v. Strings are character arrays. The last index of it contains the null character_____.
   a. \n
   b. \t
   c. \0
   d. \1

Q.3 Write TRUE/FALSE against the following statements.

i. Array is a collection of heterogeneous type of data.
ii. Array can be initialized during declaration time.
iii. char vowel[5] declares an integer type array that holds the vowel characters of English language.
iv. Loop is the best tool for handling large arrays.
v. Array elements are stored in the memory where ever free space is available.

Q.4 Declare an array to hold the high temperature (to the nearest tenth of a degree) for each day of a year. Assign a value of 0 to each day.

Q.5 Write down a C++ program to find the multiplication of two matrices A[3][2] and B[2][3].

Q.6 Write down a C++ program to find the subtraction and addition of two matrices A[3][3] and B[3][3].

Q.7 Write a C++ program to find the transpose of a matrix A[3][3].

Q.8 Write a C++ program to read the temperature of the whole week in an array and then find the hottest day of the week.

Q.9 Write a C++ program to read ten alphabets of English from the keyboard into a character type array and then sort them in descending order.

Q.10 Write a C++ program to find sum of the values of a two dimensional array int Test[2][3] and display the result on the screen.

UNIT 6

FUNCTIONS

After the completion of Unit-6, Students will be able to:

- Explain the concept and types of functions
- Explain the advantages of using functions
- Explain the signature of functions (Name, Arguments, Return type)
- Explain:
  - Function prototype
  - Function definition
  - Function call
- Differentiate among local, global and static variables
- Differentiate between formal and actual parameters
- Know the concept of local and global functions
- Use inline functions
- Pass the arguments:
  - constants
  - By value
  - By reference
- Use default arguments
- Use **return** statement
- Define function overloading
- Know advantages of function overloading
- Understand the use of function overloading with:
  - Number of arguments
  - Data types of arguments
  - Return type

INTRODUCTION

A good programming technique is to modularize the programs into small segments of codes. These small segments are called modules or procedures and perform some specific tasks. In C++ language, these modules are usually called functions. This unit describes the basic concepts of functions, its parts, parameters and arguments. Here, the concept of function overloading has also been discussed with the help of examples.

6.1 Functions

In C++ program, when a set of related statements are grouped together, in a particular format, to accomplish a specific task is called *function*. *Function* is one of the main building blocks of any programming language. There are some functions whose work has been fixed by the developers of C++ programming language. These are called *built-in functions*. *Built-in functions* are specific in its functionalities; therefore, the programmers develop their own functions called *user-defined functions*. These functions are the main focus point of this unit.

6.1.1 Concept and types of functions

Function is an important feature of any programming language. It makes the work easier by writing the code once and executing it again and again. Mainly, functions are divided into two types built-in and user defined.

a. Concept of function

Whenever we want to perform a task again and again in a program such as calculating the square or cube for multiple integer values then it is too difficult to write the same code multiple times, say 5 times for 5 different integer values, in the same program.

b. Types of functions

C++ functions are categorized into two types namely *built-in* function and *user defined* functions. The following sections describe these types in detail.

i. Built-in Functions

C++ language provides a number of pre-compiled functions for some commonly used operations. These functions are called *built-in functions*. *Built-in* functions are part of the C++ language and are highly valuable, pre-tested and completely reliable. The C++ built-in functions are made for various assignments ranging from algebra, geometry, trigonometry, finance, text and graphics to some complex operations.

Consider the following example that requests the value of angle and calculate its *sine* and display it on the screen.

```
// built-in function(sin())program
#include <iostream.h>
#include<conio.h>
#include<math.h>
int main()
{
float angle;
cout << "Enter value for angle:"<<endl;
cin >> angle;
cout<<"The Sine of "<<angle<<" degree is =
"<<sin(angle);
getch();
return 0;
}
```

Output of the program

Enter value for angle:

45.0

The Sin of 45.0 degree is = 0.850904

Here, **sin()** is a built-in function defined in the **math.h** header file. The value inside the **sin()** is converted to its corresponding *sin* value and displayed on the screen. Similarly we can calculate cosine, tangent and tangent inverse by using cos(x), tan(x) and cot(x) built-in functions respectively.

