tOBJECT ORIENTED PROGRAMMING USING C++

1.1 The Evolution of C++:

Computer languages have come a long way since the 1940's. Back then, scientists punched instructions in to mammoth, room-sized computer systems. These instructions were long series of zeroes and ones. These machine language instructions are called the *first generation* of computer languages.

The 1950's saw the emergence of the *second generation* computer languages-*assembly language*-easier to write than machine language but still extremely complicated for a lay person. However, the computer could still understand only machine language. Therefore, the assembler software was developed to translate the code written in assembly language into machine language.

In 1967, Martin Richard developed a language called BPCL for writing *operating* systems. An operating system is a set of programs that manages the resources of a computer and its interactions with users.

The era of the *third generation* of computer languages had arrived. In 1970's Ken Thompson modified BPCL to create a new language called B. working for Bell Laboratories, Thompson teamed up with Dennis Ritchie and wrote an early version of the Unix operating system for a DEC PDP-7 computer.

Dennis Ritchie was working on a project to further develop the Unix operating system. He wanted a low-language, like the assembly language, the could control hardware efficiently. At the same time, he wanted the language to provide the features of a high level language, that is, it should be able to run on different types of hardware. B had performance drawbacks, so in 1972, he rewrote B and called in C.

Therefore, C is categorized as both a second and third generation language. Thompson and Ritchie rewrote the Unix operating system in C. In the years that followed, C was widely accepted and used over different hardware platforms. In 1989, the *American National Standards Institute(*ANSI), along with the International Standards Organization (ISO), approved a machine independent and standard version of C.

In the early 1980's , **Bjarne Stroustrup** working for Bell Labs developed the C++ language. In his own words , " C++ was designed primal so that my friends an I would not have to program in assembly, C, or various modern high-level languages. Its main purpose was to make writing food programs easier and more pleasant for the individual programmer". (Bjarne Stroustup *The C++ Programming Language,* Third Edition. Reading, MA: Addition Wesley Publishing Company 1997). C++ was originally known as 'C with classes' as two languages contributed to its design. C which provided low-level features, and *simula67,* which provided the *class* concept.

The C++ language is a superset of C. Like the C language, C++ is compact and can be used for system programming. It can use existing C software *libraries.* (Libraries are collections of programs that you can reuse in your program.) C++ has object-oriented programming (OOP) capabilities similar to an earlier computer language called Simula67. C++ is called a hybrid language because it can be used both as a procedural language like C and as an object-oriented language like Simula67, other object-oriented languages include *Smalltalk* and *Ada.*

In 1990's, the ANSI/ISO committee began working on a standard version of the C++ language. By June 1998,the committee had approved the Final Draft International Standard for C++. It was released in the form of a document. It extended the language to include exceptions, templates, and the *Standard Template Library* (STL).

Software Evolution:

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 figure. Each layer representing an improvement over the previous one.

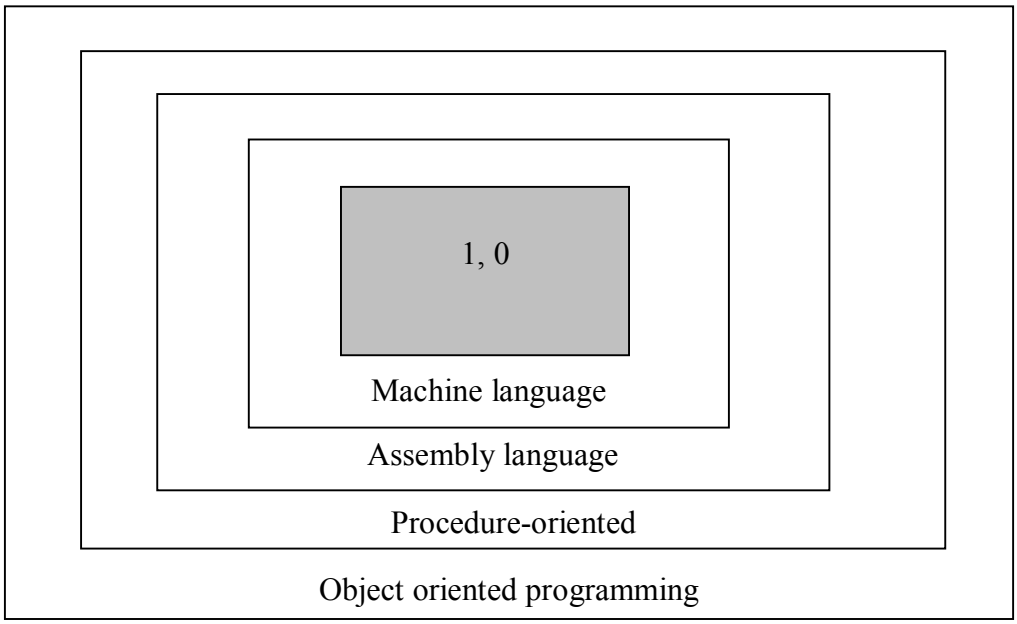


Fig. Layers of computer software

Object-Oriented Programming (OOP) is an approach program organization and development that attributes to eliminate some of pitfalls of conventional programming methods by incorporating the best of structured programming features with several powerful new concepts.

1.2 Procedure – Oriented Programming:

The high level languages such as COBOL, FORTRAN and C are commonly known as ***procedure-oriented programming*** **(POP).** In the procedure-oriented approach, the problem is viewed as a sequence of things and a number functions are written to accomplish these tasks. The primary focus is on functions.

Procedural Programming Model: Each problem is divided into smaller problems and solved using specified modules that act on data.

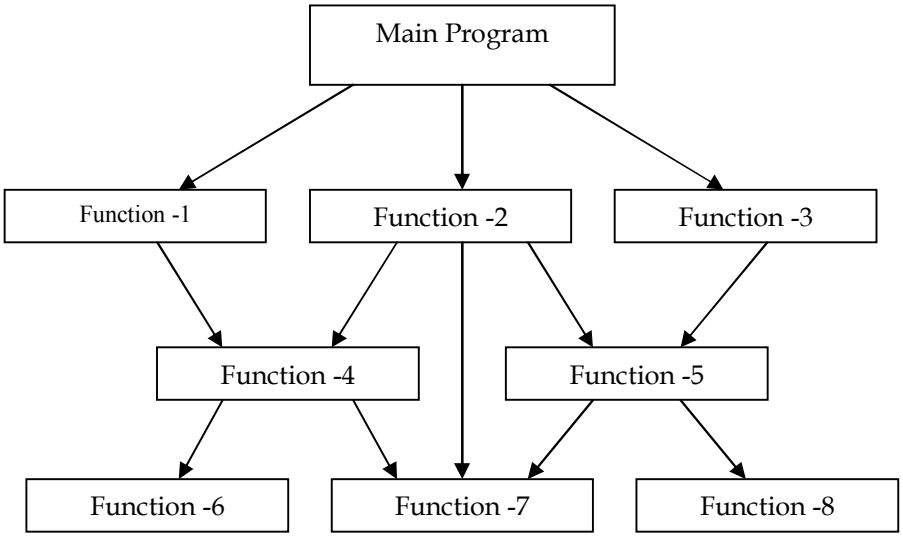


Fig – Structure of procedural-oriented programs

In a multi-function program, many important data items are placed as ***global*** so that they may be accessed by all the functions. Each function may have its own ***local data***. Fig shows the relationship of data and functions in a procedure-oriented program.

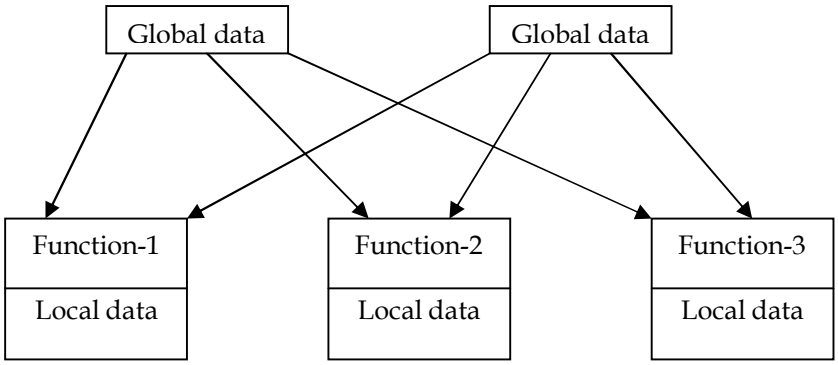


Fig- Relationship of data and functions in procedural programming

Global data are more vulnerable to an inadvertent change by a function. In a large program it is very difficult to identify what data is used by which function.

Another serious drawback with the procedural approach is that it does not model real world problems very well. This is because functions are action-oriented and do not really corresponding to the elements of the problem.

Some characteristics exhibited by procedure-oriented programming are:

- **Emphasis is on doing things (algorithms).**
- **Large programs are divided into smaller programs known as functions.**
- **Most of the functions share global data.**
- **Data move openly around the system from function to function.**
- **Functions transform data from one form to another.**
- **Employs *top-down* approach in program design.**

1.3 Object-Oriented Programming Paradigm:

The major motivating factor in the invention of object-oriented approach is to remove some of the flaws encountered in the procedural approach. OOP treats data as a critical element in the program development and does not allow it to flow freely around the system. It ties data more closely to the functions that operate on it, and protects it from accidental modification from outside functions.

OOP allows decomposition of a problem into a number of entities called ***objects*** and then builds data and functions around these objects. The organization of data and functions in object-oriented program is shown in figure.

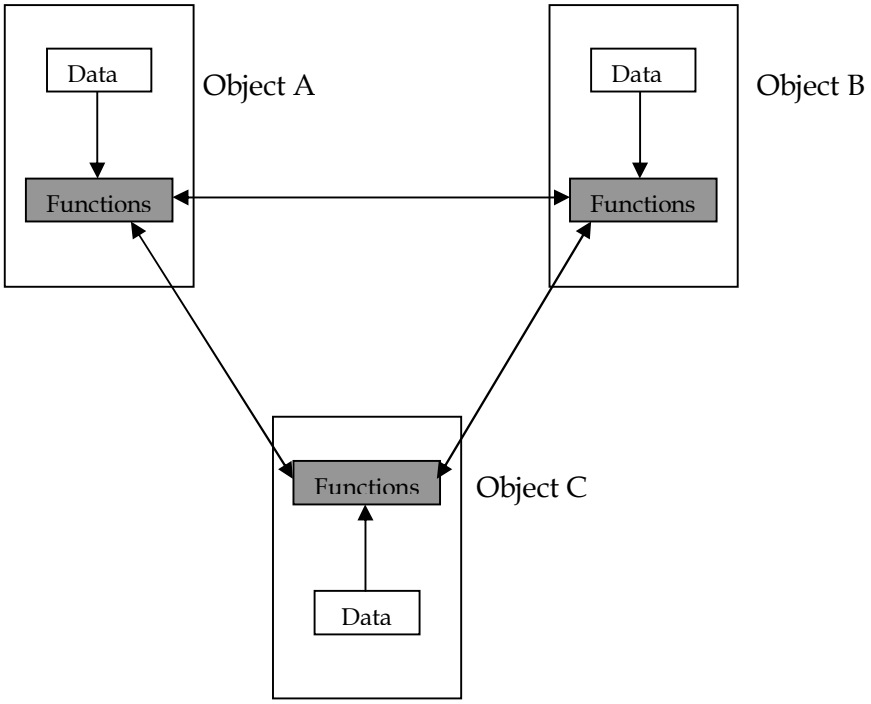


Figure - Organization of data and functions in OOP

The data of an object can be accessed only by the functions associated with the object. However, functions of one object can access the functions of other objects.

Some of the striking features of object-oriented programming are:

- **Emphasis is on data rather than procedure**
- **Programs are divided into what are known as object**
- **Data structures are designed such that they characterize the object.**
- **Functions that operate on the data of on object are tied together in the data structure.**
- **Data is hidden and cannot be accessed by external functions.**
- **New data and functions can be easily added whenever necessary.**
- **Follows *bottom-up* approach in program design.**

"Object-oriented programming 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 template for creating copies of such modules on demand*". Memory partitions are independent; the objects can be used in a variety of different programs without modifications.

Object oriented programming model: It perceived the entire software system as a collections of objects which contain attributes and behaviors.

BASICS CONCEPTS OF OBJECT –ORIENTED PROGRAMMING

Some of the concepts used extensively in object-oriented programming. These include:

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

Object:

- **Object is a triangle of entity that may be exhibiting some well-defined behavior. (Or) Object is an instance of Class (or) Everything is an object**

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 the program has to handle. They may be represents user-defined data such as vectors, time and lists.

When a program executed, the object interact by sending messages to one another. Each object contain data, and code to manipulate the data. Fig shows two notations that are popularly used in object-oriented analysis and design.

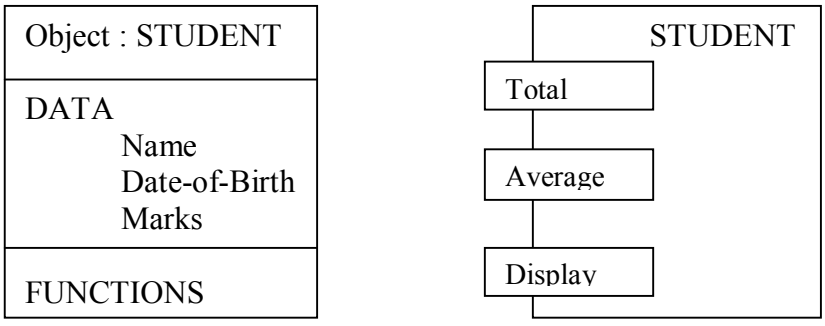


Fig – Two ways of representing an object

Classes:

- **Class is a set of attributes and behavior shared by similar objects (or) in simple way collection of objects of similar type is called a Class.**

The entire set of data and code of an object can be made a user-defined data type with the help of a *class*. **Objects are variable of the type *class***. Once a class has been defined, we can create any number of objects belonging to that class.

If **fruit** has been defined as a **class**, then the statement.

fruit apple;

will create an object **apple** belonging to the class **fruit**.

Data Abstraction and Encapsulation:

- **Abstraction focuses on the essential characteristics of objects**
- Abstraction refers to the act of representing essential features without including the background details or explanations. Classes use the concept of abstraction and are defined as a list of abstract *attributes* such as size, weight and cost, and *functions* to operate on these attributes.
- Since the classes use the concept of data abstraction, they are known as ***Abstract Data Types*** (ADT).
- The attributes are sometimes called *data members* because they hold information. The functions that operate on these data are sometimes called ***methods*** or ***member functions***.
- **Encapsulation: The wrapping up of data and function into a single unit (called class) is known as *encapsulation*. The data is not accessible to the outside world, and only those functions which are wrapped in the class can access it.**
- This instruction of the data from direct access by the program is called ***data hiding*** or ***information hiding***.

Inheritance:

- ***Inheritance* is the process by which object of one class acquire the properties of objects of another class. Create a new class derived from the old class.**
- It supports the concept of *hierarchical classification*. For example, the bird 'robin' is a part of the class 'flying bird' which is again a part of the class 'bird'. The principle behind this sort of division is that each derived class shares common characteristics with the class from which it is derived as illustrated in fig.

