```
            num2 += num1;
            num1 = num2 - num1;
        }
        return 0;
}
```

Output

```
1
1
2
3       As you can see, each number in the series is the
4       sum of the two preceding numbers.
5
8
13
21
34
55
89
```

2.3.3 do-while loop

Sometimes, however, it is desirable to have a loop with the test for continuation at the end of each pass. This can be accomplished by means of the *do-while* statement.

The general form of the do-while statement is:

```
do
statement
while (expression);
```

The statement will be executed repeatedly, as long as the value of expression is true (i.e., is nonzero). Notice that the statement will always be executed at least once, since the test for repetition does not occur until the end of the first pass through the loop. The statement can be either simple or compound, though most applications will require it to be a compound statement. It must include some feature that eventually alters the value of expression so the looping action can terminate.

Here is another example to do the same thing, using the to display the consecution digits (0, 1, 1, 2, 9) using the *do-while* loop.

```
#include<iostream.h>
main()              //display the integers 0 through 9 //
{
      int digit = 0;
      do {
      cout<<digit;
      digit;
      }while (digit <= 9);
}
```

The *do-while* loop displays the current value of digit, increases its value by 1, and then tests to see if the current value of digit exceeds 9. If so, the loop terminates, otherwise, the loop continues, using the new value of digit. Note that the test is carried out at the end of each pass through the loop. The net effect is that the loop will be repeated 10 times, resulting in 10 successive lines of output.

This programme below accepts characters from keyboard until the character, 'I', is entered and displays whether the total number of consonant characters entered is more than, less than, or equal to the total number of vowels entered.

```
#include<iostream.h>
int main( )
{
      int constant, vowel;
      char inp;
      consonant = vowel 0;
      do
      {
            inp = `;
            cout<<endl.<<"Enter a character ( ! to quit )";
            cin>>inp;
            switch(inp)
            {
                  case `A`    :
                  case `a`    :
                  case `E`    :
                  case `e`    :
                  case `I`    :
                  case `i` :
                  case `O`    :
                  case `o`    :
                  case `U`    :
                  case `u`    :      vowel  = vowel +  1;
                  break;
                  case ` !`   :
```

```
                break;
                default      :      consonant = consonant + 1;
            }
        } while ( inp ! = `!`)
        if ( constant > vowel )
            cout<<endl<<" Consonant count greater than vowel count" ;
        elseif ( consonant < vowel )
            cout<<endl<"Vowel count greater than consonant count";
        else
             cout<<endl<<"Vowel ! and consonant counts are equal ";
        return 0;
}
```

2.4 BREAKING CONTROL STATEMENTS

The breaking control statements (or jump statements) unconditionally transfer programme control within a function.

2.4.1 break Statement

The break statement causes the programme flow to exit the body of the while loop. The following programme code illustrates the use of the break statement.

```
#include<iostream.h>
int main()
{
        int num1 = 1, num2 = 1;
        cout << num1 << endl;
        while (num2 < 150)
        {
                cout << num2 << endl;
                num2 += num1;
                num1 = num2 - num1;
                if (num2 == 89)
                break
        }
        return 0;
}
```

The output of the above programme is:

```
1
1
2
```

```
3
5
8
13
21
34
55
```

The control exits the loop when the condition, num2 == 89, becomes true.

2.4.2 continue Statement

The continue is another jump statement somewhat like the break statement as both the statements skip over a part of the code. But the continue statement is little different from break. Instead of forcing termination, it forces the next iteration of the loop to take place, skipping any code in between. The following figure explains the working of continue statement.

```
while (expression)
{
      statement 1;
      if (condition)
      continue;
      statement 2:
      }
      Statement 3;
```

```
for (init; test expression; update)
       {
       statement 1;
if (condition)
              continue;
       statement 2;
        }
       statement 3;
```

```
do
      {statement 1;
      if (condition)
             continue;
      statement 2;
      }
      while (expression);
      statement3;
```

For the for loop, continue causes the next iteration by updating the variable and then causing the test-expression's evaluation. For the while and do-while loops, the programme control passes to the conditional test.

2.4.3 goto Statement

A goto statement can transfer the programme control anywhere in the programme. The target destination of a goto statement is marked by a label. The target label and goto must appear in the same function.

The syntax of goto statement is:

```
goto label;
.
.
.
label:
```

where label is a user supplied identifier and can appear before or after goto. For example,

The following code fragment,

```
a=0;
start:
      cout << "\n" << ++ a;
if (a<50) goto start;
```

prints numbers from 1 to 50. The cout prints the value of + +a and then checks if a is less than 50, the control is transferred to the label start:, otherwise the control is transferred the statement following if.

Check Your Progress

Fill in the blanks:

1. The .... statement is used to carry out a logical test and then take on of two possible actions.
2. The .... is another jump statement somewhat like the break statement as both the statements skip over a part of the code.
3. The .... loop displays the current value of digit.
4. The loop statements allow a set of instructions to be .... executed until a certain condition is fulfilled.

2.5 LET US SUM UP

Statement are the instructions given to the computer to perform any kind of action. The simplest statement is null statement (;) . C+ + provides two types of selection statements: if and switch.

The if-else statement tests an expression and depending upon its truth value one of the two sets-of-actions is executed. The **if – else** statement can have another if statement.

C+ + provides one more selection statement know as switch that tests a value against a set of integer constants (that includes characters also). A switch statement can be nested also.

C++ provides three loops: for, while and do-while. The for loop gathers all its loop control elements on the top of the loop. A for loop can have multiple initializations and update expressions separated by commas. The loop control elements in a for loop are optional. The while loop evaluates a test-expression before allowing entry into the loop. The do-while is executed at least once always as it evaluates the test expression at the end of the loop.

A goto statement transfers the programme control anywhere in the programme. A break statement can appear in any of the loops and a switch statement. Whichever statement it appears in, it causes the termination of the statement there and then and the control passes over to the statement following the loop containing break.

The continue statement abandons the current iteration of the loop by skipping over the rest of the statements in the loop-body. It immediately transfers control to the evaluation of the test expression of the loop for the next iteration of the loop.

2.6 KEYWORDS

Infinite loop: A loop that never ends.

Nested loop: A loop that contains another loop inside its body.

Statement: Instruction given to the computer to perform any kind of action.

Loop: A group of instructions the computer executes repeatedly until some terminating condition is satisfied.

2.7 QUESTIONS FOR DISCUSSION

1. What is iteration construct?
2. What is the purpose of break statement?
3. What is conditional statement?
4. Can if statement be nested?
5. What is the purpose of for loop?
6. What is nested if statement?
7. Can a switch statement be nested?
8. What is abnormal exit?
9. Which is the best loop in C++?
10. Find the syntax error (s), if any, in the following programme:

```
#include<iostream.h)
void main()
{
      int x, y;
      cin>>x;
```

```
            for (y = 0; y< 10, Y++)
            if x = = y
                  cout << Y+X;
            else
                  cout>>Y;
```

11. Given the following for loop:

```
    .
    .
    cout int sz= 25;
    for (int i = 0, sum = 0; i< sz; i++)
    sum + = i;
    cout < < sum;
```

Write the equivalent while loop for the above code.

12. Write a C++ programme to print fibonacci series i.e. 0, 1, 1, 2, 3, 5, 8—

Check Your Progress: Model Answer

1. if-else
2. Continue
3. do-while
4. repeatedly

2.8 SUGGESTED READINGS

Robert Lafore, *Object-oriented Programming in Turbo* C++, Galgotia Publications.

E Balagurusamy, *Object-Oriented Programming with* C++, Tata Mc Graw-Hill

Herbert Schildt, *The Complete Reference* C++, Tata Mc Graw Hill

UNIT II

LESSON

3

FUNCTIONAL AND PROGRAMME STRUCTURES

CONTENTS

3.0 AIMS AND OBJECTIVES

After studying this lesson, you will be able to:

- Explain the concept of defining a function
- Discuss the types of functions
- Describe the significance actual and formal arguments
- Identify and explain the local and global variables
- Discuss the default arguments
- Explain the multifunction programme
- Identify storage class specifiers
- Explain the concept of recursive function
- Discuss the preprocessors
- Define the header files
- Identify the standard functions
- Explain the array and functions
- Discuss the multidimensional arrays

3.1 INTRODUCTION

Those who are familiar with language C would agree that writing a C programme is nothing more than writing C functions (including the main function). C++ on the other hand is all about writing codes for defining and manipulating classes and objects.