Consider another example:

```
//built-in function abs( ) program
#include <math.h>
#include <iostream.h>
#include <conio.h>
int main()
{
int X;
cout<<"Enter value for X"<<endl;
cin>>X;
cout<< "Absolute value of " <<X << " is: " <<abs(X);
getch();
return 0;
}
```

Output of the program

Enter value for X

-12

Absolute value of -12 is: 12

In this example, the value entered by the user, from the keyboard, is taken into the *abs( )* function and the result is displayed on the screen. Like abs( ), some other functions that are included in <math.h> header file are pow(x,n), sqrt(x), etc.

The built-in functions are defined in the C++ library which should be included in the program before using these functions. These functions are used for the most commonly and frequently used operations and cannot be modified according to the users need. This feature of the functions is its main limitation. To overcome this problem, C++ provides a new set of functions that can be defined by the users itself. The following section describes this set of functions in detail.

ii . User-defined functions

The functions defined by the users, according to their needs, to perform their own tasks are called users-defined functions. These functions are used to overcome the limitations of build-in functions.

Defining user-defined function

A function must be defined first to use it in the program. The general syntax for defining a user-defined function is given below.

```
type name (parameter1, parameter2, ...)
{
Statements;
}
```

Where:

- ***Type*** is the data type specifier of the data returned by the function.
- ***Name*** is the identifier by which a function will be called.

- ***Parameters*** are the variables that receive arguments. Each parameter has its data type like variable. There can be as many parameters as needed but they should be separated by commas.
- ***Statements*** constitute the body of the function. It is a block of statements surrounded by braces { }.

Consider a function named *addition* to take two integer values, add them together and return the result to the calling program. The following code segment defines this function:

```
// user-defined function (addition) program
#include <iostream.h>
#include <conio.h>
int addition (int a, int b)
{
int sum;
sum=a+b;
return (sum);
}
int main()
{
int z;
z = addition (55,33);
cout << "The sum of a and b is= " << z;
getch();
return 0;
}
```

Output of the program

The sum of a and b is =88

Working of user-defined functions

In the above program, addition (int a, int b) is the example of user-defined function defined by the user to perform the addition of two integer values (55 and 33 in this case). These values are passed by the function call *addition (55,33)*. After performing addition, the function returns the sum (88) back to calling program (in this case, main) using the following statement.

return (sum);

The sum is assigned to the variable z, in *main*(), as shown below:

z = addition (55, 33);

The following line of code, in *main*, is used to display the result on the screen as shown in the above program.

cout << "The sum of a and b is= " << z;

6.1.2 Advantages of using functions

Using functions we can structure our programs in a more modularized way. The use of functions provides many benefits, including:

- ***Code reusability***: The code inside the function can be reused by calling it again and again in the program.
- ***Updating code***: It is much easier to change or update the code in a function, which needs to be done once.
- ***Readability and understandability***: It makes the code easier to read and understand.

6.1.3 Function signature

The signature of a function comprises of the parts given below.

- Name of the function
- The number, order and data types of the arguments
- Return type of function

Let us consider the *add* function to demonstrate function signature

```
double add(int x, double y)
{
return x+y;
}
```

The signature of the above mentioned function is:

double *add*(int x, double y)

The Signature of this *add* function comprises of:

- Name of the function: **add**
- The data types of the arguments: **int , double**
- Return type of the function: **double**

6.1.4 Components of functions

Each function consists of three components. These are:

- Function prototype/declaration
- Function definition
- Function call

a. Function prototype

A *function prototype* is that part of a function that tells the compiler the name of the function, the type of data returned by the function, the number of parameters the function expects to receive, the types of the parameters, and the order of parameters.

A function prototype is also called *function declaration* and is used in situation whenever the function is defined after the **main()** function. The function prototype ends with a semicolon (;).