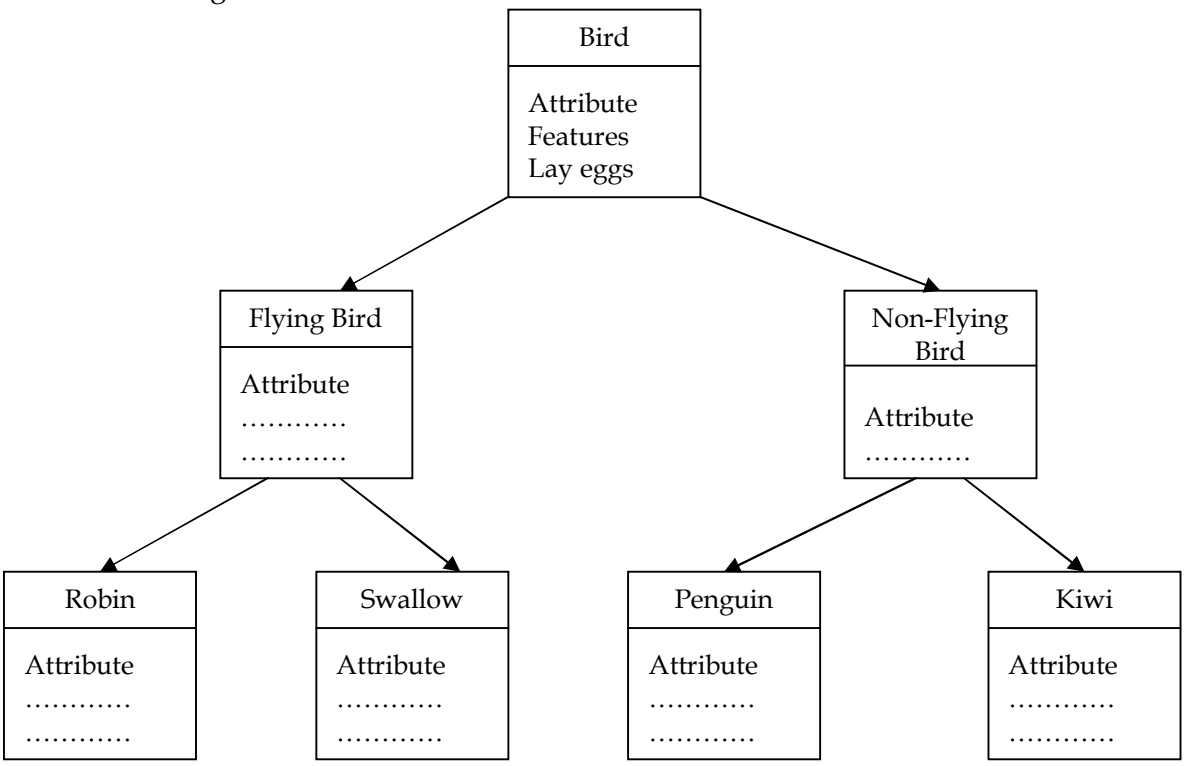


Fig- property inheritance

- 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.
- This is possible by deriving new class from the existing one. The new class will have the combined features of both the class.

Polymorphism:

- ***Polymorphism* is another important OOP concept. Polymorphism, a Greek term, means the ability to take more than one form. An operation may exhibit different behaviors in different instances.**
- The behavior depends upon the types of data used in the operation.
- For example, consider the operation of addition. For two numbers, the operation will generate a sum. If the operands are string, then the operation would produce a third string by concatenation.
- This process of making an operator to exhibit different behaviors in different instances is known as ***operator overloading.***
- Using a single function name to perform different types of tasks is known as ***function overloading.***

Figure illustrates that a single function name can be used to handle different number and different types of arguments.

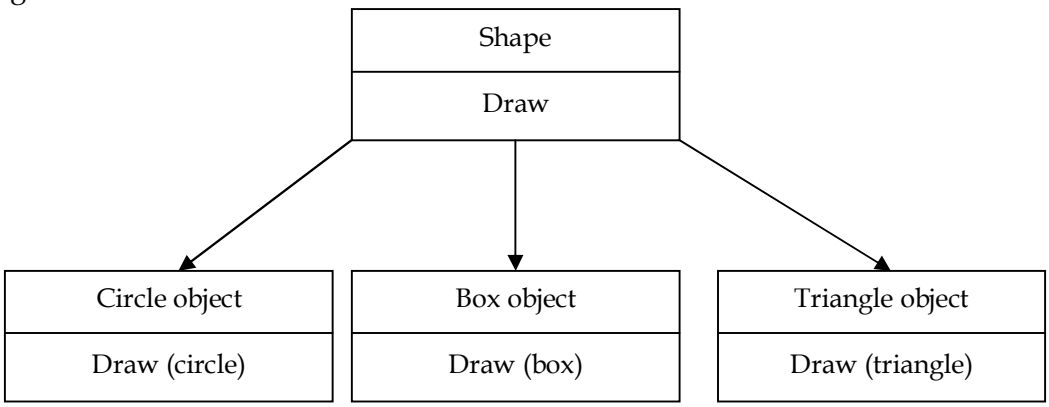


Fig. – Polymorphism

Polymorphism plays an important role in allowing object having different internal structures to share the same external interface. This means that a general class of operation may accessed in the same manner even though specific actions associated with each operation may differ. Polymorphism is extensively used in implementing inheritance.

Dynamic Binding:

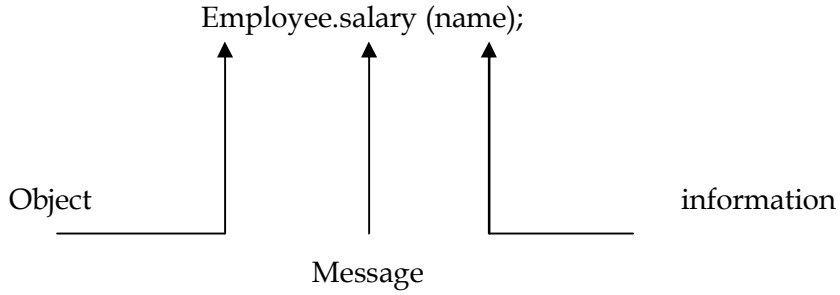
- Binding refers to the linking of a procedure call to the code to be executed in response to the call.
- ***Dynamic Binding* (also known as late binding) means that the code associated with a given procedure call is not known until the time of the call at run-time. It is associated with polymorphism and inheritance.**
- A function call associated with a polymorphic reference depends on the dynamic type of that reference.

Consider the procedure "draw" in above figure by inheritance; every object will have this procedure. Its algorithms are, however, unique to each object and so the draw procedure will be redefined in each class that defines the object.

Message Passing:

- An object-oriented program consists of a set of objects that communicate with each other. The process of programming in an object-oriented language, therefore, involves the following basic steps:
  i. Creating classes that define objects and their behavior.
  ii. Creating objects from class definitions, and
  iii. Establishing communication among objects.
- **A message for an object is a request for execution of a procedure, and therefore will invoke a function (procedure) in the receiving object that generates the desired result.**

- *Message passing* involves specifying the name of the object, the name of the function (message) and the information to be sent. Example:



Objects have a life cycle. They can be created and destroyed. Communication with an object is feasible as long as it is alive.

Benefits of OOPs:

OOP offers several benefits to both the program designer and user.

- **Through inheritance, we can eliminate redundant code and extend the use of existing classes.**
- **The principle of data hiding helps the programmer to build secure programs that cannot be invaded by code on other parts of the program.**
- **It is possible to have multiple instances of on object to co-exist without any interference.**
- **It is possible to map objects in the problem domain to those objects in the program.**
- **It is easy to partition the work in a project based on objects.**
- **Object-oriented systems can be easily upgraded from small to large systems.**
- **Message passing techniques for communication between objects makes the interface descriptions with external systems much simpler.**
- **Software complexity can be easily managed.**

Applications of OOP:

Applications of OOP are beginning to gain importance in many areas. The most popular application of object-oriented programming, up to now, has been in the area of user interface design such as windows. **Windows are developed using OOP techniques. OOP is simplifying the complex problem.**
The other areas for application of OOP includes:

- **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.**

Object-Oriented Languages:

Object-oriented programming is not the right of any particular language. Like structured programming, OOP concepts can be implemented using languages such as C and Pascal. A language that is specially designed to support the OOP concepts makes it easier to implement them.

The languages should support several of the OOP concepts to claim that they are object-oriented. Depending upon the features they support, they can be classified into the following two categories:

- Object-based programming Languages
- Object-oriented programming Languages.

➢ ***Object-based programming*** **is the style of programming that primarily supports encapsulation and object identity. Major features that are required for object-base programming are:**

- Data encapsulation
- Data hiding and access mechanisms
- Automatic initialization and clear-up of objects
- Operator overloading

Languages that support programming with objects are said to be object-based programming languages. They do not support inheritance and dynamic binding. Ada is a typical object-based programming language.

- ***Object-oriented programming*** **incorporates all of object-based programming features along with two additional features, namely, inheritance and dynamic binding. It's simply**

  Object-based features + inheritance + dynamic binding.

Languages that support these features include C++ , Smalltalk and object Pascal and java. There are a large number of object-based and object-oriented programming languages.

C++ STREAMS

A stream is a sequence of bytes. The *source* stream that provides data to the program is called the *input* stream and the *destination* stream that receives output from the program is called the *output* stream. A program extracts the bytes from and input stream and inserts bytes into an output stream.

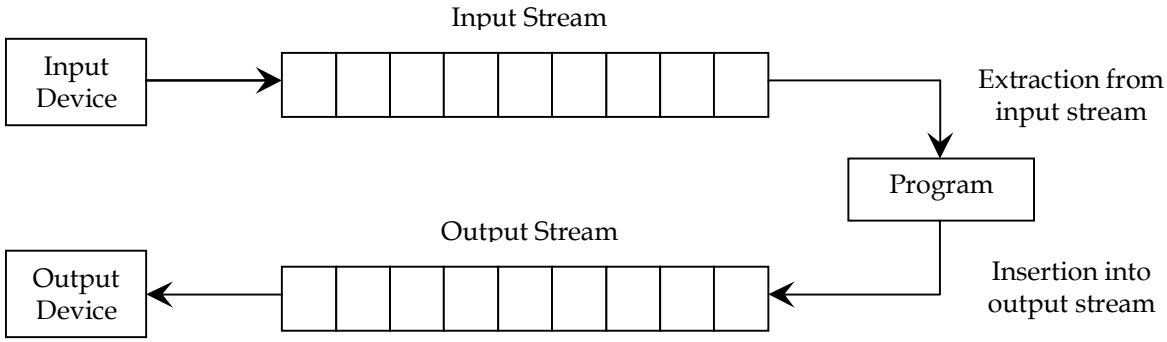


Fig. Data Streams

C++ STREAM CLASSES

The C++ I/O system contains a hierarchy of classes that are used to define various streams to deal with both the console and disk files. These classes are called stream classes.

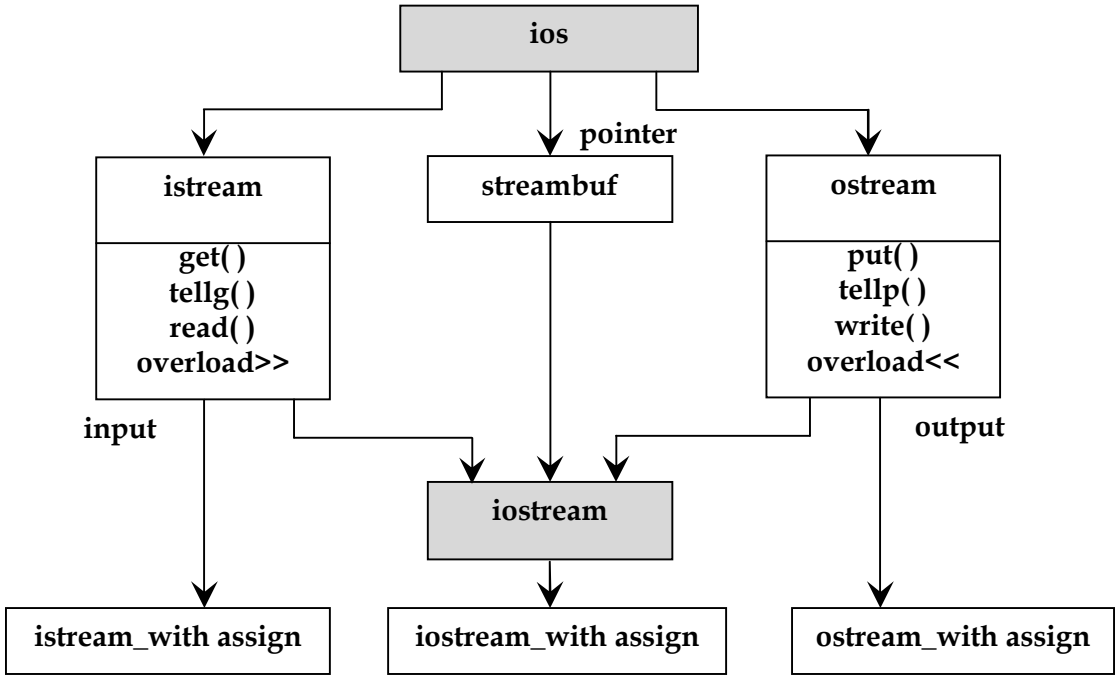


Fig. Stream classes for console I/O operations

Class name		Contents
ios	:	Contains basic facilities that are used by all other input and output classes
istream	:	Declares input functions. Inherits the properties of **ios**
ostream	:	Declares output functions. Inherits the properties of **ios**
iostream	:	Contains all the input and output functions. Inherits ios, istream and ostream classes.
streambuf	:	Provides an interface to physical devices through buffers.

UNFORMATTED I/O OPERATIONS:

The following functions are used to implement the unformatted console I/O operations. They are the member functions of the istream and ostream classes. So, we should use the cin and cout objects with the functions.

Input functions (istream class)

overload operator >>,
get( ),
getline( )

Output functions (ostream class)

overload operator <<,
put( ),
write( )

Overload operator>> : Used to get input from the keyboard.

Syntax : cin >> var1 >> var2 >> …>> varn;
Example : cin >> x >> y >> z;

get( ) : This function is used to get input from the keyboard. We can use this function in two ways.

1. get(char)

Syntax : cin.get(var_name);
Example : char c;
cin.get(c);

2. get(void)

Syntax : var_name = cin.get( );
Example : char c;
c = cin.get( );

getline( ) : This function is used to get a line of text as input.

Syntax : cin.getline(var_name, size);
Example : char name[10];
cin.getline(name, 10);

Overload operator <<: This function is used to display the output on VDU.

Syntax : cout << var1<< var2<< … << varn;
Example : cout<< x<< y<<"Thank you";

put( ) : This function is used to display a single character as output on VDU.

Syntax : cout.put(var_name);
Example : char c = 'z';
cout.put(c);

write( ) : This function is used to display a line of text as output on the screen.

Syntax : cout.write(var_name, size);
Example : char name[10] = "Thank You";
cout.write(name, 10);

FORMATTED CONSOLE I/O OPERATIONS

Formatted console I/O operations are used to format the output of a program. These functions are the member functions of **ios** class. So, we should use the **cout** object for using these functions.

i. width( )
ii. prescision( )
iii. fill( )
iv. setf( )
v. unsetf( )

width( ): Used to specify the required field size for displaying an output value.

Syntax : cout.width(size);
Example : int a = 25;
cout.width(4);
cout<<a;

		2	5

precision( ) : Used to specify the number of digits to be displayed after the decimal point of a float value.

Syntax : cout.precision(size);
Example : int a = 4522.123456;
cout.precision(2);
cout<<a;

4	5	2	2	.	1	2

fill( ) : Used to specify a character that is used to fill the unused portion of a field.

Syntax : cout.fill(arg);

Example : int a = 750;
cout.width(5);
cout.fill('*');
cout<<a;

*	*	7	5	0

setf( ) : Used to specify format flags that can control the form of output display (such as left-justification and right-justification)

Format required	Flag (arg1)	Bit-field (arg2)
Left-justified output	ios::left	ios::adjustfield
Right-justified output	ios::right	ios::adjustfield
Padding after sign or base	ios::internal	ios::adjustfield
Scientific notation	ios::scientific	ios::floatfield
Fixed point notation	ios::fixed	ios::floatfield
Decimal base	ios::doc	ios::basefield
Octal base	ios::oct	ios::basefield
Hexadecimal base	ios::hex	ios::basefield

Syntax : cout.setf(arg1, arg2);

Example : char a[7] = "PRGC++";
cout.setf(ios::left, ios::adjustfield);
cout.width(10);
cout<<a;

P	R	G	C	+	+				

unsetf( ) : This function is used unset the flags and bit field groups using setf( ) function.

Flags that do not have bit fields :

The setf( ) function can be used with the following flags as a single arguments to achieve the required output.

Flag	Meaning
ios::showbase	Use base indicator on output
ios::showpos	Print + before positive numbers
ios::showpoint	Show trailing decimal point and zeroes
ios::uppercase	Use uppercase letters for hex output
ios::skipus	Skip white space on input
ios::unitbuf	ios::Flush all streams after insertion
ios::stdio	ios::Flush stdout and stderr after insertion

Example :
cout.setf(ios::showpoint);
cout.setf(ios::showpos);
cout.precision(3);
cout.setf(ios::fixed, ios::floatfield);
cout.setf(ios::internal, ios::adjustfield);
cout.width(10);
cout << 275.5 << "\n";

Output

+			2	7	5	.	5	0	0

Manipulators and User Defined Manipulators:

Manipulators are used to format the output. C++ has provided some predefined and user defined manipulators. There are number of predefined manipulators present in the header file 'iomanip.h'. Two or more manipulators can be used as a chain in one statement as shown below:

cout << manip1 << manip2 << manip3 << item;
cout << manip1 << item1 << manip2 << item2;

Manipulators and their meanings

Manipulator	Use
setw(int w)	Set field width to w
setprecision(int d)	Set the floating point precision to d
setfill(int c)	Set the fill character to c
setioflags(long f)	Set the format flag to f
resetiosflags(long f)	Clear the flag specified by f

setw () manipulator:-

- Takes integer type of variable as its parameter. The parameter specifies the width of the column.

