

# ***Introduction to Programming With C++***

***Dept. of Computer Science  
Institute of Statistical Studies & Research  
Cairo University***

***January 2011***



# Chapter 1

## Problem Solving and Algorithms

# Chapter 1

## Problem Solving and Algorithms

Solving a problem on computer involves the following activities:

1. Define the problem.
2. Analyze the problem.
3. Develop an *algorithm* for solving the problem.
4. Write a *program* corresponding to the algorithm.
5. Test and *debug* the program.

An *algorithm* is a set of instructions that, if followed, will produce a solution to a given problem. We must write the instructions in a form that they can be easily converted into *computer instructions*.

Computer instructions fall into three main categories:



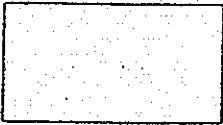
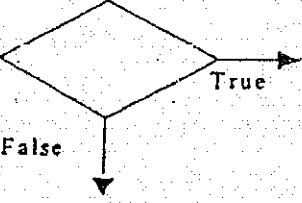

- *Input* instructions, used for supplying data to the computer.
- *Processing* instructions, for manipulating data inside the memory.

*Output* instructions, used for getting information out of the computer.

The data we supply via the input instructions is stored in the *memory* of the computer. In order to refer to the stored data, we give a name to each data item. Since the values of the data items can vary, the given names are called *variable names*, or simply *variables*. Thus a variable is a name associated with a particular memory location.

In this chapter, we provide algorithms for a number of small problems. We begin with a statement of the problem. As part of the problem analysis, we identify the problem inputs, the desired outputs and the relevant formula. Finally, we develop the algorithm. The given algorithms are specified using a set of *flowchart symbols* connected by arrows. Each symbol contains information about what must be done at that point and the arrows show the *flow of execution* of the algorithm.

Now, we illustrate the symbols most often used in flowcharting.

<u>Symbol</u>	<u>Function</u>
	Indicates Start or Stop
	Used to specify an input operation
	Used to specify an operation or process
	Used to specify a condition. The arrows lead to the actions corresponding to the value of the condition (True or False)
	Used to specify an output operation.

A flowchart is a useful tool when the algorithm is short and the flowchart can fit conveniently on a page. But if the algorithm is

long and/or complicated, many programmers prefer to use *pseudocode* for specifying algorithms.

Specifying an algorithm in 'pseudocode' means that the algorithm is written in a language similar to a real programming language. An important feature of pseudocode is that we are not forced to follow the strict rules imposed by a programming language. The only requirement is that the 'code' is clear and unambiguous. Usually, one uses English-like statements while observing some general conventions.

**Example 1: Converting units of measurement.**

**Problem statement:** You work in a store that imports textile. Most of the textile you receive is measured in square meters. However, the store's customers want to know the equivalent amount in square yards. You need to write an algorithm that performs this conversion.

**Problem analysis:** The purpose of this activity is to make sure we have a clear understanding of the given problem. It includes

determining the given inputs, the required outputs and the relevant formulas. These requirements are summarized below:

Problem Input:

The textile size in square meters (represented by the variable,  $M$ )

Problem Output:

The textile size in square yards (represented by the variable,  $Y$ )

Relevant Formula:

1 square meter = 1.196-square yards

Algorithm:

Please see figure 1.



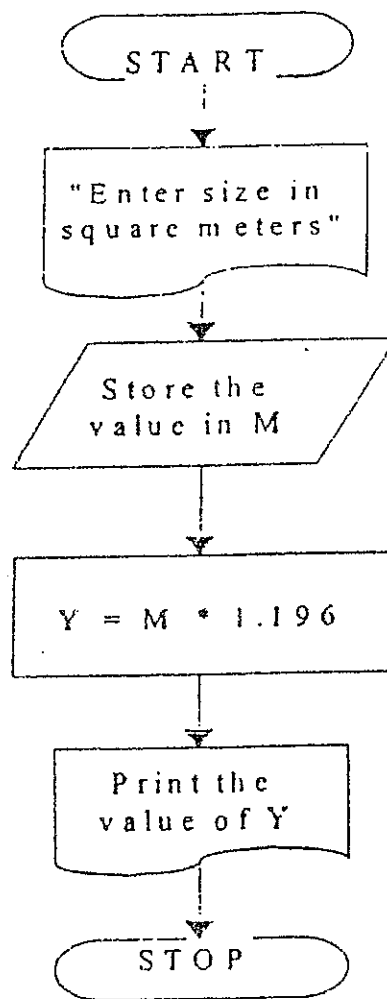


Figure 1. Converting units of measurements

Example 2: Finding the largest number.

Problem statement: Write an algorithm that finds and prints the largest among three numbers.

Problem analysis:

Problem Input:

Three numbers (represented by the variables:  $X$ ,  $Y$ ,  $Z$ )

Problem Output:

The largest number (represented by the variable:  $L$ )

Algorithm: This problem can be solved by first comparing  $X$  and  $Y$  and saving the largest in  $L$ . Then  $L$  can be compared to  $Z$  to find the largest.

Please see figure 2.

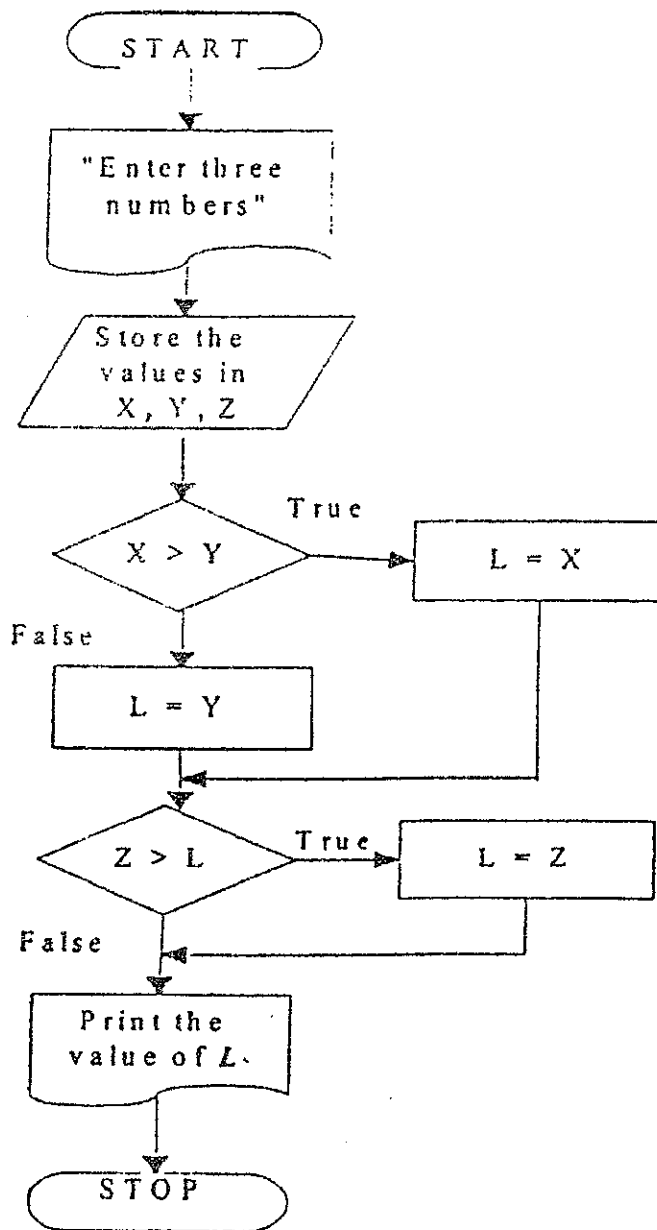


Figure 2. Finding the largest number

### Example 3: Finding the remainder and quotient.

Problem statement: Write an algorithm that finds and prints the *remainder* and *quotient* upon dividing a non-negative integer by a positive integer.

#### Problem analysis:

##### Problem Input:

The dividend (represented by the variable: X)

The divider (represented by the variable: Y)

##### Problem Output:

The remainder (represented by the variable: R)

The quotient (represented by the variable: Q)

Algorithm: One way to solve this problem is to repeat subtracting the value of the divider from the value of dividend, until the remaining value of dividend becomes less than the value of divider. In this case, the quotient will be the number of performed subtractions, and the remainder will be the remaining value of the dividend.

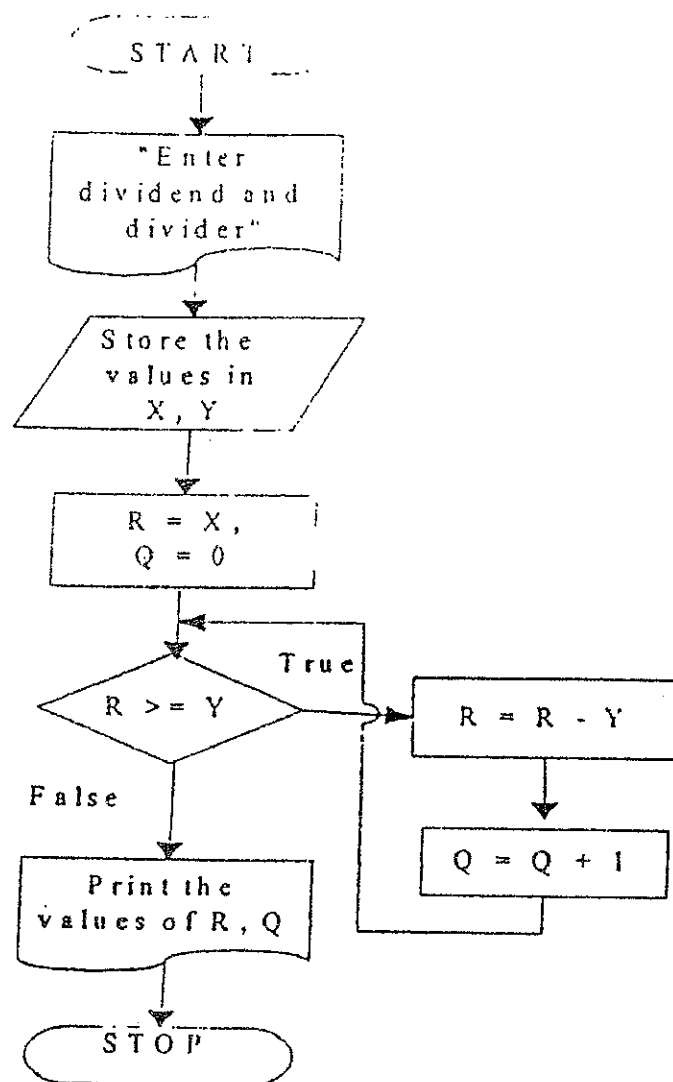


Figure 3. Finding the remainder and quotient

## Exercises

- [1] Write an algorithm that determines for a pair of integers whether the second integer is a multiple of the first. The function should take two integer arguments and return 1(true) if the second is a multiple of the first and 0(false) if otherwise.
- [2] An integer number is said to be a perfect number if its factors, including 1 (but not the number itself), sum to the number. For example, 6 is a perfect number because  $6=1+2+3$ . Write an algorithm that determines if parameter number is a perfect number.
- [3] An integer is said to be prime if it is divisible only by 1 and itself. For example, 2, 3, 5, and 7 are prime, but 4, 6, 8 and 9 are not. Write an algorithm that determines if a number is prime.
- [4] Write an algorithm that finds the number of days in a month of a given year. The input parameters should be a pair of integers: the given year and the number of the month.

