DEPARTMENT OF COMPUTER APPLICATIONS
TRICHY-20

B.C.A
OBJECT ORIENTED PROGRAMMING WITH C++

M.BALAKRISHNAN MCA, M.Phil.
DEPARTMENT OF CA
JMC, TRICHY-20

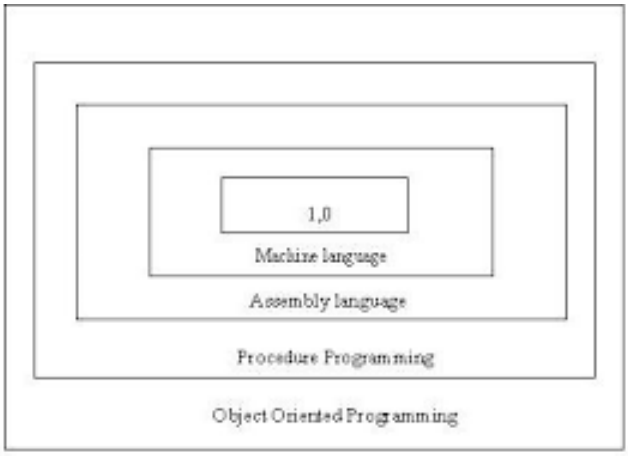
UNIT I

PRINCIPLES OF OBJECT-ORIENTED PROGRAMMING

What is C++?

C++ is an object-oriented programming language. It was developed by Bjarne Stroustrup at AT&T Bell Laboratories in Murray Hill, New Jersey, USA, in the early 1980's.

Ernest Tello, a well-known writer in the field of artificial intelligence, compared the evolution of software technology to the growth of a tree. Like a tree, the software evolution has had distinct phases or "layers" of growth. These layers were built up one by one over the last five decades as shown in fig, with each layer representing an improvement over the previous one.



Object-Oriented Programming (OOP) as an approach that provides a way of modularizing programs by creating partitioned memory area for both data and functions that can be used as templates for creating copies of such modules on demand.

Basic concepts of Object-Oriented Programming

- Objects
- Classes
- Data abstraction and encapsulation
- Inheritance
- Polymorphism
- Dynamic binding
- Message passing

Objects

Objects are the basic run-time entities in an object-oriented system. They may represent a person, a place, a bank account, a table of data or any item that the program has to handle. They may also represent user-defined data such as vectors, time and lists

Classes

A class is a collection of objects of similar type. For example, mango, apple and orange are members of the class fruit.

Classes are user-defined data types and behave like the built-in types of a programming language. If fruit has been defined as a class, then the statement

```
fruit mango;
```

Will create an object **mango** belonging to the class **fruit**.

Data Abstraction and Encapsulation

The wrapping up of data and functions into a single unit (called class) is known as *encapsulation.* Data encapsulation is the most striking feature of a class. The data is not accessible to the outside world, and only those functions which are wrapped in the class can access it. This insulation of the data from direct access by the program is called *data hiding or information hiding*.

Abstraction refers to the act of representing essential features without including the background details or explanations.

Inheritance

Inheritance is the process by which objects of one class acquire the properties of objects of another class. In OOP, the concept of inheritance provides the idea of *reusability*. This means that we can add additional features to an existing class without modifying it.

Polymorphism

Polymorphism is another important OOP concept. Polymorphism, a Greek term, means the ability to take more than one form. An operations may exhibit different behaviours in different instances. The process of making an operator to exhibit different behaviours in different instances is known as *operator overloading*.

A single function name can be used to handle different number and different types of arguments. Using a single function name to perform different types of tasks is known as *function overloading*.

Dynamic Binding

Binding refers to the linking of a procedure call to the code to be executed in response to the call. *Dynamic binding* (also known as late binding) means that the code associated with a given procedure call is not known until the time of the call at run-time.

Message passing

An object-oriented program consists of a set of objects that communicate with each other. The following basic steps:

1. Creating classis that define objects and their behaviour,
2. Creating objects from class definitions, and
3. Establishing communication among objects.

Benefits of OOP

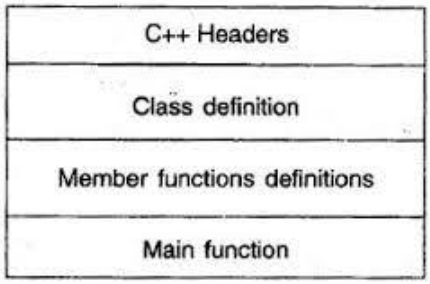
- Through inheritance, we can eliminate redundant code and extend the use of existing classes.
- We can build programs from the standard working modules that communicate with one another, rather than having to start writing the code from scratch. This leads to saving of development time and higher productivity.
- The principle of data hiding helps the programmer to build secure prorams that cannot be invaded by code in other parts of the program.
- It is possible to have multiple instances of an object to co-exist without any interference.
- It is possible to map objects in the problem domain to those in the program.
- It is easy to partition the work in aa project based on ojects.
- The data-centered design approach enable us to capture more details of a model in implemenable form.
- Object-oriented systems can be easily upgraded from small to large systems.
- Message passing techniques for communication between objects makes thee interface descriptions with external systems much simpler.
- Software complexity can be easily managed.