Example:
1. cout<< 123<<endl;
2. cout<< setw (3) << 10;

Output:
123
10

setprecision () manipulator:

- C++ displays values of float and double type with six digits by default after decimal point.
- By using the setprecision () manipulator we can pass number of digits we want to display after decimal point.

Example:
1. cout<<setprecision(3);
2. cout<< sqrt (3) << endl;

Output:
1.732 (the value of √3 is 1.7320508075689)

setfill () manipulator:

- The main task of setfill () manipulator is to fill the extra spaces left in the output of setw() manipulator with characters.

Example:
1. cout<<setfill('*')
2. cout<<setw (5) << 10;
3. cout<<setw (5) << 257<<endl;

Output:
***10**257

setiosflags () manipulator:

- There are two important parameter setiosflags () takes. They are ios::showpos and ios::showpoint.
- When ios::showpos is passed as parameter to setiosflags () than a positive sign is prefixed to the output.

Example:
1. cout<<setiosflags(ios::showpos) << 20;

Output: +20

- When ios::showpoint is passed as a parameter than it follows the setprecision parameter ().

Example:
1. cout<< setprecision(3);
2. cout<<1.7 <<endl;
3. cout<<setiosflags (ios::showpoint)
4. cout<<1.7<<endl;

Output:
1.7
1.700

resetiosflags () manipulator:

- This manipulator cancels the setiosflags parameter. This manipulator resets again to setiosflags () manipulator.

Example:
1. cout<< setprecision (3)<<1.7<<setiosflags(ios::showpoint)<<1.7<<endl;
2. cout<<resetiosflags(ios::showpoint)<<1.7<<endl;

Output:
1.7
1.700
1.7

User defined manipulators:

- The manipulators defined by the users are called as user defined manipulators.

Syntax :

```
ostream &  manipulator(ostream & output)
{
        //statements//
        return output;
}
```

Calling the user defined manipulators :

```
cout << name of the manipulator << list to print;
```

Example:

```
ostream & unit (ostream & output)
{
        output << "inches";
        return output;
}
```

Calling: `cout << 36 << unit;`

Output: **36 inches.**

Example Program using setw and endl manipulators

```
#include<iostream>
#include<iomanip>    // for setw

int main()
{
        int basic = 950, allowance=95, total=1045;
        cout<<setw(10)<<"Basic"<<setw(10)<<basic<<endl;
        cout<<setw(10)<<"Allowance"<<setw(10)<<allowance<<endl;
        cout<<setw(10)<<"Total"<<setw(10)<<total<<endl;

return 0;
}
```

Output of this program is given below.

```
Basic          950
Allowance       95
Total         1045
```

UNIT - II

INTRODUCTION TO C++

C++ is an object-oriented programming language. Initially named 'C with classes', C++ was developed by Bjarne Stroustrup at AT& T Bell Laboratories in Murray Hill, New Jersey, USA, in the early eighties. In 1983, the name("C with classes") was changed to C++. The idea of C++ comes from the C increment operator ++, thereby suggesting that C++ is an incremented version of C.

C++ is a superset of C. The three most important facilities that C++ adds on to C are classes, inheritance, function overloading, and operator overloading. These features enable us to create abstract data types, inherit properties from existing data types and support polymorphism, thus making C++ a truly object- oriented language.

The addition of new features has transformed C from a language that to one that provides bottom-up, object-oriented design.

Applications of C++:

C++ is a versatile language for handling very large programs. It is suitable for virtually any programming task including development of editors, compilers, databases, communication systems and any complex real-life application systems.

- Since C++ allows us to create hierarchy-related objects, we can build special object oriented libraries which can be used later by many programmers.
- While C++ is able to map the real-world problem properly, the C part of C++ gives the language the ability to get close to the machine-level details.
- C++ programs are easily maintainable and expandable. When a new feature needs to be implemented, it is very easy to add to the existing structure of an object.

Simple C++ Program:

```
# include<iostream>                    // include header file
int main()
{
cout<<" C++ is better than C.\n";      // C++ statement
return 0;
}                                      // end of example
```

Program Features:

Like C, the C++ program is a collection of functions. The above example contains only one function, **main()**. Execution begin at main(). Every C++ program must have a **main()**. C++ is a free-form language, the C++ statements terminate with semicolons.

Comments:

C++ introduces a new comment symbol // (double slash). A comment may start anywhere in the line, and whatever follows till the end of the line is ignored. There is no closing symbol.

The double slash comment is basically a single line comment. Multi line comments can be written follows:

```
// this is an example file
// C++ program to illustrate
// some of it's feature
```

The C comment symbol /*, */ are still valid are more suitable for multi line comments. The following command is allowed.

```
/*      this is an example of
        C++ program to illustrates
        Some of it's features
*/
```

Output Operator:

The output statement → **cout << "C++ is better than C.";**

causes the string in quotation marks to be displayed on the screen. This statement introduces two new C++ features, **cout** and **<<**. The identifier cout(pronounced as 'C out') is a predefined object that represents the standard output stream in C++.

The operator << is called the ***insertion or put to*** operator. It inserts (or sends) the content of the variable on it's to the rights of the object on it's left (figure).

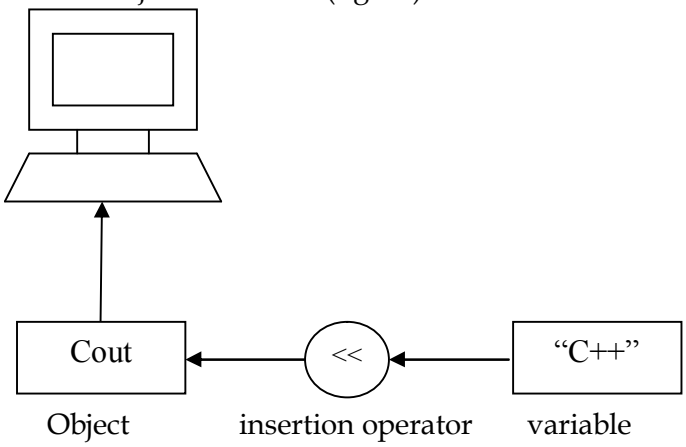


Figure: Output using insertion operator

The operator << is the bit-wise left-shift operator and it can still be used for this purpose. This is an example of how one operator can be used for different purposes, depending on the context. This concept is known as ***operator overloading***.

Input operator:

The statement → **cin >> number1;**

The operator >> is known as ***extraction or get from*** operator. It extracts (or takes) the value form the keyboard and assign it to the variable on it's right(figure).

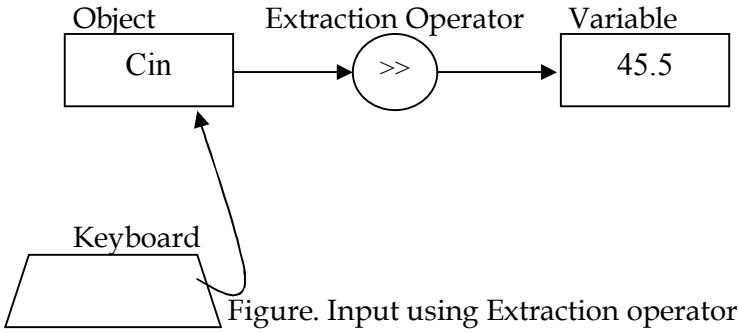


Figure. Input using Extraction operator

Cascading of I/O operators:

The statement → **cout <<"Sum = "<<sum<<"\n";**

first send the strings "Sum=" to cout and then sends the value of sum. Finally, it sends the new line character so that the next output will be in the new line. The multiple use of << in one statement is called ***cascading***.

The iostream file:

The following #include directive in the program

 # include <iostream>

This directive causes the preprocessor to add the contents of the iostream file to the program. It contains declarations for the identifier **cout** and the operator **<<**. Some old versions of C++ use a header file called iostream.h. The header file iostream should be included at the beginning of all programs that use input/output statements.

Return type of main():

In C++ main() returns an integer type value to the operating system. Therefore, every main() in C++ should end with a return(0) statement; otherwise a warning or error might occur. Since **main()** returns a integer type value, return type for main() is explicitly specified as **int**. the default return type for all functions in C++ is **int**.

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

Structure of C++ program:

A typical C++ program would contain four sections as shown in figure. These sections may be placed in separate code files and then compiled independently or jointly.

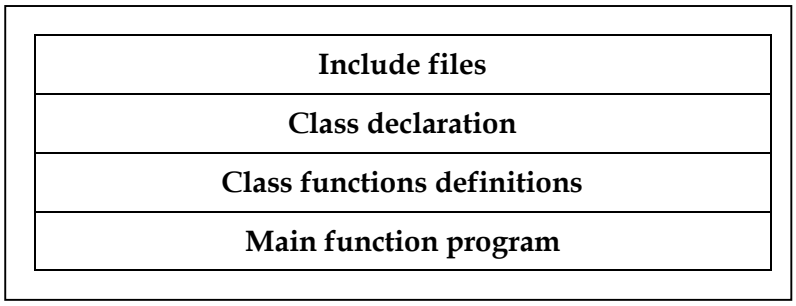


Fig. Structure of C++ program

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 and syntax of the language.

Keywords:

The keywords implement specific C++ language features. They are explicitly reserved identifiers and cannot be used as names for the program variables or other user-defined program elements.

Some Keywords are:

auto	break	case	catch	char	class	const
continue	default	delete	do	double	else	enum
extern	float	for	friend	goto	if	inline
int	long	new	operator	private	protected	public
register	return	short	signed	sizeof	static	struct
switch	template	this	throw	try	typedef	union
unsigned	virtual	void	volatile	while		

Identifiers:

Identifiers refer to the names of variables, functions, arrays, classes, etc. created by programmer. The following rules are common to both C and C++.

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

Variables:

Variables are locations in the memory that can hold values. Before assigning any value to a variable, it must be declared. To use the variable *number* storing an integer value, the variable *number* must be declared and it should be of the type *int* as follows:

int number**;**

Constants:

Constants refer to fixed values that do not change during the execution of a program. Like C, C++ support several kinds of literal constants. They include integers, characters, floating point numbers and string.

Example:	123	// decimal integer	12.34	// floating point integer
	O37	// octal integer	OX2	// hexadecimal integer
	"C++"	// string constant	'A'	// character constant

Basic Data Types:

Data types in C++ can be classified under various categories as shown in Fig.

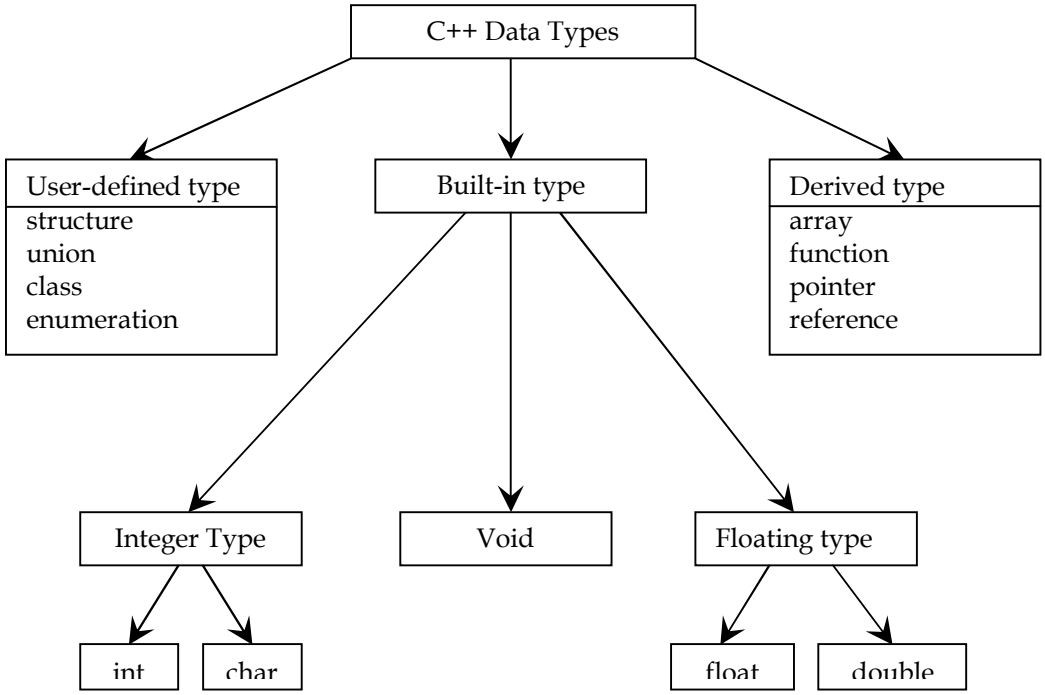


Figure- Hierarchy of C++ data types

With the exception of void, the basic data types may have several modifiers preceding them to serve the needs of various situations. The modifiers signed, unsigned, long, and short may be applied to character and integer basic data types. However, the modifier long may also be applied to double.

Size and Range of C++ Basic Data types

Type	Bytes	Range	Use
char	1	-128 to +127	Used to store single character
unsigned char	1	0 to 255	Used to store unsigned (positive only) single character
signed char	1	-128 to +127	Used to store signed (both positive and negative) single character
int	2	-32,768 to + 32,767 (-2^{15} to $2^{15}-1$)	Used to store integer (whole numbers) values
unsigned int	2	0 to 65,535	Used to store unsigned integer (positive whole numbers) values
signed int	2	-32,768 to + 32,767 (-2^{15} to $2^{15}-1$)	Used to store signed integer (whole numbers both positive and negative) values

short int	2	-32,768 to + 32,767 (-2^{15} to $2^{15}-1$)	Used to store short integer (whole numbers) values
unsigned short int	2	0 to 65,535	Used to store unsigned short integer (positive whole numbers) values
signed short int	2	-32,768 to + 32,767 (-2^{15} to $2^{15}-1$)	Used to store signed short integer (whole numbers both positive and negative) values
long int	4	-2,147,483,648 to +2,147,483,647	Used to store long integer (whole numbers) values
signed long int	4	-2,147,483,648 to +2,147,483,647	Used to store unsigned long integer (positive whole numbers) values
unsigned long int	4	0 to 4,294,967,295	Used to store signed long integer (whole numbers both positive and negative) values
float	4	3.4E-38 to 3.4E+38	Used to store numbers with decimal point with single precision.
double	8	1.7E-308 to 1.7E+308	Used to store numbers with decimal point with double precision.
long double	10	3.4E-4932 to 1.1E+4932	Used to store numbers with decimal point with more than double precision.

Void:

The normal use of void is

(1) To specify the return type of a function when it is not returning any value and

(2) To indicate an empty argument list to a function.

Example: **void funct1(void);**

User—Defined Data Types:

Structures and Classes:

1. struct → To store **structure**
2. union → To store **union**

C++ also permits us to define another user-defined data type known as **class** which can be used, just like any other basic data type, to declare variables. The class variables are known as **objects.**

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. The syntax of an **enum** statement is similar to that of the **struct** statement.

Ex: enum shape { circle, square, triangle };
enum color { red, green, blue, yellow };

By default, the enumerators are assigned integer values starting with 0 for the first enumerator, 1 for the second, and so on.

For example,

enum color{red, blue=4, green=8};
enum color{red=5, blue, green};

are valid definitions. In first case, **red** is 0 by default. In the second case, **blue** is 6 and **green** is 7.

Derived Data Types:

Arrays:

An *array* is a group of like-typed variables that are referred to by a common name. The compiler will allow declaring the array size as the exact length of the string constant.

Ex : char string[3]= ”xyz”; is valid in C.

It assumes that the programmer to leave out the null character \0 in the definition.

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

char string[4]= ”xyz”;

Functions:

It is a subprogram to reduce the size of the program and to use different places in a program for more than one time.

Pointers:

Pointers are declared and initialized as in C. For example

int *a; it is an int pointer.

a = &x; it is an address of x assigned to a.

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

char *const pt="soft"; it is an constant pointer.

Symbolic Constants:

There are two ways of creating symbolic constants in C++.

- Using the qualifier **const.**
- Defining a set of integer constants using **enum** keyword.

Any value declared as **const** cannot be modified by the program in any way. It allows us to create typed constants instead of having to use #define to create constants that have no type information. As with **long** and **short,** we can use the **const** modifier alone, it defaults to **int.** For example,

To declare **const** a=10; (it means const int a=10).

Another method of naming integer constants is as follows:

enum{x,y,z};

This defines x,y and z as integer constants with values 0,1,2 respectively. This is equivalent to const x=0; const y=1; const z=2;

We can also assign values to x,y and z explicitly.

enum{x=100,y=200,z=300};

Type Compatibility:

C++ defines **int, short int, long int** as three different types. They must be cast when their values are assigned to one another. Similarly, **unsigned char, char,** and **signed char** are considered as different types.