# Chapter 2

## An Overview of C++

## Chapter 2

### An Overview of C++

The programming language C++ was developed, as an extension of C, in the early 1980's at Bell Laboratory. C++ provides a number of features that improve the C language, but more importantly, it provides capabilities for *object-oriented programming*. C++ is a hybrid language: it is possible to program in C++ in a structured style, an object-oriented style, or both.

In this chapter, we introduce C++ programming through a programming example that illustrates the basic features of C++. After presenting the program and its output window, the language's features of that program will be discussed. Then the C++ data types as well as a collection of C++ operators will be introduced.

#### Programming Example: Converting Units of Measurement

Problem statement: You work in a store that imports textile. Most of the textile you receive is measured in square meters. However, the store's customers want to know the equivalent



amount in square yards. You need to write an algorithm that performs this conversion.

Problem analysis:

Problem Input:

The textile size in square meters (represented by the variable,  $m$ )

Problem Output:

The textile size in square yards (represented by the variable,  $y$ )

Relevant Formula:

1 square meter = 1.196 square yards

Program:

// This program converts square meters to square yards.

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

```
int main( )
```

```
{
```

```
    float m, y;
```

```
    cout << "Enter the textile size in square meters:\n";
```

```
    cin >> m;
```

```
    y = 1.196 * m;
```

```
    cout << "The textile size in square yards is " << y;
```

```
    return 0;
```

```
}
```

### Output Window

Enter the textile size in square meters:

2.0

The textile size in square yards is 2.392

Based on the above program, we describe some basic features of the C++ programming language. The line

```
// This program converts square meters to square yards.
```

begins with `//` indicating the remainder of the line is a *comment*. Comments provide supplementary information to the person reading the program. Program comments are ignored by the compiler and are not translated into machine language.

The line

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

is a *preprocessor directive*, i.e. a message to the C++ preprocessor. It tells the preprocessor to include in the program the contents of the *input/output stream header file* `iostream.h`. This file must be included for any program that outputs data to the

screen or inputs data from the keyboard using C++ style stream input/output. Lines beginning with # are processed by the preprocessor before the program is compiled.

The above C++ program consists of one function. A function definition consists of a *header* and a *body*. The function header states the function name and the type of the return value. The header also specifies variables, known as *formal parameters*, which receive the incoming arguments.

The function body consists of a sequence of *declarations* and *statements* enclosed in braces, {}. A declaration supplies information to the C++ compiler, and a statement specifies actions for execution. The general form of a function is:

```
value-type name (type arg1, type arg2, ... )  
{  
    Declarations and Statements  
}
```

The *value-type* specifies the type of the value returned by the function. If it is omitted, the function is presumed to return a

value of type integer (*int*). If the function does not return a value, then its *value-type* is given as *void*.

The line

```
int main( )
```

is a function header. It specifies the function name, *main*, type of the return value, *int*, and an empty list of formal parameters.

The line

```
float m, y;
```

is a declaration. It specifies that the two *variables*: *m* and *y* are of type **float**. A variable is a location in the computer's memory where a value can be stored for use by a program. The data type **float** is used to specify real numbers, i.e. numbers with decimal points like 3.4, 0.0, -11.19.

The line

```
cout << "Enter the textile size in square meters:\n";
```

is a statement that instructs the computer to print on the screen the *string* of characters contained between the quotation marks. Every statement must end with a *semicolon* (;).

The operator << is referred to as the *stream insertion operator*. When the above statement executes, the value to the right of the operator is inserted in the built-in *output stream object* – `cout` – that is “connected” to the screen.

The characters between the double quotes normally are printed on the screen exactly as they appear in the statement. Notice, however, that the characters `\n` are not printed on the screen. When a backslash (`\`) is encountered in a string of characters, the next character is combined with the backslash to form an *escape sequence*. The escape sequence `\n` means *new-line*. It causes the *cursor* (i.e. the current screen position indicator) to move to the beginning of the next line on the screen.

The line

```
cin >> m;
```

is a statement that instructs the computer to obtain a value from the keyboard. The operator >> is referred to as the *stream extraction operator*. When this statement executes, the computer waits for the user to enter a value for variable `m`. The user responds by typing a real number then pressing the *Enter* key

(sometimes called the *Return* key). The built-in *input stream object* – `cin` – reads from the keyboard this number and assigns it to the variable `m`.

The assignment statement

```
y = 1.196 * m;
```

calculates the value of the expression, `1.196 * m`, and assigns the result to variable `y` using the *assignment operator* `=`. Most calculations are performed in assignment statements.

The line `return 0;`

is included at the end of every main function. The keyword `return` is used to *exit* a function. The value `0` indicates that the program has terminated successfully.

## Data Types and Variable Declarations

Variables are named locations for storing values. Each variable can store a single value of a particular data type. All variables must be declared with a *name* and a *data type* before they can be

used in a C++ function. A variable name is any valid *identifier*. An identifier is a series of characters consisting of letters, digits, and underscores ( `_` ) that does not begin with a digit. C++ is *case sensitive* – uppercase and lowercase letters are different, so `a1` and `A1` are different identifiers.

There are four *basic data types* in C++: `char` (for specifying *characters*), `int` (for specifying *integer* numbers), `float` (for specifying single-precision *floating-point* or *real* numbers), `double` (for specifying double-precision *floating-point* or *real* numbers), and `bool` (boolean).

### Data Type `char`

A variable of type `char` requires 8 *bits* of memory to hold one character. What characters are available and how they are represented internally depends on the machine on which the program runs. The most common character set is ASCII (*American Standard Code for Information Interchange*). In ASCII, each character is represented by a nonnegative integer code. For instance, 90 and 122 represents the characters Z and z, respectively. Character constants are specified within single

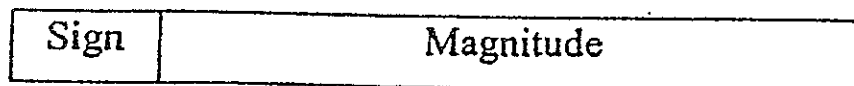
quotes, as in 'A', '9', 'z', or '+'. The value of a character constant is merely its integer code.

### Data Type int

A variable of type `int` holds an integer number. The number of bits required to hold an integer number depends on the *machine word size*. In addition to `int`, there are `short int` and `long int` types. Again, their sizes are machine dependent.

We can write `short int` and `long int` more concisely as `short` and `long`, respectively. On a 16-bit machine, C++ offers a 16-bit `short`, a 16-bit `int`, and 32-bit `long`. While, on a 32-bit computer, C++ offers a 16-bit `short`, a 32-bit `int`, and 64-bit `long`.

Integers can be either signed or unsigned. Signed integers use one bit for the sign of the number (where the bit is usually zero if it's positive and one if it's negative). A `signed int` (or just `int`, the default) uses one bit for the sign and 31 bits for the magnitude.





An unsigned int uses all the bits for the magnitude, doubling the range of positive values they can represent.

### Data Type float

Real or floating-point numbers are stored differently than integers. Internally, they are broken into a *fraction* and *exponent*, as shown below.

Sign	Exponent	Fraction
------	----------	----------

The number of bits for each of the exponent and fraction parts is machine dependent, but a typical representation for a 32-bit float uses 23 bits plus a sign bit for the fraction and 8 bits for the exponent.

Unlike integers, which are represented exactly, a floating-point value is represented approximately. The internal representation for floating point numbers can only support a fixed number of *significant* digits. A 32-bit float supports 6 significant digits, and a 64-bit double supports 10 significant digits. Therefore, we use floats when we want to save storage or avoid the overhead of double-precision operations. We use doubles when we want more significant digits and storage isn't important.

### Data Type bool

Boolean (bool) data type has only two values: 0(false) and 1(true). If a variable of type bool is assigned an integer value, C++ convert non-zero values into 1 and leave 0 as it is. For example, the following program:

```
bool b;  
b = 10;  
cout << b;
```

prints 1 on the screen.

### Arithmetic Operators

C++ provides the following arithmetic operators:

Operation	C++ operator	Expression
Addition	+	$x + y$
Subtraction	-	$x - y$
Multiplication	*	$x * y$
Division	/	$x / y$
Modulus	%	$x \% y$

*Integer division* (i.e., both the numerator and the denominator are integers) yields an integer result; for example, the expression  $7/4$  evaluates to 1, and the expression  $17/5$  evaluates to 3. Note that any fractional part in integer division is simply truncated – no rounding occurs.

The *modulus operator*, %, yields the remainder after integer division. The modulus operator is an integer operator that can be

used only with integer operands. Thus  $7\%4$  yields 3, and  $17\%5$  yields 2.

C++ applies the operators in arithmetic expressions in a sequence determined by the following rules of *operator precedence*.

Operators	Order of evaluation (precedence)
( )	<i>Evaluated first.</i> If the parentheses are nested, the expression in the innermost pair is evaluated first.
*, /, or %	<i>Evaluated second.</i> If there are several, they are evaluated from left to right.
+ or -	<i>Evaluated last.</i> If there are several, they are evaluated from left to right.

## Assignment Operators

In C++, an assignment not only assigns the value of an expression to a variable but also returns this value. One benefit is that we can use assignment operators more than once in a single statement. As an example, we can initialize more than one variable to some initial (say 0) with:

```
sum = i = 0;
```

The assignment operator associates *right to left*, so this multiple assignment is equivalent to:

```
sum = (i = 0);
```

## Relational Operators

We use *relational operators* to test whether a particular relationship holds between two values. C++ provides the following relational operators. A relational operator returns an int: 1 (*true*) if the relation holds, 0 (*false*) if it doesn't.