Conceptually an object may have only data members specifying its attributes. However, such an object would serve no useful purpose. For the purpose of establishing communication with the object it is necessary that the object provide methods, which are C like functions. Though C++ functions are very similar to C functions, yet they differ significantly as you will discover in this lesson.

Arrays are data structure, which hold multiple variables of same data type. An array is a sequence of objects all of which have the same type. The objects are called elements of array and are numbered consecutively 0, 1, 2, ... These numbers are called index values or subscripts of the array. The term 'subscript' is used because as a mathematical sequence, an array would be written with subscripts; n_0, n_1, n_2, ... These numbers locate the element's position within the array, thereby giving direct access into the array. In this lesson we will discuss various types of arrays, their declaration and memory allocation.

3.2 DEFINING A FUNCTION

A functions is a named unit of a group of programme statements. This unit can be invoked from other parts of the programme.

In C++, a function must be defined before it is used anywhere in the programme. The general form of a function definition is as given below:

Type function-name (parameter list)

{body of the functions}

Where the type specifies the type of value that the return statement of the function returns. If no type is specified, the compiler assumes that the functions returns an integer value. The parameter list is a comma-separated list of variables of a function referred to as its arguments. A function may be without any parameters, in which case, the parameter list is empty.

E.g: int absval (int a)

```
{ return (a<0? - a : a);
}
int get (int n1, int n2 ) // returns the grc.d.
{       int temp;
        while (n2)
{       temp = n2;
        n2= n1 % n2;
        return n1;
}
```

A function definition must have a return statement.

Function Prototyping

Prototype of a function is the function without its body. The C++ compiler needs to about a function before you call it, and you can let the compiler know about a function is two ways-by defining it before you call it or by specifying the function prototypes before you call it. Many programmers prefer a 'Top-Down' approach in which main() appears ahead the user-defined function definition. In such approach, function access will precede function definition. To overcome this, we use the function prototypes or we declare the function. Function-prototype are usually written at the beginning of a programme, ahead of any programmer-defined functions including main(). In general, function prototype is written as:

Return-type name (type 1 arg. 1, type 2 arg. 2, ... type n arg. n);

Where return-type represents the data type of the value returned by the function, name represents the function name, type 1, type 2 .... type n represents the data-type of arguments art. 1, arg. 2 ... arg n. the function prototypes resemble first line to function definition, though it ends with the semi-colon. Within the function declaration, the names of arguments are optional but data-types are necessary for eq. Int count (int);

The code snippet given below will cause compilation error because the main function does not know about the called function one().

```
void main()
{
      ...
      one();      //incorrect!!No function prototype available for one()
      ...
}
void one()
{
            //function definition
}
```

To resolve this problem you should define the function before it is called as shown below:

```
void one()
{
            //function definition
}
void main()
{
      ...
      one();      //Correct!! Function one is defined before calling
            ...
}
```

Another elegant alternative to this problem is to include a function prototype before it called no matter where the function has been actually defined as shown below:

```
void one(void);        //prototype of function one()
void main()
{
      ...
      one();      //Correct!! Function one is defined before calling
      ...
}
void one()
{
            //function definition
}
```

3.3 TYPES OF FUNCTIONS

There are two types of functions in C++:

1. Built-in (or library functions)
2. User-defined functions

Built-in Functions

These functions are part of the compiler package. These are part of standard library made available by the compiler. For example, exit(), sqrt (), pow ( ), strlen(), etc. are library functions.

User-defined Functions

The user-defined functions are created by you, i.e., the programmer. These functions are created as per requirements of your program.

3.4 ARGUMENT

An argument is the input to the function; it is placed inside the parentheses following the function name. The function then processes the argument and returns a value; this is the output from the function.

Any C++ function can be invoked or called by another, e.g. in the previous example, main() invokes another function starline( ). The invoking functions say main( ) here may pass information to the invoked function say star line (), and the invoked function may return information to its invoke.

An argument is a piece of data can int value, for example) passed from a programme to the function. Arguments allow a function to operate with different values, or even to do different things, depending on the requirements of the programme calling it.

3.4.1 Formal and Actual Arguments

Although argument and parameters may have different names, a function's parameters should match the function's arguments in number and data type. For example, if a function is invoked with two arguments, it should have two parameters. If the arguments are of type int and char, respectively, the first parameter should be of type int and the second of type char. The match between arguments and parameters is important, for parameters are simply variables that initialized to the values of the arguments when the function is invoked.

3.4.2 Default Argument

C++ functions can have arguments having default values. The default values are given in the function prototype declaration. Whenever a call is made to a function without specifying an argument, the programme automatically assigns values to the parameters as specified in the default function prototype declaration.

Default arguments facility allows easy development and maintenance of programmes. To establish a default value, you have to use the function prototype. Because the compiler looks at the prototype to see how many arguments a function uses, the function prototype also has to alert the programme to the possibility of default arguments. The method is to assign a value to the argument in the prototype. The following programme explains the use of default arguments:

```
#include<iostream.h>

void rchar (char ch = '*', int num = 10); //default arguments with values '*'
and 10

void main (void)
```

```
{
rchar( );
      rchar ('=');
rchar('+', 30);
}
      void rchar (char ch, int num)
{
for (int j = 0; j < num; j ++)
      cout << ch;
      cout << endl;
}
```

The function rchar() is called 10 times in the main( ) function with different number of arguments passed to it. This is allowed in C++.

The first function call to rchar() takes no arguments in the main() function. The function declaration provides default values for the two arguments required by rchar(). When values are not provided, the compiler supplies the defaults '*' and 10.

In the second call, one argument is missing which is assumed to be the last argument. rchar() assigns the single argument '=' to the *ch* parameter and uses the default value 10 for *num*.

In the third case, both arguments are present and no defaults are considered. A few points worth remembering here.

The missing or default arguments must be the trailing arguments, those at the end of the argument list. The compiler will flag an error when you leave something out. Default arguments may be useful when an argument has the same value.

void abc (int a, float b, int c = 10, char d = 'P'); //correct defaulting!

The above declaration is correct because no un-defaulted arguments appear after defaulted arguments. For this reason the following declaration is incorrect.

void abc (int a, float b = 2.5, int c, char d = 'P'); //incorrect defaulting!

3.5 LOCAL AND GLOBAL VARIABLES

The scope for variables is determined by the place of their declaration. If a variable declaration appears outside all the functions, it is said to be global variable. A global variable is available to all the functions and blocks defined in the file. A global variable comes into existence when the programme execution stars and is destroyed when the programme terminates. Global variables hold their values throughout the programme execution. Any expression may access them regardless of what block of code that expression is in. They can be accessed from any where in the file.

Unlike global variables, the local variables are the ones that are defined within a function. A local variable comes into existence when the function is entered and is destroyed upon exit. That is, a local variable cannot hold its value between function calls. It is defined and initialized every time a function call (for the function that declares it) occurs. The only exception to this rule is static local variable. Such variable is defined and initialized at the time of first function call and it holds its value throughout the programme run, but its scope is still the function scope i.e., it cannot be accessed beyond its parent function.

3.6 MULTIFUNCTION PROGRAMME

Programme to illustrate working of call-by-reference method of a function invoking.

```
#include<iostream.h>
#include<conio.h> //for clrscr ( )
int main( )
{ clrscr ( );
vold change (int &);      //notice prototype
      int orig = 5;       //original value is 5
      cout < <" The original value is:" < < orig < <"\n";

      change(orig);
      cout < <"Value after change ( ) is over:" < < orig < < "\n";
      return 0;
}
vold change (int & a)
{a = 10;
      cout<<"Value of orig in function change( ) is:" < < a < < "\n";
      return;
}
```

The above programme will produce the following output.

```
The original value is 5
Value of orig in function change ( ) is: 10
Value after change( ) is over : 10
```

In the above programme, change( ) refers to original value orig which is (5) by its reference a. That is same memory location is being referred to as orig by main( ) and as a by change( ). So the change in a (to 10) is done in the same location as that of orig's and hence, this change is reflected back to main( ). Thus, the above output is produced.