To use the **sizeof, sizeof('x');** it will be equals to **sizeof(char)** and print the size of character type data

Declaration of Variables:

In C, all variables must be declared they are used in executable statements. C++ allows the declaration of a variable any where in the scope. This means that a variable can be declared right at the place of its first use.

Example;

```
int main()
{
        float x;                        //declaration
        float sum = 0;
        for(int i=1;i<5;i++)            //declaration
        {
                cin>>x;
                sum=sum+x;
        }
        float average;                  //declaration
        average=sum/i;
        cout<<average;

        return 0;
}
```

Dynamic Initialization Variables:

C++, permits initialization of the variable at run time. this is referred to as ***dynamic initialization***.

Example:

Normal declaration	**Dynamic Declaration**
…….	……
…….	……
int n;	int n=strlen(string);
n=strlen(string);	……
float area;	…….
area=3.14159*rad*rad;	float area=3.14159*rad*rad;

Dynamic initialization is extensively used in object-oriented programming. We can create exactly the type of object needed, using information that is known only at the run time.

Reference Variables:

C++ introduces a new kind of variable known as the ***reference variable***. A reference variable provides an ***alias*** *(alternative name)* for a previously defined variable. For example, if we make the variable **sum** a reference to the variable **total**, then **sum** and **total** can be used interchangeably to represent that variable.

The reference variable created as follows:

data-type & reference-name=variable-name;

C++ assigns additional meaning to the symbol **&**. Here, **&** is not an address operator. The notation **float &** means reference to **float**. Example,

```
int      n[10];
int & x = n[10];          // x is alias for n[10]
char & a ='\n';           // initialize reference to a literal
```

the variable **x** is an alternative to the array element **n[10]**. The variable **a** is initialized to the newline constant.

The following reference are also allowed

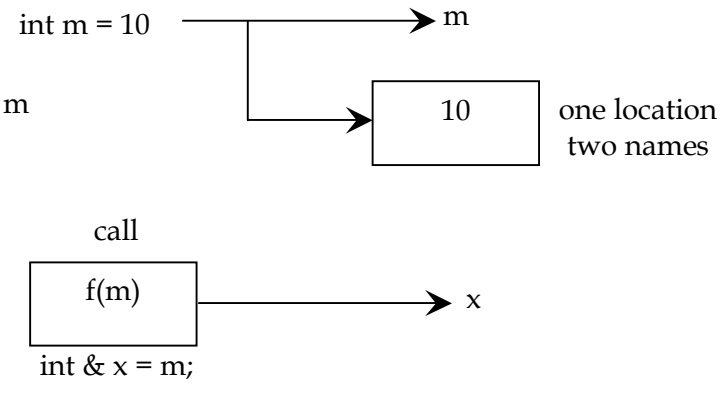
i.
```
int x;
int *p=&x;
int &m=*p;
```

ii.
```
int &n=50;
```

the first set of declarations causes **m** to refer to **x** which is pointed to by the pointer p and the statement in ii. Creates an **int** object with value 50 and name **n**.

A major application of reference variables is in passing arguments to functions. Consider the following:

```
void f(int & x)          // uses reference
{
x =x+10;                 // x is incremented; so also m
}
int main()
{
int m=10;
f(m);                    // function call
------
------
}
```



when the function is call **f(m)** is executed, the following initialization occurs:

int &x = m;

such function calls are known as ***call by reference.*** Since the variable **x** and **m** are aliases, when the function increments **x**, **m** is also incremented. The value of m becomes 20 after the function is executed.

Operators:

C++ has a rich set of operators. All operators in C are valid in C++ also.

a. Assignment Operators b. Arithmetic Operators
c. Relational Operators d. Logical Operators
d. Bit wise Operators e. Increment/Decrement operators
f. Shortcut Operators g. Conditional Operators
h. Special operators

Assignment operators:

The assignment operator is used to store the constants, variables, and result of an expression and return value of a function from right side into variable placed in left side.

Operator : **=** Ex : **a = 6; a = b; a = b+c;**

Arithmetic operators:

The arithmetic operators perform the basic arithmetic operations such as addition, subtraction, multiplication, division and modulus.

The operators are : **+, – , *, /, %** Ex : **x=a+b, x=a-b, x=a*b, x=a/b, x=a%b**

Unary operators:

The operators that work with only one operand are called as unary operators. The operators used in C++ language are : **+, –, ++, – –, !**.

Increment/Decrement operators:

The Increment operators **++** increase the value of operand by one.

The Decrement operators **--** decrease the value of operand by one.

Ex : **i++, ++i, i--, --i.**

Shortcut assignment operators:

When same variable is used in both left and right side of an arithmetic expression, we can use the shortcut assignment operators .

The operators are : **+=, – =, *=, /=, %=** Ex : **a += b, a –=b, a*=b, a/=b, a%=b**

Sizeof and Comma operators:

The **sizeof** operator returns the memory size occupied by an operand in terms of bytes.

Ex : **a = sizeof(int)**;

The **comma** operator (,) adjoins several expressions. Ex : **x = (y = 3, y + 1);**

Conditional operators:

The **?:** are conditional operators. They do the job done by if..else control structure.

Syntax : **expression1 ? expression2 : expression3;** Ex : **big = (a>b) ? a : b;**

Relational operators:

The operators that do relationship tests with the operands are called relational operators. The relational operators are:

< (less than) → **Ex : if(a < b)**
<= (less than or equal to) → **Ex : if(a <= b)**
> (greater than) → **Ex : if(a > b)**
>= (greater than or equal to) → **Ex : if(a >= b)**
== (is equal to) → **Ex : if(a == b)**
!= (not equal to) → **Ex : if(a != b)**

Logical operators:

Logical operators are used when more than one relationship should be considered in evaluation. It gives either True value or False value.

AND operator : **&& → AND** **Ex: if((a>b) && (a>c))**
OR operator : **|| → OR** **Ex: if((a>b) || (a>b))**
NOT operator : **! → NOT** **Ex: if(!(a>b))**

Truth table of NOT :

A	! A
0	1
1	0

Truth table for AND, OR:

A	B	A&&B	A || B
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	1

Bit-wise operators

The bit wise logical operators work with those decimal numbers in its binary form. The bit wise OR, bit wise AND, bit wise XOR, bit wise <<, and bit wise >> are binary operators. The bit wise 1's complement is a unary operator.

Bit wise OR (|) : Ex : 0101 **|** 1010 = 1111
Bit wise AND(&): Ex : 0101 **&** 0101 = 0101
Bit wise XOR (^): Ex: 0101 **^** 1010 = 1111

Bit wise Complement(~) : If a bit is **1**, then the operator converts it into **0** and vice versa.

Ex : ~1010 = 0101.

Bit wise Left shift (<<) : It is used to shift the bits to the left.

Ex: x = 5 **<<** 2; 0000 0101 = 0001 0100

Bit wise Right shift (>>) : It is used to shift the bits to the right.

Ex: x = 5 **>>** 2; 0000 0101 = 0000 0001

C++ introduces some new operators. They are :

<< **----→ insertion operator (or output operator)**
>> **----→ extraction operator (or input operator)**
:: **----→ Scope resolution operator**
::* **----→ pointer-to-member declarator**
->* **----→ pointer-to-member operator**
.* **----→ pointer-to-member operator**
delete **----→ memory release operator**
endl **----→ line feed operator**
new **----→ memory allocation operator**
setw **----→ field width operator**

Scope Resolution Operator (::)

C++ is a block structured language. Blocks and scopes can be used in constructing programs. The same variable name can be used to have different meanings in different blocks. The scope of the variable extends from the point of its declaration till the end of the block containing the declaration. A variable declared inside the block is said to be ***local*** to that block.

Example 1:

```
-----
-----
{
        int x =10;
        -----
        -----
}
-----
-----
{
        int x =1;
        -----
        -----
}
```

Example 2:

```
-----
-----
{
        int x = 10;
        -----
        -----
        {
                int x = 1;
                -----
                -----
        }
        -----
        -----
}
```

Block 2 Block 1

The two declarations of x refers to two different memory locations containing different values. The statement second block cannot refer to the variable x declared in the first block, and vice-versa.

Block2 is contained in Block1. Note that a declaration in an inner block ***hides*** a declaration of the same variable in an outer block and therefore, each declaration of **x** causes it to refer to a different data object.

In C, the global version of a variable cannot be accessed from within the inner block. C++ resolves this problem by introducing a new operator **::** called the ***scope resolution operator***.

This can be used to uncover a hidden variable. It take the following form

:: variable-name

This operator allows access to the global version of variable. ::count means the global version of the variable count.

Example1:

```
#include <iostream>
int m=10;                    //global m
int main()
{
int m=20;                    // m redeclared, local to main
        {
            int k = m;
            int m = 30;   //m declared again
                          // local to inner block
                cout<<" we are in inner block \n";
                cout<<"k ="<<K<<"\n";
                cout<<"m ="<<m<<"\n";
                cout<<"::m ="<<::m<<"\n";
            }
            cout<<"\n we are in outer block\n");
            cout<<"m="<<m<<"\n"};
            cout<<"::m="<<::m<<"\n"};
            return 0;
}
```

Output:

We are in inner block
k = 20
m = 30
::m = 10
We are in outer block
m = 20
::m = 10

Member Dereferencing Operators:

::* -----→ To declare a pointer to a member of a class
* -----→ To access a member using object name and a pointer to that the member
->* -----→ to access a member using a pointer to the object and a pointer to that member

Memory Management Operators:

We use dynamic allocation techniques when it is not known in advance how much of memory space is needed. Although C++ support these functions, it also defines two unary operators **new** and **delete** that perform the task of allocating and freeing the memory in a better and easier work. These operators manipulate memory on free store; they are also known as ***free store*** operators.

An object can be created by using **new**, and destroyed by using **delete**. The **new** operator can be used to create objects of any type. It take the following general form

pointer-variable = **new** data-type;

Here, ***pointer-variable*** is a pointer of type ***data-type***. The **new** operator allocates sufficient memory to hold a data object of data-type and returns the address of the object. The ***pointer-variable*** holds the address of the memory space allocated. The declaration of pointer and their assignments as follows.

```
int *p =new int;
float *q = new float;
```

When a data object is no longer needed, it is destroyed to release the memory space reuse. The general form of its use is

delete pointer-varaible;

the ***pointer-variable*** is the pointer that points to a data object created with **new**. Example

```
delete p;
delete q;
```

The **new** operator offers the following advantages:

1. It automatically computes the size of the data object. We need not use the operator **sizeof**.
2. It automatically returns the correct pointer type, so that there is no need to use a type cast.
3. It is possible to initialize the object while creating the memory space.
4. Like any other operator, **new** and **delete** can be overloaded.

Manipulators:

Manipulators are operators that are used to format the data display. The most commonly used manipulators are **endl** and **setw.**

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;
cout<<"n="<<n<<endl;
cout<<"p="<<p<<endl;
…….
```

Output

m = | 2 | 5 | 9 | 7 |

n = | 1 | 4 |

p = | 1 | 7 | 5 |

Three lines of output, one for each variable. the values of the variable as 2597,14 and 175 respectively. The output will appear as shown below.

Here, the numbers are ***right-justified***. This form of output is possible only if we can specify common field width for all the numbers and force them to be printed right-justified. The **set** manipulator does this job. It is used as follows.

cout<<setw(5)<<sum<<endl;

The manipulator **setw(5)** specifies a field width 5 for printing the value of the variable sum. This value is right-justified within the field s shown below.

		3	4	5

The program illustrate the use of **endl** and **setw.**

Example

```
#include<iostream>
#include<iomanip>       // for setw
int main()
{
        int basic = 950,  allowance=95,  total=1045;
        cout<<setw(10)<<"Basic"<<setw(10)<<basic<<endl;
        cout<<setw(10)<<"Allowance"<<setw(10)<<allowance<<endl;
        cout<<setw(10)<<"Total"<<setw(10)<<total<<endl;

return 0;
}
```

Output of this program is given below.

```
Basic          950
Allowance       95
Total         1045
```

Type Cast Operators:

C++ permits explicit type conversion of variables or expression using the type cast operator.

```
(type-name) expression      //C notation
type-name (expression)      //C++ notation
```

Examples:

```
average = sum/(float)       // C notation
average=sum/float(i);       // C++ notation
```

For example,

p=int*(q); is illegal. In Such cases, we must use C type notation.

p=(int*)q; alterativley, we can use typedef to create an identifier of the required type and use it in the functional notation.

```
typedef int * int_pt
p=int_pt(q)
```

Expressions and their types:

An expression is a combination of operators, constants and variables arranged as per the rules of the language. Expressions consist of one or more operands, and zero or more operators to produce a value. Expressions may be of the following seven types:

- Constant expression
- Integral expression
- Float expression
- Pointer expression
- Relational expression
- Logical expression
- Bitwise expression

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

Constant Expression:

Constant expression consists of only constant values. Examples:

```
15
20 + 5 / 2.0
```

Integral Expressions:

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

```
m
m * n - 5
m * 'x'
5 + int(2.0)
```

where **m** and **n** are integer variables.

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.

Pointer Expressions:

Pointer expressions produce address values. examples:

```
&m
ptr
ptr+1
"xyz"
```

where **m** is a variable and **ptr** is a pointer.

Relational Expressions:

Relational expressions yield results of type **bool** which takes a value **true** or **false**. Examples:

```
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 results compared. Relational expressions are also known as ***Boolean Expressions***.

Logical Expressions:

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

```
a > b && x == 10;
x == 10 || y == 5
```

Bitwise Expressions:

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

```
x<<3    //shift three bit position to left
y>>1    //shift one bit position to right
```

shift operators are often used for multiplication and division by powers of two.

Special Assignment Expressions

Chained Assignment

```
x = ( y = 10 );        or        x = y = 10;
```

First 10 assigned to y and then to x.

A chained assignment cannot be used to initialize variables at the time of declaration.

Embedded Assignment

```
x = (y = 50) + 10.
```

(y=50) is an assignment expression known as embedded assignment. Here, the value 50 is assigned to y and then the result 50 + 10 = 60 is assigned to x.

Compound Assignment

C++ supports compound assignment operator which is a combination of the assignment operator with a binary arithmetic operators.

Example: x = x + 10 may be written as → **x += 10**;

The operator += is known as compound assignment operator or short-hand assignment operator.

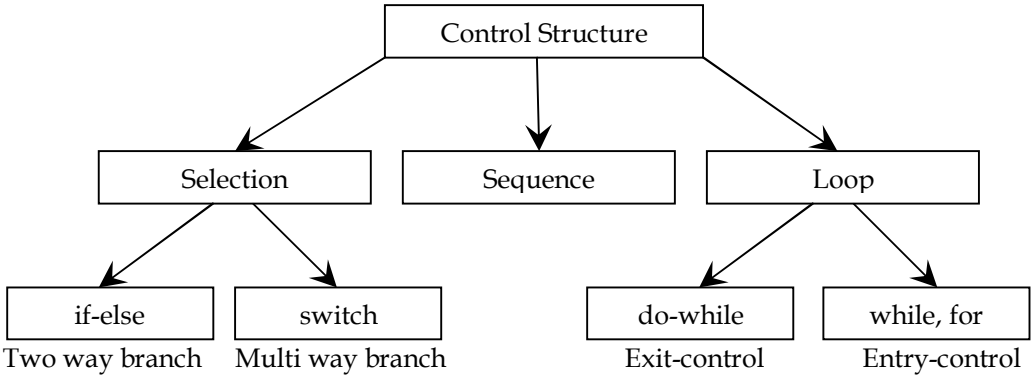
The general form is : **variable op = variable 2;**

OPERATOR PRECEDENCE

Associativity defines in the case that there are several operators of the same priority level- which one must be evaluated first, the rightmost one or the leftmost one.