The function prototype for a function named "*maximum*" to calculate maximum number out of three integers is given below:

int maximum(int a, int b, int c);

The compiler use function prototypes to validate function call. In a prototype, argument names are optional; however, the types are necessary as shown below:

int maximum(int , int , int);

Function prototype is used in C++ program only when the function is defined after the definition of *main()* function. If the function definition lies before the *main()* function then there is no need for the function prototype.

b. Function definition

A function definition specifies what a function does. It has two parts: a *declaration /header* and *function body* enclosed in { }. Function *declarator* is similar to *function* prototype with the only difference that it has the variables name for the parameters and without semicolon (;). Consider the following function *declarator*:

int maximum(int x, int y, int z)

The second part of function definition is the function body. It contains the actual statements that perform the intended task and is always enclosed in { }.

The whole function definition is shown below:

```
int maximum(int x. int y, int z) // function declarator
{
int max = x;
if ( y > max )
max = y;
if ( z > max )
max = z;
return max;
}
```

function body

c. Function call

A *function call* is a statement in the calling function (e.g. **main()** function) to execute the code of the function. Consider the following statement:

maximum(a, b, c);

The function call has the list of arguments that are passed to the *function declarator*. The arguments in this call are a, b and c. The following program shows different parts of the function i.e. prototype, definition and call.

```
// finding maximum out of three integers using function
#include <conio.h>
```



```
#include <iostream.h>
int maximum( int, int, int ); // function prototype
int main()
{
int a, b, c;
cout << "Enter three integers: ";
cin >> a >> b >> c;
cout << "Maximum is: " << maximum ( a, b, c ) << endl; // Function call
getch();
return 0;
}

int maximum( int x, int y, int z ) // function definition
{
int max = x;
if ( y > max )
max = y;
if ( z > max )
max = z;
return max;
}
```

Output of the program

```
Enter three integers: 4 -2 6
Maximum is: 6
```

6.1.5 Scope of variables

By the scope of a variable we mean that if a variable is defined in a program then in which parts of the same program this variable can be accessed. There are three scopes of variables.

- Local scope
- Global scope
- Static scope

a. Local/Automatic variables

The variables defined within a block are called local variables and its scope is local to block. These variables are not accessible from outside the block and thus their visibility remains only to that block. For example, in the following code segment, the variables a, b and c are local variables and cannot be used outside this boundary.

```
int main()
{
int a, b, c;
cout << "Enter three integers: ";
cin >> a >> b >> c;
// a, b and c are the local variables and cannot be used outside the {} block
cout << "Maximum is: " << maximum ( a, b, c ) << endl; getch();
return 0;
}
```

It is impossible to use the variables x, y or z defined in maximum( ) function directly within *main* function because these variables are local to the *maximum* function.

The following program makes use of a function *square()* that has its own local variable 'y' which is not accessible in the *main()* function.

```
// local variables (example) to calculate square of integers
#include <iostream.h>
#include<conio.h>
int square( int y); // function prototype
```



```
int main()
{
for ( int x = 1; x <= 10; x++ )
cout << square( x ) << " ";
cout << endl;
getch();
return 0;
}
// Function definition
int square( int y )
{
return y * y;
}
```

b. Global variables

The variables defined usually at the top of a program before the *main()* function are called global variables. The variables declared *globally* are visible from any point of the code. They are accessible from inside and outside of all the functions of the same program. In order to declare global variables we simply have to declare the variable outside any function or block; i.e. directly in the body of the program.

The following program describes local and global variables.

```
#include <iostream.h>
#include <conio.h>
int add(); /* function prototype */
int val1, val2, val3; /*These are global variables */
int add()
{
```

```
int sum; // here, sum is local variable
sum = val1 + val2 + val3;
return sum;
}
int main()
{
int total; //here, total is local variable
val1 = 10;
val2 = 20;
val3 = 30;
total = add();
cout<<"The sum of the three global variables="<<total;
getch();
return 0;
}
```

Output of the program

The sum of the three global variables=60

In this example, the variables *val1, val2, val3* are defined globally and can be accessed both in *main()* function and *add()* function without generating any error message. Here in the same program, the variables *sum* in function *add()* and *total* in function *main()* are local to the respective functions and are not visible outside the boundaries of these functions.

c. Static variables

Static variables are those variables which are preceded by the keyword **static** while declaring. For example:

```
static Static_Var;
```