3.7 STORAGE CLASS SPECIFIERS AND VARIABLES

The storage class specifiers tell the compiler how to store the subsequent variable. The storage specifier precedes the rest of the variable declaration. Its general form is as follows:

storage – specifier type var-name:

Four storage class specifiers in C++ are:

auto, register, extern and static

Auto

The storage specifier auto refers to automatic variable. A variable is declared automatic as follows:

auto type variable-name;

that is the keyword auto precedes the normal variable declaration.

By default, variables in a function are auto unless specified otherwise. The auto variable is automatically created when the variable (that defines the auto variable) is called and automatically destroyed when the function terminates. Auto variable is alive as long as the function (that defines it) is executing. Thus, the lifetime (the time period between the creation and destruction of a variable) of the variable is the time during which its parent function (that defines it) is running.

The scope of an auto variable is the function scope. They can be accessed only from their parent function.

Register

A register declaration is an auto declaration. A register variable has all the characteristics of an auto variable. The only difference between the two is that register variables provide fast access as they are stored inside CPU registers rather than in memory. So, the time required to read from memory is saved. The register as well as auto can be applied only to local variables.

Extern

If a programme is spread across files, then a global variable declared once cannot be declared again and again for separate files. Rather, you can put the global variable's declaration preceded by the keywords extern. The extern specifier tells the compiler that the variable types and names that follow it have been declared elsewhere, so that, fresh memory is not allocated to these variables.

The lifetime of external variables is the life of the programme i.e., as long as the programme runs. They hold their values throughout the programme and are destroyed only when the programme terminates. (Do not get confused with programme and a function. A programme can have many functions within it.)

The scope of external variables is the file scope. They are accessible in the file(s) where their declaration (with or without extern) appears.

Static

There can be static local variable and static global variables. When a global variable is declared static, it means that this variable is globally available for the very file it appears in. From all other files of the programme, it is hidden and hence cannot be accessed from there.

Where static modifier applies to a local variable, it is initialized only when the very first call to the function (that defines it) occurs. It is not destroyed when the function terminates, rather it holds its value even after function's termination but it can be only accessed within its own function.

A static local variable (or static automatic variable) has the scope of a local variable but the life time of a global variable. A static global variable has a file scope but its life time is the entire programme run.

3.8 RECURSIVE FUNCTION

In C++, a function can call itself, this is called recursion. A function is said to be recursion if a statement in the body of the function calls itself. For example,

```
fac (int n)
{
      if (n = 1) returnl;
      else
            return n*fac(n-1);   //recursive call, fac( ) calling itself.
}
```

See, the above function called fac( ) invokes itself to find the factorial of n-1. This function works as follows:

Say for the first time n=4

1. fac (4) = { (4 = =1) is false thus return 4* fac (3)}
2. fac (3) = { (3 = =1) is false thus return 3* fac (2)}
3. fac (2) = { (2 = =1) is false thus return 2* fac (1)}
4. fac (1) = { (1 = =1) is true thus return 1}

The result of fac (1) is returned to step (iii) i.e., fac (2) which returns 2* 1 (the result of fac (1)) = 2 to fac (3), which in turn returns 3*2 (the result of fac (2)) = 6. Now fac (4) returns 4* 6=24, which is the final result.

3.9 PREPROCESSOR

As it relates to C++, the preprocessor is largely a hold over from C. Moreover, the C++ preprocessor is virtually identical to the one defined by C. The main difference between C and C++ in this regard is the degree to which each relies upon the preprocessor. In C, each preprocessor directive in necessary. In C++, some features have been rendered redundant by never and better C++ language elements. One of the longer term design goals of C++ is the elimination of the preprocessor altogether. But the preprocessor will still be widely used.

The preprocessor contains the following directions

#define	#elif	#else	#end if
#error	#if	#if def	#ifndef
#include	#line	#pragma	#undef.

All processor directives begin with a # sign. In addition each preprocessing directive must be on its own line.

For example,

#include<stdio.h> #include<stdlib.h>

will not work. They should be in separate lines.

#define

The #define directive defines an identifier and a character sequence (i.e., a set of characters) that will be substituted for the identifier each time it is encountered in the source file. The identifier is referred to as a macro name and the replacement process as macro replacement.

The general form of the directive is,

#define macro-name char-sequence.

For example, if you wish to use the word LEFT far the value 1 and the word RIGHT for the value 0, you could declare the two #define directives:

```
#defined LEFT 1
#define RIGHT 0
```

This causes the compiler to substitute a '1' or a '0' each time LEFT or RIGHT is encountered in your source file. For example, the following prints 012 on the screen:

```
cout<<RIGHT<<LEFT<<LEFT+1;
```

Once a macro name has been defined, it may be used as a part of the definition of other macro names. For example, this code defines the values of ONE, TWO and THREE.

```
#define ONE 1
#define TWO ONE + ONE
#define THREE TWO + ONE
```

Macro substitution is simply the replacement of an identified by the character sequence associated with it. Therefore, if you wish to define a standard error message, you might write something like this

```
#define E-MS "standard error on output\n"
cout<<E-MS;
```

The compiler will actually substitute the string "standard error on output\n" when the identifier E-MS is encountered.

Defining function-like macros (Macro with Parameter)

The #define directive has another powerful feature: The macro name can have arguments. Each time the macro name is encountered, the arguments used in its definition are replaced by the actual argument found in the programme. This form of a macro is called a function-like macro, or Macro with Parameter.

For example,

```
#include <Stdio.h>
#define ABS (a)          (a)<0 ? -(a) : (a)
int main (void).
{printf("abs of -1 and 1: % d % d", ABS (-1), ABS (1)); return 0;}
```

when this programme is compiled, a in the macro definition will be substituted with the values -1 and 1.

For example, if the parentheses around a were removed this expression

ABS(10-20)

Would be converted to

10 - 20 < 0? -10 -20 : 10 - 20

after Macro replacement and would yield the wrong result.

The use of a Macro with parameters in place of real functions has one major benefit: It increases the execution speed of the code because there is no function call overhead. However, if the size of the function - like Macro is very large, this increased speed may be paid for with an increase in the size of the programme because of duplicate code.

#error

The error directive forces the compiler to stop compilation. It is used primarily for debugging. The general form of the #error directive is

#error error-message.

The error message is not between double quotes. When the #error directive is encountered, the error message is displayed, possibly along with other information defined by the compiler.

#include

The #include directive instructs the compiler to read another source file in addition to the one that contains the #include directive. The name of the additional source file must be enclosed between double quotes or angle brackets. For example,

#include "stdio.h"

#include < stdio.h >

both instruct the compiler to read and compile the header for the I/O system library functions.

3.10 HEADER FILES

The C++ language contains some statements only and not any built-in function. Rather it provides standard library that contains files storing the standard functions (I/O functions, mathematical functions, string functions, character functions, etc). These files are known as Header files. Header files provide, function Prototypes definitions of library functions, declarations of data types and constants used with the library functions. There are lots of header files in C++ standard library. A few of them have been listed below:

1. ***stdio.h:*** This header file defines types and macros needed for the standard I/O package.
2. ***string.h:*** This header file declares various string manipulation and memory manipulation routines.
3. ***math.h:*** This header file declares various math functions and math error handlers.
4. ***stdlib.h:*** This header file declares several commonly used routines like conversion routines, search/sort routines and other miscellaneous things.
5. ***iostream.h:*** This header file declares basic C++ stream I/O routines.

6. ***iomanip.h:*** This header file declares stream I/O manipulators and contains macros for creating parameterized manipulators.