Operation	C++ operator	Expression
Equality	<code>==</code>	<code>x == y</code>
Inequality	<code>!=</code>	<code>x != y</code>
Greater than	<code>&gt;</code>	<code>x &gt; y</code>
Less than	<code>&lt;</code>	<code>x &lt; y</code>
Greater than or equal to	<code>&gt;=</code>	<code>x &gt;= y</code>
Less than or equal to	<code>&lt;=</code>	<code>x &lt;= y</code>

### Logical Operators

We often need to combine relational tests in a single expression. To do so we use *logical operators*. C++ provides the following logical operators:

Operation	C++ operator	Expression
Logical negation	<code>!</code>	<code>!x</code>
Logical AND	<code>&amp;&amp;</code>	<code>x &amp;&amp; y</code>
Logical OR	<code>  </code>	<code>x    y</code>

Like the relational operators, logical operators return an int, 1(true) or 0(false). An expression such as `x && y` will evaluate to

True only if both  $x$  and  $y$  are true. An expression such as  $x \parallel y$  will evaluate to true if either  $x$  or  $y$  is true. Logical negation returns true if its operand is false, and false otherwise.

### Shorthand Assignment Operators

C++ doesn't just provide a single assignment operator – it provides an entire collection. These operators have the form:

$$lhs\ op = rhs$$

where  $lhs$  is the left-hand side of the assignment,  $rhs$  is an expression, and  $op$  is any one of arithmetic or bitwise operators (so the shorthand operators are  $+=$ ,  $-=$ ,  $*=$ ,  $/=$ ,  $\%=$ ,  $>>=$ ,  $<<=$ ,  $\&=$ ,  $|=$ , and  $\wedge=$ ). The shorthand form is equivalent to

$$lhs = lhs\ op\ rhs$$

With the shorthand operators, for example, we can multiply  $p$  by  $x$  with:

$$p *= x;$$

which is equivalent to  $p = p * x$ ;

### Postfix and Prefix Assignment

C++ also provides two special shorthand operators for the common operations of *incrementing* and *decrementing* by 1:  $++$  adds 1 to its operand and  $--$  subtracts 1 from its operand. There

are two forms of these operators, *prefix* (preceding its operand) and *postfix* (following its operand).

The form doesn't matter, *so long as the operator and its operand are not part of a larger expression or statement*. By itself,  $i++$  or  $++i$  is equivalent to  $i += 1$ , and  $i--$  or  $--i$  is equivalent to  $i -= 1$ . However, if we used these forms as a piece of a more complicated expression or statement, then  $i++$  first provides the value of  $i$  to the rest of the expression or statement, and then adds 1 to  $i$ , while  $++i$  does the addition first and makes the new value of the variable available to the rest of the expression. Postfix and prefix  $--$  behave similarly to  $++$ , except that they decrement their operand.

The following program demonstrates the difference between the pre-incrementing version and the post-incrementing version of the  $++$  operator. The decrement operator ( $--$ ) works similarly.

```
// pre-incrementing and post-incrementing.
#include <iostream.h>

int main( )
{
    int c = 5;
    cout << c;
    cout << c++;
    cout << c << endl;
    c = 5;
    cout << c;
    cout << ++c;
    cout << c;
    return 0;
}
```

#### Output Window

```
5 5 6
5 6 6
```

The **endl** (an abbreviation for “*end line*”) in the above program output a new line then “flushes the output buffer”. This simply means that on some systems where outputs accumulate in the machine until there are enough to “make it worthwhile to print on the screen” **endl** forces any accumulated outputs to be printed at that moment.



### The Evaluation Order of Operators

If an expression has more than one operator, C++ evaluates it with the following order of evaluation of the operators

Operator type	Operation	Associativity
Prefix	++, --	
Unary	!	left to right
Arithmetic	*, /, %	left to right
	+, -	left to right
Relational	<, >, <=, >=	left to right
	=, !=	left to right
Logical	&&	left to right
		left to right
Assignment	=, +=, -=, *=, /=, %=	right to left
Postfix	++, --	

The following table shows some examples:

```
int x, y, z, w;
x = 1;
y = 2
z = 3;
w = 4;
```

Expression	Equivalent Expression	value
$x > y \parallel z \leq y$	$(x > y) \parallel (z \leq y)$	$(0) \parallel (1) \rightarrow 1$
$--x \parallel y \&\& z$	$(--x) \parallel (y \&\& z)$	$(0) \parallel (1) \rightarrow 1$
		x is decremented by 1
$++x + y > lz / w$	$((++x) + y) > ((lz) / w)$	$((2)+3)) > ((0)/4) \rightarrow 1$
		x is incremented by 1

Programming with C++

## Exercises

- ✓ [1] Write a program that asks the user to enter the radius of a circle and then computes and displays the circle's area and circumference. Use the formulas

$$\text{area} = \pi \times \text{radius} \times \text{radius}$$

$$\text{circumference} = 2 \times \pi \times \text{radius}$$

where  $\pi$  is the constant 3.14159.

- [2] Write a program to convert a temperature in degrees Fahrenheit to degrees Celsius. Use the formula

$$\text{Celsius} = (5/9) \times (\text{Fahrenheit} - 32)$$

- [3] Write a program that reads in the weight (in pounds) of an object and then computes and prints its weight in kilograms  
(Hint: 1 pound equals 0.453592 kilograms)

- ✓ [4] Write a program that reads in the length and width of a rectangular yard and the length and width of a rectangular house situated in the yard. Your program should compute the

time required to cut the grass at the rate of 2 square meters a second.

- [5] Write a program to print your income after taxes, assuming your gross salary is \$78,000, you have to pay 7.5% in social security on the first \$58,000 of income, your federal tax is \$3000 plus 28% of all income over \$30,000, and your state tax is 10% of your gross.
- [6] Write a program to compute the actual cost of a set of items, given a list price per item, a percentage discount, and a sales tax rate.

Here is its output when we used it to compute the actual cost of buying 23 Compact Disks (CDs), given that each CD is \$13.95, we get an 18% discount for buying more than 20, and the state sales tax is 7.5%:

List price per item:	13.950
List price of 23 items:	320.850
Price after 18% discount:	263.097
Sales tax after 7.5%:	19.732
Total cost:	282.829

[7] Write a program to compute and print the actual cost of a new car. Assume that the sticker price is \$89,950, that the dealer will discount the car price 12%, that there's a 10% luxury tax on the amount over \$30,000, and that the state sales tax is 8.75%.

[8] Write a program to calculate and print the amount invested in a saving account for a number of years. The amount on deposit at the end of the  $n$ th year is computed using the formula:

$$p(1 + r)^n$$

where  $p$  is the original amount invested, and  $r$  is the annual interest rate. Here's the output we want to see when we run your program.

Interest Rate:	6.0%
Starting Balance:	\$ 5000.00
Balance at the end of 7-year period:	\$ 7518.15

## Chapter 3

# Control Structures

## Chapter 3

### Control Structures

All programs could be written in terms of only three *control structures*, namely the *sequence structure*, the *selection structure*, and the *repetition structure*. The sequence structure is built into C++. Unless directed otherwise, the computer executes C++ statements one after the other in the order in which they are written. The programs given in chapter 2 illustrate typical sequence structures.

#### Selection Structures

A selection structure is used to choose among alternative courses of action. C++ provides three types of selection structures; we discuss each of these in this chapter. The **if** selection structure either performs (selects) an action if a condition is true or skips the action if the condition is false. The **if/else** selection structure performs an action if a condition is true and performs a different action if the condition is false. The **switch** selection structure is

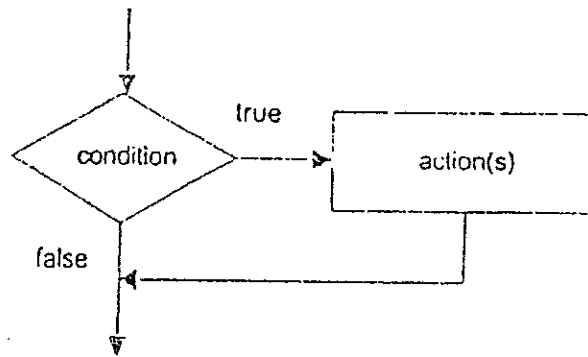
called a *multiple-selection structure* because it selects the action to perform from many different actions.

### Repetition Structures

A repetition structure allows the programmer to specify that an action to be repeated. C++ provides three types of repetition structures, namely *while*, *do/while* and *for*. The *while* repetition structures specify that an action to be repeated while some condition remains true. In the *while* structure, the condition is tested *before* the action is performed. The *do/while* repetition structure is similar to the *while* structure. However, in the *do/while* structure, the condition is tested *after* the action is performed. The *for* repetition structures specify that a counter variable controls the repetition of an action. All manipulations of the counter is specified in the *for* structure header.

### The if Selection Structure

The *if* selection structure either performs (selects) an action if a condition is true or skips the action if the condition is false. The following flowchart illustrates the *if* selection structure.



The syntax of the if selection structure is as follows:

```
if (condition)
    statement;
```

The effect is to execute the given statement only if the *condition* is true. If the condition is false, control flows to the next statement after the if structure.

Generally, in C++, statements fall into two categories:

1. *Simple statements*: One single statement terminated by a semicolon (;).
2. *Compound statements*: Zero or more statements grouped together by { and }.

A compound statement can be used anywhere a simple statement can.



- Example 1: The following if selection structure displays the message "Hi Mom" only when the character variable mom\_dad has the value 'M'.

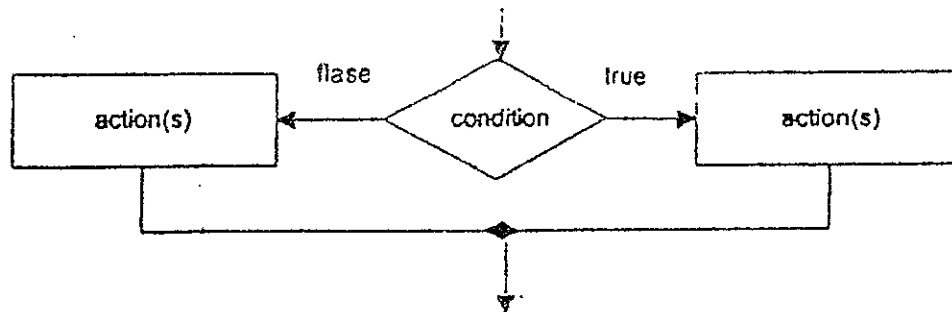
```
if (mom_dad == 'M')  
    Cout << "Hi Mom";
```

- Example 2: The following if selection structure rearranges any two values stored in x and y so that the smaller number will always be in x and the larger number will always be in y.

```
if (x > y) {  
    t = x;  
    x = y;  
    y = t; }
```

### The if/else Selection Structure

The if/else selection structure performs an action if a condition is true and performs a different action if the condition is false. The following flowchart illustrates the if/else selection structure.