Static variables are initialized once in the program and remains in memory until the end of the program. These variables have the capability to preserve information about the last value a function returned, until the program completely ends. Static variables are *local in scope* to their module or function in which they are defined, but *life is throughout the program.*

Consider the following example to demonstrate the use of static variables.

```
/* static and automatic variables program*/
#include <conio.h>
#include <iostream.h>
void demo(){
auto int Auto_Var = 0; // automatic variable
static int Static_Var = 0; // static variable
cout<<"Auto ="<<Auto_Var<< ","<< "Static="<<Static_Var<<endl;
++ Auto_Var; ++ Static_Var; }
int main()
{
int i=0;
while( i < 3 )
{ demo(); i++; }
getch();
return 0;
}
```

Output of the program

```
Auto = 0, Static = 0
Auto = 0, Static = 1
Auto = 0, Static = 2
```

Here, the automatic variable **Auto_Var** losses its values when the control goes out of the function body but the static variable **Static_Var** keeps its last value even if the control goes out of the function.

6.1.6 Formal parameters and actual parameters

In function prototypes and function calls, variables and values are written in the parenthesis of the functions. These are divided into two categories; ***formal parameters*** and ***actual parameters***.

a. Formal parameters

Formal parameters are those parameters which appear in function *declarator/header* and also in *function prototype*. These are also called *formal arguments* and are used to receive values for functions.

For example:

```
void foo(int x); // prototype -- x is a formal parameter
void foo(int x) // declarator -- x is a formal parameter
{
body
}
```

b. Actual parameters

Actual parameters are those parameters which appear in function calls and are used to send data to formal parameters. These are also called *actual arguments*. Consider the following function call:

```
foo(6); // 6 is the argument passed to parameter x
foo(y+1); // the value of (y+1) is the argument passed to parameter x
```


The actual parameters can be fixed values or variables holding values or expressions resulting in some values.

In the function **main,** in the following example, **rice, chicken** and **fruit** are actual parameters when used to call **calculate_bill**. On the other hand, the corresponding variables in calculate_bill (namely **diner1, diner2** and **diner3**, respectively) are formal parameters because they appear in the function definition and receive values passed in the function call.

Let's look at calculate_bill function in the following program:

```
/* formal and actual parameters program */
#include <stdio.h>
#include<conio.h>
#include <iostream.h>
int calculate_bill (int, int, int);
int main()
{
int bill;
int rice = 25;
int chicken = 32;
int fruit = 27;
bill = calculate_bill (rice, chicken, fruit);
cout<<"The total bill comes to "<<bill<< "Rs.";
getch();
return 0;
}
int calculate_bill (int diner1, int diner2, int diner3)
{
```

Actual Parameters

Formal Parameters

```
int total;
total = diner1 + diner2 + diner3;
return total;
}
```

Output of the Program

The total bill comes to 84 Rs.

Although, formal parameters are always variables but actual parameters may not be variables. Numbers, expressions, or even function calls can also be used as actual parameters. Here are some examples of valid actual parameters in the function call to calculate_bill:

```
bill = calculate_bill (25, 32, 27);
bill = calculate_bill (50+60, 25*2, 100-75);
bill = calculate_bill (rice, chicken, (int) sqrt(27));
```

6.1.7 Local and Global functions

The terms local and global functions specify the scope of functions within programs. Based on the scope of the functions, functions are categorized into two types; ***local functions*** and ***global functions***.

a. Local functions

Those functions which are defined inside the body of another function are called local function. Normally, we use *built-in* function inside the body of *main ()* function to perform our activities. Such declarations are termed as local.

b. Global functions

A function declared outside any function is called *global function*. A *global function* can be accessed from any part of the program. Normally, *user-defined* functions are considered as *global function* if they are defined before the *main ()* function and are thus accessible to every part of the program.

6.1.8 inline function

Using function calls in program, a lot of time is wasted in passing control from the calling function to the called function and returning control back to the calling function. This limitation can be overcome by the use of **inline** function.

