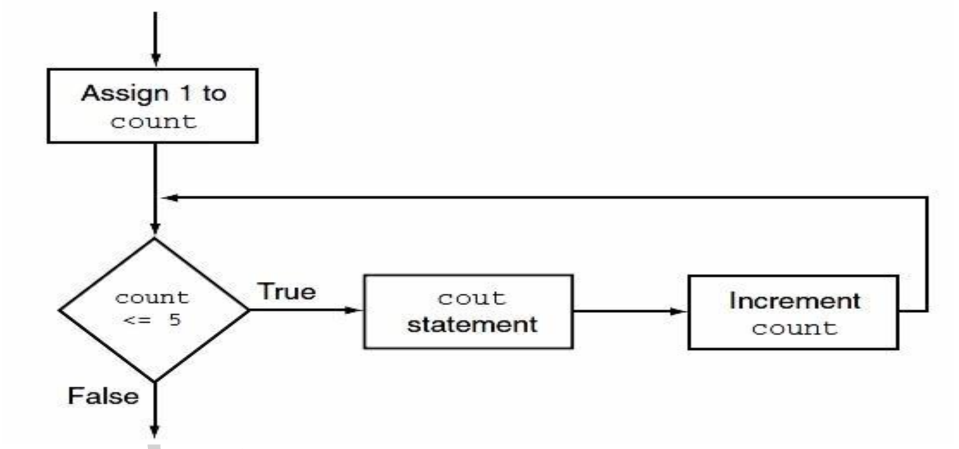


Figure 7-2

Program 7-1 displays the numbers 1-5 and their squares by using a for loop.

Program 7-1

```
1 // This program uses a for loop to display the numbers 1-5 and their squares.
2 #include <iostream>
3 #include <iomanip>
4 using namespace std;
5 int main()
6 { int num;
7    cout << "Number Square\n";
8    cout << "--------------\n";
9    for (num = 1; num <= 5; num++)
10     cout << setw(4) << num << setw(7) << (num * num) << endl;
11  return 0;
12}
```

Program Output

```
Number    Squared
--------------------------
  1          1
  2          4
  3          9
  4         16
  5         25
```

- **The for Loop is a Pretest Loop**

Because the for loop tests its test expression before it performs an iteration, it is possible to write a for loop in such a way that it will never iterate. Here is an example:

```
for (count = 11; count <= 10; count++)
    cout << "Hello" << endl;
```

Because the variable count is initialized to a value that makes the test expression false from the beginning, this loop terminates as soon as it begins.

- **Avoid Modifying the Counter Variable in the Body of the for Loop**

Be careful not to place a statement that modifies the counter variable in the body of the for loop. All modifications of the counter variable should take place in the update expression, which is automatically executed at the end of each iteration. If a statement in the body of the loop also modifies the counter variable, the loop will probably not terminate when you expect it to. The following loop, for example, increments x twice for each iteration:

```
for (x = 1; x <= 10; x++)
   {
   cout << x << endl;
   x++; // Wrong!
   }
```

- **Other Forms of the Update Expression**

You are not limited to incrementing the loop control variable by just 1 in the update expression. Here is a loop that displays all the even numbers from 2 through 100 by adding 2 to its counter:

```
for (num = 2; num <= 100; num += 2)
    cout << num << endl;
```

And here is a loop that counts backward from 10 down to 0:

```
for (num = 10; num >= 0; num--)
    cout << num << endl;
```

- **Defining a Variable in the for Loop's Initialization Expression**

Not only may the counter variable be initialized in the initialization expression, it may be defined there as well. The following code shows an example. This is a modified version of the loop in Program 5-9.

```
for (int num = 1; num <= 5; num++)
   cout << setw(4) << num << setw(7) << (num * num) << endl;
```

In this loop, the num variable is both defined and initialized in the initialization expression.
If the counter variable is used only in the loop, it is considered good programming practice to define it in the loop header. This makes the variable's purpose clearer.
When a variable is defined in the initialization expression of a for loop, the scope of the variable is limited to the loop. This means you cannot access the variable in statements outside the loop. For example, the following program segment will not compile because the last **cout** statement cannot access the variable count.

```
for (int count = 1; count <= 10; count++)
    cout << count << endl;
cout << "count is now " << count << endl; // ERROR!
```

- **Creating a User-Controlled for Loop**

In next program we allow the user to control how many times a loop should iterate by having the user enter the **final value** for the counter variable. The following program segment illustrates this.

```
// Get the final counter value
cout << "How many times should the loop execute? ";
cin >> finalValue;
for (int num = 1; num <= finalValue; num++)
{
// Statements in the loop body go here.
}
```

Repetition Statements

13.1.The while Loop

The while loop has two important parts:
(1) an expression that is tested for a true or false value, and
(2) a statement or block that is repeated as long as the expression is true.
Next figure shows the general format of the while loop and a flowchart visually depicting how it works.

```
while (condition)
{
      statement;
      statement;
      // Place as many statements
      // here as necessary
}
```

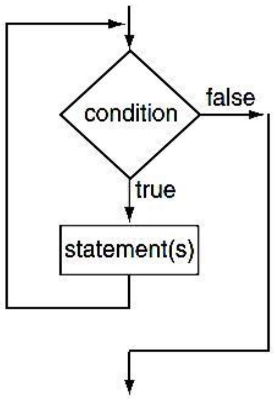


Figure 8-1

Let's look at each part of the **while loop**. The first line, sometimes called the **loop header**, consists of the key word **while** followed by a condition to be tested enclosed in parentheses.
The condition is expressed by any expression that can be evaluated as true or false. Next comes the **body** of the loop. This contains one or more C++ statements.

Here's how the loop works. The condition expression is tested, and if it is true, each statement in the body of the loop is executed. Then, the condition is tested again. If it is still **true**, each statement is executed again. This cycle repeats until the condition is **false**.