The syntax of the if/else selection structure is as follows:

```

if (condition)
    statement-1;
else
    statement-2;
  
```

The condition is first tested to decide which one of the two given statements is to be executed. If the condition is true, then only *statement-1* is executed; otherwise, only *statement-2* is executed.

- Example 1: The following if/else selection structure displays either “Hi Mom” or “Hi Dad”, depending on the character stored in the variable `mom_dad`.

```

if (mom_dad == 'M')
    cout << "Hi Mom";
else cout << "Hi Dad";
  
```

- Example 2: The following if/else selection structure assigns to `m` the absolute value of `n`.

```
if (n > 0)
    m = n;
else m = -n;
```

- Example 3: The following if/else selection structure determines whether the integer value stored in *x* is even or odd.

```
if (x%2 == 0)
    cout << x << " is even";
else cout << x << " is odd";
```

### Programming Example: Finding the largest number.

Problem statement: Write a program that finds and prints the largest among three numbers.

#### Problem analysis:

#### Problem Input:

Three numbers (represented by the variables: *x*, *y*, *z*)

#### Problem Output:

The largest number (represented by the variable: *l*)

#### Program:

```
// This program finds and prints the largest among three numbers.
```

```

#include <iostream.h>

int main( )
{
    int x, y, z, l;
    cout << "Enter three numbers:\n";
    cin >> x >> y >> z;
    if (x > y)
        l = x;
    else
        l = y;
    if (z > l)
        l = z;
    cout << "The largest number = " << l;
    return 0;
}

```

#### Output Window

```

Enter three numbers:
12 20 8
The largest number = 20

```

Notice that the above program first stores the largest between the first two numbers (x and y) in the variable l. Then it compares the third number (z) with the value of l, and replaces the value of l with the value of z only if  $z > l$ .

Programming Example: Converting exam percentages into grades.

Problem statement: Write a program that will print grade A for exam percentage greater or equal to 90, B for scores in the range 80 to 89, C for scores in the range 70 to 79, D for grades in the range 60 to 69, and F for all other grades.

Problem analysis:

Problem Input:

An exam percentage (represented by the variable: *p*)

Problem Output:

The corresponding grade (represented by the variable: *g*)

Program (1):

```
// This program converts an exam percentage into the corresponding
// grade. It uses if selection structures.
```

```

#include <iostream.h>

int main( )
{
    int p; char g;
    cout << "Enter an exam percentage:\n";
    cin >> p;
    if (p >= 90)
        g = 'A';
    if (p < 90 && p >= 80)
        g = 'B';
    if (p < 80 && p >= 70)
        g = 'C';
    if (p < 70 && p >= 60)
        g = 'D';
    if (p < 60)
        g = 'F';
    cout << "The corresponding grade = " << g;
    return 0;
}

```

*Output Window*

Enter an exam percentage:  
72  
The corresponding grade = C

*Program (2):*

// This program converts an exam percentage into the corresponding  
// grade. It uses nested if/else selection structures.

```

#include <iostream.h>

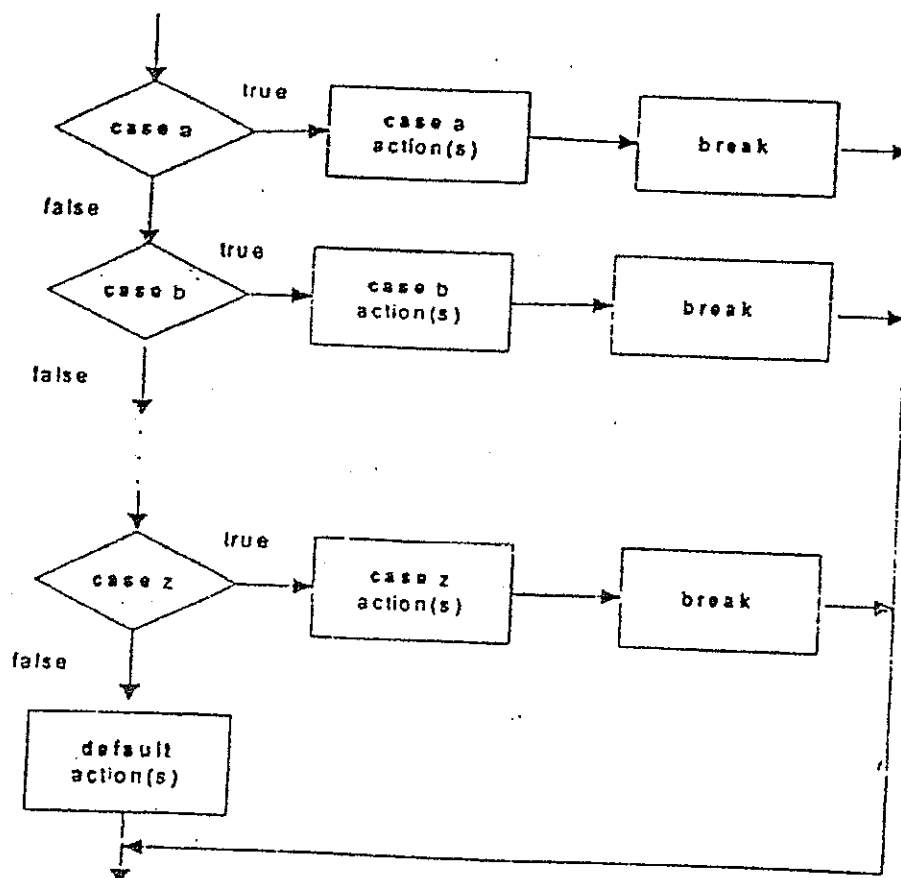
int main( )
{
    int p; char g;
    cout << "Enter an exam percentage:\n";
    cin >> p;
    if (p >= 90)
        g = 'A';
    else
        if (p >= 80)
            g = 'B';
        else
            if (p >= 70)
                g = 'C';
            else
                if (p >= 60)
                    g = 'D';
                else
                    g = 'F';
    cout << "The corresponding grade = " << g;
    return 0;
}

```

Notice that for any valid input exam percentage, only one condition of the five if selection structures, in program 1, will be true. However, in program 2, if the input exam percentage is greater than or equal 90, the four conditions will be true, but only the statement, `g = 'A';` after the first test will be executed. The else-part of the "outer" if/else selection structure will be skipped.

## The switch Selection Structure

The switch selection structure is called a *multiple-selection structure* because it selects the action to perform from many different actions. The switch structure consists of a series of case labels, and an optional default case. Each case can have one or more actions. The following flowchart illustrates the switch selection structure.





The syntax of the **switch** selection structure is as follows:

```
switch (expression)
{
    case constant1:
        statement(s)
        break;
    case constant2:
        statement(s)
        break;
    ...
    default:
        statement(s)
}
```

The *expression* is evaluated. If it matches any of the *constants* the *statement(s)* associated with that constant is performed. The **break** statement causes program control to proceed with the first statement after the **switch** structure. Note that unlike other control structures, it is not necessary to enclose a multi-statement **case** in braces.

The following program illustrates the use of the **switch** structure. Its input is a triplet containing a floating-point operand, a single-character operator (such as +, -, \*, or /), and another floating-point operand. The output of the program is the result of applying the operator to its operands.

## Programming Example: Simple Calculator.

Problem statement: Write a program that simulates a simple calculator.

Problem analysis:

Problem Input:

An operand (represented by the variable: *x*)

An operator (represented by the variable: *op*)

Another operand (represented by the variable: *y*)

Problem Output:

The result of applying the operand to its operands (represented by the variable: *r*)

Program:

// This program simulates a simple calculator.

```

#include <iostream.h>
#include <iomanip.h>

int main( )
{
    float x, y; char op;
    cout << "Enter an operand, an operator and another operand.\n";
    cin >> x >> op >> y;
    switch (op)
    {
        case '+': r = x + y;
                break;
        case '-': r = x - y;
                break;
        case '*': r = x * y;
                break;
        case '/': if ( y != 0.0)
r = x / y;
                else
                { cout << "Warning: division by zero.\n";
                  r = 0.0;
                }
                break;
        default: cout << "Unknown Operator: " << op << "\n";
                r = 0.0;
    }
    cout << "The result =" << setw(7) << r;
    return 0;
}

```

### Output Window

Enter an operand, an operator and another operand:

12.3 + 16.4

The result = 28.7

```
sum = 0; count = 0;
cin >> grade;
while (grade != -1) {
    sum += grade;
    count++;
    cin >> grade; }
```

### Programming Example: A Base Conversion Program.

Problem statement: Write a program that converts an integer value from base 10 to a user-selectable base between 2 and 10.

#### Problem analysis:

#### Problem Input:

A 10 base integer value (represented by the variable: *v*)

A user-selectable base (represented by the variable: *b*)

#### Problem Output:

The value of *v* in base *b*.

---

The syntax of the while repetition structure is as follows:

```
while (condition)  
    statement;
```

where the loop body is again a simple or compound statement. The effect is to evaluate the *condition* and, if the condition is true, executes the *statement*. This process is repeated until the condition is false.

- Example 1: The following while structure computes  $n$  factorial ( $n!$ ) for nonnegative integer  $n$ . Recall  $n! = n * (n-1) * \dots * 2 * 1$ . This is an example of the so-called *counter-controlled repetition*.

```
fact = 1;  
while (n > 1) {  
    fact = fact * n;  
    n = n - 1; }
```

- Example 2: The following while structure computes the sum and the count of a collection of grades. Because grades are normally nonnegative integers, -1 is an acceptable *flag value* to indicate "end of data entry". This is an example of the so-called *flag-controlled repetition*.