Applications of OOP

- ✓ Real-time systems
- ✓ Simulation and modeling
- ✓ Object-oriented databases
- ✓ Hypertext, hypermedia and expertext
- ✓ AI and expert systems
- ✓ Neural networks and parallel programming
- ✓ Decision support and office automation systems
- ✓ CIM/CAM/CAD systems

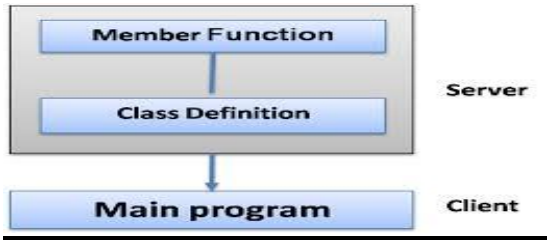
STRUCTURE OF C++ PROGRAM

C++ program would contain four sections as shown. These sections may be placed in separate code files and then compiled independently or jointly. It is a common practice to organize a program into three separate files. The class declarations are placed in a header file and the definitions of member functions go into another file.



Structure of a C++ Program

This approach is based on the concept of client-server model as shown in fig. The class definition including the member functions constitute the server that provides services to the main program known as client. The client uses the server through the public interface of the class.



TOKENS:

- The smallest individual units in a program are known as tokens.
  c++ has the following tokens:
- Keywords
- Identifiers
- Constants
- Strings
- Operators

A C++ program is written using these tokens, white spaces, ant the syntax of the language.

KEYWORDS:

A keyword is a reserved word. You cannot use it as a variable name, constant name etc. **A list of 32 Keywords in C++ Language which are also available in C language are given below.**

auto	break	case	char	const	continue	default	do
double	else	enum	extern	float	for	goto	if

int	long	register	return	short	signed	sizeof	static
struct	switch	typedef	union	unsigned	void	volatile	while

IDENTIFIERS:

- Identifiers refer to the names of variable, functions, arrays, classes, etc. created by the programmer.

The following rules are common to both C and C++:

- Only alphabetic characters, digits and underscores are permitted.
- The name cannot start with a digit.
- Uppercase and lowercase letter are distinct.
- A declared keyword cannot be used as a variable name.

Example:

auto, break, case, char, class, const, continue, delete, new, private, protected, this, throw.

CONSTANTS:

Constants refer to fixed values that do not change during the execution of a program.

Like C, C++ supports several kinds of literal constants. They include integers, characters, floating point numbers and strings. Literal constant do not have memory locations.

Example:

123 // Decimal integer.

12.34 // Floating point integer

The **wchar_t** type is a wide-character literal introduced by ANSI C++ and is intended for character sets that cannot fit a character into single byte. Wide-character literals begin with the letter L.

BASIC DATA TYPES:

Types	Data Types
Basic Data Type	int, char, float, double, etc
Derived Data Type	array, pointer, etc
Enumeration Data Type	enum
User Defined Data Type	structure

Type	Size (Bytes)	Range
char	1	-128 to 127
unsigned char	1	0 to 255
signed char	1	-128 to 127
int	2	-32768 to 32767
unsigned int	2	0 to 65535
signed int	2	-32768 to 32767
short int	2	-32768 to 32767
unsigned short int	2	0 to 65535
signed sort int	4	-32768 to 32767
long int	4	-2147483648 to 2147483647
unsigned long int	4	0 to 4294967295
signed long int	4	-2147483648 to 2147483647
float	4	3.4E-38 to 3.4E+38
double	8	1.7E-308 to 1.7E-308
long float	10	3.4E-4932 to 1.1E+4932

USER-DEFINED DATA TYPES:

Structure and classes:

User-defined data types such as **struct** and union in C. While these data types are legal in C++, some more features have been added to make them suitable for object oriented programming.

Enumerated Data Type:

- An enumerated data type is another user-defined type which provides a way for attaching names to numbers, thereby increasing comprehensibility of the code.
- The **enum** keyword automatically enumerates a list of words by assigning them values 0, 1, 2 and so on. This facility provides an alternative means for creating symbolic constants. The syntax of an **enum** statement is similar to that of the **struct** statement.

Example:

enum shape { circle, square, triangle} ;

enum colour {red, blue, green, yellow } ;

enum position {off, on };

In C++, the tag names shape, colour, and position become new type names. By using these tag names, we can declare new variable.

Example:

Shape ellipse; // ellipse is of type shape

Colour background; // background is of type colour

- C++ does not permit an int value to be automatically converted to an enum value.

Example:

```
enum shape
{
circle,
rectangle,
triangle
} ;

int main ()
{
     cout << "enter shape code: " ;
      int code;
     cin >> code;
     while (code >= circle && code <= triangle )
{
       switch (code)
{
       case circle:
       ……..
        ……..
        break;
        case rectangle:
        …….
        ……
         break;
          case triangle:
         ……
          ……
           break;
}
cout << "bye \n";
return 0;
}
```

ANSI C permits an enum to be defined within a structure or a class, but the enum is globally visible. In C++ an enum defined within a class (or structure is local to that class(or structure) is local to that class (or structure) only.