Notice that, as with an if statement, each statement in the body to be conditionally executed ends with a semicolon, **but there is no semicolon after the condition expression in parentheses**. This is because the while loop is not complete without the statements that follow it.

Next program uses a while loop to print “Hello” five times.

Program 8-1

```
// This program demonstrates a simple while loop.
#include <iostream>
int main()
{
int number = 1;
while (number <= 5)
  {
  cout << "Hello ";
  number++;
  }
cout << "\nThat's all!\n";
return 0;}
```

Program Output

```
Hello Hello Hello Hello Hello
That's all!
```

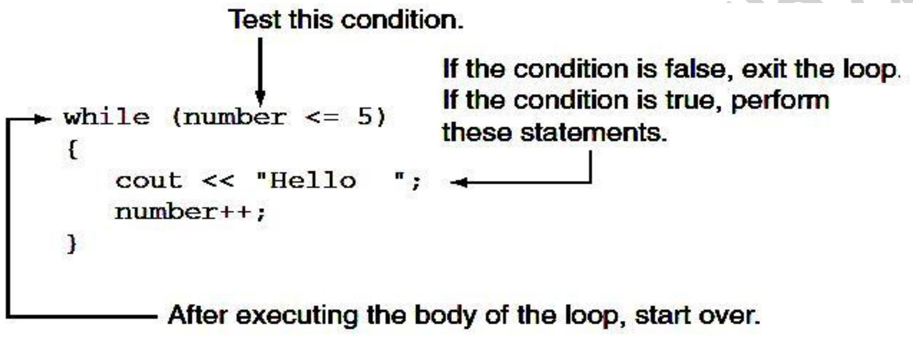
Let's take a closer look at this program. In line 6 an integer variable number is defined and initialized with the value **1**. In line 7 the **while loop** begins with this statement:

```
while (number <= 5)
```

This statement tests the variable **number** to determine whether its value is less than or equal to **5**. Because it is, the statements in the body of the loop (lines 9 and 10) are executed:

```
cout << "Hello ";
number++;
```

The statement in line 9 prints the word "Hello". The statement in line 10 uses the increment operator to add one to number, giving it the value 2. This is the last statement in the body of the loop, so after it executes the loop starts over. It tests the expression **number <= 5** again, and because it is still true, the statements in the body of the loop are executed again. This cycle repeats until the value of **number** equals **6**, making the expression **number <= 5** false. Then the loop is exited. This is illustrated in Figure 8-2.


Test this condition.
If the condition is false, exit the loop.
If the condition is true, perform these statements.
After executing the body of the loop, start over.

```
while (number <= 5)
{
    cout << "Hello  ";
    number++;
}
```

Figure 8-2

1. While is a Pretest Loop

The while loop is known as a **pretest loop**, which means it tests its expression before each iteration. Notice the variable definition of number in line 6 of Program 8-1:

```
int number = 1;
```

The **number** variable is initialized with the value **1**. If **number** had been initialized with a value greater than **5**, as shown in the following program segment, the loop would never execute:

```
int number = 6;
while (number <= 5)
{
cout << "Hello ";
number++;
}
```

An important characteristic of the while loop is that **the loop will never iterate if the test expression is false to start with**. If you want to be sure a while loop executes the first time, you must initialize the relevant data in such a way that the test expression starts out as true.

2. Infinite Loops

In all but rare cases, loops must contain within themselves a way to terminate. This means that something inside the loop must eventually make the test expression false. The loop in Program 8-1 stops when the expressions **number <= 5** is false. If a loop does not have a way of stopping, it is called **an infinite loop**. **Infinite loops** keep repeating until the program is interrupted. Here is an example:

```
int number = 1;
while (number <= 5)
{cout << "Hello ";}
```

This is an infinite loop because it does not contain a statement that changes the value of the number variable. Each time the expression number <= 5 is tested, number will contain the value 1.
It's also possible to create an infinite loop by accidentally placing a semicolon after the first line of the while loop. Here is an example:

```
int number = 1;
while (number <= 5); // This semicolon is an ERROR!
{cout << "Hello ";
  number++;}
```

3. Don’t Forget the Braces with a Block of Statements
If you write a loop that conditionally executes a block of statements, don't forget to enclose all of the statements in a set of braces. If the braces are accidentally left out, the while statement conditionally executes only the very next statement. For example, look at the following code.

```
int number = 1;
// This loop is missing its braces!
while (number <= 5)
cout << "Hello ";
number++;
```

In this code the **number++** statement is not in the body of the loop. Because the braces are missing, the while statement only executes the statement that immediately follows it. This loop will execute infinitely because there is no code in its body that changes the **number** variable.
Another common pitfall with loops is accidentally using the **=** operator when you intend to use the **==** operator.

Remember, any nonzero value is evaluated as true.

4. Using the while **Loop for Input Validation**
The while loop can be used to create input routines that repeat until acceptable data is entered.
Input validation is the process of inspecting data given to a program by the user and determining if it is valid.
The while loop is especially useful for validating input. If an invalid value is entered, a loop can require that the user re-enter it as many times as necessary. For example, the following loop asks for a number in the range of 1 through 100:

```
cout << "Enter a number in the range 1 - 100: ";
cin >> number;
while ((number < 1) || (number > 100))
{cout << "ERROR: Enter a value in the range 1 - 100: ";
  cin >> number;}
```

The general logic of performing input validation is shown in Figure 8-3.