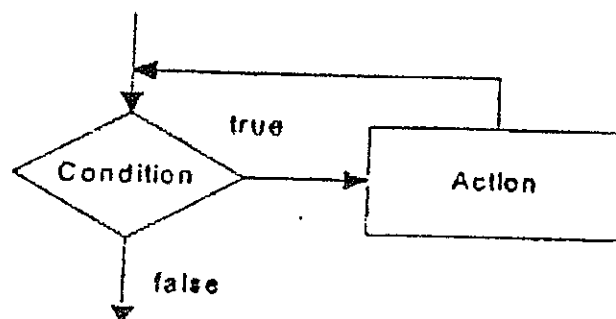
Programs that perform formatting tasks must contain the preprocessor directive:

```
#include <iomanip.h>
```

The call `setw(7)`, which is referred to as a *stream manipulator*, specifies that the next value to be output is printed in a *field width* of 7, i.e., the value is printed with at least 7 character positions. If the value to be output is less than 7 character positions wide, the value is *right justified* in the field. If the value to be output is more than 7 character positions wide, the field width is extended to accommodate the entire value.

### The while repetition Structure

The while repetition structures specify that an action to be repeated while some condition remains true. In the while structure, the condition is tested *before* the action is performed. The following flowchart illustrates the flow of control in the while repetition structure.



### Algorithm:

1. Set k to the number of digits the result will have.
2. (Hint:  $k = \lfloor \log_b v \rfloor + 1 = \lfloor \log_{10} v / \log_{10} b \rfloor + 1$ )
3. while  $k > 0$ ,
  - i) Display  $v/b^{k-1}$  (the most significant remaining digit)
  - ii) Set v to  $v \% b^{k-1}$  (the remaining part of the base 10 value)
  - iii) Subtract 1 from k (the number of digits we still have to display)

### Program:

// This program converts a base 10 value into a value in a specified base.

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

```
int main( )
{
    int v, b, k, d;
    cout << "Enter a value in base 10 >";
    cin >> v;
    cout << "Enter a base >";
    cin >> b;
    cout << v << " in base 10 is ";
    if ( v == 0)
        cout << "0";
```

```

else {
    k = floor(log10(v) / log10(b)) + 1;
    while (k > 0) {
        d = pow(b, k-1);
        cout << v/d;
        v = v%d;
        k = k - 1; }
    cout << " in base " << b;
    return 0;
}

```

### Output Window

```

Enter a value in base 10 > 175
Enter a base > 2
175 in base 10 is 10101111 in base 2

```

Notice that the math library `math.h` provides the `floor`, `log10`, and `pow` functions.

- `floor(x)`: finds the largest integer not  $> x$ .
- `log10(x)`: calculate the base 10 logarithm of  $x$ .
- `pow(b,n)`: computes the exponent  $b^n$ .

The problem with directly implementing the above algorithm is that it is inefficient: we have to compute an exponent each time through the loop. However, the efficiency of the above



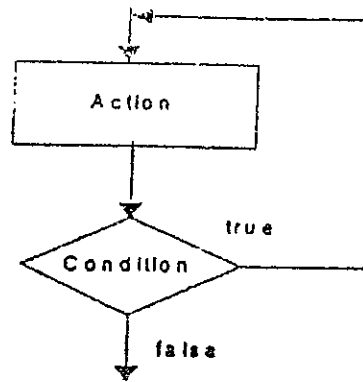
implementation can be improved by calculating  $b^{k-1}$  before we enter the loop.

Then instead of using  $k$  to index the loop, we can just use  $d$ , dividing it by the base each time and stopping once it reaches zero. Thus, the while structure, in the above program, should be replaced by:

```
d = pow(b, k-1);  
while (d > 0) {  
    cout << v/d;  
    v = v%d;  
    d = d/b; }
```

### The do/while repetition Structure

The do/while repetition structure is similar to the while structure. However, in the do/while structure, the condition is tested *after* the action is performed. The following flowchart illustrates the flow of control in the do/while repetition structure.



The syntax of the **do/while** repetition structure is as follows:

```
do  
    statement  
while (condition);
```

where the loop body is again a simple or compound statement. The effect is to execute the statement, and then evaluate the condition. This process is repeated until the condition is false.

Example: The following **do/while** structure prints the numbers from 1 to 10.

```
int n = 1;  
do  
    cout << n << " ";  
while (++n <= 10);
```

### The for Repetition Structure

The general format of the for repetition structure is

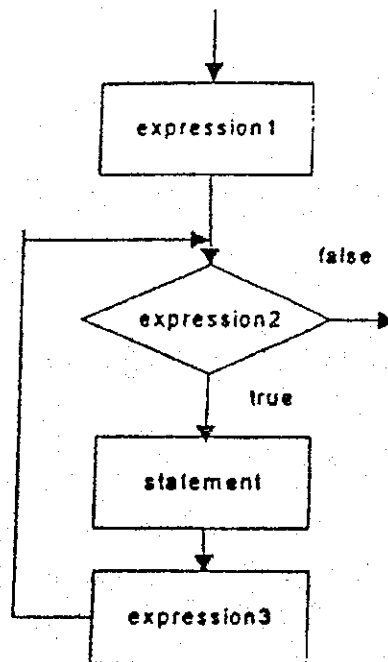
```
for ( expression1; expression2; expression3)  
    statement;
```

where *expression1* initializes the loop's control variable, *expression2* is the loop-continuation condition, and *expression3* increments the control variable. The loop body, *statement*, can be simple or a compound statement.

In most cases, the for structure can be represented with an equivalent while structure as follows:

```
expression1;  
while (expression2) {  
    statement  
    expression3;  
}
```

The following flowchart illustrates the for structure control flow.



Example 1: The following **for** structure computes the exponent  $x^n$ , where  $x$  and  $n$  are integers variables. The result is stored in an integer variable  $p$ .

```

for( int p=1, i=1; i<=n; i++)
    p = p * x;
  
```

Example 2: The following program computes and prints the amount invested in a saving account at the end of each year for 10 years. The amount on deposit at the end of the  $n$ th year is computed using the formula:

$$p(1+r)^n$$

where

$p$  is the original amount invested, and  
 $r$  is the annual interest rate.

```
// This program computes compound interest.
```

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

```
#include <iomanip.h>
```

```
#include <math.h>
```

```
int main( )
```

```
{
```

```
    double amount, p, r;
```

```
    cout << "Enter the original amount invested > ";
```

```
    cin >> p;
```

```
    cout << "Enter the annual interest rate > ";
```

```
    cin >> r;
```

```
    cout << "Year" << setw(21) << "Amount on deposit" << endl;
```

```
    for ( int year = 1; year <= 10; year++) {
```

```
        amount = p * pow(1.0 + r, year);
```

```
        cout << setw(4) << year
```

```
            << setiosflags( ios::fixed | ios::showpoint)
```

```
            << setw(21) << setprecision(2) << amount << endl;
```

```
    }
```

```
    return 0;
```

```
}
```

### Output Window

Enter the original amount invested > 1000

Enter the annual interest rate > 0.05

Year	Amount on deposit
------	-------------------

1	1050.00
---	---------

2	1102.50
---	---------

3	1157.62
---	---------

4	1215.51
---	---------

5	1276.28
---	---------

6	1340.10
---	---------

7	1407.10
---	---------

8	1477.46
---	---------

9	1551.33
---	---------

10	1628.89
----	---------

In the above program, the function `pow(x,y)` calculates the value of `x` raised to the `y`th power. The function `pow` takes two arguments of type `double` and returns a `double` value. The `math.h` file includes information that tells the compiler to convert the value of `year` to a temporary `double` representation before calling the function. This information is contained in `pow`'s *function prototype*.

The values of the variables `year` and `amount` are printed using the formatting specified by the parameterized stream manipulators `setw`, `setiosflags` and `setprecision`. The call `setw(4)` specifies that the next value output is printed in a *field width* of 4, i.e., the value is printed with at least 4 character positions. If the value to be output is less than 4 character positions wide, the value is *right justified* in the field by default. The call `setiosflags ios::left` can be used to specify that the values should be output *left justified*. If the value to be output is more than 4 character positions wide, the field width is extended to accommodate the entire value.

The call `setiosflags( ios::fixed | ios::showpoint )` indicates that variable `amount` is printed as a fixed-point value with a decimal

point. The call `setprecision( 2 )` specifies two digits of precision to the right of the decimal point.

### Nested Loops

Nested loops consist of an outer loop with one or more inner loops. Each time the outer loop is repeated, the inner loops are reentered and all required iterations are performed.

Example 1: The following nested loop draws the number triangle:

```
1
1 2
1 2 3
1 2 3 4
1 2 3 4 5
```

---

*Programming with C++*

```

    for (lineNumber = 1; lineNumber <= 5; lineNumber++)
    { for (i = 1; i <= lineNumber; i++)
        cout << i << " ";
        cout << endl;
    }

```

In the example, the outer loop is responsible to output the five lines of the triangle. The inner loop is responsible to output the sequence of numbers on a line.

Example 2: The following nested loop draws the triangle:

```

*
***
*****
*****
*****

```

```

    for (lineNumber = 1; lineNumber <= 5; lineNumber++)
    { for (i = 1; i <= 5 - lineNumber; i++)
        cout << " ";
        for (i = 1; i <= 2*lineNumber - 1; i++)
            cout << "*";
        cout << endl;
    }

```



In the above example, the outer loop is responsible to output the five lines of the triangle. The first inner loop prints the leading blank spaces; the second inner loop prints the required number of asterisks.

Example 3: The following code prints the addition table:

```
+ 1 2 3 4 5
1 2 3 4 5 6
2 3 4 5 6 7
3 4 5 6 7 8
4 5 6 7 8 9
5 6 7 8 9 10
```

```
cout << '+' ;
for (i = 1; i <= 5; i++)
    cout << setw(3) << i;
cout << endl;
for (i = 1; i <= 5; i++)
{
    cout << i;
    for (j = 1; j <= 5; j++)
        cout << setw(3) << i + j;
    cout << endl;
}
```

## Exercises

- [1] Write a program that reads a temperature and prints out what sport is appropriate for that temperature, using the following guidelines.

<i>Sport</i>	<i>Temperature</i>
Swimming	$\text{temp} > 85$
Tennis	$70 < \text{temp} \leq 85$
Golf	$32 < \text{temp} \leq 70$
Skiing	$\text{temp} \leq 32$

- [2] Write a program that reads a student number and the hours that the student has completed, and prints his or her classification. A student's classification is based on the following table:

<i>Classification</i>	<i>Hours completed</i>
Freshman	$\text{hours} < 30$
Sophomore	$30 \leq \text{hours} < 60$
Junior	$60 \leq \text{hours} < 90$
Senior	$90 \leq \text{hours}$

- [3] Write a program that reads someone's age, sex, and marital status, and prints his or her insurance rate. Below is the table that agents use for quoting rates.