DERIVED DATA TYPES:

Arrays:

When initializing a character array in ANSI C, the compiler will allow us to declare the array size as the exact length of the string constant. For instance,

```
Char string [3] = "xyz";
```

Is valid in ANSI C. But in C++, the size should be one larger than the number of characters in the string.

```
Char string [4] = "xyz";
```

Function:

Dividing a program into functions is one of the major principles of top-down, structured programming. Another advantage of using functions is that it is possible to reduce the size of a program by calling and using them at different places in the program.

Pointers:

Pointers are declared and initialized as in C.

Example:

```
I n t *I p ;      // int pointer
I p= &x ;    // address of x assigned to I p
*I p = 10;      // 10   assigned to x through indirection
```

- C++ adds the concept of **constant pointer and pointer to a constant.**

```
Char * const ptr1= "GOOD"       // constant pointer
```

We cannot modify the address that ptrl is initialized to.

```
int const * ptr2 = &m;  // pointer to a constant
```

Ptr2 is declared as pointer to a constant. It can point to any variable of correct type, but the contents of what it points to cannot be changed.

- We can also declare both the pointer and the variable as constants in the following ways:

```
Const char * const cp =    "xyz";
```

This statement declares cp as a constant pointer to the string which has been declared a constant.

- Pointers are extensively used in C++ for memory management and achieving polymorphism.

DECLARATION OF VARIABLES

Variables in C++ is a name given to a memory location. It is the basic unit of storage in a program.

- The value stored in a variable can be changed during program execution.
- A variable is only a name given to a memory location, all the operations done on the variable effects that memory location.
- In C++, all the variables must be declared before use.

Rules for Declaring Variable

1. The name of the variable contains **letters**, **digits**, and **underscores**.
2. The name of the variable is **case sensitive** (ex **Arr** and **arr** both are different variables).
3. The name of the variable does not contain any whitespace and **special characters** (ex **#,$,%,*,** etc).
4. All the variable names must begin with a letter of the **alphabet** or an **underscore** (_).
5. We cannot used **C++ keyword** (ex float, double,class)as a variable name.

Examples:

```
// Declaring float variable

float simpleInterest;

// Declaring integer variable

int time, speed;
```

// Declaring character variable

char name;

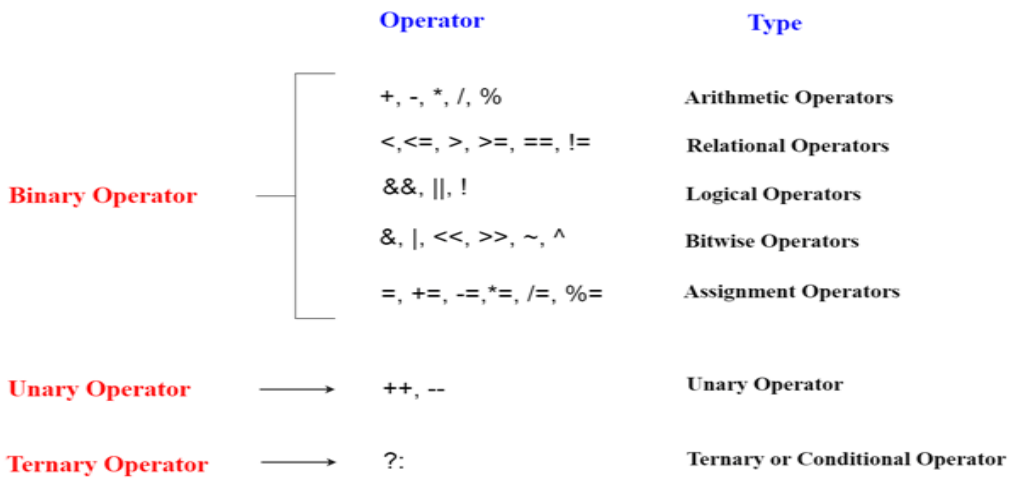
Types of Variables
There are three types of variables based on the scope of variables in C++

- **Local Variables**
- **Instance Variables**
- **Static Variables**

OPERATORS IN C++:

An operator is simply a symbol that is used to perform operations. There can be many types of operations like arithmetic, logical, bitwise etc.
There are following types of operators to perform different types of operations in C language.

- **Arithmetic Operators**
- **Relational Operators**
- **Logical Operators**
- **Bitwise Operators**
- **Assignment Operator**
- **Unary operator**
- **Ternary or Conditional Operator**
- **Misc Operator**



ARITHMETIC OPERATORS
The arithmetic operators are used to perform the arithmetic operations on the operands. The operations can be addition, multiplication, subtraction and division.

RELATIONAL OPERATORS
The relational operators are those operators that are used to compare the values of two operands.

LOGICAL OPERATORS

The logical operators are those operators that are used to combine two or more conditions. The logical operators are AND (&&) and OR (||).

ASSIGNMENT OPERATORS

The assignment operators are those operators which are used to assign value to a variable. On the left side of the assignment operator the operand is a variable and the right side of the operator the operand is a value.