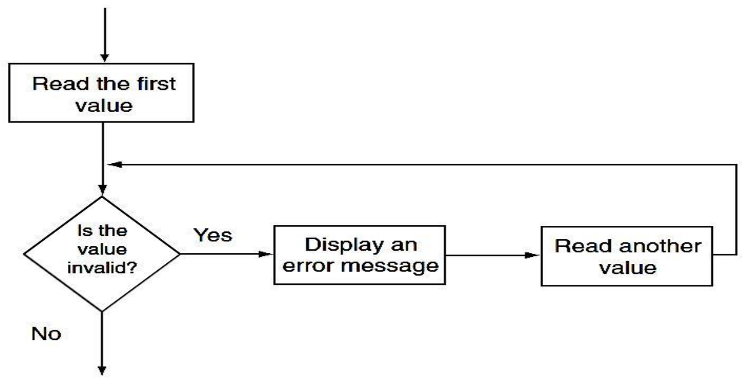


Figure 8-3

Program 8-2 calculates the number of soccer teams a youth league may create, based on a given number of players and a maximum number of players per team. The program uses while loops (in lines 20 through 25 and lines 31 through 35) to validate the user's input.

Program 8-2

```
1 // This program calculates the number of soccer teams a youth
2 // league may create from the number of available players.
3 // Input validation is done with while loops.
4 #include <iostream>
5 using namespace std;
6
7 int main()
8 {
9 int players, // Number of available players
10 teamPlayers, // Number of desired players per team
11 numTeams, // Number of teams
12 leftOver; // Number of players left over
13
14 // Get the number of players per team
15 cout << "How many players do you wish per team?\n";
16 cout << "(Enter a value in the range 9 - 15): ";
17 cin >> teamPlayers;
18
19 // Validate the input
20 while ((teamPlayers) < 9) || (teamPlayers > 15))
21 {
22 cout << "Team size should be 9 to 15 players.\n";
23 cout << "How many players do you wish per team? ";
24 cin >> teamPlayers;
25 }
26 // Get the number of players available
27 cout << "How many players are available? ";
28 cin >> players;
```

```

// Validate the input
while (players <= 0)
{
cout << "Please enter a positive number: ";
cin >> players;
}
// Calculate the number of teams
numTeams = players / teamPlayers;

// Calculate the number of leftover players
leftOver = players % teamPlayers;

// Display the results
cout << "\nThere will be " << numTeams << " teams with ";
cout << leftOver << " players left over.\n";
return 0;
}
```

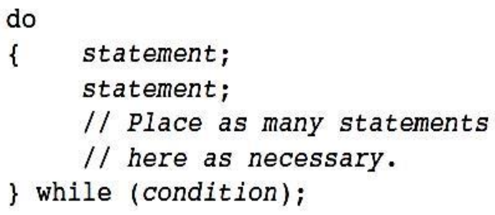
Program Output with Example Input Shown in Bold

```
How many players do you wish per team?
(Enter a value in the range 9 - 15): 4[Enter]
Team size should be 9 to 15 players.
How many players do you wish per team? 12[Enter]
How many players are available? –142[Enter]
Please enter a positive number: 142[Enter]
There will be 11 teams with 10 players left over.
```

Repetition Statements

14.1. The do-while Loop

The do-while loop is a posttest loop, which means its expression is tested after each iteration.
In addition to the **while loop**, C++ also offers the **do-while loop**. The do-while loop looks similar to a while loop turned upside down. Figure 8-4 shows its format and a flowchart visually depicting how it works.



```
do
{     statement;
      statement;
      // Place as many statements
      // here as necessary.
} while (condition);
```

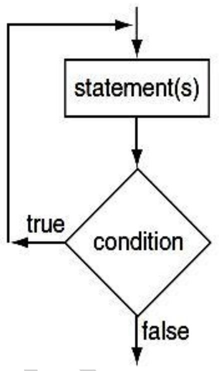


Figure 8-4

As with the while loop, if there is only one conditionally executed statement in the loop body, the braces may be omitted.

NOTE: The do-while loop must be terminated with a semicolon after the closing parenthesis of the test expression.

Besides the way it looks, the difference between the do-while loop and the while loop is that do-while is a post test loop. It tests its expression after each iteration is complete.
This means do-while always performs at least one iteration, even if the test expression is false at the start. For example, in the following **while loop** the **cout** statement will not execute at all:

```
int x = 1;
while (x < 0)
cout << x << endl;
```

But the **cout** statement in the following **do-while loop** will execute once because the **do-while loop** does not evaluate the expression x < 0 until the end of the iteration.

```
int x = 1;
do
cout << x << endl;
while (x < 0);
```

You should use the do-while loop when you want to make sure the loop executes at least once.
For example, Program 8-3 computes and displays the average of a set of test scores before asking if the user wants to repeat the process with another set of scores. As with the while loop, a do-while loop can be written to iterate a set number of times or to allow the user to control how many times to loop.
Program 8-3 illustrates another method for letting the user control the loop. It will repeat as long as the user enters a Y or y for yes.

Program 8-3

```
1 // This program averages 3 test scores. It uses a do-while loop
2 // that allows the code to repeat as many times as the user wishes.
```

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

int main()
{
int score1, score2, score3; // Three test scores
double average; // Average test score
char again; // Loop again? Y or N

do
{ // Get three test scores
cout << "\nEnter 3 scores and I will average them: ";
cin >> score1 >> score2 >> score3;

// Calculate and display the average
average = (score1 + score2 + score3) / 3.0;
cout << "The average is " << average << ".\n";

// Does the user want to average another set?
cout << "Do you want to average another set? (Y/N) ";
cin >> again;
} while ((again == 'Y') || (again == 'y'));
 return 0;}
```

Program Output with Example Input Shown in Bold

```
Enter 3 scores and I will average them: 80 90 70[Enter]
The average is 80.
Do you want to average another set? (Y/N) y[Enter]
Enter 3 scores and I will average them: 60 75 88[Enter]
The average is 74.3333.
Do you want to average another set? (Y/N) n[Enter]
```

14.2. Using do-while with Menus

The do-while loop is a good choice for repeating a menu.

Program 8-4 is use a do-while loop to repeat the program until the user selects item 4 from the menu.

Program 8-4