Level	Operator	Description	Grouping
1	::	scope	Left-to-right
2	() [] . -> ++ -- dynamic_cast static_cast reinterpret_cast const_cast typeid	postfix	Left-to-right
3	++ -- ~ ! sizeof new delete	unary (prefix)	Right-to-left
	* &	indirection and reference (pointers)	
	+ -	unary sign operator	
4	(type)	type casting	Right-to-left
5	.* ->*	pointer-to-member	Left-to-right
6	* / %	multiplicative	Left-to-right
7	+ -	additive	Left-to-right
8	<< >>	shift	Left-to-right
9	< > <= >=	relational	Left-to-right
10	== !=	equality	Left-to-right
11	&	bitwise AND	Left-to-right
12	^	bitwise XOR	Left-to-right
13	|	bitwise OR	Left-to-right
14	&&	logical AND	Left-to-right
15	||	logical OR	Left-to-right
16	?:	conditional	Right-to-left
17	= *= /= %= += -= >>= <<= &= ^= |=	assignment	Right-to-left
18	,	comma	Left-to-right

CONTROL STRUCTURES:



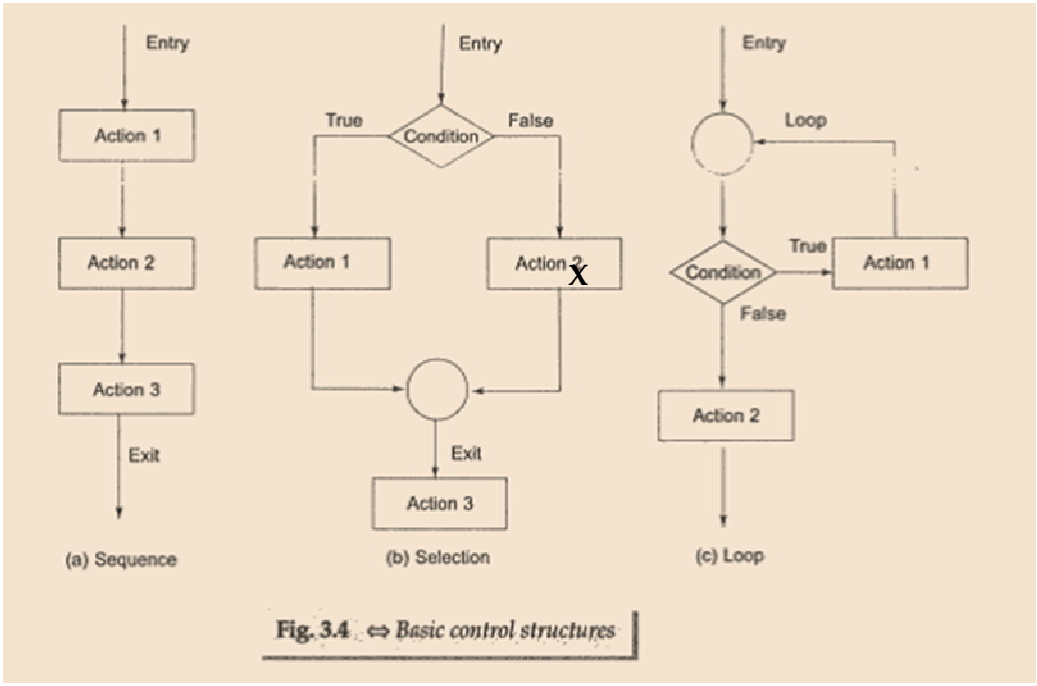


Fig. 3.4 ⇔ *Basic control structures*

The if statement : The **if** statement is implemented in two forms:

- Simple **if** statement
- **if…else** statement

a) The **if** structure tests a condition and transfers the control accordingly.

Syntax : **if(expression)**
{ statements }

Example : if(a>b)
printf("A is big");
printf("B is big");

b) **if…else** structure works according to the true & false value generated by the expression.

Syntax : **if(condition)**
{ statements }
else
{ statements }

Example : **if(marks >= 40)**
printf("Pass");
else
printf("Fail");

c) **Nested if structure:**

If any **if** or **else** block contains another **if, if…else, else…if ladder** then it can be called as **Nested if…else**.

Syntax :
```
if(exp)
{
        if(exp)
                { statements }
        else
                { statements }
}
else
{       if(exp)
                { statements }
        else
                { statements }
}
```

Example :
```
if(mark >= 60)
{
if(mark >= 75)
        printf("Honours");
        else
        printf("I class");
}
else
{       if(mark >= 40)
                printf("II Class");
        else
                printf("Fail");
}
```

Switch structure:

switch is called Multi-directional control structure. This structure is used when decision-making is depending upon more than two values.

Syntax :
```
switch(exp)
{
case 1 :
        action1;

case 2 :
        action2;

case 3:
        action3;
default :
        action4;

}
```

Example :
```
switch(n)
{
case 1 :
        printf("ONE");
        break;
case 2 :
        printf("TWO");
        break;
case 3 :
        printf("THREE");
        break;
default :
        printf("Thank you");
        break;
}
```

while loop structure:

The while structure is called as entry controlled loop. The condition is tested first. If the condition satisfies, the loop block is executed, otherwise the program execution continues after the while structure.

Syntax :
```
initialization;
while(condition)
{       statements;
        increment/decrement;
}
```

Example :
```
i = 1;
while(i < 10)
{       cout<<i<<"\n";
        i++;
}
```

do…while loop structure:

The do…while structure is called as exit controlled loop. The block is executed first and the condition is tested next. If the condition satisfies, the block is executed again, otherwise the program execution continues after the do…while structure.

Syntax :
```
initialization;
do
{       statements;
        increment/decrement;
} while(condition);
```

Example :
```
i = 1;
do
{       cout<<i<<"\n";
        i++;
}while(i < 10);
```

for loop structure:

for statement is used to execute a statement or a group of statements repeated for a known number of times.

Syntax :
```
for(initialization; condition; incr/decr)
{
        statements;
}
```

Ex :
```
for(i=0; i<10; i++)
    cout<<i;
```

Other forms of for loop :

It is possible to omit any one of the sections in for construct. In case, if we omit, a semi colon must be present.

Examples : a. **for(; i<10 ; i++)** b. **for(; i<10 ;)** c. **for(; ; ;)**

We can use more than one initial value, condition and counter inside a for loop by using comma operator.

d. **for(i =1, j=1; i<5, j<3; i++, j++)**

ARRAYS

An array is a group of related data items that share a common name. An array reserves continuous memory location to place the values consecutively. The types are :

1. One-dimensional array.
2. Two-dimensional array.
3. Multi-dimensional array.

Rules for subscripted variables :

a. The subscript is always an integer
b. The subscript value cannot be negative
c. The subscript must be given within square brackets.
d. If there is more than one subscript, they should be given within separate square brackets. No other delimiters are used.

One-dimensional array :

An array having only one subscript is a one-dimensional array.

Declaration : **type array_name[size];**
Initialization : **type array_name[size] = { values };**
Example : **int a[10];**
int a[10] = { 1,2,3,4 };

Two-dimensional array :

An array having two subscripts is a two-dimensional array.

Declaration : **type array_name[size1][size2];**
Initialization : **type array_name[size1][size2] = { values };**
Example : **int a[10][10];**
int a[10][10] = { 1,2,3,4 };

Multi-dimensional array:

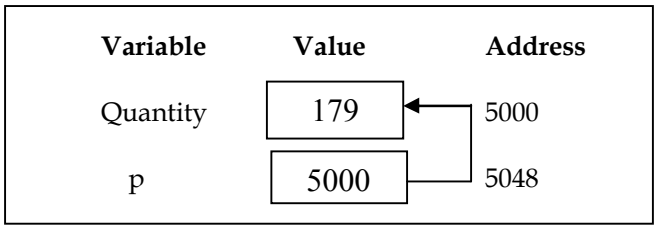
An array having more than two subscripts is a multi-dimensional array.

Declaration : **type array_name[size1][size2]…[sizeN];**
Initialization : **type array_name[size1][size2]…[sizeN]={ values};**
Example : **int a[2][2][2];**
int a[2][2][2] = { 1,2,3,4,5,6,7,8 };

Pointer :

A pointer is a variable that contain the address of any variable. It point to variable of the data type for which it has been declared. It is declared with the '*' preceding it.

- The unary operator **&** gives the "address of a variable".
- The indirection/dereference operator ***** gives the "content of an object pointed to by a pointer".



The value of the variable p is the address of the variable quantity. Access the value of quantity by using the value of p. the variable p 'points' to the variable quantity thus p gets the names pointer.

Ex. **int *p;**
p=&quantity;

assign the address 5000 (location of quantity) to variable p. the & operator remembered "address of ".

UNIT - III

FUNCTIONS

A function is said to be a set of instructions that can perform a specific task. It will be reduce the size of the program by calling and using them at different places in the program. The advantages are: **Re-usability, modularity, overall programming simplicity.** Functions are classified into library functions and user-defined functions.

Library Functions: The functions that are already available in the C++ library are called Library functions. Some important categories of library functions are : Input/Output functions (iostream.h), Mathematical functions (Math.h), String handling functions (String.h), Console input/output functions (conio.h) and Manipulator functions (iomanip.h).

User-Defined Functions: A function defined by the user is called as user-defined function. The advantage of user-defined function is that the complexity of a program is reduced and the errors can be traced easily.

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 arg);

The function that have a return value should use the **return** statement for termination. The **main( )** function in C++, therefore, defined as follows:

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

The return type of function is **int** by default, the keyword **int** in the main( ) header is optional. Most C++ compilers will generate an error or warning if there is no **return** statement.

Function Prototyping (or declaration):

- **The prototype describes the function interface to the compiler by giving details such as the number and type of arguments and the type of return values.**
- **When a function is called, the compiler uses the template to ensure that proper arguments are passed, and the return value is treated correctly.**

Function prototype is a declaration statement in the calling program and is of the following form:

type function-name (argument-list);

The *argument-list* contains the types and names of arguments that must be passed to the function.

Example:
```
int cal(int, int, float);
float sum(int a, float b, float c);
```

Function Definition:

Function definition defines the operations of a function which is already declared. When the function definition comes before the main() function, we need not declare it.

Example:
```
float sum(int a, float b, float c)
{
        float v=a+b+c;
          ----------
}
```

Function call : The process of calling a function for execution is known as function call. Any user defined function can call any other function. There are two types of function call: **Call by value** and **Call by Reference**.

The function **sum( )** can be invoked in a program as follows:

float addval = sum(b1,w1,h1); //function call

The variable **b1**,**w1** and **h1** are known as the **actual parameters** (**arguments**) which specify the dimensions of **addval.** Their types should match with the types declared in the prototype. The calling statement should not include type names in the argument list.

Different types of Functions

1. Function **with parameters** (arguments) and **without return values**.

```
#include <iostream.h>
#include <conio.h>
void main( )
{
        int a=10, b=10;
        void add(int,int);
        clrscr();
        add(a,b);
        getch();
}
void add(int x, int y)
{
        cout<< x+y;
}
```

2. Function **with parameters** (arguments) and **with a return value**.

```
#include <iostream.h>
#include <conio.h>
void main( )
{
        int a=10, b=10;
        int add(int,int);
        clrscr();
        cout << add(a,b);
        getch();
}
int add(int x, int y)
{
        return(x+y);
}
```

2. Function **without parameter** (argument) and **with a return value**.

```
#include <iostream.h>
#include <conio.h>
void main( )
{
        int add();
        clrscr();
        cout << add();
        getch();
}
int add()
{

        int x=10, y=10;
        return(x+y);
}
```

2. Function **without parameter** (argument) and **without a return value**.

```
#include <iostream.h>
#include <conio.h>
void main( )
{
        void add();
        clrscr();
        add();
        getch();
}
void add()
{

        int x=10, y=10;
        cout << (x+y);
}
```

Parameter Passing in Functions

Call By Value :

When a function is called by **supplying values as parameters** (arguments), it is called **Call By Value**. The values placed in the **actual parameters are copied into the formal parameters**. So any **changes made to the formal parameters do not affect the actual parameters**.

Call By Reference:

When we pass **parameters (arguments) by reference**, the **formal parameters** in the called function **become aliases** to the **actual parameters** in the calling function. So any **change made in the formal parameters affect the original value**.

Return By Reference:

A function can also return a reference.

```
int & max(int &x, int &y)
{       if(x>y)
                return x;
        else
                return y;
}
```

Since the return type of **max( )**is **int &**, the function returns reference to **x** and **y** (and not the value). Then function call such as **max(a,b)** will yield a reference to either **a** or **b** depending on their values. This means that this function call can appear on the Left-hand side of an assignment statement. That is, the statement

```
max(a,b) = -1;
```

is legal and assigns -1 to **a** if it is larger, otherwise -1 to **b**.

Inline Function:

One of the objectives of using functions in a program is to save some memory space, which becomes appreciable when a function is likely to be called many times. Every time a function is called, it takes a lot of extra time is executing a series of instructions for tasks such as jumping to the functions, saving registers, pushing arguments into stack, and returning to the calling function.

To eliminate the cost of calls to small functions, C++ proposes a new feature called *inline function*. Inline function is a function that is expanded in line when it is invoked.

```
Syntax:     Inline function-header
            {
                    function body
            }
Example:    inline double cube(double a)
            {
                    return(a*a*a);
            }
```

It will be reduce the memory size and processing time will be very fast more than the normal functions.

Some of the situations where inline expansion may not work are:

1. For functions returning values, if a loop, a **switch**, or a **goto** exists.
2. For functions not returning values, if a return statement exists.
3. If functions contain **static** variables.
4. If **inline** functions are recursive.

Default Arguments:

C++ allows us call a function without specifying all its arguments. In such cases, the function assigns a ***default value*** to the parameter which does not have a matching argument in the function call. Default values are specified when the function is declared. The compiler looks at the prototype to see how many arguments a function uses and alerts the program for possible default values:

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

the above prototype declare a default value of 0.15 to the argument **rate**. Subsequent function call like

```
value  amount(5000,7);          // one argument missing
```

passes the value of 5000 to **principal** and 7 to **period** and then let the function use default value of 0.15 for rate. The call

```
value  amount( 5000,5,0.12);
```

passes the value of 0.12 to **rate**.

One important point to note is that only the trailing argument can have default values. We must add default from ***right to left***. We cannot provide a default value to a particular argument in the middle of argument list.

```
int mul (int i, int j=5, int k=10);      // legal
int mul (int i=5, int j=5);              // illegal
int mul (int i=0, int j, int k=10);      // illegal
int mul (int i=2, int j=5, int k=10);    // legal
```

const Arguments:

In C++, an argument to a function can be delivered as const as shown below.

```
int strlen(const  char  *p);
int length(const string &s);
```

The qualifier **const** tells the compiler that the function should not modify the argument. The compiler will generate an error when this condition is violated. This type of declaration is significant only when we pass arguments by reference or pointers.

Function Overloading:

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

Using the concept of function overloading; we can design a family of functions with one function name but different argument list. The function would perform different operations depending on the argument list in function call. The correct function to be invoked is determined by checking the number and type of the argument but not on the function type.

For example, an overload **add( )** function handles different type of data as below:

Function prototypes (declarations)	*Function calls*
int add(int a, int b);	cout << add(5,10);
int add (int a, int b, int c);	cout << add(15,10.0);
double add(double a, double b);	cout << add(12.5,7.5);
double add(int a double q);	cout << add(5,10,15);
double add(double p, int q);	cout << add(0.75,5);

Recursive Function:

A function call by itself is called Recursive Function.

Example:

```
#include<iostream.h>
int fact(int n);                    // Function declaration
void main()
{
    cout<<"6 factorial value is"<<fact(6)<<endl;   //Calling fact() function in main().
    cout<<"5 factorial value is"<<fact(5)<<endl;  // Calling fact() function in main().
}
int fact(int n)
{
    int res;
    if(n==1)
        return 1;
    else
        res=fact(n-1)*n;    //Calling fact() function inside the same function's body(Function calls itself)
    return res;


}
```

```
/* Lab Exercise -2 : Program to implement function overloading, inline function, default argument,
                     Constant argument, Call by value and Call by reference*/
#include<iostream.h>
#include<conio.h>

int volume(int&);
double volume(double, double);
long volume(long, int, int);
inline float volume(const float &,float);

int main()
{
        int a;
        clrscr();
        cout << "Function Overloading Program to find the volume of 4 Geometric Primitives \n \n";
        cout << "---------------------------------------------------------------------------------------------------- \n";
        cout << "Enter side value for cube : \n";
        cin >> a;
        cout << "Call by reference method : int volume(int &); \n";
        cout << "---------------------------------------------------------- \n";
        cout << "Volume of a cube : " << volume(a) << "\n \n";
        cout << "Call by value method : double volume(double, int); \n";
        cout << "---------------------------------------------------------------- \n";
        cout << "Volume of cylinder : " << volume(2.5, 8.0) << "\n \n";
        cout << "Default Argument method : long volume(long l, int b, int h=10); \n";
        cout << "------------------------------------------------------------------------------ \n";
        cout << "Volume of Rectangular box : " << volume(100L, 75, 10) << "\n \n";
        cout << "Inline function & const Argument   : inline float volume(const float&,float); \n";
        cout << "---------------------------------------------------------------------------------------------- \n";
        cout << "Volume of a Sphere : " << volume(1.33, 5.0) << "\n \n";
        getch();
        return 0;
}

int volume(int & s)
{
        return(s * s * s);
}

double volume (double r, double h)
{
        return(3.14519 * r * r * h);
}

long volume (long l, int b, int h)
{
        return (l * b * h);
}

inline float volume(const float &c, float r=10.0)
{
        return (c * 3.14519 * r * r * r);
}
```

C Structures Revisited:

A structure is a convenient tool for handling a group of logically related data item. They provide a method for packing together data of different types. It is a user defined data type with a *template* that serves to define its data properties.