BITWISE OPERATORS

The bitwise operators are those are used to perform bit level operations on the operands.

SCOPE RESOLUTION OPERATOR:

The scope resolution operator is use for the Unary scope operator , if a namespace scope (or) global Scope name is hidden by an explicit declaration of the Name in block or class.

```
int  count=0;
int main(void)
{
int  count=n;
::count=1;
count=2;
return 0;
}
```

The declaration of count is declared in the main function Hides the integer named count declared in global namespace scope.The statement :: count =1 accesses the variable named Count declared in global namespace scope.

SCOPE RESOLUTION OPERATOR IN C++:

The scope resolution operator (::) in c++ used to Define the already declared in the member functions of the class.

C++ supports to the global variable from a function,Local variable is to defined in the same function name.

The syntax of the scope resolution operator:

:: global variable name

Resolution operator is placed between the front of the variable name then the global variable is affected.If no resolution operator is placed between the local variable is affected.

Example:

```
#include<iostream.h>
int n=12;              //global variable
int main()
{
int n=13;              //local variable
cout<<::n<<endl;       //print global variable:12
cout<<n<<endl;         //print the local variable:13
}
```

If the resolution operator is placed between the class name and the data member belonging the class then data name belonging to the particularly class is affected.

If it is place front of the variable name then the global variable is affected. It is no resolution operator is placed then the local variable is affected.

TYPE CAST OPERATOR:

Converting an expression of a given type into another type is known as *type-casting*. We have already seen some ways to type cast:

Implicit conversion

Implicit conversions do not require any operator. They are automatically performed when a value is copied to a compatible type. For example:

```
short a=2000;
int b;
b=a;
```

Implicit conversions also include constructor or operator conversions, which affect classes that include specific constructors or operator functions to perform conversions. For example:

```
class A {};
class B { public: B (A a) {} };

A a;
B b=a
```

Explicit conversion

C++ is a strong-typed language. Many conversions, specially those that imply a different interpretation of the value, require an explicit conversion. We have already seen two notations for explicit type conversion: functional and c-like casting:

```
short a=2000;
int b;
b = (int) a;    // c-like cast notation
b = int (a);    // functional notation
```

dynamic_cast

dynamic_cast can be used only with pointers and references to objects. Its purpose is to ensure that the result of the type conversion is a valid complete object of the requested class.

Therefore, dynamic_cast is always successful when we cast a class to one of its base classes:

```
class CBase { };
```

```
class CDerived: public CBase { };

CBase b; CBase* pb;
CDerived d; CDerived* pd;

pb = dynamic_cast<CBase*>(&d);     // ok: derived-to-base
pd = dynamic_cast<CDerived*>(&b);  // wrong: base-to-derived
```

MANIPULATORS:

Manipulator are operator that are used to format the data display. The most commonly used manipulator are **endl** and **stew**.

The **endl** manipulator,when used in an output statement ,causes a linefeed to be inserted . It has the same effect as using the newline character "\n". For example, the statement

```
……
……
Cout <<"m="<<m<<endl
      <<"n="<<n<<endl
      <<"p="<<p<<endl;
……
……
```

Program illustrates the use of endl and setw.

```
#include <iostream.h>
    #include <iomanip.h>
    void main (void)
    {
       int  a,b;
       a = 200;
       b = 300;
      cout << setw (5) << a << setw (5) << b << endl;
      cout << setw (6) << a << setw (6) << b << endl;
      cout << setw (7) << a << setw (7) << b << endl;
      cout << setw (8) << a << setw (8) << b << endl;
    }
```

Output of the above program

```
200     300
200     300
200     300
200     300
```

EXPRESSIONS AND THEIR TYPES:

An expression is a combination of operators, constants and variables arranged as per the rules of the language.it may also include function calls which return values.an expression may consist of one or more operands,and zero or more operators to produce a value.

Expressions may be of the following seven types:

- constant expressions
- integral expression
- float expression
- pointer expression
- relation expression
- logical expression
- bitwise expression

An expression may also use combinations of the above expressions. Such expressions are known as compound expressions.

1.Constant expressions

Constant expressions consists of only constant values.

Example :

15

20+5/2.0

'x'

2.Integral expressions

Integral expressions are those which produce integer results after implementing all the automatic and explicit type conversation.

Example:

m

M*n-5

M*'x'

5+int(2.0)

Where m and n are integer variables.

3.Float expressions

FLOAT EXPRESSIONS are those which,after all conversions, produce floating-point results.

Examples:

X+Y

X*Y/10

5+float(10)

10.75

Where x and y are floating-point variables.

4. Pointer Expressions

Pointer expressions produce address values.

Example:

&m

Ptr

Ptr+1

"xyz"

Where m is a variable and ptr is a pointer.

5. **Relational Expressions**

Relational expressions are yield results of type **bool** which takes a value **true** and **false** .

Example:

X<=y

a+b==c+d

m+n> 100

when arithmetic expressions are used on either side of a relational operator,they will be evaluated first and then the results compared . Relational expressions are also known as Boolean expressions .

6. Logical expressions

Logical expressions combine two or more relational expressions and produces **bool type** results .