C++ offers a way to combine the advantages of functions with the speed of code written in-place with the help of **inline functions**. In inline function, the function return type is preceded by the **inline** keyword which requests the compiler to treat the function as an inline and do not jump again and again to the called function. In the case of inline function, when the compiler compiles the code, all inline functions are expanded in-place, that is, the function call is replaced with a copy of the code of the function itself, which removes the function call overhead.

The disadvantage of the *inline function* is that it can make the compiled code quite larger, especially if the inline function is long and/or there are many calls to the inline function.

The format for the declaration of inline function is given in the following segment.

```
inline type name ( arguments ... )
{
Statements;
}
```

Calling an inline function is simple and is just like the call to any other function. In the call, the **inline** keyword is not needed to be included.

The following program demonstrates the use of inline function in a program.

```
/* use of inline function to find minimum out of two integers*/
#include<iostream.h>
#include<conio.h>
inline int min(int a, int b)
{
return a > b ? b : a;
}
int main()
{
cout <<"Minimum out of 25 and 26 is: "<<min(25, 26);
cout <<"\nMinimum out of 23 and 22 is:"<<min(23,22);
getch() ;
return 0;
}
```

Output of the program

Minimum out of 25 and 26 is: **25**

Minimum out of 23 and 22 is: **22**

6.2 Passing arguments and returning values

When we want to execute functions, we need to pass arguments (values) to a function. The result (values) produced within the function body is then returned back to the calling program. The following section describes different methods used for passing arguments (values) to functions.

6.2.1 Passing arguments

The following are the most commonly used methods of passing arguments to functions.

- Passing arguments by constants
- Passing arguments by values
- Passing arguments by reference

a. Passing arguments by constants

While calling a function, arguments are passed to the calling function. In passing constants as arguments, the actual constant values are passed instead of passing the variables.

Consider the following program:

```
// passing integer constant as argument.
#include<iostream.h>
#include<conio.h>
void display(int x)
{
cout << " x= " << x << endl;
}.
int main()
{
display(25); // function call
getch();
return 0;
}
```

Output of the program

x = 25

In the above program, the function call, *display (5)*, passes the constant argument (5) to the function.

Consider another program defining a function *displayGender ()* that takes a character constant, 'F' or 'M', as argument from the calling function:

```
// passing character constants as argument
#include<iostream.h>
#include<conio.h>
void displayGender(char ch)
{
cout << "The gender marker you passed is : " << ch << endl;
}
int main()
{
displayGender('F'); // function call
getch();
return 0;
}
```

Output of the program

The gender marker you passed is : F

b. Passing arguments by values

By default, arguments in C++ are passed by value. When arguments are **passed by value**, a copy of the argument value is passed to the function formal parameter.

Consider the following program to demonstrate the use of passing arguments by value.


```
// passing parameters by value
#include<iostream.h>
#include<conio.h>
void foo(int y)
{
cout << "y in foo() = " << y << endl;
y = 6;
cout << "y in foo() = " << y << endl;
} // y is destroyed here
int main()
{
int x = 5;
cout << "x in main() before call= " << x << endl;
foo(x); //argument passed by value
cout << "x in main() after call= " << x << endl;
getch();
return 0;
}
```

Output of the program

```
x in main() before call=5
y in foo() = 5
y in foo() = 6
x in main() after call=5
```

In this program, the original value of x is not changed before and after calling the functi*foo()*.

Consider another example that uses a function **addition** that gets arguments by values.

```
// passing parameters by value
#include <iostream.h>
#include <conio.h>
int addition (int a, int b)
{
int result;
result=a+b;
return (result);
}
int main()
{
int z, x=5, y=3;
z = addition (x, y); //arguments passed by value
cout << "The sum of both the values is =" << z;
getch();
return 0;
}
```

Output of the program

```
The sum of both the values is =8
```

c. Passing arguments by reference

In **pass by reference**, the reference to the function parameters are passed as arguments rather than value of variables. Using pass by reference, the value of arguments can be changed within the function.