7. ***conio.h:*** This header file declares various functions used in calling DOS console I/O routines.

3.11 STANDARD FUNCTIONS

3.11.1 put() and get() Functions

The classes *istream* and *ostream* define two member functions get() and put() respectively to handle the single character input/output operations. There are two types of get() functions. We can use both get(char*) and get(void) prototypes to fetch a character including the blank space, tab and the newline character. The get(char*) version assigns the input character to its argument and the get(void) version returns the input character. . .

Since these functions are members of the input/output stream classes, we must invoke them using an appropriate object. For instance, look at the code snippet given below:

```
char c;
cin.get(c); //get a character from keyboard and assign it to c
while (c!= '\n')
{
cout << c; //display the character on screen cin.get (c);
//get another character
}
```

This code reads and displays a line of text (terminated by a newline character). Remember, the operator> >can also be used to read a character but it will skip the white spaces and newline character. The above while loop will not work properly if the statement

```
cin >> c;
```

is used in place of

```
cin.get(c);
```

Try using both of them and compare the results. The get(void) version is used as follows:

```
char c;
c = cin.get();          //cin.get(c) replaced
```

The value returned by the function get() is assigned to the variable c.

The function put(), a member of *ostream* class, can be used to output a line of text, character by character. For example,

```
cout << put('x');
```

displays the character x and

```
cout << put(ch);
```

displays the value of variable ch.

The variable ch must contain a character value. We can also use a number as an argument to the function put (). For example,

cout < < put(68); .

displays the character D. This statement will convert the int value 90 to a char value and display the character whose ASCII value is 68.

The following segment of a programme reads a line of text from the keyboard and displays it on the screen.

```
char c;
cin.get (c);                    //read a character
while(c!= '\n')
{
cout << put(c);    //display the character on screen cin.get (c );
}
```

3.11.2 getline() and write() Functions

We can read and display a line of text more efficiently using the line-oriented input/output functions getline() and write(). The getline() function reads a whole line of text that ends with a newline character. This function can be invoked by using the object *cin* as follows:

cin.getline(line, size);

This function call invokes the function which reads character input into the variable *line*. The reading is terminated as soon as either the newline character '\n' is encountered or *size* number of characters are read (whichever occurs first). The newline. character is read but not saved. Instead, it is replaced by the *null* character. For example, consider the following code:

char name[20];

cin.getline(name, 20); .

Assume that we have given the following input through the keyboard:

Neeraj good

This input will be read correctly and assigned to the character array *name*. Let us suppose the input is as follows:

Object Oriented Programming

In this case, the input will be terminated after reading the following 19 characters:

Object Oriented Pro

After reading the string, *cin* automatically adds the terminating null character to the character array.

Remember, the two blank spaces contained in the string are also taken into account, i.e. between *Objects* and *Oriented* and *Pro*.

We can also read strings using the operator > > as follows:

cin > > name;

But remember *cin* can read strings that do not contain white space. This means that *cin* can read just one word and not a series of words such as "Neeraj good".

3.12 ARRAYS AND FUNCTIONS

Arrays can be used as arguments to functions. Lets consider an example to understand the concept.

```
// funesale. Cpp
#include<iostream.h>
#include<iomainp.h>
const int districts = 5;          //array dimensions
const int months = 3;
void display (floa + funsales [Districts] [months]); //prototype
void main( )
// initialize two-dimensional array
float sales [Districts] [months]
 ={ { 1423.07, 243.50, 645.01}
      { 328.00, 13872.62, 17584.86}
      { 9329.32, 972.00, 4942.30}
      { 12843.28, 7331.63, 38.94}
      { 17628.72, 4283.66, 429.12}  };
display(sales); // call function, array as an argument
} // end main
void display(float funsales[districts] [Months])
{
      int d, m;
      cout<< "n\n";
      cout<< "Months\n";
      cout<< "          1     2     3\n";
      for (d = 0, d<districs; d++)
      {
            cout<< "district" <<d+1;
      for (m = 0; m < Months; m++)
            cout<<setiosflags (ios::fixed)<setw(10)
            <<setiosflags(ios::showpoint)<<setprecision(2)
            <<funsales[d][m]; //array element
      }
}
```

3.12.1 Function Declaration with Array Argument

Array arguments are represented by the data type and sizes of the array in a function declaration. The declaration of the function display() is as given.

void display(float[Districts] [Months]);

The following statement works just as well:

void display(float[ ] [Months]);

Since a two-dimensional array is an array of arrays. The function first thinks of the argument as an array of districts. It doesn't need to know how many districts are there, but it does need to know how big each districts element is, so it can calculate where a particular element is (by Multiplying the bytes per element times the index). So, we must tell it the size of each element, i.e. Months, but not how many are there, which is districts.

3.12.2 Function Call with Array Argument

When the function is called, only the name of the array is used as an argument:

display(sales);

Here, sales actually represents the memory address of the array. Using an address for an array argument is similar to using a reference argument, in that the values of the array elements are not duplicated(copied) into the function. Instead, the function works with the original array, although it refers to it by a different name.

However, an address is not the same as a reference. No ampersand(&) sign is used with the array name in the function declaration.

3.12.3 Function Definition with Array Argument

In the function definition, the declarator looks like this:

void display (float funsales [Districts] [Months])

The array argument uses the data type, a name, and the sizes of the dimensions. The array name used by the function (funsales in above example) can be different from the name that defines the array (sales), but they both refer to the same array. All the array dimensions must be specified; the function need them to properly access the array elements.

3.13 MULTI-DIMENSIONAL ARRAY

An example of initializing a 4-dimensional array:

```
static int arr [3] [4] [2]   =     {{{2, 4}, {7, 8}, {3, 4}, {5, 6},},
                                    7, 6}, {3, 4}, {5, 3}, {2, 3}, },
                                    8, 9}, {7, 2}, {3, 4}, {6, 1}, }
                                    };
```

In this example, the outer array has three elements, each of which is a two dimensional array of four rows, each of which is a one dimensional array of two elements.

Let us consider some applications of multidimensional array programming.

e.g.:

1. Sorting an integer array.

```
#include<iostream.h>
void main( )
{
        int arr [5];
        int i, j; temp;
        cout<<"\n Enter the elements of the array:";
        cin>>&arr[i]);
        for (i = 0; i < = 4; i ++);
        {
                for (j = 0; j<=3; j++)
                if (arr[j] > arr[j+1])
                {
                        temp = arr[j];
                        arg[j] = arr[j+1];
                        arr[j+1] = temp;
                }
        }
        cout<<"\ n The Sorted array is:";
        for (i = 0; i < 5; i++)
        cout<<"\ t ", arr[i];
}
```

2. To insert an element into an existing sorted array (Insertion Sort).

```
#include<iostream.h>
void main( )
{
        int i, k, y, x [20], n;
        for (i = 0; i < 20; i++)
                x [ i] = 0;
        cout<<"\ Enter the number of items to be inserted:\n";
        cin>>&n;
        cout<<"\n Input %d values \n", n;
```

```
            for (k = 0; k < n; k++)
            {
                  cin>>&x [k];
                  y = x [x]
            for (i = k-1; i > = 0 && y < x [i]; i - -)
                  x [i+1] = x[i];
                  x [i+1] = y;
      }
            cout<<"\n The sorted numbers are:";
            for (i = 0; i < n; i++)
            cout<<"\n ", x [i];
      }
```

3. Accept character string and find its length.

We will solve this problem by looping.

```
#include<iostream.h>
void main( )
{
      char name [20];
      int i, len;
      cout<<"\n Enter the name:";
      cin>>name;
      for (i = 0; name [i] ! = '\0'; i++);
      len = i - 1;
      cout<<"\n Length of array is % d", len;
}
```

Check Your Progress

Fill in the blanks:

1. A functions is a named unit of a group of .... statements.
2. We can read and display a line of text more efficiently using the line-oriented input/output functions ....
3. Array arguments are represented by the data type and .... of the array in a function declaration.
4. The .... directive defines an identifier and a character sequence.