Examples :

a>b && x==10

x==10 || y==5

7. Bitwise expressions

Bitwise expressions are used to manipulate data at bit level . They are basically used for testing or shifting bits.

Example:

X<<3 || shift three bit position to left

y>>1 || shift one bit position to right

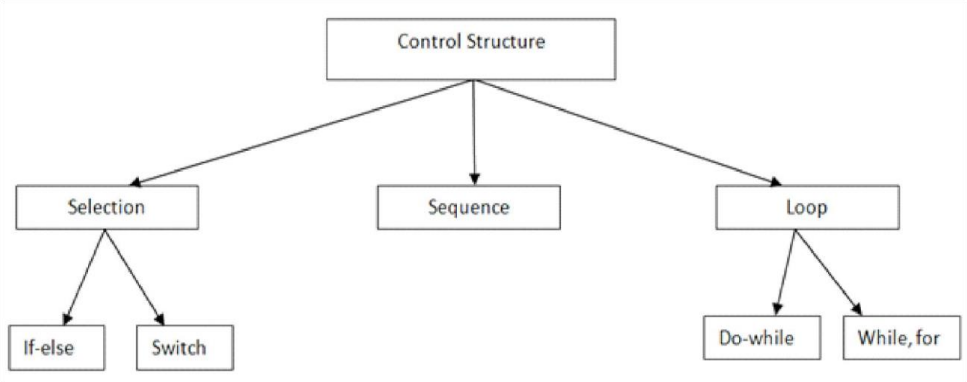
shift operator are often used for multiplication and division by power of two.

CONTROL STRUCTURES:

The following three control structures:

1. sequence structure(straight line)
2. selection structure(branching)
3. loop structure(iteration or repetition)

The following figure shows how these structures are implemented using one –entry,one-exit concept, a popular approach used in modular programming .



It is important to understand that all programs can be coded by using only these basic control constructs in programming is known as structured programming, an important technique in software engineering.

IF Statement

The C++ if statement tests the condition. It is executed if condition is true.

```
if(condition)
{
//code to be executed
}
```

EX:

```
#include <iostream>
using namespace std;
int main ()
{
  int num = 10;
        if (num % 2 == 0)
        {
           cout<<"It is even number";
         }
       return 0;
}
```

Output:

```
It is even number
```

If. Else statement:

The C++ if-else statement also tests the condition. It executes if block if condition is true otherwise else block is executed.

```
if(condition)
{
//code if condition is true
}
else
```

EX:

```
    #include <iostream>
    using namespace std;
int main ()
{
    int num = 11;
if (num % 2 == 0)
    {
    cout<<"It is even number";
    }
    else
    {
```

```
    {
//code if condition is false
}
```

For loop:

The C++ for loop is used to iterate a part of the program several times. If the number of iteration is fixed, it is recommended to use for loop than while or do-while loops.

```
    for(initialization; condition; incr/decr)
    {
//code to be executed
}
```

EX:

```
#include <iostream>
using namespace std;
    int main()
     {
    for(int i=1;i<=10;i++)
    {
    cout<<i <<"\n";
}
}
```

While loop:

C++, while loop is used to iterate a part of the program several times. If the number of iteration is not fixed, it is recommended to use while loop than for loop.

```
while(condition)
{
//code to be executed
}
```

EX:

```
    #include <iostream>
    using namespace std;
    int main()
    {
    int i=1;
    while(i<=10)
    {
    cout<<i <<"\n";
i++;
}
}
```

Do..While loop:

The C++ do-while loop is executed at least once because condition is checked after loop body.

do

```
    {
    //code to be executed
    }
    while(condition);
```

EX:

```
    #include <iostream>
    using namespace std;
    int main()
    {
    int i = 1;
    do{
    cout<<i<<"\n";
    i++;
    }
    while (i <= 10) ;
}
```

Switc statement:

The C++ switch statement executes one statement from multiple conditions. It is like if-else-if ladder statement in C++.

```
switch(expression)
{
case value1:
//code to be executed;
    break;
case value2:
//code to be executed;
break;
.....
default:
    //code to be executed if all cases are not matched;
    break;
    }
```

```
#include <iostream>
      using namespace std;
      int main ()
      {
           int num;
     cout<<"Enter a number to check grade:";
     cin>>num;
        switch (num)
            {
               case 10: cout<<"It is 10"; break;

               case 20: cout<<"It is 20"; break;

              case 30: cout<<"It is 30"; break;

default: cout<<"Not 10, 20 or 30"; break;
        }
  }
```

UNIT -2

FUNCTION:

- A function is a group of statements that together perform a task. Every C++ program has at least one function, which is **main()**, and all the most trivial programs can define additional functions.
- A function **declaration** tells the compiler about a function's name, return type, and parameters. A function **definition** provides the actual body of the function.