<i>Sex</i>	<i>Age</i>	<i>Marital Status</i>	<i>Rate</i>
m	under 25	m	400
m	under 25	s	500
m	25 or over	m	100
m	25 or over	s	200
f	under 25	m	300
f	under 25	s	300
f	25 or over	m	100
f	25 or over	s	200

[4] Write a program to compute the roots of the quadratic equation  $ax^2 + bx + c = 0$ .

(*Hint*: if  $b^2 - 4ac$  (called the 'discriminant') is positive, two distinct real roots exist. If the discriminant is zero, two equal real roots exist. If the discriminant is negative, we have two complex roots.)

[5] Write a program that reads a person's birthdate and the current date, and prints the person's age in the form of three integers: *day, month, year*.

[6] Write a program to accept a date typed in the form

*day/month/year*

with the year specified by two digits, and to expand it by naming the month, quoting the full four digits of the year

and following the day by st, nd, rd, or th as appropriate. For example, 23/ 5/ 81 should appear as 23rd May 1981.

[7] A Telephone Company has the following rate structure for long-distance calls:

- \* Any call started after 8:00 p.m.(2000 hours) but before 7:00 a.m.(0700) receives a 50-percent discount.
- \* Any call started after 7:00 a.m.(0700 hours) but before 8:00 p.m.(2000 hours) is charged full price.
- \* All calls are subject to a 4-percent Federal tax.
- \* The regular rate for a call is \$0.40 per minute.
- \* Any call longer than 60 minutes receives a 15-percent discount (after any other discount is subtracted and before tax is added).

[8] Write a program that reads the start time for a call based on a twenty-four-hour clock and the length of the call. The gross cost (before any discounts or tax) should be printed followed by the net cost (after discounts are deducted and tax is added).

- [9] There are 9,870 people in a town whose population increases by 10 percent each year. Write a loop that determines how many years it will take for the population to go over 30,000.
- [10] Write a program segment that allows the user to enter values and prints out the number of positive and negative values entered. Use 0 as the sentinel value.
- [11] Write a program that finds the product of a collection of data values. Your program should ignore any negative data and should terminate when a zero value is read.
- ✓ [12] Write a loop that prints a table of equivalent Fahrenheit and Celsius temperature values. Use the formula

$$\text{Fahrenheit} = 1.8 \times \text{Celsius} + 32.0$$

to convert from degrees Celsius to Fahrenheit. Assume that the initial and final Celsius values are available in *InitCel* and *FinalCel*, respectively, and that the change in Celsius values between table entries is given by *StepCel*.

- [13] Write a program to read a collection of integer data items and find and print the index of the first occurrence and last

occurrence of the number 12. Your program should print index values of 0 if the number 12 is not found.

- [14] Write a program to read in a collection of exam scores ranging in value from 1 to 100. Your program should count and print the number of *outstanding* scores (90-100), the number of *satisfactory* scores (60-89), and the number of *unsatisfactory* scores (1-59).
- [15] Write a program to find the largest, smallest, and average value in a collection of N numbers, where the value of N is the first data item read.
- [16] Write a program that reads in a positive real number and finds and prints the number of digits to the left of the decimal point. *Hint:* Find the whole part and repeatedly divide by 10 until the number becomes less than 1.
- [17] Write a program which takes a positive value n and produces an inverted pyramid comprising n lines such that the value n-i+1 appears n-i+1 times in line i of the pyramid. For n = 5, the pyramid should have the following form:

5 5 5 5 5

4 4 4 4

3 3 3

2 2

1

- [18] Write a program which draws a diamond of the form illustrated below. The letter which is to appear at the widest point of the figure (E in the example) is specified as input data.

```
      A
    B - B
  C - - - C
D - - - - - D
E - - - - - - - E
  D - - - - - D
    C - - - C
    B - B
      A
```

- [19] Write a program to print all integers in the range 100 to 500 which have the property that the digits in the number are different. Print the results 4 numbers per line.

- [20] Write a program that prints the following diamond shape.

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

- [21] A palindrome is a number or a word that reads the same backwards as forwards. For example, each of the following five-digit integers are palindromes: 12321, 55555, 45554 and 11611. Write a program that reads in a five-digit integer and determines whether or not it is a palindrome.
- (*Hint:* Use the division and modulus operations to separate the number into its individual digits,)
- [22] Write a program that computes and presents the multiplication table from  $1 \times 1$  to  $n \times n$ . A suitable heading should be part of the output.
- [23] Write a program that prints the calendar for a particular month in a given year. The input parameters should be three integers: the given year, the number of the month and the number of the first days of the month (0 for Saturday, 1 for Sunday, and so on). Here is a sample output for a January that starts on Monday (i.e., the number of the month is 1 and the number of its first day is 2)

January						
Sat	Sun	Mon	Tue	Wed	Thu	Fri
		1	2	3	4	5
6	7	8	9	10	11	12
13	14	15	16	17	18	19
20	21	22	23	24	25	26
27	28	29	30	31		



[24] Write a program that counts the number of decimal digits in a given integer. The input parameter is an integer number.

## *Chapter 4*

### *Arrays*

# Chapter 4

## Arrays

In many situations, we need to group many data items of the same type into one group. The most basic structure in c++ that accomplishes that is the *array*. Array can hold a few data items or tens of thousands.

Data items in array are accessed by index since all of them have the same name of the array, but can have different values since each item has different location in the array.

Array can be considered as a list of items of the same data type.

### Array Declaration

Array is declared using subscript [] operator as the following example:

```
int age [4];
```

this means that *age* is an array of 4 integers data items. Figure 4.1 shows the array syntax.

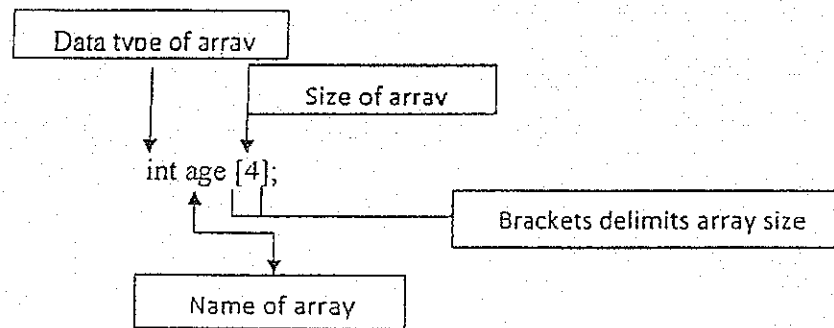


Figure 4.1 : Syntax of array definition

### Array Elements

The items in an array are called elements, the index of the first elements is 0 while the index of the last item is size-1. Figure 4.2 shows the elements of the array *age*.

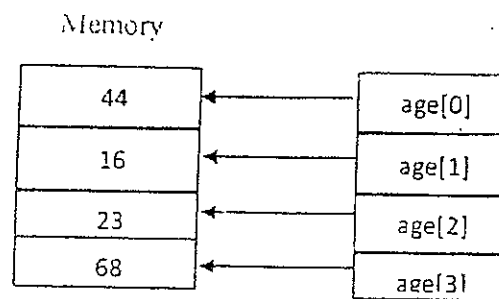


Figure 4.2: Array elements

Notice that first array element is numbered 0. Thus, since there are four elements, the last one is number 3. This is a potentially confusing situation; you might think the last element in a four-elements array would be number 4, but it's not.

**Example:**

```
#include <iostream.h>
int main()
{
    int age[4];
    for ( int j=0; j<4;j++) // get 4 ages
    {
        cout<< " Enter an age: ";
        cin>>age[j];      // access array element
        cout<<endl;       // display 4 ages
    }
    for (j=0;j<4;j++)
        cout<<" You entered " <<age[j]<<endl;
    return 0;
}
```

Here 's sample interaction with the program:

```
Enter an age: 44
Enter an age: 16
Enter an age: 23
Enter an age: 68
```

You entered 44

You entered 16

You entered 23

You entered 68

The first for loop gets the ages from the user and places them in the array, while the second reads from the array and displays them.

### Accessing Array Elements

In the above example, we access each array element twice. The first time, we insert a value into the array, with the line

```
cin>> age[j];
```

The second time, we read it out with the line

```
cout<< "You entered "<< age[i];
```

In both cases the expression for the array element is

```
age[j]
```

This consists of the name of the array, followed by brackets delimiting a variable *j*. Which of the four array elements is specified by this expression depends on the value *j*; `age[0]` refers to the first element, `age[1]` to the second element, `age[2]` to the third, and `age[3]` to the fourth. The variable (or constant) in the brackets is called the array index.

Since *j* is the loop variable in both for loops, it starts at 0 and is incremented until it reaches 3, thereby accessing each of the array elements in turn.

### Averaging Array Elements

Here's another example of an array at work. This one, `SALES`, invites the user to enter a series of six values representing widgets sales for each day of the week (excluding Sunday), and then calculates the average of these values. We use an array of type `double` so that monetary values can be entered.

```
// sales.cpp
```

```
// average a week's widgets sales (6 days)
```

```

#include <iostream.h>
int main()
{
    const int SIZE =6;           // size of array
    double sales[SIZE];         // array of 6 variable
    cout<<" Enter widget sales for 6 days\n";
    for (int j=0;j<SIZE;j++)    // puts figures in array
        cin>>sales[j];
    double total=0;
    for (j=0;j<SIZE;j++)        // read figures from array
        total+=sales[j];        // to find total
    double average=total/SIZE;   // to find average
    cout<<" Average = "<<average<<endl;
    return 0;
}

```

Here's some sample interaction with SALES:

Enter widget sales for 6 days

352.64

867.70

781.32

867.35

746.21

189.45

Average= 634.11

A new detail in this program is the use of a const variable for the array size and loop limits. This variable is defined at the start of the listing:

```
const int SIZE = 6;
```

Using a variable (instead of a number, such as the 4 used in the last example) makes it easier to change the array size: Only one program line needs to be changed to change

the array size, loop limits, and anywhere else the array size appears. The all-uppercase name reminds us that the variable cannot be modified in the program.