3.14 LET US SUM UP

A function is a named unit of a group of programme statements that can be invoked from other parts of the programme. Function helps reduce the programme size and avoid ambiguity. C++ requires the function definition before it is used anywhere in the programme.

A function prototype is a declaration that tells the programme about the type of the return value of the function and the number and type of each argument. If a function does not return a result, its return type becomes as void. Default arguments can also be specified for a function by writing the default values in its prototype.

C++ provides two kinds of variable specification: local and global. If a variable declaration occurs inside a function, it is a local variable. If a variable declaration occurs outside all functions, it is a global variable. C++ provides four storage class modifiers: auto, register, static and extern.

3.15 KEYWORDS

Function: The best way to develop and maintain a large programme is to divide it into several smaller programme modules of which are more manageable than the original programme. Modules are written in C++ as classes and functions. A function is invoked by a function call. The function call mentions the function by name and provides information that the called function needs to perform its task.

Return: The C++ keyword return is one of several means we use to exit a function. When the return 0 statement is used at the end of main function, the value 0 indicates that the programme has terminated successfully or returns NULL value to the operating system.

Function pass by value: The phrase 'passing by value" and "passing by copy" means the same thing in computer terms. Some textbooks and C++ programmemers say that arguments are passed by value and some say that they are passed by copy. Both describes one of the two methods by which arguments are passed to receiving functions.

Function pass by address: The phrases "passing by address" and "passing by reference" mean the very same thing. Some textbooks and C++ programmers say that arguments are passed by address, and some say that they are passed by reference. When you pass an argument by address, the variable's address is assigned to the receiving function's parameter.

Function prototype: A function prototype declares the return-type of the function that declares the number, the types and order of the parameters, the function expects to receive. The function prototype enables the compiler to verify that functions are called correctly.

Function overloading: In C++, it is possible to define several function with the same name, performing different actions. The functions must only differ in their argument lists. Otherwise, function overloading is the process of using the same name for two or more functions. Each redefinition of a function must use different type of parameters or different sequence of parameters or different number of parameters.

Function recursion: A recursion function is a function that called itself either directly or indirectly. When a loop repetitively call a function, it is called a recursion.

Function declaration: Statement that declares a function's name, return type, number and type of its arguments.

Reference: Alias name of a variable.

Scope: The programme part(s) in which a function or a variable is accessible.

3.16 QUESTIONS FOR DISCUSSION

1. What are actual and formal parameters of a function?
2. When can a function prototype be omitted?
3. What are global and local variable?
4. When is a default argument value used inside a function?
5. What is the principal reason for passing argument by value?
6. What is meant by scope? What all kinds of scope is supported by C++?
7. If the return type of a function is missing, What happen?
8. What is the role of storage class specifiers.
9. Write a C++ programme that uses a function called area ( ) to calculate area of a circle. The function receives radius of float type and returns area of double type.

Check Your Progress: Model Answer

1. programme
2. getline() and write()
3. sizes
4. #define

3.17 SUGGESTED READINGS

Robert Lafore, *Object-oriented Programming in Turbo C++*, Galgotia Publications.

E Balagurusamy, *Object-Oriented Programming with C++*, Tata Mc Graw-Hill

Herbert Schildt, *The Complete Reference C++*, Tata Mc Graw Hill

LESSON 4

POINTERS

CONTENTS

4.0 AIMS AND OBJECTIVES

After studying this lesson, you will be able to:

- Explain the concept of pointers
- Discuss how to declare a pointer
- Define pointer arithmetic
- Identify and explain the pointer and functions
- Discuss the pointer and array
- Explain the concept of pointer and string
- Discuss the concept of pointer to pointer

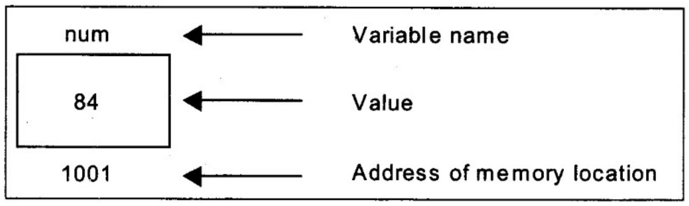
4.1 INTRODUCTION

Computers use their memory for storing instructions of the programmes and the values of the variables. Memory is a sequential collection of storage cells. Each cell has an address associated with it. Whenever we declare a variable, the system allocates, somewhere in the memory, a memory location and a unique address is assigned to this location.

Pointer is a variable which can hold the address of a memory location rather than the value at the location. Consider the following statement

int num = 84;

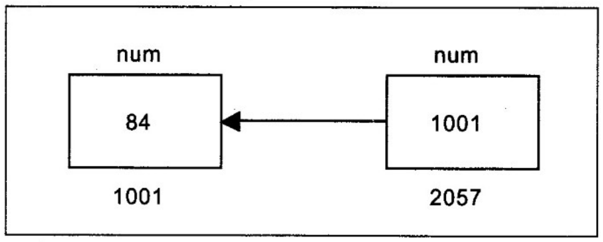
This statement instructs the system to reserve a 2-byte memory location and puts the value 84 in that location. Assume that a system allocates memory location 1001 for num. Diagrammatically, it can be shown as



As the memory addresses are numbers, they can be assigned to some other variable.

Let ptr be the variable which holds the address of variable num.

Thus, we can access the value of num by the variable ptr. Thus, we can say "ptr points to num". Diagrammatically, it can be shown as



4.2 POINTER VARIABLES

Pointers are variables that follow all the usual naming rules of regular, non-pointer variables. As with regular variables, you must declare pointer variables before you use them. A type of pointer exists for every data type in C++; you can use integer pointers, characters pointers, floating-point pointers, floating-point pointer, and so on. You can declare global pointer or local pointer, depending on where you declare them.

The only difference between pointer variables and regular variables is what they hold. Pointer do not contain values, but the address of a value.

C++ has two operators:

1. & - The "address of" operator
2. * - The de-referencing operator

Whenever you see '&' used with pointer, think of the phrase "address of". The '&' operator provides the memory address of whatever variable it precedes. The * operator, when used with pointer, de-references the pointer's value. When you see * operator used, think of the phrase "value pointed to".

The * operator retrieves the value that is "pointer to" by variable it procedes.

4.3 DECLARING POINTERS

A pointer is a special variable that return the address of a memory. If you need to declare a variable to hold your age, you might use the following variable declaration:

int age = 18 // declares to variable to hold my age

Declaring age this way does several things. Because C++ knows that you need a variable called age, C++ reserves storage for that variable. C++ also knows that you will store only integers in age, not floating-point or double floating point data. You also have requested that C++ store the value of 18 in age after reserving storage for that variable.

Suppose that you want to declare a pointer variable, not to hold your age but to point to age, the variable that hold your age. p_age might be a good name for this pointer variable. Figure given below shows an illustration of what you want to do. This example assumes that C++ store age at address 1002, although your C++ compiler arbitrarily determines the address of age, and it could be anything.

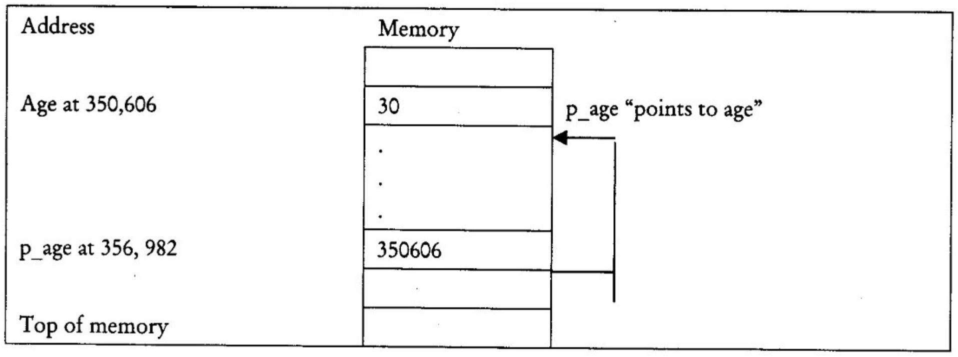


Figure 4.1: p_age contains the address of age; p_age to the age variable.