The Main function:

A program shall contain a global function named **main**, which is the designated start of the program.

int **main () {** body **}** (1)

int **main (**int argc**,** char *argv[]**) {** body **}** (2)

int **main (**int argc**,** char *argv[] **,** other_parameters **) {** body **}** (3)

argc - Non-negative value representing the number of arguments passed to the program from the environment in which the program is run.

argv - Pointer to the first element of an array of pointers to null-terminated multibyte strings that represent the arguments passed to the program from the execution environment (argv[0] through argv[argc-1]). The value of argv[argc] is guaranteed to be 0.

body - The body of the main function

other_parameters - Implementations may allow additional forms of the main function as long as the return type remains int. A very common extension is passing a third argument of type char*[] pointing at an array of pointers to the execution environment variables.

The names argc and argv are arbitrary, as well as the representation of the types of the parameters: int main(int ac, char** av) is equally valid.

Functions in C++

Dividing a program into functions is one of the major principles of top-down, structured programming. Another advantages of using functions is the it is possible to reduce the size of a program by calling and using them at different places in the program.

```
void show();  /*Function declaration*/
main()
{
…….....
show();    /*Function call*/
…….....
}
void show()   /*Function definition*/
{
…….....
…….....   /*Function body*/
…….....
}
```

The Main Function

In C++, the main() returns a value of type int to the operating system. C++, therefore, explicitly defines main() as matching one of the following prototypes:

```
int main();

int main(int argc,char*argv[]);
```

The functions that have a return value should use the return statement for termination. The main() function

in C++ is, therefore, defined as follows:

```
int main()
{
….....
….....
return 0;
}
```

Since the return type of functions is int by default, the keyword int in the main() header is optional.

CALL BY REFERENCE

In call by reference, original value is modified because we pass reference (address).A address of the value is passed in the function, so actual and formal arguments share the same address space. Hence, value changed inside the function, is reflected inside as well as outside the function.

```
#include<iostream>
    using namespace std;
    void swap(int *x, int *y)
    {
     int swap;
     swap=*x;
     *x=*y;
     *y=swap;
    }
    int main()
    {
     int x=500, y=100;
     swap(&x, &y);  // passing value to function
     cout<<"Value of x is: "<<x<<endl;
     cout<<"Value of y is: "<<y<<endl;
     return 0;
    }
```

Output:

```
Value of x is: 100
Value of y is: 500
```

INLINE FUNCTION

To eliminate the costs of calls to small functions , c++ proposes a new feature called **inline function .** An inline function is a function that is expanded in line when it is invoked . That is , the compiler replaces the functions call with the corresponding function code.

The inline functions are defined as follows:

```
Inline  function-header
    {
        Function body
```

}

Example:

```
inline double  cube(double a)
{
    Return (a*a*a);
}
```

Program :

```
#include<iostream>
using namespace std;
inline float mul(float x, float y)
{
      return(x*y)
}
inline double div(double p, double q)
{
return(p/q)
}
int main()
{
float a=12.345;
float b=9.82;
cout<<mul(a,b)<<"\n";
cout<<div(a,b)<<"\n";
return 0;
}
```

The out of the program would be

121.228

1.25713

DEFUALT ARGUMENTS

C++ allows us to call functions without specifying all its argument . the functions assigns a default value to the parameter which does not have a matching argument in the function call . Default values are specified when the functions is declared .Here is an example of a prototype (i.e.function declared) with default values:

```
Float amount (float principle ,int period ,float rate =0.15);
```

The above prototype declares a default value of 0.15 to the argument rate.

A default argument is checked for type at the time of the declaration and evaluated at the time of calls.

Some example of function declarations with default values are:

```
Int mul  (int i, int j=5 , int k=10);          //legal

Int mul  (int i=5,int j);                      //illegal

Int mul  (int i=0 , int j , int k=10);         //illegal

Int mul  (int i=2 , int j=5 , int k=10);       //legal
```

PROGRAM

```
#include <iostream.h>
void sum ( int a, int b, int c= 6, int d = 10);
```

```
 void main ( )
 {
    int a, b, c, d;
    cout << " enter any two numbers \n";
    cin >> a >>b;
    sum (a, b) ; / / sum of default values
 }
 void sum (int a1, int a2, int a3, int a4)
 {
    int temp;
    temp = a1 + a2 + a3 + a4;
    cout << " a = " << a1 << endl;
    cout << " b = " << a2 << endl;
    cout << " c = " << a3 << endl;
    cout << " d = " << a4 << endl;
    cout << " sum = " << temp;
 }
```

Output of the above program

```
   enter any two numbers
   11   21
   a = 11
   b = 21
   c = 6
   d = 10
  sum = 48
```

FUNCTION OVERLOADING

Overloading refers to the use of the same thing for different purposes, c++ also permits overloading of functions. This means that we can use the same functions name to create functions that perform a variety of different tasks. This is known as functions polymorphism in **OOP**.