In passing arguments by reference, the formal parameters should precede by the ampersand sign &. Consider the following example.

```
#include<iostream.h>
#include<conio.h>
void AddOne(int &y)
{
y++; // changing values in function
}
int main()
{
int x = 55;
cout<<"x in main() before call= " << x << endl;
AddOne(x); //passing arguments by reference
cout<<"x in main() after call = " << x << endl;
getch();
return 0;
}
```

Output of the program

```
x in main() before call= 55
x in main() after call = 56
```

In the above example, the value of x is passed by reference and changed inside the function **AddOne().** This change affects the values of x in **main()** because the formal argument **&y** in the function declarator is a reference to x and any change to y results in change in x.

Consider another example having a function named **duplicate** that receive arguments by reference:

```
#include <iostream.h>
#include<conio.h>
void duplicate (int& a, int& b, int& c)
{
a=a*2;
b=b*2;
c=c*2;
}
int main()
{
int x=1, y=3, z=7;
cout<<"values of x, y and z in main()before calling function\n";
cout << "x=" << x << ", y=" << y << ", z=" << z<<endl;
duplicate (x, y, z);
cout<<"values of x, y and z in main() after calling function\n";
cout << "x=" << x << ", y=" << y << ", z=" << z;
getch() ;
return 0;}
```

Output of the program

```
values of x, y and z in main()before calling function
x=1,y=3,z=7
values of x, y and z in main() after calling function
x=2, y=6, z=14
```


Sometimes we need a function to return multiple values. However, functions can only return one value. One way to return multiple values is the use of the method of passing arguments by reference.

Passing arguments by this method allow programmers to change the value of the arguments, which is sometimes useful. Similarly, it is a fast approach to passing and processing values in a function and also has the facility of returning multiple values from a function.

6.2.2 Default arguments

When declaring a function we can specify a default value for each of the last parameters. This value will be used if the corresponding argument is left blank when calling to the function. To do this, we simply have to use the assignment operator and a value for the arguments in the function declaration. If a value for that parameter is not passed when the function is called, the default value is used, but if a value is specified this default value is ignored and the passed value is used instead.

To demonstrate the concept of default arguments, consider the following general syntax.

Type function_name (parameter1=value, ...);

Here in this line of code, the *type* is any valid data type, *function_name* is the name of the function and *parameter1* is the parameter having a *default value* named *value* assigned to it in the prototype. In the example given below, parameter b has a default value 2 assigned to it in the declaration. Now, if, one value is passed in the function call then the default value of b will be taken as the value of parameter 'b'.

int divide (int a, int b=2)

The following program demonstrates the concept of default arguments.

```
#include <iostream.h>
#include<conio.h>
int divide (int a, int b=2)
{
int r;
r=a/b;
return (r);
}
int main()
{
cout<<"Result by providing one argument="
<< divide (12); // function call with one arguments
cout<< endl;
cout<<"Result by providing both the
arguments=" <<divide (20,4);
getch();
return 0;
}
```

Output of the program

```
Result by providing one argument=6
Result by providing both the arguments=5
```

As we can see in the body of the program there are two calls to function *divide()*. In the first one, we have only passed one argument, but the function **divide()** gets the second value from the default argument, int b=2, in the function prototype and thus results is 6.

In the second call, there are two parameters passed, so the default value for b is ignored and b takes the value passed as argument, that is 4, making the result returned equal to 5.

6.2.3 return statement

If the return type of a function is any valid data type such as *int, long, float, double, long double* or *char* then *return* statement should be used in the function body to return the result to the calling program.

The general syntax of the use of *return* statement is given below.

return (expression or variable holding results or constant);

Consider the following function:

```
// the use of return statement in the cube() function
int cube(int x)
{
int c;
c= x*x*x;
return c; // or return (c);
}
```

If a function returns a value by the use of *return* statement then the call to the function should be from a statement or an expression that utilizes it.
For example:

```
cout<<cube(x);
```

```
int y=cube(x);
```

6.3 Function overloading

6.3.1 Definition

In programming languages, two or more variables or functions with the same name cannot be used in a single program or block of code. **Function overloading** is a feature of C++ that allows us to create multiple functions with the same name, so long as they have different number or types of parameters.