```
// This menu-driven Health Club membership program carries out the
// appropriate actions based on the menu choice entered. A do-while loop
// allows the program to repeat until the user selects menu choice 4.
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
// Constants for membership rates
const double ADULT_RATE = 40.0;
const double CHILD_RATE = 20.0;
const double SENIOR_RATE = 30.0;

int choice; // Menu choice
int months; // Number of months
double charges; // Monthly charges

do
{ // Display the menu and get the user's choice
cout << "\n Health Club Membership Menu\n\n";
cout << "1. Standard Adult Membership\n";
cout << "2. Child Membership\n";
cout << "3. Senior Citizen Membership\n";
cout << "4. Quit the Program\n\n";
cout << "Enter your choice: ";
cin >> choice;

// Validate the menu selection
while ((choice < 1) || (choice > 4))
{
cout << "Please enter 1, 2, 3, or 4: ";
cin >> choice;
}
// Process the user's choice
if (choice != 4)
{ cout << "For how many months? ";
cin >> months;

// Compute charges based on user input
switch (choice)
{
case 1: charges = months * ADULT_RATE;
break;
case 2: charges = months * CHILD_RATE;
break;
case 3: charges = months * SENIOR_RATE;
}
// Display the monthly charges
cout << fixed << showpoint << setprecision(2);
cout << "The total charges are $" << charges << endl;
}
} while (choice != 4); // Loop again if the user did not
// select choice 4 to quit
return 0;
}
```

Program Output with Example Input Shown in Bold

```
   Health Club Membership Menu
1. Standard Adult Membership
2. Child Membership
3. Senior Citizen Membership
4. Quit the Program

Enter your choice: 1[Enter]
For how many months? 12[Enter]
The total charges are $480.00

   Health Club Membership Menu
1. Standard Adult Membership
2. Child Membership
3. Senior Citizen Membership
4. Quit the Program


Enter your choice: 4[Enter]
```

14.3.**Nested Loops**

A loop that is inside another loop is called a nested loop. The first loop is called the **outer loop**. The one nested inside it is called the **inner loop**. We can put loops one inside another to solve a certain programming problems. Loops may be nested as follows:

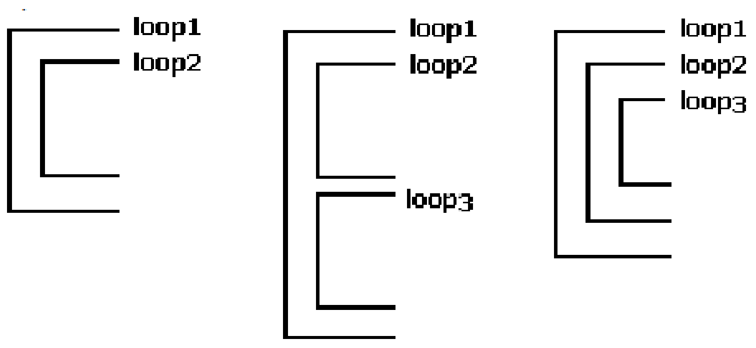


Figure 8-5

Notice how the inner loop must be completely contained within the outer one.
This is illustrated by the following two while loops.

```
while (condition1)   // Beginning of the outer loop
   { ------
    while (condition2)  // Beginning of the inner loop
        {-------
        }                        // End of the inner loop
    }                      // End of the outer loop
```

When we have program averages test scores by asking the user to input the number of students and the number of test scores per student. Next program show that:

Program 8-5

```
#include <iostream>
using namespace std;
int main()
{
int numStudents, // Number of students
numTests; // Number of tests per student
double average; // Average test score for a student
// Get the number of students
cout << "This program averages test scores.\n";
cout << "How many students are there? ";
cin >> numStudents;
// Get the number of test scores per student
cout << "How many test scores does each student have? ";
cin >> numTests;
cout << endl;
// Read each student's scores and compute their average
for (int snum = 1; snum <= numStudents; snum++) // Outer loop
{ double total = 0.0; // Initialize accumulator
for (int test = 1; test <= numTests; test++) // Inner loop
     { int score; // Read a score and add it to the accumulator
        cout << "Enter score " << test << " for ";
        cout << "student " << snum << ": ";
        cin >> score;
        total += score;
      } // End inner loop
average = total / numTests; // Compute and display the student's average
cout << "The average score for student " << snum;
cout << " is " << average << "\n\n";
} // End outer loop
return 0;
} // End main function
```

Program Output with Example Input Shown in Bold