## Initializing Arrays

You can give values to each array element when the array is first defined. Here is an example. DAYS, that sets 12 array elements in the array `days_per_month` to the number of days in each month.

```
// days.cpp
// shows days from start of year to date specified
include <iostream.h>
int main()
{
    int month, day, total_days;
    int days_per_month[12]= { 31,28,31,30,31,30,31,31,30,31,30,31};
    cout<< "\n Enter month (1 to 12) : "; // get date
    cin>>month ;
    cout<< "\n Enter day (1 to 31) : ";
    cin>> day ;
    total_days=days;    //separate days
    for (int j=0; j<month-1; j++) // add days each month
        total_days += days_per_month[j];
    cout<<" Total days from start of year is: "<< total_days<<endl;
    return 0;
}
```

The program calculates the number of days from the beginning of the year to date specified by the user. (Beware: It doesn't work for leap years.) Here's some sample interaction:

Enter month (1 to 12): 3

Enter day (1 to 31): 11

Total days from start of year is : 70

Once it gets the month and day values, the program first assigns the day value to the `total_days` variable. Then it cycles through a loop, where it adds values from the `days_per_month` array to `total_days`. The number of such values to add is one less

than the number of months. For instance, if the user enters month 5, the values of the first four array elements (31, 28, 31, and 30) are added to the total.

The values to which `days_per_month` is initialized are surrounded by braces and separated by commas. They are connected to the array expression by an equal sign.

Figure 4.3 shows the syntax.

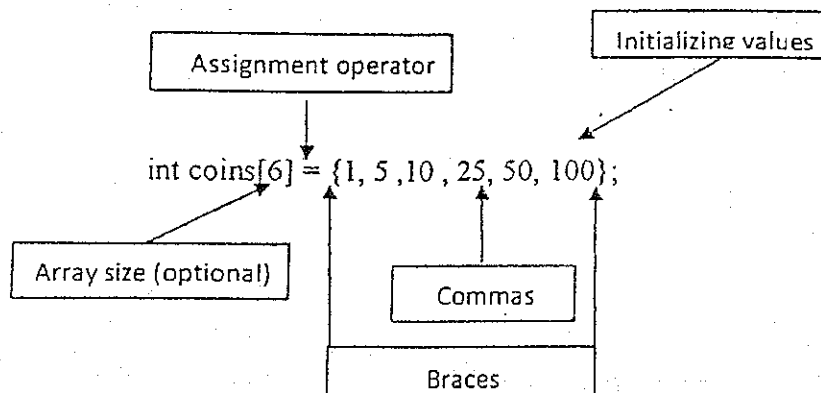


Figure 4.3 : Syntax of array initialization

Actually, we don't need to use the array size when we initialize all the array elements, since the compiler can figure it out by counting the initializing variables. Thus we can write `int day_per_month[] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};`

What happens if you do use an explicit array size, but it doesn't agree with the number of initializers? If there are too few initializers, the missing elements will be set to 0. If there are too many, an error is signaled.

## Multidimensional Array

so far we have looked at arrays of one dimension: A single variable specifies each array element. But arrays can have higher dimensions. Here's a program, SALEMONT, that uses a two dimensional array to store sales figures for several districts and several months:

```
// salemont.cpp
// display sales chart using 2.d array
#include <iostream.h>
#include <iomanip.h>
const int DISTRICTS = 4;
const int MONTHS = 3;
```



```

int main()
{
    int d, m;
    double sales[DISTRICTS][MONTHS]; // two-dimensional array definition
    cout<<endl;
    for (int d=0; d< DISTRICTS; d++)
        for (int m=0; m<MONTHS; m++)
            {
                cout<< " Enter sales for district "<<d+1;
                cout<<" , month "<<m+1<<": ";
                cin>>slaes[d][m];           //put numbers in array
            }
    cout<<"\n\n";
    cout<<"                               Month \n";
    for (int d=0; d< DISTRICTS; d++)
        {
            cout<<"\n District "<<d+1;
            for (int m=0; m<MONTHS; m++) //display array value
                cout<< setiosflags(ios::fixed) // not exponential
                    << setiosflags(ios::showpoint) // always use point
                    << setprecision(2)           // digits to right
                    << setw(10)                  //field width
                    << sales[d][m];              // get number from array
        } // end for (d)
    cout<<endl;
    return 0;
} // end main

```

This program accepts the sales figures from the user and then display them in a table.

Enter sales for district 1, month 1: 3964.23

Enter sales for district 1, month 2: 4135.87

Enter sales for district 1, month 3: 4397.98

Enter sales for district 2, month 1: 867.75

Enter sales for district 2, month 2: 923.59

Enter sales for district 2, month 2: 1037.01

Enter sales for district 3, month 1: 12.77  
 Enter sales for district 3, month 2: 378.32  
 Enter sales for district 3, month 3: 798.22  
 Enter sales for district 4, month 1: 2983.53  
 Enter sales for district 4, month 2: 3983.73  
 Enter sales for district 4, month 3: 9494.98

	Month		
	1	2	3
District 1	3964.23	4135.87	4397.98
District 2	867.75	923.59	1073.01
District 3	12.77	378.32	798.22
District 4	2983.53	3983.73	9494.98

## Defining Multidimensional Arrays

The array is defined with two size specifiers, each enclosed in brackets:

```
double sales[DISTRICTS][MONTHS];
```

You can think about sales as a two-dimensional array, laid out like a checkerboard. Another way to think about it is that sales is an array of arrays. It is an array of DISTRICTS elements, each of which is an array of MONTHS elements. Figure 4.4 shows how this looks.

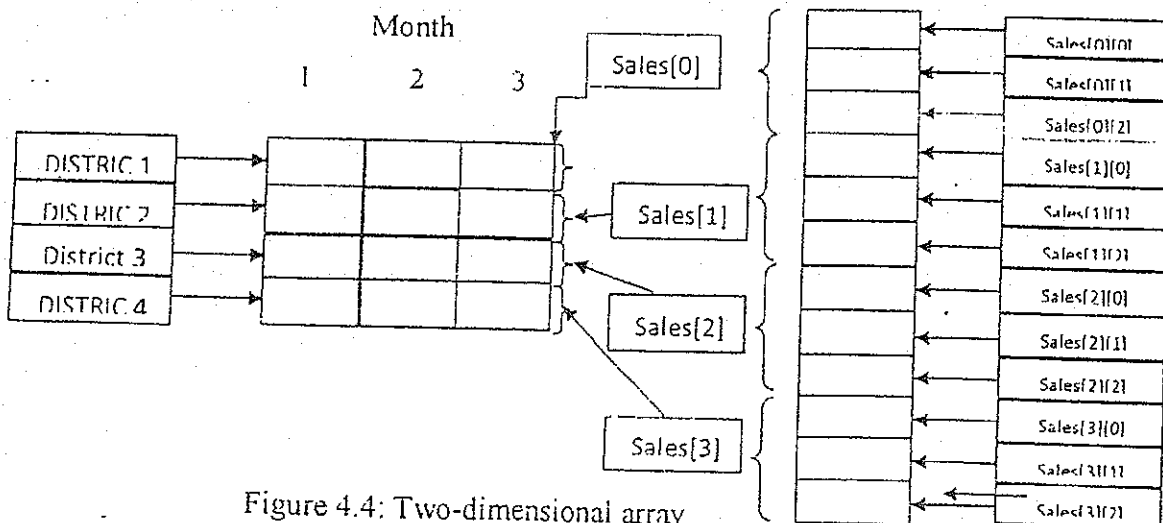


Figure 4.4: Two-dimensional array

Of course there can be arrays of more than two dimensional. A three-dimensional array is an array of arrays of arrays. It is accessed with three indexes:

```
elem = dimen3[x][y][z];
```

This is entirely analogous to one-and two-dimensional arrays.

### Accessing Multidimensional Array Elements

Array elements in two-dimensional arrays require two indexes:

```
sales[d][m]
```

Notice that each index has its own set of brackets. Commas are not used. Don't write `sales[d, m]`; this works in some language, but not in C++.

### Formatting Numbers

The SALEMONT program displays a table of dollar values. It's important that such values be formatted properly, so let's digress to see how this is done in C++. With dollar values you normally want to have exactly two digits to the right of the decimal point, and you want the decimal points of all the numbers in a column to line up. It's also nice if trailing zeros are displayed; you want 79.50 not 79.5.

Convincing the C++ I/O streams to do all this requires a little work. You have already seen the manipulator `setw()`, used to set the output field width. Formatting decimal numbers requires several additional manipulators.

Here's a statement that prints a floating-point number called `fpn` in a field 10 characters wide, with two digits to the right of the decimal point:

```
cout<< setiosflags (ios::fixed)    // fixed (not exponential)
      << setiosflags (ios::showpoint) // always show decimal point
      << setprecision (2)           // two decimal places
      << setw(10)                   // field width 10
      << fpn;                       // finally, the number
```

A group of one-bit formatting flags in a long int in the ios class determines how formatting will be carried out. At this point we don't need to know what the ios class is, or the reasons for the exact syntax used with this class, to make the manipulators work.

We are concerned with two of the ios flags: fixed and showpoint. To set the flags, use the manipulator `setiosflags`, with the name of the flag as an argument. The name must be preceded by the class name, ios, and the scope resolution operator (`::`).

The first two lines of the `cout` statement set the ios flags. (If you need to unset--that is, clear--the flags at some later point in your program, you can use the `resetiosflags` manipulator.) The fixed flag prevents number from being printed in exponential format, such as `3.45e3`. The showpoint flag specifies that there will always be a decimal point, even if the number has no fractional part: `123.00` instead of `123`.

To set the precision to two digits to the right of the decimal place, use the `setprecision` manipulator, with the number of digits as an argument. We have already seen how to set the field width by using the `setw` manipulator. Once all these manipulators have been sent to `cout`, you can send the number itself; it will be displayed in the desired format.

## Initializing Multidimensional Arrays

As you might expect, you can initialize multidimensional arrays. The only prerequisite is a willingness to type a lot of braces and commas. Here's a variation of the SALEMON program that uses an initialized array instead of asking for input from the user. This program is called SALEINIT.

```
// saleinit.cpp
// displays sales chart, initializes 2-d array
#include <iostream.h>
#include <iomanip.h>
const int DISTRICTS = 4; // array dimensions
const int MONTHS = 3;
int main()
```

```

{
    int d, m ;           // initialize array elements
    double sales[DISTRICTS][MONTHS] = { { 1432.07, 234.50, 654.01},
                                           { 322.00, 13838.32, 17589.88},
                                           {9328.34, 934.00, 449.30},
                                           {12838.29, 2332.63, 32.93}};

    cout<<"\n\n";
    cout<<"                               Month\n";
    for (int d=0; d< DISTRICTS; d++)
    {
        cout<<"\n District "<<d+1;
        for (int m=0; m<MONTHS; m++)
            cout<< setw(10) <<setiosflags(ios::fixed) << setiosflags(ios::showpoint)
                << setprecision(2) <<sales[d][m]; // access array element
    } // end for (d)
    cout<<endl;
    return 0;
}

```

Remember that two-dimensional array is really an array of arrays. The format for initializing such an array is based on this fact. The initializing values for each subarray are enclosed in braces and separated by commas {1432.07, 234.50, 654.01, } and then all four of these subarrays, each of which is an element in main array, is likewise enclosed by braces and separated by commas, as can be seen in the listing.