To declare the p_age pointer variable. You should do the following:

int*_age; // declare an integer pointer

As with the declaration for age, this line reserves a variable called p_age. It is not a normal integer variable, however, because of the C++. knows that this variable is to be a pointer variable. The following programme example demonstrates the declaration and use of pointer in C++.

Example
```
# include<iostream.h >
int main( )
{
        int i = 18, * p_age;
        p_age  = &I;
        cout<<i<< " << * p_age << endl;
}
```

4.3.1 Accessing Variable through Pointer

Consider the following statements:

```
int q, * i, n;
q = 35;
i = & q;
n = * i;
```

i is a pointer to an integer containing the address of q. In the fourth statement, we have assigned the value at address contained in i to another variable n. Thus, indirectly we have accessed the variable q through n. using pointer variable i.

4.3.2 Pointer Expressions

Like other variables, pointer variables can be used in expressions. Arithmetic and comparison operations can be performed on the pointers.

e.g.: If p1 and p2 are properly declared and initialized pointers, then following statements are valid.

```
y = * p1 * *p2;
sum = sum + * p1;
```

4.4 POINTER ARITHMETIC

C++ allows us to add integers from pointers as well as subtract one pointer from another. Shorthand operators like sum + = *p2; can also be used with the pointers for arithmetic expressions. Following operations can be performed on a pointer:

(a) Addition of a number to a pointer

e.g.:
```
int i = 4, *J, *k;
J = &i;
J = J+1;
J = J+9;
k = J+3;
```

(b) Subtraction of a number from a pointer

e.g.:
```
int i = 4, *J, *k;
J = &i;
J = J-2;
J = J-5;
k = J-6;
```

(c) Subtraction of one pointer from another provided that both variables point to elements of the same array. The resulting value indicates the number of bytes separating corresponding array elements.

But the following operations are not allowed on pointers.

- Multiplication of pointer with a constant
- Addition of two pointers
- Division of pointer by a constant

Pointer Comparison

Pointers can be compared using relational operators. Expressions such as p1 > p2, p1 = = p2 and p1 ! = p2 are allowed. Such comparisons are useful when both pointer variables point to elements of the same array.

4.5 POINTER AND FUNCTIONS

A function is a user defined operation that can be invoked by applying the call operator ("( )") to the function's name. If the function expects to receive arguments, these arguments (actual arguments) are placed inside the call operator. The arguments are separated by commas. A functions may be involved in one of the two way;

(a) Call by value

(b) Call by reference

Call by Value

In this method, the value of each actual argument in the calling function is copied on to the corresponding formal arguments of the called function. The called function works with and manipulates its own copy of arguments and because of this, changes made to the formal arguments in the called function have no effect on the values of the actual arguments in the calling function.

Call by Reference

This method can be used in two ways:

(i) by passing the reference

(ii) by passing the pointer.

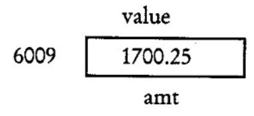
A reference is an alias name for a variable.

For instance, the following code

float value = 1700.25;

float &amt = value; //amt reference of value

declares amt as an alias name for the variable value. No separate memory is allocated for amt rather the variable value can now be accessed by two names value and amt.

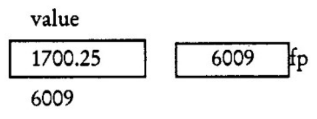


A pointer to a variable holds its memory address using which the memory area storing the data value of the variable can be directly accessed. For instance, the following code

float value = 1700.25

float fp = &value ; // fp stores the address of value

makes a float pointer *fp point to a float variable value. The variable value can now be accessed through fp also (using *fp)



The call by reference method is useful in situations where the values of the original variables are to be changed using a function. The following example programme explains it

E.g: Programme to swap values of two variables using pass by reference method.

```
#include<iostream.h>
#include<conio.h>        //for clrscr)
int main( )
{ clrscr( );
void swap(int &, int &); // prototype
int a = 7,  b = 4;
cout<<"Original values \n";
cout<<"a=" << a <<, "b=" << b << "\n";

swap(a,b);               //invoke the function
cout<<"swapped values \n";
cont<<"a=" << a << "b=" << b << "\n";
return 0;
}
\\function definition
void swap(int & x, int & y)
{ int temp;
    temp = x;
    x=y;
    y=temp;
}
```

The output produced by the above programme is as follows:

```
Original Values
a= 7, b = 4
```

Swapped Values

a = 4, b = 7

In the above programme, the function swap ()

creates reference 'x' for the first incoming integers and reference 'y' for the second incoming integer. Thus the original values are worked with but by using the names x and y. The function call is the simple one i.e.;

swap (a,b);

but the function declaration and definition include the reference symbol &. The function declaration and definition, both, start as

void swap(int &, int &)

Therefore, by passing the references the function works with the original values (i.e., the same memory area in which original values are stored) but it uses alias names to refer to them. Thus, the values are not duplicated.

4.5.1 Pointers to Functions

When the pointers are passed to the function, the addresses of actual arguments in the calling function are copied into formal arguments of the called function. This means that using the formal arguments (the addresses of original values) in the called function, we can make changes in the actual arguments of the calling function. Therefore, here also, the called function does not create own copy of original values by the addresses (passed through pointers) it receives.

Eg: Programme to swap values of two variables by passing pointers.

```
#include<iostream.h>
#include<conio.h>                                  // for clrscr( )
int main( )
{ clrscr( );
void swap(int *x, int *y);                         // prototype
int a = 7, b = 4;
cout<< "Original values \n";
cout<< "Original values \n";
cout<< " a=" << a << ",b=" << b<< "\n";
Swap(&a, &b);                                      // function call
cout<< "swapped values \n";
cout<< "a=" << a <<, "b=" << b << "\n";
return 0;
}
//function definition
void swap(int *x, int * y)
```

```
{   int temp;
    temp = *x;
    *x = *y;
    *y  = temp;
}
```

Output: Original values

a = 7, b = 4

Swapped values

a = 4, b= 7

4.5.2 Function Returning Pointers

The way a function can returns an int, a float, a double, or reference or any other data type, it can even return a pointer. However, the function declaration must replace it. That is, it should be explicitly mentioned in the function's prototype. The general form of prototype of a function returning a pointer would be type

function-name (argument list);

The return type of a function returning a pointer must be known because the pointer arithmetic is relative to its base type and a complier must know what type of data the pointer is pointing to in order to make it point to the next data item.

Eg. Programme to illustrate a function returning a pointer.

```
#include<iostream.h>
#include<conio.h>
int *big(int&, int&)                    //prototype
int main( )
{   clrscr( )
    int a, b, *c;
    cout<<"Enter two integers \n";
    cin>>a>>b;
    c= big(a, b);
    cout<<"The bigger value is "<< *c<< "\n";
return 0;
}
int *big(int &x, int &y)
{ if  (x>y)
return(&x);
else
return(&y);
}
```

```
if input = 7        13
```

then, output : The bigger value is 13

4.6 POINTER AND ARRAYS

When an array is declared, the compiler allocates a base address and sufficient amount of storage to contain all the elements of the array in contiguous memory locations. The base address is the location of the first element (index 0) of the array. The compiler also defines the array name as a constant pointer to the first element.

The array declared as

static int x[5] = {1, 2, 3, 4, 5};

is stored as follows:

Elements	x[0]	x[1]	x[2]	x[3]	x[4]
Value	1	2	3	4	5
Address	1000	1002	1004	1006	1008

The name *x* is defined as a constant pointer pointing to the first element, *x[0]* and therefore the value of x is 1000, the location where *x[0]* is stored. That is,

x = &x[0] = 1000

If we declare *p* as an *integer pointer*, then we can make the pointer *p* to point to the *array x* by the assignment statement *p = x ;* which is equivalent to *p = &x[0];*

Now, we can access every value of *x* using *p++* to move from one element to another. The relationship between p and x is shown below:

p = &x[0] (=1000)

p+1 = &x[1] (=1002)

p+2 = &x[2] (=1004)

p+3 = &x[3] (=1006)

The address of an element is calculated using its *index* and the *scale factor* of the data type.

e.g.: Address of *x[3]* = Base Address + (3 Scale Factor of *int*) = 1000 + (3 × 2) = 1006

When handling arrays, instead of using array indexing, we can use pointers to access array elements, as **(p+3)* gives the value of *x[3]*. The pointer accessing method is much faster than array indexing. *&x[i]* and *(x+i)* both represent the address of the i^{th} element of x. *x[i]* and **(x+i)* both represent the contents of that address, the value of the i^{th} element of x. The two terms are interchangeable.