Consider the following example having two functions with the same name *display* used to perform slightly different tasks in over loaded form.

```
#include <iostream.h>
#include<conio.h>
void display (int a)
{
cout<<a;
}
void display()
{
cout<< "Welcome"<<endl;
}
int main()
{
display() ;
display(55) ;
getch();
return 0;
```



```
}
```

Output of the program

```
Welcome
55
```

Here, the function *display* is overloaded with two tasks: one for displaying an integer value and the other for displaying a welcome message. While calling the overloaded functions, the compiler decides at run time that which *display* function should be called. To determine which *display* to call is decided by looking at the number and types of the arguments or parameters of the functions. In this example, *void display ()* is called for display () and *void display (int a)* for *display (55)*.

Consider another program having two functions with the same name *multiply* that operate on integers and floating points numbers.

```
#include <iostream.h>
#include<conio.h>
int multiply (int a, int b)
{
return (a*b);
}
float multiply (float a, float b)
{
return (a*b);
}
int main()
{
cout << "Output of integers="<< multiply (5, 33)<<endl;
```

```
cout << "Output of floats="<< multiply (2.1, 3.0);
getch();
return 0;
}
```

Output of the program

Output of integers=165
Output of floats=6.3

In this case, we have defined two functions with the same name, ***multiply***, that accept two arguments. One accepts two arguments of type **int** and the second of type **float**. The compiler looks at the arguments and calls the respective function. Here, ***multiply*** *(int a, int b)* is called for ***multiply*** *(5, 33)* because the arguments match with the data types of the formal parameters. Similarly, ***multiply*** *(float a, float b)* is called for the call ***multiply*** *(2.1, 3.0)* because the arguments and formal parameters match each other in types.

6.3.2 Advantages of function overloading

- Function overloading can significantly reduce complexity of programs.
- Using function overloading, we can declare multiple functions with the same name that have slightly different purposes.
- As multiple functions have same name, therefore, remembering them is easier as compared to remembering more names.
- It increases the readability of programs.
- It exhibits the behavior of polymorphism.

6.3.3 Understanding function overloading concept

For the complete understanding of the concept of function overloading, one should know those features of the overloaded functions which disambiguate them and make them unique in a single program. These features are listed below:

- Number of arguments
- Data types of arguments
- Return type

a. Number of arguments

Functions can be overloaded if they have different numbers of parameters. More than one functions with the same name but different number of parameters can be used in a single program. In the calls to these functions, the compiler disambiguates the calls by looking at the number of arguments and formal parameters in the function decelerators. Consider the following example:

```
/* overloaded functions with different number of parameters */
#include <iostream.h>
#include <conio.h>
int Addition(int X, int Y)
{
return (X + Y);
}
int Addition(int X, int Y, int Z)
{
return (X + Y + Z);
}
int main()
{
int a=55, b=22, c=100;
cout << "Output of 2 integers="<<Addition (a,b);
cout << endl;
cout << "Output of 3 integers="<<Addition (a,b,c);
cout << endl;
getch();
```

```
return 0;
}
```

Output of the program

```
Output of 2 integers=77
Output of 3 integers=177
```

In this example, two functions have been used with the name *Addition*. One has two parameters and the second has three parameters. In the *main()* function there are two calls *Addition (a,b)* and *Addition (a,b,c)* which call functions *int Addition(int X, int Y)* and *int Addition(int X, int Y, int Z)* respectively by looking at the number of parameters.

b. Data types of arguments

One way to achieve function overloading is to define multiple functions with the same name but different types of parameters. Consider the following example:

```
/*overloaded functions print()with different types of
parameters*/
#include <iostream.h>
#include <conio.h>
void print (int X)
{
cout<< "X="<<X<<endl;
}
void print (float Y)
{
cout<< "Y="<<Y<<endl;
}
int main()
{
print(55);
print(33.66);
getch();
return 0;
}
```


Output of the program

```
X=55
Y=33.66
```

Here, two functions, *print (int X)* and *print (float Y)*, have been used with different types of parameters, *int* and *float*. In *main()* function, two calls are used *print(55)* and *print(33.66)*. The compiler looks at the type of arguments and calls the corresponding function.

c. Return type

The return type of a function is not considered when functions are overloaded. It means that if two or more functions with the same name have same function signatures but different return types then they are not considered as overloaded and the compiler will generate error message.