As with other data types, strings can be variables or constants. We will look at these two entities before going on to examine more complex string operations. Here's an example that defines a single string variable. It asks the user to enter a string, and places this string in the string variable. Then it displays the string. Here's the listing for STRININ:

```

#include <iostream.h>
int main()
{
    const int MAX= 80;           //max characters in string
    char str [MAX] ;             // string variable str
    cout<<" Enter a string: ";
    cin>> str;                   //put string in str
    cout<<"You entered: "<<str<<endl; //display string from str
    return 0;
}

```

The definition of the string variable `str` looks like (and is) the definition of an array of type `char`: `char str [MAX];`

We use the extraction operator `>>` to read a string from the keyboard and place it in the string variable `str`. This operator knows how to deal with strings; it understands that they are arrays of characters. If the user enters the string "Amanuensis" (one employed to copy manuscripts) in this program, the array `str` will look something like Figure 4.5. Each character occupies 1 byte of memory. An important aspect of C-strings is that they must terminate with a byte containing 0. This is often represented by the character constant `'\0'`, which is a character with an ASCII value of 0. This terminating zero is called the *null character*. When the `<<` operator displays the string, it displays characters until it encounters the null character.

### String Constants

You can initialize a string to a constant value when you define it. Here's an example, `STRINIT`, that does just that (with the first line of a Shakespearean sonnet):

```

// strinit.cpp
// initialized string
#include <iostream.h>
int main()
{
    char str[] = "Farewell! thou art too dear for my possessing. ";
    cout<< str<<endl;
}

```

```
return 0;
```

```
}
```

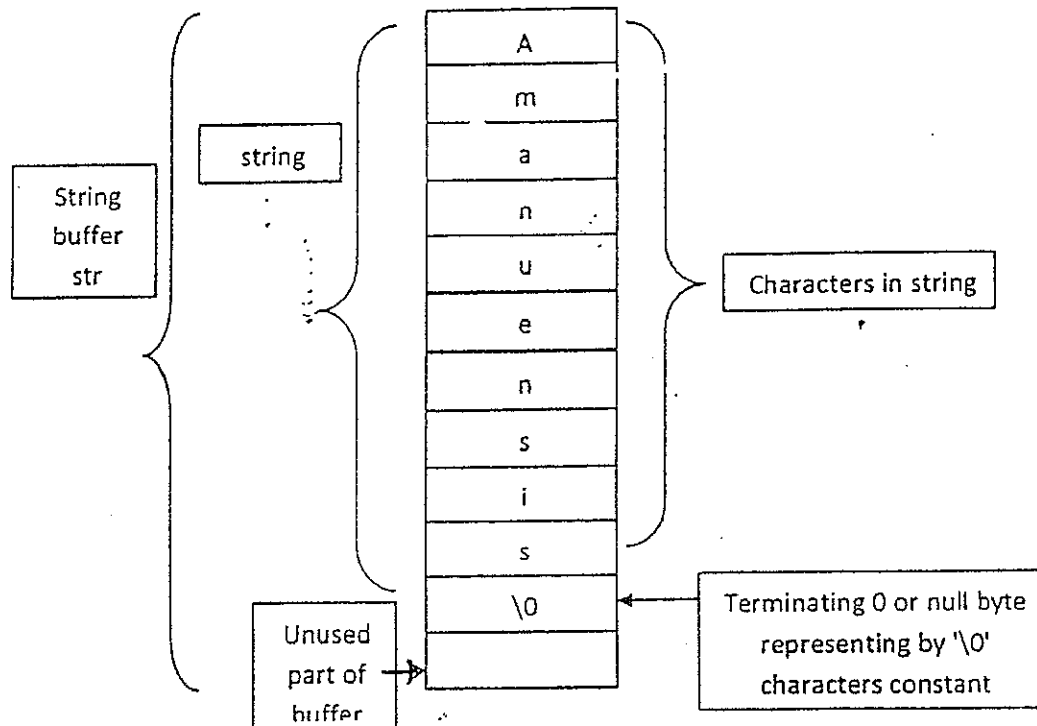


Figure 4.5: String stored in string variable

Here the string constant is written as a normal English phrase, delimited by quotes. This may seem surprising, since a string is an array of type `char`. In past examples you have seen arrays initialized to a series of values delimited by braces and separated by commas. Why is not `str` initialized the same way? In fact you could use such a sequence of character constants:

```
char str[] = {'F', 'a', 'r', 'e', 'w', 'e', 'l', 'l', 'l', 't', 'h', .....};
```

and so on. Fortunately, the designers of C++ (and C) took pity on us and provided the shortcut approach shown in `STRINIT`. The effect is the same: The characters are placed one after the other in the array. As with all C-strings, the last character is a null (zero).

## Copying a String

The best way to understand the true nature of strings is to deal with them character by character. The following program does this.

```

//strcpy1.cpp
//copies a string using a for loop
#include <iostream.h>
Int main()
{
    char str1[ ] = "Oh, Captain, my Captain! "
                  "our fearful trip is done";           // initialized string
    const on MAX=80;                                     // size of str2 buffer
    char str2[MAX];                                       //empty string
    int j=0;
    while (str1[j] != '\0')
    {
        str2[j]=str1[j];
        j++;
    }
    str2[j]='\0';                                         // insert NULL at the end
    cout<<str2<<endl;
    return 0;
}

```

This program creates a string constant, str1, and a string variable, str2. It then uses a for loop to copy the string constant to the string variable. The copying is done one character at a time, in the statement

```
str2[j]=str1[j];
```

Recall that the compiler concatenates two adjacent string constants into a single one, which allow us to write the quotation on two lines. This program also introduces C-string library functions. Because there are no string operators built into C++, C-string must usually be manipulated using library functions. Fortunately there are many such functions. The one we use in this program, strlen(), finds the length of a C-string (That is, how many characters are in it). We use this length as the limit in the for loop so that the right number of characters will be copied. When string functions are used, the header file CSTRING.H must be included with #include in the program. The copied version of the string must be terminated with a null. However the string length returned by strlen() does not include the null. We could copy one additional character, but it's safer to insert the null explicitly. We do this with the line

```
str2[j]='\0';
```



## Arrays of Strings

If there are arrays of arrays, of course there can be arrays of strings. This is actually quite a useful construction. Here's an example, STRARRAY, that puts the names of the days of the week in an array:

```
// straray.cpp
// array of strings
# include <iostream.h>

int main()
{
    const int DAYS = 7;           //number of strings in array
    const int MAX = 10;          // maximum size of each string
    char star[DAYS][MAX]={"Sunday", "Monday", "Tuesday", "Wednesday ",
                           " Thursday", "Friday", "Saturday"};

    for (int j=0; j<DAYS; j++)
        cout<<star[j]<<endl;      // display every string
    return 0;
}
```

The program prints out each string from the array:

Sunday

Monday

Tuesday

Wednesday

Thursday

Friday

Saturday

Since a string is an array, it must be true that star—an array of strings—is really a two-dimensional array. The first dimension of this array, DAYS, tells how many strings are in the array. The second dimension, MAX, specifies the maximum length of the strings (9 characters for "Wednesday" plus the terminating null makes 10). Figure 4.6 shows how this looks

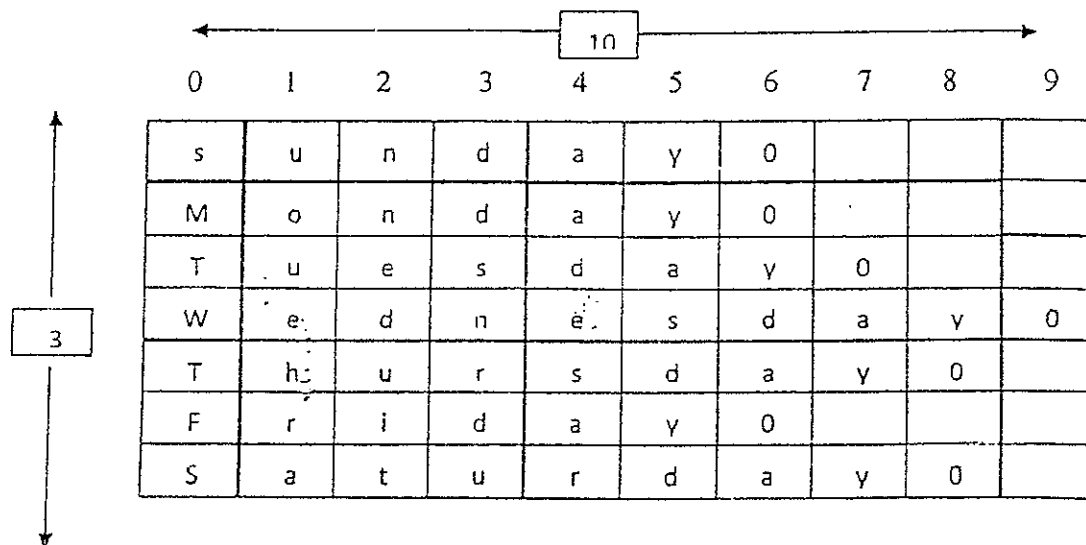


Figure 4.6: Array of strings

Notice that some bytes are wasted following strings that are less than the maximum length.

The syntax for accessing a particular string may look surprising:

```
star[j];
```

If we are dealing with a two-dimensional array, where's the second index? Since a two-dimensional array is an array of arrays, we can access elements of the "outer" array, each of which is an array (in this case a string), individually. To do this we don't need the second index. So `star[j]` is string number `j` in the array of strings.

## Exercises

1. Write a program that finds the max, min number in array of integers.
2. Write a program that reverses a string
3. Write a program copies a string into another string but in a reverse order. For example if the first string is "Welcome", the second string should be "emocleW".
4. Write a C++ program to get two strings from the user and compare them to show which one comes first in alphabetical order, e.g. "ahmed" comes before "ashraf" which comes before "baher" and so on. Note that "fayz" comes before "fayza". The program then tells if the two strings are the same, the first string comes first or the second string comes first.