```
This program averages test scores.
How many students are there? 2[Enter]
How many test scores does each student have? 3[Enter]
Enter score 1 for student 1: 84[Enter]
Enter score 2 for student 1: 79[Enter]
Enter score 3 for student 1: 97[Enter]
The average for student 1 is 86.6667
Enter score 1 for student 2: 92[Enter]
Enter score 2 for student 2: 88[Enter]
Enter score 3 for student 2: 94[Enter]
The average for student 2 is 91.3333
```

Home work:

Write a program to prints a multiplication table?

جامعة الفرات الأوسط التقنية
المعهد التقني كربلاء
قسم تقنيات أنظمة الحاسوب

البرمجة بلغة C^{++}

لطلبة السنة الاولى

أعداد

م.م. محمد ثجيل عبدالله

2023-2022

Lecture (15)

Outline

- Introducing One dimensional Arrays
- Represent One Dimensional Array in C++

Introducing One dimensional Arrays

- An **array** is a compound data structures/data form that can hold several values, all of one data type.
- An **array** works like a variable that can store a group of values, all of the same type.
- An **array** is a fixed size sequential collection of elements of identical types.
- Syntax(or general form):
  Elements_Data_type   Array_name[array_Size];
- The element in a one dimensional **array** are indexed by the integers   (**0** to **n- 1**), where **n** is the size of the array.
- **For example**, an array can hold 60 type integer values that represent five years of game sales data.

Represent One Dimensional Array in C++

- To create a one dimensional array, you use a **declaration statement**.
- A one dimensional array **declaration** should indicate three things:
  1. The type of value to be stored in each element.
  2. The name of the array.
  3. The number of elements in the array.
- This is the general form for **declaring** an array:

Datatype ArrayName[ArraySize];

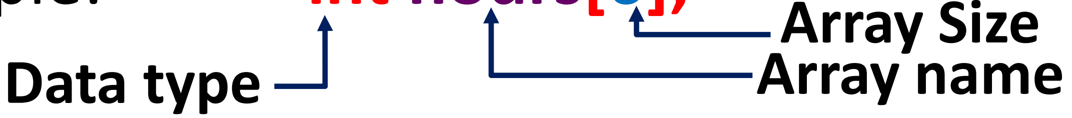
Datatype it is could be one of the primitive data structures in c++ like (int, float, char, ... etc).

ArrayName Any name that is within the terms of naming variables.

ArraySize which is the number of elements, must be an integer constant, such as 10 or a const value.

Represent One Dimensional Array in C++ cont'd

- Example: **int hours[6];**

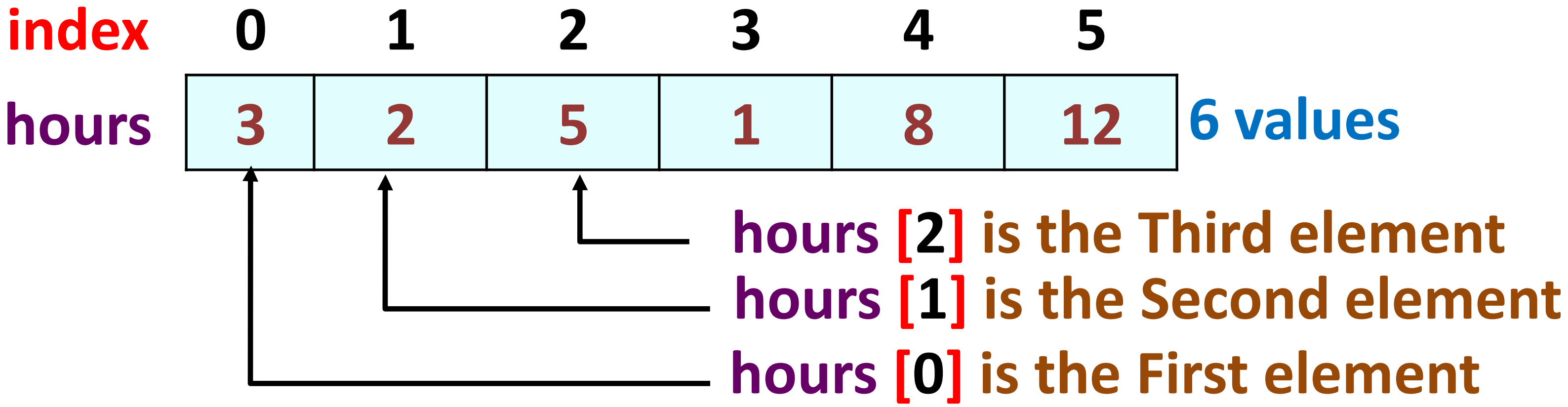


Or const int SIZE = 6;

int hours[SIZE];

- The name of this array is **hours.**

- The number inside the brackets is the array's *size* ***declarator***.
  It indicates the number of ***elements***, or values, the array can hold.


index
0
1
2
3
4
5
hours
3
2
5
1
8
12
6 values
hours [2] is the Third element
hours [1] is the Second element
hours [0] is the First element

Represent One Dimensional Array in C++ cont’d

Accessing Array Elements

- The individual elements of a one dimensional array are assigned unique subscripts/index. These subscripts are used to access the elements.
- Even though an entire array has only one name, the elements may be accessed and used as individual variables. This is possible because each element is assigned a number known as a ***subscript/index***.
- The following statement stores the integer 30 in **hours[3]**. Note that this is the fourth array element.

```
hours[3] = 30;   // hours[3] now holds 30.
```

Array name

Element value

Element index/subscript

Represent One Dimensional Array in C++ cont'd

- Why need to use array type?

  Consider the following issue: **"We have a list of 1000 students' marks of an integer type"**

- Can you imagine how long we have to write the declaration part by using normal variable declaration?

  We will declare something like the following:

  int studMark0, studMark1, studMark2, ..., studMark999;

- By using an array, we just declare like this:

```
int   studMark[1000];
```

Thank you to your attention

and

Any Question

جامعة الفرات الأوسط التقنية
المعهد التقني كربلاء
قسم تقنيات أنظمة الحاسوب

البرمجة بلغة C^{++}

لطلبة السنة الاولى

أعداد

م.م. محمد ثجيل عبدالله

2022-2021

Lecture (16)
Outline

- Initialize One Dimensional Array in C++
- Inputting and Displaying One Dimensional Array Contents in C++
- Processing Array Contents
- Copying One Array to Another
- Comparing Two Arrays

Initialize One Dimensional Array in C++

- Initializing all specified memory locations:

Arrays can be initialized at the time of declaration when their initial values are known in advance.

Ex: **int hours[6]={10,5,2,6,14,1};**

- Partial array initialization:

Partial array initialization is possible in c language. If the number of values to be initialized is less than the size of the array, then the elements will be initialized to zero automatically.

Ex: **int hours[6]={10,5,2};**

- Initialization without size:

Consider the declaration along with the initialization.

Ex: **char b[]={'C','O','M','P','U'};**

Inputting and Displaying One Dimensional Array Contents in C++

```
// This program stores employee work hours in an int array.
// It uses one for loop to input the hours and another for loop to
// display them.
 #include <iostream>
 int main() {
 const int NUM_EMPLOYEES = 6;
 int hours[NUM_EMPLOYEES]; // Holds hours worked for 6 employees
 int count; // Loop counter
 // Input hours worked by each employee
cout << "Enter the hours worked by " << NUM_EMPLOYEES << " employees: ";
for (count = 0; count < NUM_EMPLOYEES; count++)
    cin >> hours[count];
// Display the contents of the array
cout << "The hours you entered are:";
for (count = 0; count < NUM_EMPLOYEES; count++)
    cout << " " << hours[count];
cout << endl;
return 0;}
```

Processing One dimensional array contents

- Individual a **one dimensional array elements** are processed like any other type of **variable**.
- **For example**, the following statement multiplies **hours[3]** by the **variable rate**:

```
pay = hours[3] * rate;
```

- And the following are examples of pre-increment and post-increment operations on array elements:

```
int score[5] = {7, 8, 9, 10, 11};
++score[2]; // Pre-increment operation on the value in score[2]
score[4]++; // Post-increment operation on the value in score[4]
```

Copying One Array to Another

- You cannot simply assign one array to another array. To copy the contents of one array to another, you must assign each element of the first array, one at a time, to the corresponding element of the second array. The following code segment uses a for loop to do this.

```
const int SIZE = 6;
int arrayA[SIZE] = {10, 20, 30, 40, 50, 60};
int arrayB[SIZE] = { 2, 4, 6, 8, 10, 12};
for (int index = 0; index < SIZE; index++)
    arrayA[index] = arrayB[index];
```

Comparing Two Arrays

- Just as you cannot copy one array to another with a single statement, you also cannot compare the contents of two arrays with a single statement.
- That is, you cannot use the == operator with the names of two arrays to determine whether the arrays are equal.
- The following code appears to compare the contents of two arrays, but in reality does not.

```
int arrayA[] = { 5, 10, 15, 20, 25 };
int arrayB[] = { 5, 10, 15, 20, 25 };
if (arrayA == arrayB) // This is a mistake
   cout << "The arrays are the same.\n";
else
   cout << "The arrays are not the same.\n";
```

Comparing Two Arrays cont’d

- To **compare the contents of two arrays**, you must compare their individual elements. **For example**, look at the following code:

```
 const int SIZE = 5;
 int arrayA[SIZE] = { 5, 10, 15, 20, 25 };
 int arrayB[SIZE] = { 5, 10, 15, 20, 25 };
 bool arraysEqual = true; // Flag variable
 int count = 0; // Loop counter variable
// Determine whether the elements contain the same data
while (arraysEqual && count < SIZE){
        if (arrayA[count] != arrayB[count])
            arraysEqual = false;
        count++;}
// Display the appropriate message
if (arraysEqual)
  cout << "The arrays are equal.\n";
else
  cout << "The arrays are not equal.\n";
```

Prepared by : Mohammed Thajeel

Thank you to your attention

and

Any Question

جامعة الفرات الأوسط التقنية
المعهد التقني كربلاء
قسم تقنيات أنظمة الحاسوب

البرمجة بلغة C++

لطلبة السنة الاولى

أعداد

م.م. محمد ثجيل عبدالله

2022-2021

Lecture (17)
Outline

- Introducing Two dimensional Arrays
- Represent Two dimensional Array in C++
- Initialize Two dimensional Array in C++

Introducing Two dimensional Arrays

- Is a data structure consists of a set of elements which are all of the same type, it characterized by all its elements are distributed on a set of rows and columns that represent the size of these array.
- To define a **two-dimensional** array, **two size** declarators are required: The **first one is for the number** of **rows (1st dimension size)** and the **second one is for the number** of **columns (2st dimension size)** .
- Syntax(or general form):

Data_type Array_name[1st dimension size][2st dimension size];

- For processing the information in a two dimensional array, each element has **two** subscripts/**indexes**: one for its **row** and another for its **column**.

Represent Two dimensional Array in C++

- To create a two dimension array, you use a **declaration statement**.
- A two dimension array **declaration** should indicate four things:
  1. The data type of value to be stored in each element.
  2. The name of the array.
  3. The number of elements in row (1st dimension size) of array.
  4. The number of elements in column (2st dimension size) of array.
- This is the general form for **declaring** a two dimension array:

Datatype Arrayname[1st dimension size][2st dimension size];

Datatype it is could be one of the primitive data structures in c++ like (int, float, char, ... etc).

Arrayname Any name that is within the terms of naming variables.

1st dimension size which is the number of rows array, must be an integer constant, such as 10 or a const value.

2st dimension size which is the number of columns array, must be an integer constant, such as 10 or a const value.

Represent Two dimensional Array in C++ cont'd

- Example:

int S_M[3][2];

Data type

Array name

number of rows array

number of columns array

Or const int ROWSSIZE = 3;

const int COLUMNSSIZE = 2;

int S_M[ROWSSIZE][COLUMNSSIZE];

Columns index / Rows index	0	1
0	S_M[0][0]	S_M[0][1]
1	S_M[1][0]	S_M[1][1]
2	S_M[2][0]	S_M[2][1]

Represent Two dimensional Array in C++ cont’d
Accessing Array Elements

- The individual elements of a **two dimensional** array are assigned unique pair indexes one for row and another for column. These indexes are used to access the elements.
- Even though an entire array has only one name, the elements may be accessed and used as individual variables. This is possible because each element is assigned a two numbers known as a **row index and a column index**.
- The following statement stores the integer 30 in **S_M[1][1]**. Note that this is the fourth array element.