Program to illustrate function overloading

// function volume ( ) is overloaded three times

```
# include <iostream>
Using namespace std;
                                    //  declarations (prototypes)
int volume (int);
double volume (double , int );
long volume (long ,int ,int);
int main()
{
Cout<<volume(10)<<"\n";
Cout <<volume(2.5 , 8)<<"\n";
Cout<<volume(100L , 75 , 15)<<"\n";
return 0;
}                                   //function definitions
int volume (int s)                  //cube
{
return(s*s*s);
```

```
}
double volume (double r , int h);         //cylinder
{
return(3.14519*r*r*h);
}
long volume (long l , int b , int h )    // rectangle
{
return(l*b*h);
}
```

Out of program

1000
157.26
112500

MATH LIBRARY FUNCTION:

Functions come in two varieties. They can be defined by the user or built in as part of the compiler package. As we have seen, user-defined functions have to be declared at the top of the file. Built-in functions, however, are declared in **header files** using the #include directive at the top of the program file, e.g. for common mathematical calculations we include the file cmath with the #include <cmath> directive which contains the *function prototypes*for the mathematical functions in the cmath library.

Mathematical functions

Math library functions allow the programmer to perform a number of common mathematical calculations:

Function	Description
sqrt(x)	square root
sin(x)	trigonometric sine of x (in radians)
cos(x)	trigonometric cosine of x (in radians)
tan(x)	trigonometric tangent of x (in radians)
exp(x)	exponential function
log(x)	natural logarithm of x (base e)
log10(x)	logarithm of x to base 10
fabs(x)	absolute value (unsigned)
ceil(x)	rounds x up to nearest integer
floor(x)	rounds x down to nearest integer
pow(x,y)	x raised to power y

CLASSES AND OBJECTS

SPECIFYING A CLASS

A class is a way to bind the data and its associated functions together. A class specification has two parts:

1. Class declaration
2. Class function definitions

The class declaration describes the type and scope of its members. The class function definitions describe how the class functions are implemented.

The general form of a class declaration is:

```
class class_name
{
private:
variable declaration;
function declaration;
public:
variable declaration;
function declaration;
};
```

- The keyword class specifies, that what follows is an abstract data of type class_name. The body of a class is enclosed within braces and terminated by a semicolon.
- The class body contains the declaration of variables and functions. These functions and variables are collectively called class members.
- They are usually grouped under two sections, namely, private and public to denote which of the members are private and which of them are public.
- The variables declared inside the class are known as data members and the functions are known as member functions.
- The binding of data and functions together into a single class-type variable is referred to as **encapsulation.**

A Simple Class Example

A typical class declaration would look like:

```
class item
{
int number;   //variables declaration
float cost;  //private by default
public:
void getdata(int a, float b);  //functions declaration
void putdata(void);  //using prototype

};        //ends with semicolon
```

Creating Objects

For item x; //memory for x is created

creates a variable x of type item. In C++, the class variables are known as objects. Therefore, x is called an object of type item.

Objects can also be created when a class is defined by placing their names immediately after the closing brace, as we do in the case of structures. That is to say, the definition

class item

{
.....
.....
.....
}
x,y,z;

Accessing Class Members

The following is the format for calling a member function:

object-name.function-name (actual-argument);

For example, the function call statement

x.getdata(100,75.5);

DEFINING MEMBER FUNCTIONS

Member functions can be defined in two places:

- Outside the class definition.
- Inside the class definition.

Outside the Class Definition

Member functions that are declared inside a class have to be defined seperately outside the class. **The general** form of a member function definition is:

return-type class-name::function-name (argument declaration)

{

Function body

}

The membership label class-name:: tells the compiler that the function function-name belongs to the class class-name. That is, the scope of the function is restricted to the class-name specified in the header line.

The symbol :: is called the scope resolution operator.

They may be coded as follows:

```
void item :: getdata(int a, float b)
{
number=a;
cost=b;
}
```

Inside the Class Definition

Another method of defining a member funcion is to replace the function declaration by the actual function definition inside the class. For example, we could define the item class as follows:

```
class item
{
int number;
float cost;
public:
void getdata(int a, float b);  //declaration
//inline function
void putdata(void)  //definition inside the class
{
cout<<number<<"\n";
cout<<cost<<"\n";
}
};
```

STATIC DATA MEMBERS

A data member of a class can be qualified as static. A static member variable has certain special characteristics. These are:

It is initialized to zero when the first object of its class is created. No other initialization is permitted.

Only one copy of that member is created for the entire class and is shared by all the objects of that class, no matter how many objects are created.

It is visible only within the class, but its lifetime is the entire program.

STATIC CLASS MEMBER

```
#include<iostream>
using namespace std;
class item
{
static int count;
int number;
public:
void getdata(int a)
{
number=a;
count++;
}
void getcount(void)
{
cout<<"count:";
cout<<count<<"\n";
}
};
int item :: count;
int main()
{
item a,b,c;   //count is initialized to zero
a.getcount();    //display count
```

```
b.getcount();
c.getcount();
a.getdata(100);  //getting data into object a
b.getdata(200);  //getting data into object b
c.getdata(300);  //getting data into object c
cout<<"After reading data"<<"\n";
a.getcount();   //display count
b.getcount();
c.getcount();
return 0;
}
```

Output:

```
count:0
count:0
count:0
After reading data
count:3
count:3
count:3
```

STATIC MEMBER FUNCTIONS

A member function that is declared static has the following properties:
A static function can have access to only other static members (functions or variables) declared in the same class.
A static member function can be called using the class name(instead of its objects) as follows:
class-name :: function-name;