For example,
```
struct student
{
        char name[20];
        int    rollno;
        float totalmark;
};
```

The keyword **struct** declares **student** as a new data type that can hold three fields of different data types. These fields are known as ***structure members*** or ***elements***. The identifier student, which is referred to as ***structure name*** or ***structure tag***, can be used to create variables of type student.

Example **struct** student **A**; /C declaration

A is a variable of type student and has three members variable as defined by the template. Member variable can be accessed using the ***dot*** or ***period operator*** as follows:

```
strcpy(A.name, "Roever");
A.rollno = 1234;
A.totalmark = 879;
```

Specifying Class:

A Class is a way to bind the data and its associated functions together. It allows the data (and functions) to be hidden, if necessary, from external use. When defining a class, we are creating a new *abstract data type* that can be treated like any other built-in data type. Generally, 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.

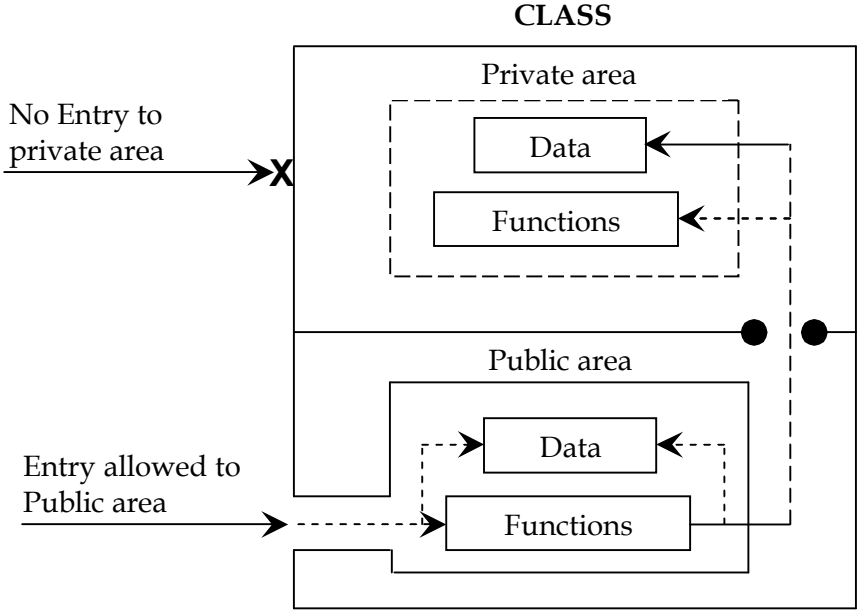
General form of a class declaration is:

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

The **class** declaration is similar to a **struct** 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***. These are usually grouped in two sections, namely, **private** and **public** known as visibility labels and the keywords are followed by a colon.

The class members that have been declared as **private** can accessed only from within the class, **public** members can be accessed from the outside the class also. The use of keyword **private** is optional. By default, the members of a class are **private**. If both labels are missing, then, default, the members are **private**.

The variable declared inside the class are known as ***data members*** and the functions are known as ***member functions***. Public members (both function and data) can be accessed from outside the class. The binding of data and functions together into a single class-type variable is referred to as ***encapsulation***.



Data hiding in classes

A Simple Class Example: The class declaration would look like:

```
class Item
{
 private:
        int number;                    // variable declaration
        int cost;                      // private by default
 public:
        void getdata (int a, float b); // function declaration
        void putdata();                // using prototype
};
```

Creating objects:

An object in C++ is essentially a block of memory that contains space to store all the instance variables. Creating an object is also referred to as ***instantiating*** an object.

To create objects of type **item**, use statements such as the following:

```
Item  x;                    //memory for x is created
```

creates a variable **x** of the type **item**. In C++, the class variables are known as ***objects***. Therefore, **x** is called an object of type **item**. We may also declare more than one object in one statement.

```
Example :   Item x, y, z;
```

Objects can also be created when a class is defined by placing their names immediately after the closing brace,

```
class Item
{       ……..
        ……..
} x, y, z;
```

would create the object **x, y** and **z** of type **item**.

Accessing class members:

The private data of a class can be accessed only through the member functions of that class. The **main( )** cannot contain statement that access **number** and **cost** directly. The following is the format for calling a member function.

```
Object_name.functionname (actual argument)
```

For example, the function call statement

```
x.getdata(100, 75.5);
```

is valid and assign the values 100 to **number** and 75.5 to **cost** of the object **x** by implementing the **getdata()** function.

Similarly, the statement

```
x.putdata();
```

would display the values of data members.

Dot Operator:

The dot notation is used to obtain the value of the instance variables. It has two parts namely the object on the left side of the dot and the variable on the right side of the dot. Dot expressions are evaluated from left to right. The general form for accessing instance variables using the dot operator is given below:

```
Objectreference.variable name
```

We can store values into instance variables using the dot operator as shown below:

```
one.c = 10;      two.c = 15;
```

We can refer to values of instance variables using the dot operator as given below:

```
cout<< "c=" << one.c;
cout<< "c=" << two.c;
```

Defining Member Function:

Member function can be defined as two places:

1. Outside the class definition
2. Inside the class definition.

Outside the class definition:

Member functions that are declared inside a class have to be defined separately outside the class. They should have a function header and a function body. An important difference between a member function and a normal function is that a member function incorporates a membership 'identity label' in the header.

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.

For instance, consider the member functions **getdata()** and **putdata()** as discussed above. They may be coded as follows:

```
void item : : getdata( int a, float b )
{       number = a;
        cost = b;
}
void item : : putdata( void)
{       cout << "Number : " << number << "\n";
       cout << "Cost : " << cost << "\n";
}
```

Note: The member functions have some special characteristics that are often used in the program development. These characteristics are:

- Several different classes can use the same function name. The 'membership label' will resolve their scope.
- Member functions can access the private data of the class. A non-member function cannot do so. (However, an exception to this rule is a ***friend function***).
- A member function can call another member function directly, without using the dot operator.

Inside the class definition:

We can define a member function inside the class definition. When a function is defined inside the a class, it is treated as an inline function. Normally, only small functions are defined inside the class definition. For example the **item** class as follows:

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

Nesting of Member functions :

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.

Private Member functions:

In some situations, we may require certain functions to be hidden from outside calls. We can place these functions in the private section.

```
/* Example program for Nesting of member
   function and private member function. */
#include <iostream.h>
#include <conio.h>
class sample
{
        int m;
        void read(void);  // Private member
                          // function declaration.
public:
        void update( void );
        void write( void );
};

void sample :: read( )
{
        cout <<"Enter a value : "<<"\n";
        cin >> m;
}
```

```
void sample :: update( )
{
        read( );        // Nesting of member function.
                        // Simple call, No objects used.
}
void sample :: write(void)
{
        cout << "M value is : " << m <<"\n";
}
void main( )
{       sample S;
        clrscr();
        S.read( );  //This function call won't work.
                    // Objects cannot access private
                    //  members.
        S.update( );
        S.write( );
        getch();
}
```

ARRAYS WITHIN A CLASS

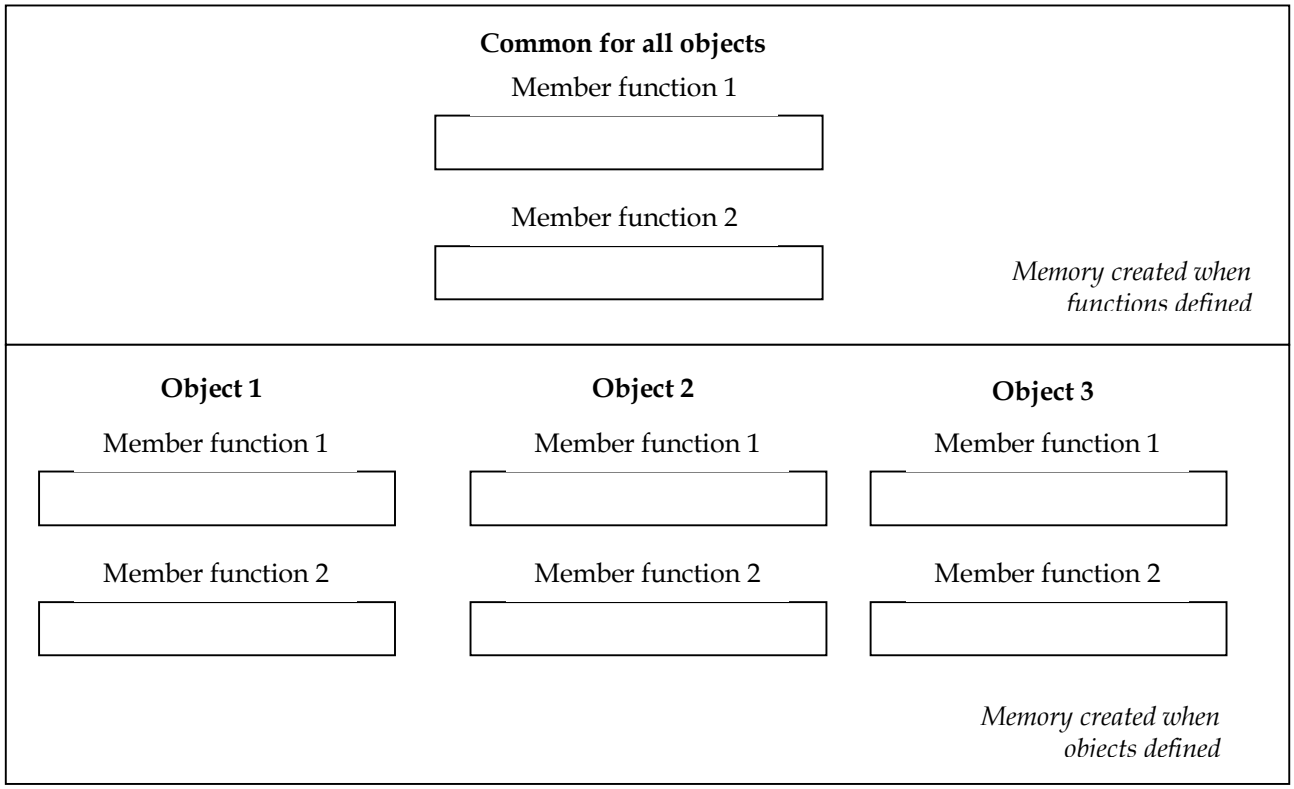
The arrays can be used as member variables in a class.

```
Example:    const int size=10;
            class array
            {
                    int a[size];      // 'a' is int type array
             public:
                    void setval(void);
                    void display(void);
            };
```

The array variable **a[ ]** declared as a private member of the class **array** can be used in the member functions, like any other variable.

MEMORY ALLOCATION FOR OBJECTS

The member functions are created and placed in the memory space only once when they are defined as part of a class specification. Since all objects belonging to that class use the same member functions, no separate space is allocated for member functions when the objects are created. Only space for member variables is allocated separately for each object. Separate member locations for the objects are essential, because the member variables will hold different data values for different objects.



Objects in memory

STATIC DATA MEMBERS AND MEMBER FUNCTIONS:

```
/* Example program to use static member variable
   and static member function */
#include <iostream.h>
#include <conio.h>
class test
{
        int code;
        static int count;  // static member variable
public:
       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;
}
```

Static Data members:

A data member of a class can be qualified as **static**. Static variables are normally used to maintain values common to the entire class. 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 member function:

Like **static** member variable, we also have **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 (function or variables) declared in the same class.
- A **static** member function can be called using the class name (instead of it's objects) as follows:

```
class-name :: function-name;
```

In the example program, the **static** function **showcount()** displays the number of objects created till the moment. A count of number of objects created is maintained by the **static** variable count.

The function **showcount()** displays the code number of each object.

ARRAYS OF OBJECTS

We can have arrays of variables that are of the type class. Such variables are called arrays of objects. Consider the following class definition:

```
class employee
{
        char name[30];
        float age;
  public:
        void getdata(void);
        void putdata(void);
};
```

The identifier **employee** is a user-defined type and can be used to create objects that relate to different categories of the employees.

```
employee manager[3];
employee foreman[15];
employee worker[75];
```

The array **manager** contains three objects(managers), namely, **manager**[0], **manager**[1] and **manager**[2], of type **employee** class. Similarly, the **foreman** array contains 15 objects (foreman) and the **worker** array contains 75 objects (workers).

OBJECTS AS FUNCTION ARGUMENTS:

Like any other data type, an object may be used as a function argument. This can be done in two ways:

- A copy of the entire object is passed to the function.
- Only the address of the object is transferred to the function.

The first method is called *pass-by value*. Since a copy of the object is passed to the function, any changes made to the object inside the function do not affect the object used to call the function.

The second method is *pass-by-reference*. When an address of the object is passed, the called function works directly on the actual object used in the call. This means that any changes made to the object inside the function will reflect in the actual object.

FRIENDLY FUNCTIONS:

The private members of a class cannot be accessed from outside class. In some situations, **C++ allows a common function to be made friendly with two or more classes, thereby allowing the common function to have access to the private data of these classes.** Such a function need not be a member of any of these classes. **The functions that are declared with the keyword friend are known as *friend functions*.**

A friend function have some 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 private part of a class without affecting its meaning.**
- **Usually, it has the objects arguments**.

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

```
/* Example program for Objects as function
   arguments and  Friend function */
#include<iostream.h>
class ABC;                  // forward declaration
class XYZ
{
        int  x;
public:
        void setvalue(int   i) {  x = i;  }
        friend float max(XYZ,  ABC);
                // Declaration of friend function in class 1
};
class ABC
{
         int a;
public:
        void setvalue(int   i) {  a = i;  }
        friend float max(XYZ,  ABC);
                // Declaration of friend function in class 2
};
```

```
void max(XYZ m, ABC n)      // definition of friend
                //and Objects as function arguments
{
        if(m.x >= n.a)
                cout << m.x;
        else
                cout<<n.a;
        }

int main()
{
        ABC   abc;
        abc.setvalue(10);
        XYZ   xyz;
        xyz.setvalue(20);
        max(xyz, abc);

        return 0;
}
```

RETURNING OBJECTS

A function can not only receive objects as arguments but also can return them.

Example:

```
complex sum(complex c1, complex c2)
{
        complex c3;             //Object c3 is created inside a function
        c3.x = c1.x + c2.x;
        c2.y = c1.y + c2.y;
        return(c3);             // Returns object c3.
 }
```

const MEMBER FUNCTIONS

If a member function does not alter any data in the class, then we may declare it as a **const** member function as follows:

```
void    mul(int, int)  const;
double   get_balance( )  const;
```

The qualifier **const** is appended to the function prototypes (in both declaration and definition). The compiler will generate an error message if such functions try to alter the data values.

CONSTRUCTORS AND DESTRUCTORS

Constructor is a 'special' member function whose task is to initialize the object of it's class. A ***constructor*** is a special method that creates and initializes an object of a particular class. It has the same name as its class and may accept arguments. In this respect, it is similar to any other function.

If you do not explicitly declare a constructor for a class, the C++ compiler automatically generates a **default constructor that has no arguments**.

A constructor is declared and defined as follows:

```
// class with a constructor
class integer
{
        int m,n;
public:
        integer(void);              // constructor declared
        …….
};
integer : : integer(void)           //constructor defined
{
        m=0; n=0;
}
```

The constructor functions have some special characteristics, These are :

- **They should be declared in the public section.**
- **They are invoked automatically when the objects are created.**
- **They do not have return values, not even void and therefore, and they cannot return values.**
- **They cannot be inherited, though a derived class can call the base class constructor.**
- **Like other C++ functions, they can have default arguments.**
- **Constructors cannot be virtual.**
- **We cannot refer to their addresses.**
- **An object with a constructor (or destructor) cannot be used as a member or union.**
- **They make 'implicit calls' to the operators new and delete when memory allocation is required.**

PARAMETERIZED CONSTRUCTORS

The constructors that can take arguments are called parameterized constructors. We must pass the initial values as arguments to the constructor function when an object is declared. This can be done in two ways:

- By calling the constructor explicitly
- By calling the constructor implicitly

```
Example:    1. Integer int1 = Integer(0,100);    //Explicit call
            2. Integer int1(0,100)               //Implicit call
```

CONSTRUCTOR OVERLOADING (or) MULTIPLE CONSTRUCTORS IN A CLASS

C++ Permits us to use more than one constructor in the same class. When more than one constructor function is defined in a class, we say that the constructor is overloaded.