When assigning a *value* to an array element such as *x[i]*, the left side of the assigned statement may be written as either *x[i]* or as **(x+i)*. Thus, a value may be assigned directly to an array element, or it may be assigned to the memory area whose address is that of the array element. While assigning an *address* to an identifier, a pointer variable must appear on the left side of the assignment statement. Expressions such as *x, (x+1)* and *&x[i]* cannot appear on the left side of an assignment statement because it is not possible to assign an arbitrary address to an array name or an array element.

Array of Pointers

A multi-dimensional array can be expressed in terms of an array of pointers rather than as a pointer to a group of contiguous arrays. In such situations, the newly defined array will have one less dimension than the original multi-dimensional array. Each pointer will indicate the beginning of a separate (n - 1) dimensional array.

In general terms, a two dimensional array can be defined as one dimensional array of pointers by writing

*data_type *array[expression1];*

rather than the conventional array definition *data_type array[expression1] [expression2];*

Similarly, a n dimensional array can be defined as a (n-1) dimensional array of pointers by writing

data_type *array[expression1][expression2]...[expressionN-1];

rather than the conventional array definition data_type array[expression1] [expression2]...[expressionN];

In these declarations *data_type* refers to the *data type* of the original n dimensional array, *array* is the array name, and *expression1, expression2, . . ., expression n* are positive-valued integer expressions that indicate the maximum number of elements associated with each subscript.

The array name and its preceding asterisk *are not enclosed in parentheses* in this type of declaration. Thus, a right-to-left rule first associates the pairs of square brackets with *array,* defining the named object as an array. The preceding asterisk then establishes that the array will contain pointers.

Moreover, note that the last (the rightmost) expression is omitted when defining an array of pointers, whereas the first (the leftmost) expression is omitted when defining a pointer to a group of arrays.

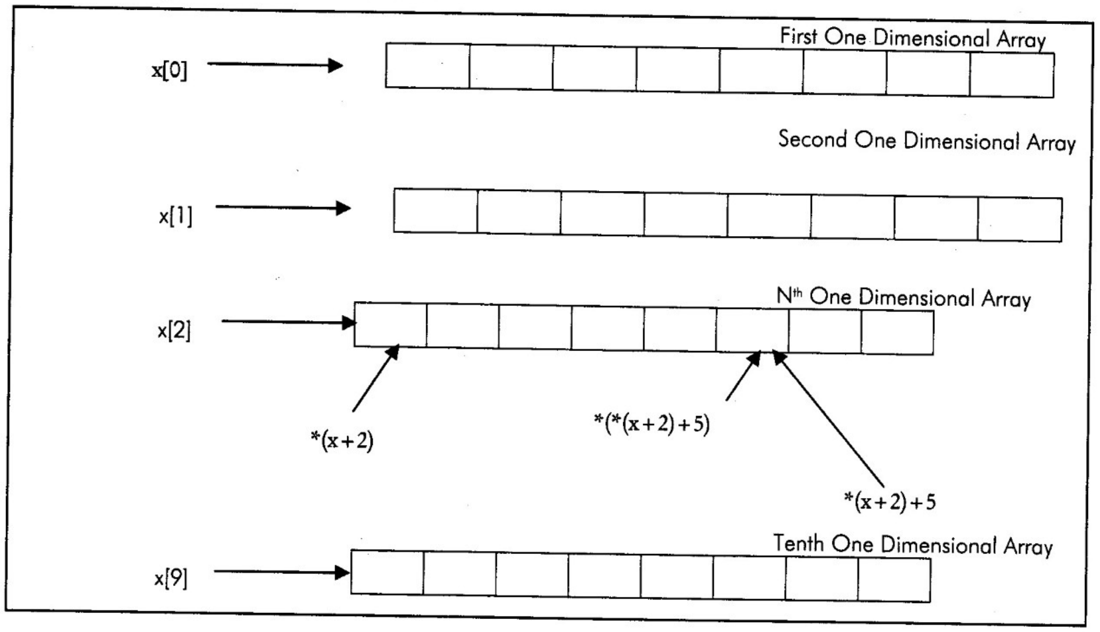
When a *n* dimensional array is expressed in this manner, an individual array element within the n dimensional array can be accessed by a single use of the indirection operator. The following example illustrates how this is done.

Suppose that x is a two dimensional integer array having 10 rows and 20 columns, we can define x as a one dimensional array of pointers by writing int *x[10];

Hence, x[0] points to the beginning of the first row, x[1] points to the beginning of the second row, and so on. The number of elements within each row is not explicitly specified.

An individual array element, such as x[2][5], can be accessed by writing **(x[2] + 5).* In this expression, x[2] is a pointer to the first element in row 2, so that (x[2] + 5) points to element 5 (actually, the sixth element) within row 2. The object of this pointer, *(x[2] + 5), therefore, refers to x[2] [5].

These relationships are illustrated below:


First One Dimensional Array
x[0]
Second One Dimensional Array
x[1]
Nth One Dimensional Array
x[2]
*(x+2)
((x+2)+5)
*(x+2)+5
Tenth One Dimensional Array
x[9]

4.7 POINTERS AND STRINGS

As you know that C++ treats a string as an array of charachers, we are discussing the string under arrays. A string is a one-dimensional array of characters Aeriminated by a null ('\o'). To print a string vertically, using a character pointer, consider the following programme:

```
Char name [ ] = "pointer".
Char * cp;                    // a char pointer
for (cp = name; *cp! = '\o'; cp++)
cout<<"\n" << *cp;
```

The output of the above given code segment is

```
P
O
I
N
T
E
R
```

Here, the for loop in the initialization section makes cp (the char pointer) point to name (the string) and then prints the data value being printed to the cp in each cycle of for. At the end of each cycle, the cp is incremented so that it points to the next character in the string. The loop continues as long as the cp is not pointing to null ('\0').

To print the entire string in the same line using the pointer, the loop will be as

```
for (cp = name; *cp! '\0'; cp++)
    cout<< * cp;
    cout<< "\n";
```

Thus, we have seen how we can access strings and their constituent characters using char pointer. An array of char pointers is very popularly used for string several strings in memory. Let us see how several strings can be stored in the array of pointers;

Char * names [ ] = { "Sachin", "Kapil", "Ajay", "Sunil", "Anil"}}

In the above declaration, names [ ] is an array of char pointers whose element. pointers contain base addresses of respective names. That is, the element pointer name {0} stores the base address of string "Sachin", the element pointer name [1] stores the above address of string "Kapil", and so forth.