Program

```
#include<iostream>
using namespace std;
class test
{
int code;
static int count;
public:                 //static member variable
void setcode(void)
{
code=++count;
}
void showcode(void)
{
cout <<"object number:"<<code<<"\n";
}
static void showcount(void)  //static member function
{
cout<<"count:"<<count<<"\n";
}
};
int test :: count;
int main()
```

```
{
test t1,t2;
t1.setcode();
t2.setcode();
test :: showcount();     //accessing static function
test t3;
t3.setcode();
test :: showcount();
t1.showcode();
t2.showcode();
t3.showcode();
return 0;
}
```

Output:

```
count:2
count:3
object number:1
object number:2
object number:3
```

NESTING OF MEMBER FUNCTION:

A member function of a class can be called only by an object of that class using a dot operator. However, there is an exception to this. A member function can be called by using its name inside another member function of the same class. This is known as nesting of member functions.

Nesting of Member Function example

```
#include <iostream.h>
using namespace std;
class set
{
int m,n;
public:
void input(void);
void display(void);
void largest(void);
};
int set :: largest(void)
{
if(m >= n)
return(m);
else
return(n);
}
void set :: input(void)
{
cout << "Input value of m and n"<<"\n";
cin >> m>>n;
}
void set :: display(void)
```

```
{
cout << "largest value=" << largest() <<"\n";
}

int main()
{
set A;
A.input();
A.display();

return 0;
}
```

The output of program would be:
Input value of m and n
25 18
Largest value=25

ARRAYS OF OBJECTS

Arrays within a class

An array can be of any data type including struct. Similarly, we can also have arrays of variables that are of the type class. Such variables are called arrays of objects.

Program:

```
#include<iostream>
using namespace std;
class employee
{
char name[30];              //string as class member
float age;
public:
void getdata(void);
void getdata(void);
};
void employee :: getdata(void)
{
cout<<"Enter name:";
cin>>name;
cout<<"Enter age:";
cin>>age;
}
void employee :: putdata(void)
{
cout<<"Name:"<<name<<"\n";
cout<<"Age:"<<age<<"\n";
}
const int size=3;
int main()
{
```

```
employee manager[size];
for(int i=0;i<size;i++)
{
cout<<"\nDetails of manager"<<i+1<<"\n";
manager[i].putdata();
}
return 0;
}
```

input:

```
Details of manager 1
Enter name:xxx
Enter age:45

Details of manager 2
Enter name:yyy
Enter age:37

Details of manager 3
Enter name:zzz
Enter age:50
```

Program output

```
Manager1
Name:xxx
Age:45

Manager2
Name:yyy
Age:37

Manager3
Name:zzz
Age:50
```

FRIEND FUNCTIONS

```
class ABC
{
----
----
----
public:
---
---
 friend void (xyz);          // syntax of friend function.
    };
```

- The function declaration should be preceded by the keyword friend. The function is defined else were in the program like a normal C++ function.
- The function definition does not use either the keyword friend or the scope operator:: .

- The functions that are declared with the keyword friend are known as friend functions. A function can be declared as a friend in any number of classes.
- A friend function, although not a member function, has full access rights to the private members of the class.

A friend function possesses certain special characteristics:

- It is not in the scope of the class to which it has been declared as friend.
- Since it is not in the scope of the class, it cannot be called using the object of that class.
- It can be invoked like a normal function without the help of any object.
- Unlike member functions, it cannot access the member names directly and has to use an object name and dot membership operator with each member name.(e.g. A.x).
- It can be declared either in the public or the private part of a class without affecting its meaning.
- Usually, it has the objects as arguments.

Friend function example program

```
#include<iostream>
using namespace std;
class sample
{
int a;
int b;
public:
void setvalue() {a=25; b=40}
friend float mean(sample s);
};
float mean(sample s)
{
return float(s.a+s.b)/2.0;
}
int main()
{
sample X;   //object X
X.setvalue();
cout<<"Mean value="<<mean(X)<<"\n";
return 0;
}
```

Output:

Mean value=32.5

RETURNING OBJECTS

A function cannot only receive objects as arguments but also can return them. The example in program illustrates how an object can be created (within a function) and returned to another function.

Program

```
#include<iostream>
using namespace std;
class comples  //x+iy form
{
float x;   //real part
float y;    //imaginary part
public:
void input(float real, float imag)
{x=real;y=imag;}
friend complex sum(complex, complex);
void show(complex);
};
complex sum(complex c1,complex c2)
{
complex c3;    //objects c3 is created
c3.x=c1.x+c2.x;
c3.y=c1.y+c2.y;
return(c3);  //returns object c3
}
void complex :: show(complex c)
{
cout<<c.x<<"+j"<<c.y<<"\n";
}
int main()
{
complex A,B,C;
A.input(3.1, 5.65)
B.input(2.75, 1.2)
C=sum(A,B);    //C=A+B
cout<<"A="; A.show(A);
cout<<"B="; B.show(B);
cout<<"C="; C.show(C);
return 0;
}
```

Output:

A=3.1+j5.65
B=2.75+j1.2
C=5.85+j6.85