Consider the following case:

```
int display();
double display(); // This prototype generates error message
```

Consider another example:

```
int RandomNumber();
double RandomNumber(); // This prototype generates error message
```

Here, the compiler generates an error message of re-declaration for the second declaration at line number 2. It is because that both the declarations have same number of parameters (zero in this case) but only the return types are different which do not play any role in the overloading. If we want to do it, for then, these functions will need to be given different names.

Summary

- A function is a self-contained program that performs a specific task.
- C++ has two types of functions, built-in and user-defined. Built-in functions are specific in their activities and cannot be used for general type of tasks.
- Every C++ program comprise of one function called main (). It is the point from where the compiler starts the execution of every C++ program.
- A good C++ programmer writes programs that comprise of small functions.
- Functions are one of the main building blocks of C++ programs. It modularizes large programs into segments and thus increase the program readability and also provides the facility of code reusability.
- Each function has its own name and when the function is called the execution of the program branches to the body of that function. When the function is finished, execution returns to the area of the program code from which it was called, and the program continues on to the next line of code.
- Function prototype tells the compiler the name of the function, number, types and order of parameters.
- A function definition is a set of instructions enclosed in a block which performs the intended task of the function.
- Local/Automatic variables have local scope and can only be accessed in the block in which they are declared. These variables cannot be accessed from outside the block.
- Global variables have global access and their visibility is throughout the program in which they are defined.

- Static variables have scope of local variables and retain its values throughout the life of the program.
- The variables that appear in the function prototype and function declaration are called formal parameters.
- The variables in function calls that hold the values to be passed to the function are called actual parameters.
- inline functions are special functions with inline keyword that are used to minimize the problem of function call overhead with the expenses of memory wastage.
- C++ provides the facility of calling a function with fewer arguments with the help of default arguments.
- The phenomenon of using same function name for related but slightly different tasks is called function overloading.
- In overloaded function, the function signature, number, types and order of arguments should be different from each other.

Exercise

Q.1 Fill in the Blanks.

i. Normally a function can return _____ value.

ii. The only difference between the function declarator and function prototype is the use of _____.

iii. The scope of global variable declared in a C++ program, is _____.

iv. Using _____, we can declare multiple functions with the same name, having slightly different purpose.

v. If a function needs to return a value to the calling function then its return type should not use the keyword _____.

Q.2 Select the correct choice for the following Multiple Choice Questions.

i. The phenomenon of having two or more functions in a program with the same names but different number and types of parameters is known as:
   a. Inline Function
   b. Nested Function
   c. Function overloading
   d. Recursive Function

ii. We declare a function with _____ if it does not have any return type
   a. long
   b. double
   c. void
   d. int

iii. Arguments of a functions are separated with_____.
   a. Comma (,)
   b. Semicolon (;)
   c. Colon (:)
   d. None of these

iv. Variables inside parenthesis of functions declarations have _____ level access.
   a. Local
   b. Global
   c. Module
   d. Universal

v. Observe the following function declaration and choose the best answer:
   int divide (int a, int b = 2)
   a. Variable b is of integer type and will always have value 2
   b. Variable a and b are of int type and the initial value of both variables is 2
   c. Variable b is international scope and will have value 2
   d. Variable b will have value 2 if not specified when calling function

Q.3 Write TRUE/FALSE against the following statements.

i. A function cannot be overloaded only by its return type.

ii. A function can be overloaded with a different return type if it has all the parameters same.

iii. Inline functions minimize the size of the program.

iv. If we have a function: int sum (int x, int y), then it will return float value to the calling program.

v. Passing arguments by address allows the function to change the value of the argument.

Q.4 Write a program with a function that takes two *int* parameters, adds them together, and then returns the sum.

Q.5 Write a program with a function name "**mean**" to read in three integers from the keyboard to find the arithmetic mean.

Q.6 Write a C++ program having a function name **rectangle** to read the length and width of a rectangle from the keyboard and find the area of the rectangle. The result should be returned to the *main* program for displaying on the screen.