CONSTRCTORS WITH DEFAULT ARGUMENTS

It is possible to define constructors with default arguments.

```
Example:    class complex
            {
                    float x, y;
              public:
                  complex( float real,   float imag=0 ); //Constructor with default argument
                    ……
            }
```

DYNAMIC INITIALIZATION OF OBJECTS

Class objects can be initialized dynamically too. That is to say, the initial value of an object may be provided during runtime.

COPY CONSTRUCTOR

A copy constructor is used to declare and initialize an object from another object. A copy constructor takes a reference to an object of the same class as itself as an argument.

For example, the statement:

```
integer I2(I1);
```

would define the object I2 and at the same time initialize it to the values of I1.

Another form of this statement is

```
integer I1 = I1;
```

The process of initializing through a copy constructor is known as copy initialization.

DYNAMIC CONSTRUCTOR

Allocation of memory to objects at the time of their construction is known as Dynamic construction of objects. The memory is allocated with the help of the new operator.

```
/* Example program for dynamic constructor and
   Destructor */
#include <iostream.h>
#include <string.h>
class String
{
        char *name;
        int length;
 public:
        String(char *s)
        {
                length = strlen(s);
                name = new char[length + 1];
                strcpy( name, s );
        }
        void display(void)
        {
                cout << name << "\n";
        }
        ~String( );
};
```

```
String :: ~String( )
{
        delete name;
}

void main( )
{
        String name1("Vinayaga ");
        String name2("College ");
        name1.display( );
        name2.display( );
}
```

DESTRUCTORS:

A ***destructor***, as the name implies, is used to destroy the objects that have been created by a constructor. Like a constructor, the destructor is a member function whose name is the same as the class name but is preceded by a **tilde(~).** For example , the destructor for the class String can be defined as shown below:

~String( ) { }

A destructor never takes any argument nor does it return any value. It will be invoked implicitly be the compiler upon exit from the program to clean up storage that is no longer accessible. It is a good practice to declare destructors in a program since it releases memory space for future use.

Whenever **new** is used to allocate memory in the constructors, we should use **delete** to free that memory.

const OBJECTS

We may create and use constant objects using **const** keyword before object declaration.

```
Example:    const  Matrix  X(m,n);
```

Any attempt to modify the values of m and n will generate compile-time error.

```
/* Program using class, object, member function, constructors and Destructors  for calculating area and perimeter of a circle */
#include <iostream.h>
#include <conio.h>
class circle
{
private:
        float radius, area, perimeter;
public:
        circle() {}         //Default constructor
        circle(float r)    //Parameterized Constructor
        {
                radius = r;
                area = 0;
                perimeter = 0;
        }
        circle(float a, float p, float r=25)
                // Constructor with Default Argument
        {
                radius = r;
                area = a;
                perimeter=p;
        }
        circle(circle & x) //Copy constructor
        {
                radius   = x.radius;
                area     = x.area;
                perimeter= x.perimeter;
        }

        void read(); //Member function1 Declaration
       // Member function 2 defined inside the class
        void display()
        {
        cout<<"\n\n Given radius is :";
        cout<<radius;
        cout<<"\n Area of the circle is :";
        cout<<3.14 * radius * radius;
        cout<<"\n Perimeter of the circle is :";
        cout<<2 * 3.14 * radius << endl;
        }
        ~circle() {}
        //Destructor
};
       //Member function1 defined outside the class
void circle::read()
{
        cout<<"\n Enter the radius of circle :";
        cin>>radius;
}
```

```
int main()
{
        clrscr();
        cout<<"\n Program for calculating area and
                perimeter of a circle";
        cout<<"\n---------------------------------------";
        float n;

        circle C1;

        circle C3(0,0);

        circle C4[10];

        C1.read();
        cout << "\n ** Default Constructor **\n";
        C1.display();    //

        cout<<"\n ** Parameterized Constructor and
                Dynamic Initialization of Objects **";
        cout<<"\n Enter radius value :";
        cin>>n;
        circle C2 = circle(n);
        C2.display();

        cout<<"\n ** Constructor with Default
                        arguments **";
        C3.display();

        cout<<"\n ***** Array of Objects *****";
        for(int i=0;i<2;i++)
        {
          C4[i].read();
          C4[i].display();
        }

        circle C5 = C1;
        circle C6(C2);

        cout<<"\n ***** Copy Constructor *****";
        C5.display();
        C6.display();
        getch();
        return 0;
}
```

OPERATOR OVERLOADING:

The mechanism of giving such special meanings to an operator is known as ***operator overloading***. We can overload (give additional meaning to) all the C++ operators except the following:

The operators that cannot be overloaded are:

- class member access operators (**. , .***)
- scope resolution operator(**: :**)
- sizeof operator(**sizeof**)
- conditional operator(**?:**)

To define an additional task to an operator, we must specify what it means in relation to the class to which operator is applied. This is done with the help of a special function, called ***operator function***.

General form of operator function is:.

```
return type  classname :: operator op(argument list)
{
        function body
}
```

where return type is the type of value returned by the specified operation and ***op*** is the operator being overloaded. The ***op*** is preceded by the keyword **operator**. **Operator *op*** is the function name.

Operator functions must be either member functions (non-static) or friend functions.

The operator functions are declared in the class using prototypes as follows:

```
vector operator+(vector);          // vector addition using member function
vector operator-( );               // unary minus using member function
friend vector operator+(vector, vector);   // vector addition using friend function
friend vector operator-(vector &a);        // unary minus using frien function
```

The process of operator overloading involves the following steps:

1. Create a class that defines the data type that is to be used in the overloading operation.
2. Declare the operator function operator ***op( )*** in the public part of the class. It may be either a member function or a **friend** function
3. Define the operator function to implement the required operations.

OVERLOADING UNARY OPERATOR: (An operator with only one operand is called Unary operator)

Unary operators are operators that work with only one operand. Example of unary operators include unary plus, unary minus operators(+,-), increment, decrement operators (++,--) etc.,

Unary Operator overloading → Using member function : **No argument**
→ Using friend function : **One argument**

OVERLOADING BINARY OPERATOR:(An operator with two operands is called Binary operator)

Binary operators are operators that work with two operands. Example of binary operators include arithmetic operators(+,-,*,/,%), arithmetic assignment operators(+=, -=, *=,/=), and comparison operators (<, >, <=, >=,==, !=).

Binary Operator overloading → Using member function : **One argument**
→ Using friend function : **Two arguments**

Rules for Overloading operators:

- Only existing operators can be overloaded. New operators cannot be created.
- The overload operator must have at least one operand that is of user-defined type.
- We cannot change the basic meaning of an operator.
- Overloaded operators follow the syntax rules of the original operator.
- Unary Operator overloading → Using member function : No argument
  → Using friend function : One argument
- Binary Operator overloading → Using member function : One argument
  → Using friend function : Two arguments
- When using binary operator overloading, the left hand operand must be an object of the relevant class.

Example Program for Unary operator overloading

```
/*Program for unary operator overloading [++, minus(-)] with member & friend functions. */
#include<iostream.h>
#include<conio.h>
class space
{       int a,b;
public:
        void input(int,int);
        void display();
        void operator++();
        friend void operator-(space &s);
};
void space::input(int x,int y)
{       a=x;
          b=y;
}
void space::display()
{     cout<<"a value is " <<a<<endl;
        cout<<"b value is: "<<b<<endl<<"\n";
}
void space::operator ++()
{       a = ++a;
        b = ++b;
}
void operator-(space &s)
{       s.a = -s.a;
        s.b = -s.b;
}
void main()
{       space S;
        clrscr();
        S.input(10,-20);
        S.display();
        ++S;
        S.display();
        -S;
        S.display();
        getch();
}
```

Example program for binary operator overloading

```
#include <iostream.h>
#include <conio.h>
const s=2;
class matrix
{
        int m[s][s];
public:
        matrix(){ }
        matrix(int x[][s]);
        matrix operator +(matrix B);
        friend matrix operator -(matrix A, matrix B);
        void display(matrix M);
};
matrix::matrix(int x[][s])
{       for(int i=0;i<s;i++)
                for(int j=0;j<s;j++)
                        m[i][j]=x[i][j];
}
matrix matrix::operator +(matrix B)
{       matrix C;
        for(int i=0;i<s;i++)
                for(int j=0;j<s;j++)
                        C.m[i][j] = m[i][j] + B.m[i][j];
        return C;
}
matrix operator -(matrix A, matrix B)
{       matrix C;
        for(int i=0;i<s;i++)
            for(int j=0;j<s;j++)
                C.m[i][j] = A.m[i][j] - B.m[i][j];
        return C;
}
void matrix::display(matrix M)
{       for(int i=0;i<s;i++)
        {
                for(int j=0;j<s;j++)
                {
                   cout<<M.m[i][j]<<"\t";
                }
                cout<<"\n";
        }
}

void main()
{
        int X[][s]={11,12,13,14};
        int Y[][s]={1,2,3,4};
        matrix M1(X);
        matrix M2(Y);
        matrix M3, M4;
        M3 = M1+M2;
        M4 = M1-M2;
        clrscr();
        cout<<"\n\n Matrix A \n";
        M1.display(M1);
        cout<<"\n\n Matrix B \n";
        M2.display(M2);
        cout<<"\n\n Matrix Addition \n";
        M3.display(M3);
        cout<<"\n\n Matrix Subtraction \n";
        M4.display(M4);
        getch();
}
```

TYPE CONVERSIONS:

Every expression has a type that is determined by the components of the expression. Consider the following statement:

```
int x = 5.5 / 2 ;          // x contains 2, the fraction part is lost.
```

Data can be lost when it is converted from a higher data type to a lower data type.

Casts: We can force an expression to be of a specific type by using a type cast operator. The general form:

```
type-name (expression).        //C++ notation
```

where type is a valid data type.

For example, to make sure that the expression x/2 evaluates to type float, write

```
float y = 5.5 / float (2)
```

Since the user-defined data types are designed by us to suit our requirements, the compiler does not support automatic type conversions for such data types.

Three types of situations might arise in the data conversion between incompatible types:

1. Conversion from basic type to class type.
2. Conversion from class type to basic type.
3. Conversion from one class type to another class type.

Type Conversions

Conversion required	Conversion takes place in	
	Source class	Destination class
Basic → Class	Not Applicable	Constructor
Class → Basic	Casting Operator	Not Applicable
Class → Class	Casting Operator	Constructor

Conversion function / Casting operator

```
operator typename( )
{       ……
        …… (Function statements
}
```

Important Note :

1. The constructors used for the type conversions take a single argument whose type is to be converted.
2. The casting operator should satisfy the following conditions : a. It must be a class member
   b. It must not specify a return type
   c. It must not have any arguments

```
/*Example program for Type conversion & String
 manipulation using binary operator overloading */
class String
{       char *name;
        int length;
  public:
        String() {}
        String(char *s)
        {       length = strlen(s);
                name = new char[length + 1];
                strcpy( name, s );
        }
        void display(void)
        {       cout << name << "\n";
        }
        String operator +(const String &t);
        operator char*()
        {       return(name);
        }
        ~String();
};
String :: ~String()
         {      delete name;
         }
```

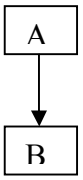
```
String String::operator +(const String &t)
{   String temp;
    temp.length = length + t.length;
    temp.name = new char[temp.length+1];
    strcpy(temp.name, name);
    strcat(temp.name, t.name);
    return (temp.name);
}
void main( )
{
        char* name1="Vinayaga ";
        char* name2="College";
        clrscr();
        String S1(name1);
        String S2 = name2;
        cout<<"\nGiven Strings are : \n";
        S1.display( );
        S2.display( );
        String S3;
        S3 = S1 + S2;
        char* N = S3;
        cout<<"The Joined string is : “<<N<<"\n";
        getch();
}
```

UNIT – IV - INHERITANCE

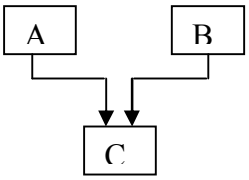
C++ support the concept of ***reusability***. The mechanism of deriving a new class from an old one is called ***inheritance (or derivation)***. The old class is referred to as the ***base class*** or ***super class*** and new one is called ***derived class*** or ***sub class***.

Forms of inheritance:

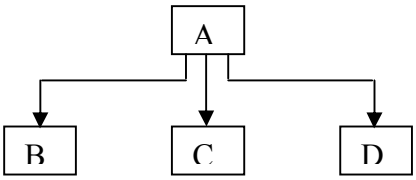
1. Single Inheritance (A derived class with only one base class).
2. Multiple inheritance (A derived class with several base classes).
3. Hierarchical inheritance (One base class with several derived classes).
4. Multilevel inheritance (Deriving a class from another 'Derived class').
5. Hybrid Inheritance (Combination of two or more types of inheritances).



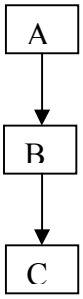
i Single inheritance



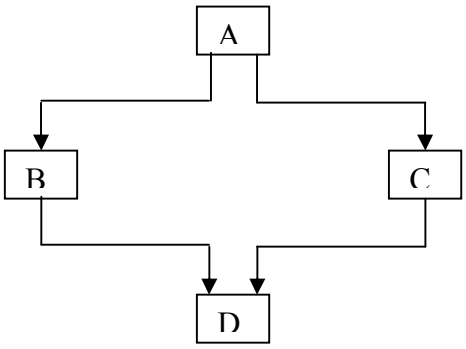
ii Multiple inheritance



iii Hierarchical inheritance



iv. Multilevel Inheritance



v. Hybrid inheritance

DEFINING DERIVED CLASSES:

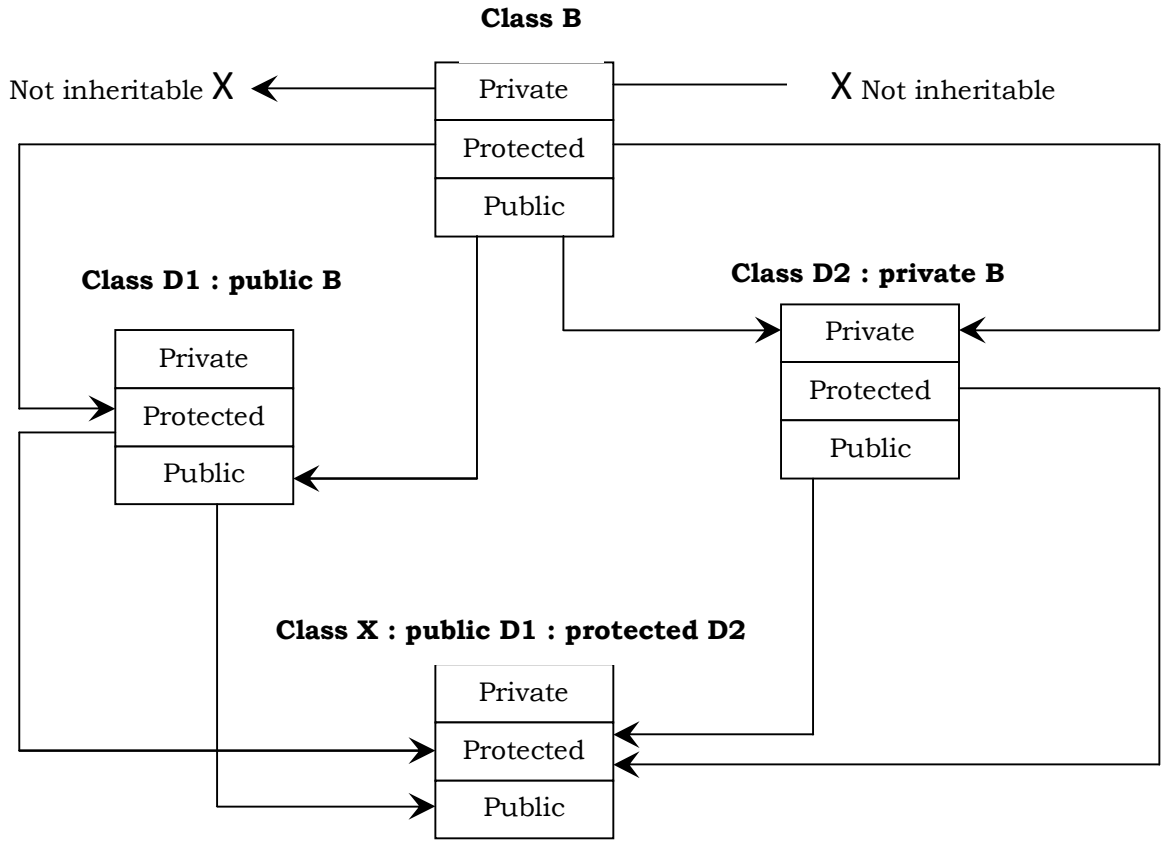
```
class derived-class-name : visibility-mode base-class-name
{       ……//
        ……//  members of derived class
};
```

The colon indicates that the ***derived-class-name*** is derived from the ***base–class–name***. The ***visibility-mode*** is optional and, if present, may be either private or public. The default visibility-mode is private. Visibility mode specifies whether the features of the base class are privately derived or publicly derived.