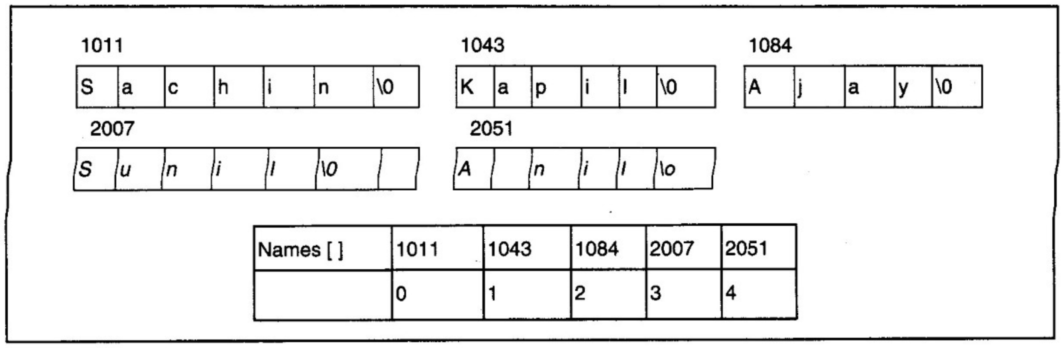


Figure 4.2: Array of Char Pointers, Pointing to Several Strings

The array of char pointers is generally preferred over two dimensional array of characters. There are two reasons behind this, which are:

1. An array of pointers makes more efficient use of available memory by consuming lesser number of bytes to store the strings.
2. An array of pointers makes more the manipulation of the strings much easier. One can easily exchange the positions of strings in the array using pointers without actually touching their memory locations. The following example programme illustrates it.

e.g: Programme to exchange the positions of strings stored in array using array of pointers.

```
#include<iostream.h>
#include<conio.h>
#include<string.h>
int main( )
{ clscr ( );
Char *names[ ] = {"Sachin", "Kapil", "Ajay", "Sunil", "Anil" };
int len = 0;
```

```
    len = strlen (name [1]) ;        // length of string no.
    cout<< "\n Originally string 2 is";
    cout.write (names [1], len).put('\n');
    cout<< "and string is";
    cout.write(names [3], len).put('\n');
    // exchange the positions of string 2 and 4 //
    char * t;
    t = names [1];
    names [1] = names [3];
    names [3] = t;
    // print exchanged strings
    len= strelen(names [1]);
    cout<< "Exchanges string 2 is";
    cout.write(names [1], len).put('\n');
    cout<< "and string 4 is";
    cout.write(names [3], len).put('\n);
    return 0;
}
```

The above programme produces the following output:

Originally string 2 is Kapil

and string 4 is Sunil

Exchanged string 2 is Sunil

and string 4 is Kapil

The above programme just exchanges two addresses of the names stored in the array of pointers, rather than the names themselves. Thus, by effecting just one exchange we are able to interchange names. This makes managing string very convenient.

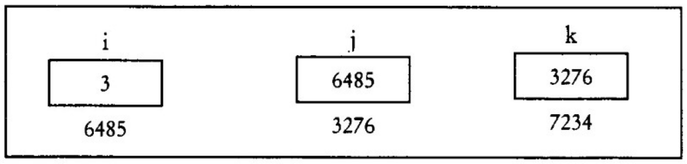
4.8 POINTER TO POINTERS

Pointer is a variable, which contains address of a variable. This variable itself could be another pointer. Thus, a pointer contains another pointer's address as shown in the example given below:

```
void main()
{
   int i = 3; int *j, **k;
   j = &i; k = &j;
   cout<<"Address of i = \n"<< &i;
   cout<<"Address of i = \n"<<j;
```

```
    cout<<"Address of i = \n"<<*k;
    cout<<"Address of j = \n"<<&j;
    cout<<"Address of j = \n"<<*k;
    cout<<"Address of k = \n\n"<<&k;
    cout<<"Value of j = \n"<<j;
    cout<<"Value of k = \n"<<k;
    cout<<"Value of i = \n"<<i;
    cout<<"Value of i = \n"<<*(&i);
    cout<<"Value of i = \n"<<*j;
    cout<<"Value of i = \n"<<**k;
}
```

The following figure would help you in tracing out how a programme prints the output.



Output:
```
Address of i  = 6485
Address of i  = 6485
Address of i  = 6485
Address of j  = 3276
Address of j  = 3276
Address of k  = 7234
Value of j    = 6485
Value of k    = 3276
Value of i    = 3
Value of i    = 3
Value of i    = 3
Value of i    = 3
```

Check Your Progress

Fill in the blanks:

1. Pointers are .... that follow all the usual naming rules of regular, non-pointer variables.
2. C++ allows us to add integers from .... as well as subtract one pointer from another.
3. The way a function can .... an int, a float, a double, or reference or any other data type, it can even return a pointer.
4. Pointer is a variable, which contains .... of a variable.

4.9 LET US SUM UP

A pointer is a variable that holds the memory address of the location of another variable in memory. A pointer is declared in the following form:

type * var_name ;

where type is a predefined C++ data type and var_name is the name of the pointer variable. The operator &, when placed before a variable, returns the memory address of its operand. The operator * returns the memory address of its operand. The operator * returns the data value stored in the area being pointed to by the pointer following it. The pointer variables must always point to the correct type of data. Pointers must be initialized properly because uninitialized pointers result in the system crash. In pointer arithmetic, all pointers increase and decrease by the length of the data type point to.

An array name is a pointer that stores the address of its first element. If the array name is incremented, It actually points to the next element of the array. Array of pointers makes more efficient use of available memory. Generally, it consumes lesser bytes than an equivalent multi-dimensional array.

Functions can be invoked by passing the values of arguments or references to arguments or pointers to arguments. When references or pointers are passed to a function, the function works with the original copy of the variable. A function may return a reference or a pointer also.

4.10 KEYWORDS

Pointer: A variable holding a memory address.

Base address: Starting address of a memory location holding array elements.

Memory location: A container that can store a binary number.

Reference: An alias for a pointer that does not require de-referencing to use.

Function Pointer: A function may return a reference or a pointer variable also. A pointer to a function is the address where the code for the function resides. Pointer to functions can be passed to functions, returned from functions, stored in arrays and assigned to other pointers.

Alias: A different name for a variable of C++ data type.

4.11 QUESTIONS FOR DISCUSSION

1. What is pointer?
2. How does pointer variable differ from simple variable?
3. What does the character '&' stand for?
4. What does the character '*' stand for?
5. Find the syntax error (s), if any, in the following programme:

```
{
   int x [5], *y [5]
   for (i = 0; i< 5; i++)
   { x [i] = I;
   x[i] = i + 3;
   y = z;
   x = y;
}
```

6. What are the relation between array and pointer?
7. Discuss two different ways of accessing array elements.
8. Write a programme to traverse an array using pointer.
9. Write a programme to compare two strings using pointer.

Check Your Progress: Model Answer

1. Variables
2. Pointers
3. Returns
4. Address

4.12 SUGGESTED READINGS

Robert Lafore, *Object-oriented Programming in Turbo* C++, Galgotia Publications.

E Balagurusamy, *Object-Oriented Programming with* C++, Tata Mc Graw-Hill

Herbert Schildt, *The Complete Reference* C++, Tata Mc Graw Hill

UNIT III

LESSON

5

STRUCTURES UNIONS AND BIT FIELDS

CONTENTS

5.0 AIMS AND OBJECTIVES

After studying this lesson, you will be able to:

- Explain the concept of structures, unions and bitfield
- Discuss how to identify nested structures
- Define unions
- Identify and explain the bitfields
- Define enumeration

5.1 INTRODUCTION

Until now you have been working with atomic data types. While these data types are suitable for manipulation of raw data, they fail to represent structured data. In practical problems related data items are often grouped into a single unit much as in the case of records. This is possible if the programming language allows the programmers to create their own grouped data types. C allows programmer to group data types into what is known as structure.

Sometimes creating appropriate data items sharing the same memory space enhances the processing by making the program space efficient. Unions are the way to this in C.

This lesson deals with structures and unions in detail.

5.2 STRUCTURE

A structure is a collection of variables referenced under one name providing a convenient means of keeping related information together. The structure definition creates a format that may be used to declare structure variables in a programme later on.

The general format of structure definition is as follows:

```
      struct tag_name
            {
                  data_type member1;
                  data_type member2;
                  - - - - - - -
                  - - - - - - -
            };
```

A keyword struct declares a structure to hold the details of fields of different datatypes.

At this time, no variable has actually been created. Only a format of a new data type has been defined.

Consider the following example:

```
struct addr
{
    char name [30];
    char street [20];
    char city [15];
    char state [15];
    int pincode;
};
```

The keyword struct declares a structure to hold the details of fine fields of address, namely, #name, street, city, state, pin code. The first four members are character array and fifth one is an integer.