```
S_M[1][1] = 30;   // S_M[1][1] now holds 30.
```

Array name

Element value

Element column index

Element row index

Initialize Two dimensional Array in C++

- Initializing all specified memory locations:

Arrays can be initialized at the time of declaration when their initial values are known in advance.

Ex: **int S_M[3][2]={{60,59},{68,48},{88,78}};**

- Partial array initialization:

Partial array initialization is possible in c language. If the number of values to be initialized is less than the size of the array, then the elements will be initialized to zero automatically.

Ex: **int S_M[3][2]={{60,59},{68}};**

- Initialization without size:

Consider the declaration along with the initialization.

Ex: **int S_M[][]={{60,59},{68,48},{88,78}};**

جامعة الفرات الأوسط التقنية
المعهد التقني كربلاء
قسم تقنيات أنظمة الحاسوب

البرمجة بلغة C^{++}

لطلبة السنة الاولى

أعداد

م.م. محمد ثجيل عبدالله

2022-2021

Lecture (18)
Outline

- Inputting and Displaying Two Dimensional Array Contents in C++
- Processing Array Contents
- Square Array

Inputting and Displaying Two dimensional Array Contents in C++

```
// This program stores number of students and for each student
// number of marks in an int array.
// It uses for loop to input the marks for students and another for
// loop to display them.
 #include <iostream>
 int main() {
 const int ROW_SIZE= 10, COLUMN_SIZE= 10;
 int S_M[ROW_SIZE][COLUMN_SIZE];
 int num_students,num_marks; // number of rows and number of columns
 int i,j; // Row index and Column index
// Input the number of rows and number of columns
 cout << "Enter the number of students : ";
cin>>num_students;
cout << "Enter the number of marks : ";
cin>>num_marks;
```

Inputting and Displaying Two dimensional Array Contents in C++ cont'd

```
// Input the marks for each students (input array)
for (i = 0; i < num_students; ++i){
    cout << "\n Input the marks to the student number "<<i+1;
    for (j= 0; j < num_marks; ++j){
        cout << "\n Input the mark number "<< j+1 <<" : ";
        cin >> S_M[i] [j];}
   }
// Display the marks for each students (display array)
for (i = 0; i < num_students; ++i){
    cout << "\n these marks for the student number "<<i<<endl;
    for (j= 0; j < num_marks; ++j)
         cout<<"  "<<S_M[i] [j];
   }
return 0;}
```

Processing Two dimensional array contents

- Individual a **two dimensional array elements** are processed like any other type of **variable**.
- **For example**, the following statement calculate the summation marks for the first student and store it in **variable SUM**:

```
SUM = S_M[0][0]+S_M[0][1]+S_M[0][2] ;
```

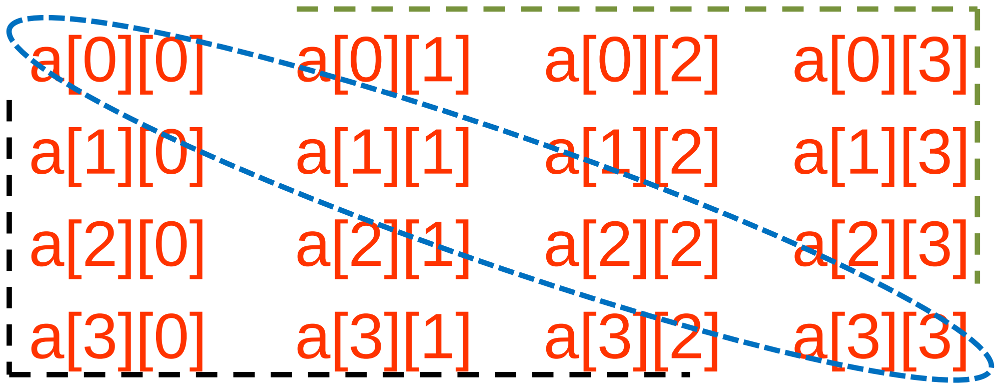
OR

```
for (j = 0; j <= 2; ++j)
   SUM = SUM + S_M[0][j];
```

Square Array

- When you have square array which mean the number of rows equal to the number of columns as:

int a[n][n];

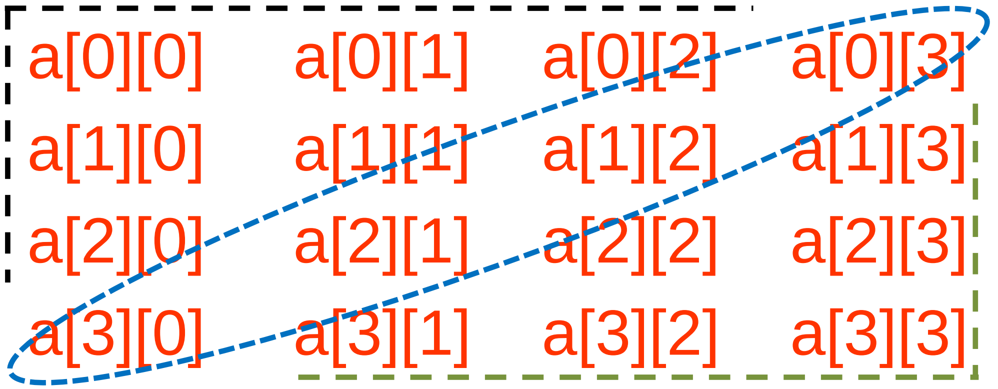

a[0][0] a[0][1] a[0][2] a[0][3]
a[1][0] a[1][1] a[1][2] a[1][3]
a[2][0] a[2][1] a[2][2] a[2][3]
a[3][0] a[3][1] a[3][2] a[3][3]

- **a**(i,j) **is on first diagonal if i = j.**
- **a**(i,j) **is on upper triangle of the first diagonal if i < j.**
- **a**(i,j) **is on lower triangle of the first diagonal if i > j.**

Square Array cont'd

- When you have square array which mean the number of rows equal to the number of columns as:

int a[n][n];



a[0][0]	a[0][1]	a[0][2]	a[0][3]
a[1][0]	a[1][1]	a[1][2]	a[1][3]
a[2][0]	a[2][1]	a[2][2]	a[2][3]
a[3][0]	a[3][1]	a[3][2]	a[3][3]

- **a**(i,j) **is on second diagonal if i + j=n-1.**
- **a**(i,j) **is on lower triangle of the second diagonal if i+j>=n.**
- **a**(i,j) **is on upper triangle of the second diagonal if i+j<n-1.**

جامعة الفرات الأوسط التقنية
المعهد التقني كربلاء
قسم تقنيات أنظمة الحاسوب

البرمجة بلغة C++

لطلبة السنة الاولى

أعداد

م.م. محمد ثجيل عبدالله

2022-2021

Lecture (19)
Outline

- Introduction to String
- Input and Display String in C++
- Useful String Functions and Operators

Introduction to String

- Standard C++ provides a special data type for storing and working with strings.
- The **first step** in using the string is to **#include** the **string header file**. This is accomplished with the following preprocessor directive:

```
#include <string>
```

- The **next step** is to define a **string type variable**, called a string object. Defining a string object is similar to defining a variable of a primitive type.

Syntax(or general form) to declaration string is:

```
string variable_name;
```

Input and Display String in C++

- You can use **cin** with the **>>** operator to input **strings** or use **cout** with **<<** operator to display **strings**, after you use a **declaration statement** to it. For example the next program (11-1) read and print student name.

```
#include <iostream>
#include <string>
int main() {
  string Stname;
  cout << "Please enter student name: ";
  cin >> Stname;
  cout << "Hello, "<< Stname << endl;
return 0;}
```

Input and Display String in C++

- With use **cin** to input **string** you can not use space in your **string** when you have to strings. For example the next program (11-2)read your name and your city name.

```
#include <iostream>
#include <string>
int main() {
string Name, City;
cout << “Enter Your name: ";    cin >> Name;
cout << “Enter Your City name: ";   cin>>City;
cout<<"Hello,"<<Name<<“ You live in “<<City<<endl;
return 0;}
```

Input and Display String in C++

- To solve this problem, C++ provides a special function called **getline**. Syntax(or general form) to use **getline** function is:

  getline(cin,string_varible);

- The same program (11-2) it will be:

```
#include <iostream>
#include <string>
int main() {
string Name, City;
cout << “Enter Your name: ";           getline(cin,Name);
cout << “Enter Your City name: ";    getline(cin,City);
cout<<"Hello,"<<Name<<“ You live in “<<City<<endl;
return 0;}
```

Useful String Functions and Operators

- The C++ provides a number of functions, called member functions, for working with strings. One that is particularly useful is the **length function**, which tells you how many characters there are in a string. Here is an example of how to use it.

```
string state = "New Jersey";
int size = state.length();
```

- The C++ also has special operators for working with strings. One of them is the + operator. when this operator is used with string operands it *concatenates* them, or joins them together. Here is an example of how to use it.

```
string Fname = “Ali", Sname=“Abd”;
string Fullname = Fname+Sname;
```

Member Function Example	Description
`theString.erase(x, n);`	Erases n characters from `theString`, beginning at position x.
`theString.find(str, x);`	Returns the first position at or beyond position x where the string `str` is found in `theString`. The parameter `str` may be either a string object or a C-string. If `str` is not found, a position beyond the end of `theString` is returned.
`theString.find('z', x);`	Returns the first position at or beyond position x where `'z'` is found in `theString`.
`theString.insert(x, str);`	Inserts a copy of `str` into `theString`, beginning at position x. `str` may be either a string object or a character array.
`theString.insert(x, n, 'z');`	Inserts `'z'` n times into `theString` at position x.
`theString.length();`	Returns the length of the string in `theString`.
`theString.replace(x, n, str);`	Replaces the n characters in `theString` beginning at position x with the characters in string object `str`.
`theString.resize(n, 'z');`	Changes the size of the allocation in `theString` to n. If n is less than the current size of the string, the string is truncated to n characters. If n is greater, the string is expanded and `'z'` is appended at the end enough times to fill the new spaces.
`theString.size();`	Returns the length of the string in `theString`.
`theString.substr(x, n);`	Returns a copy of a substring. The substring is n characters long and begins at position x of `theString`.
`theString.swap(str);`	Swaps the contents of `theString` with `str`.

جامعة الفرات الأوسط التقنية
المعهد التقني كربلاء
قسم تقنيات أنظمة الحاسوب

البرمجة بلغة C^{++}

لطلبة السنة الاولى

أعداد

م.م. محمد ثجيل عبدالله

2022-2021

Lecture (20)
Outline

- Breaking Out of a Loop
- Using break in a Nested Loop
- The continue Statement

Breaking Out of a Loop

- **The break statement causes a loop to terminate early.**
- Sometimes it's necessary to stop a loop before it goes through all its iterations.
- The **break statement**, which was used with switch statement, can also be placed inside a loop.
- When it is encountered, the loop stops and the program jumps to the statement immediately following the loop.
- The while loop in the following program segment appears to execute 10 times, but the **break statement** causes it to stop after the fifth iteration.

Breaking Out of a Loop

```
int count = 1;
while (count <= 10){
    cout << count << endl;
    count++;
    if (count == 6)
      break;}
```

Using break in a Nested Loop

- In a nested loop, the **break statement** only interrupts the loop it is placed in.
- The following program segment displays three rows of asterisks on the screen. The outer loop controls the number of rows and the inner loop controls the number of asterisks in each row. The inner loop is designed to display 20 asterisks, but the break statement stops it during the 11th iteration.

```
for (int row = 0; row < 3; row++){
   for (int star = 0; star < 20; star++){
      cout << '*';
      if (star == 10)   break;}
   cout << endl;}
```

The output of this program segment is:

```
***********
***********
***********
```

The continue Statement

- The **continue statement** causes a loop to stop its current iteration and begin the next one.
- The **continue statement** causes the current iteration of a loop to end immediately. When continue is encountered, all the statements in the body of the loop that appear after it are ignored, and the loop prepares for the next iteration.
- The following program segment demonstrates the use of continue in a while loop:

```
int testVal = 0;
while (testVal < 10){
   testVal++;
   if (testVal == 4)
      continue; // Terminate this iteration of the loop
   cout << testVal << " ";}
```

Here is the output:

```
1 2 3 5 6 7 8 9 10
```