Example:

```
class ABC : private XYZ //Private derivation
 {
       members of ABC;
 };

class ABC : XYZ   //private declaration by default
{
     members of ABC;
};
```

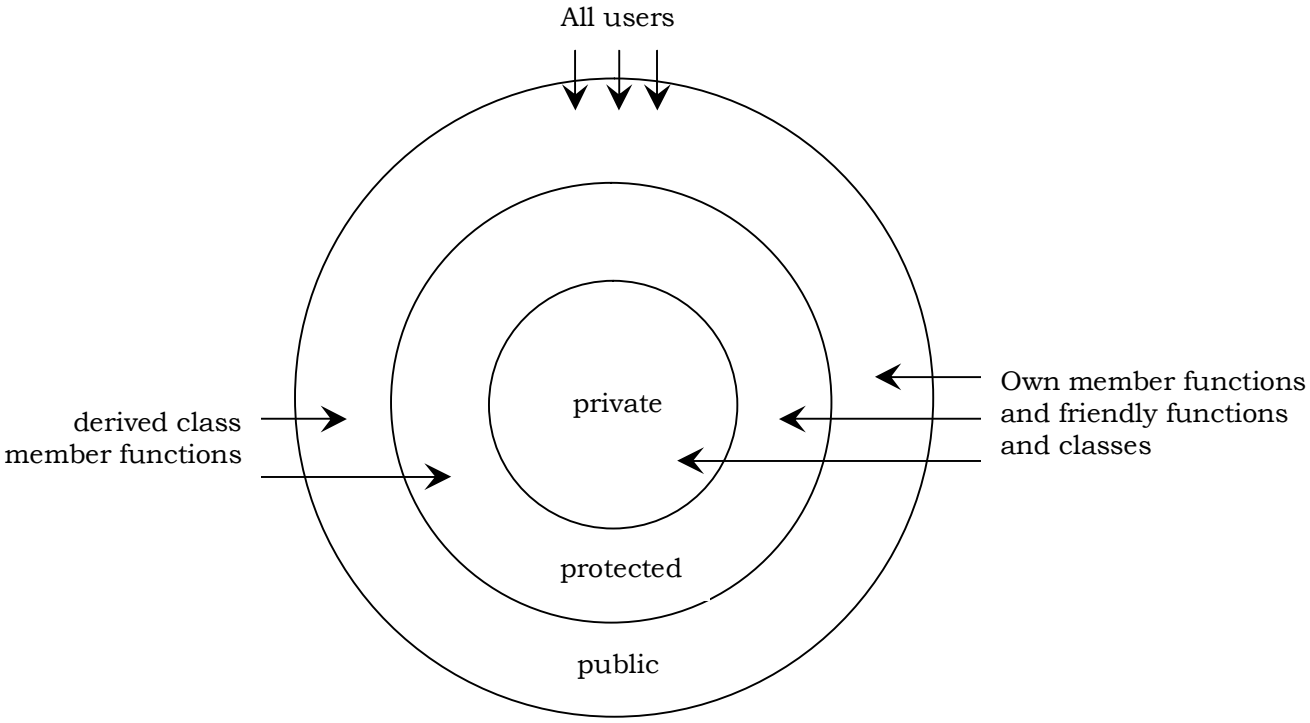
```
class ABC : public XYZ //Public derivation
{
     members of ABC
};
```



Effect of inheritance on the visibility of members

Visibility of inherited members

Base class Visibility	Derived class visibility		
	Public derivation	**Private derivation**	**Protected derivation**
Private	Not inherited	Not inherited	Not inherited
Protected	Protected	Private	Protected
Public	Public	Private	Protected



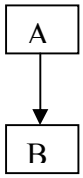
Private: The private members of a class can be accessed only by the member functions of that class.

Protected: The protected members of a class can be accessed only by the member functions of that class, and the member functions of the classes derived from it.

Public: The public members of a class can be accessed anywhere in the application.

SINGLE INHERITANCE

A derived class with only one base class**.**



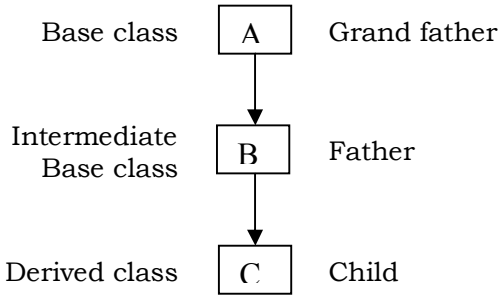

```
/* Lab 4(a): Example Program using
   Single Inheritance */

#include <iostream.h>
#include <conio.h>

class base
{       int x;
public:
        void setx(int n)
        {
                x = n;
        }
        void showx ( )
        {
                cout<<"X = "<<x<<"\n";
        }
};
```

```
class derived : public base
{       int y;
public:
        void sety(int n)
        {
                y = n;
        }
        void showy ( )
        {
                cout<<"Y = "<<y<<"\n";
        }
};
void main()
{
        derived D;
        D.setx(10);
        D.sety(20);
        clrscr();
        cout<<"Single Inheritance program \n";
        D.showx();
        D.showy();
        getch();
}
```

MULTILEVEL INHERITANCE:

A class is derived from another derived class is called multilevel inheritance

```
class A {….};                       // base class
class B      :  public A {….};      // B derived from A
class C      :  public C {….};      // C derived from B
```

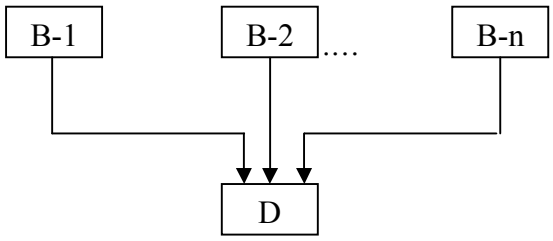


```
/* Program using Multilevel Inheritance */
#include <iostream.h>
#include <conio.h>
class base
{       int x;
public:
        void setx(int n)
        {       x = n;
        }
        void showx ()
        {       cout<<"X = "<<x<<"\n";
        }
};

class derived1 : public base
{       int y;
public:
        void sety(int n)
        {       y = n;
        }

        void showy ()
        {       cout<<"Y = "<<y<<"\n";
        }
};
```

```
class derived2 : public derived1
{       int z;
public:
        void setz(int n)
        {       z = n;
        }

        void showz ()
        {       cout<<"Z = "<<z<<"\n";
        }
};

void main()
{
        derived2 D;
        D.setx(10);
        D.sety(20);
        D.setz(30);
        clrscr();
        cout<<"Multilevel Inheritance program \n";
        D.showx();
        D.showy();
        D.showz();
        getch();
}
```

MULTIPLE INHERITANCE :

Create a new class from more than one base class is called multiple inheritance.



Multiple inheritance

The syntax of a derived class with multiple base class :

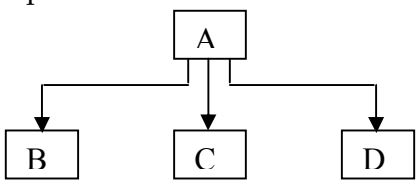
```
class D :  visiblility B-1,  visibility B-2…
{       ……
        …… (Body of D)
        ……
};
```

Example :

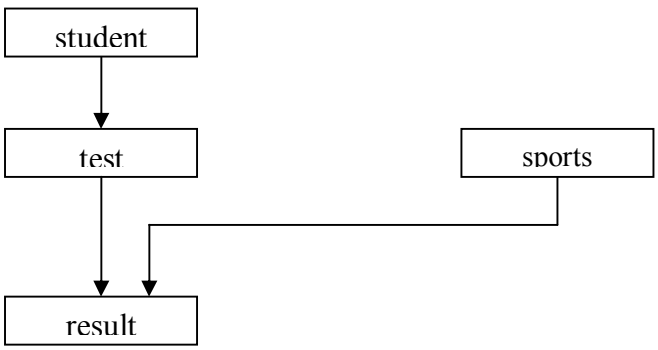
```
class P : public M, public N
{
  public:
        void display(void);
}
```

```
/* Lab 4(b) : Program using Multiple Inheritance */
#include <iostream.h>
#include <conio.h>
class base1
{
        int x;
public:
        void setx(int n)
        {
                x = n;
        }
        void showx ()
        {
                cout<<"X = "<<x<<"\n";
        }
};
class base2
{
        int y;
public:
        void sety(int n)
        {
                y = n;
        }
        void showy ()
        {
                cout<<"Y = "<<y<<"\n";
        }
};
class derived : public base1, public base2
{
        int z;
public:
        void setz(int n)
        {
                z = n;
        }
        void showz ()
        {
                cout<<"Z = "<<z<<"\n";
        }
};

void main()
{
        derived D;
        D.setx(10);
        D.sety(20);
        D.setz(30);
        clrscr();
        cout<<"Multiple Inheritance program \n";
        D.showx();
        D.showy();
        D.showz();
        getch();
}
```

HIERARCHICAL INHERITANCE: One base class many sub class is called hierarchical inheritance. The base class will include all the features that are common to the subclasses. A subclass can be constructed by inheriting the properties of the base class.



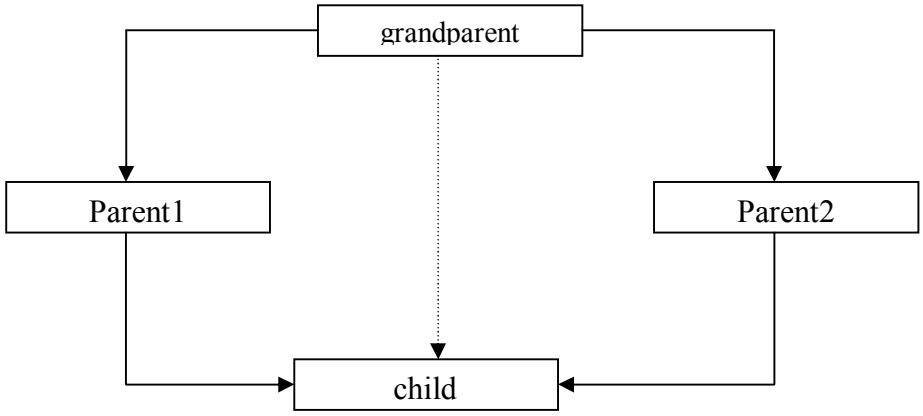
HYBRID INHERITANCE: Combination of two or more types of inheritances is called hybrid inheritance.



Multilevel, Multiple inheritance

VIRTUAL BASE CLASS:

The 'child' has two ***direct base classes*** 'parent1' and 'parent2' which themselves have a common base class 'grandparent'. The 'child' inherits the traits of 'grandparent' via two separate paths. It can also inherit directly as shown by the broken line. The 'grandparent' is sometimes referred to as ***indirect base class***.



Inheritance by the 'child' some problem. All the public and protected members of 'grandparent' are inherited into 'child' twice, first via 'parent1' and again via 'parent2'. This means 'child ' have ***duplicate*** set of the members inherited from 'grandparent'.

Duplication of inherited members due to multiple paths can be avoided by making the common base class (ancestor class) as *virtual base class* while declaring the direct or intermediate base classes.

```
class A                          // grandparent
{
……
……
};
class B1:virtual public A        // parent1
{
……
……
};
class B2 : public virtual A      // parent2
{
……
……
};
class C : public B1, public B2   // child
{
……                               // only one copy of A
…...                             // will be inherited
};
```

When a class is made a ***virtual*** base class, c++ takes necessary care to see that only one copy of that class is inherited, regardless of how many inheritance paths exist between the virtual base class and a derived class.

ABSTRACT CLASS:

An abstract class is one that is not used to create objects. An abstract class is designed only to act as a base class to be inherited by other classes. It is a design concept in program development and provides a base upon which other classes may be built. (Example: Class **base** in Multilevel Inheritance program)

```
/* Program using Multiple, Multilevel,
   Hierarchical, Hybrid Inheritance, Virtual Base
   class and Pure Virtual function, Abstract class */

#include <iostream.h>
#include <conio.h>

class GP
{
protected:
        int x;
public:
        virtual void set( ) = 0;
        virtual void show( ) { };
};

class P1 : virtual public GP
{
protected:
        int y;
};

class P2 : public virtual GP
{
protected:
        int z;
};
```

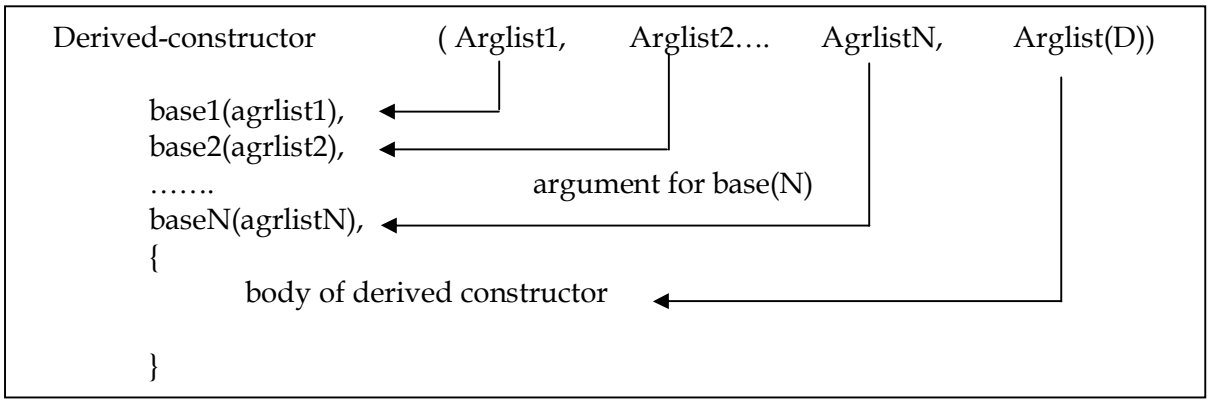
```
class child : public P1, public P2
{
public:
        void set( )
        {       cout << "\n Enter X value : ";
                cin >> x;
                cout << "\n Enter Y value : ";
                cin >> y;
                cout << "\n Enter Z value : ";
                cin >> z;
        }
        void show ( )
        {       cout<<"\n X = "<<x<<"\n ";
                cout<<"\n Y = "<<y<<"\n ";
                cout<<"\n Z = "<<z<<"\n ";
        }
};
void main()
{
        GP *G;
        child D;
        G = &D;
        clrscr();
        cout<<"Inheritance, Virtual Base class, ";
        cout<<"Virtual Function program \n";
        G->set();
        G->show();
        getch();
}
```

Ambiguity may arise in inheritance applications. A function with the same inheritance appears in both base class and derived classes. We may solve this problem by using the scope resolution operator. For instance, consider the following situation:

```
class A
{
public:
        void display( )
        {  cout << "A \n";  }
};

class B
{
 public:
        void display( )
        { cout << "B \n";   }
};

void main( )
{
        B b;                    // derived class object
        b.display( );           // invokes display( ) in class B
        b.A::display( );        // invokes display( ) in class A
        b.B::display( );        // invokes display( ) in class B
}
```

CONSTRUCTORS IN DERIVED CLASS:

The constructor of the derived class receives the entire list of values as its arguments and passes them on to the base constructors in the order in which they are declared in the derived class. The base constructors are called and executed before executing the statements in the body of the derived constructor. The general form of defining a derived constructor is :




```
/* Example program for Constructors in Derived
   classes */
#include<iostream.h>
#include<conio.h>
class one
{
protected:
        int a;
public:
        one(int x)
        {
              a=x;
        }
        void show( )
        {
              cout<<"\n a value is : "<<a<<endl;
        }
};
class two
{
protected:
        float b;
public:
        two(float y)
        {
             b=y;
        }
        void show1( )
        {
             cout<<"\n b vlaue is : "<<b<<endl;
        }
};
```

```
class three : public one, public two
{
protected:
        int c,d;
public:
        three(int x,float y,int k,int p):one(x),two(y)
        {
        c=k;
        d=p;
        }

        void show2( )
        {
        cout<<"\n c vlaue is : "<<c<<endl;
        cout<<"\n d vlaue is : "<<d<<endl;
        }
};

void main()
{
      three T(4,  6.0,  10,  20);
      clrscr( );
      T.show( );
      T.show1( );
      T.show2( );
      getch();
}
```

POINTERS, VIRTUAL FUNCTIONS AND POLYMORPHISM

POLYMORPHISM

- **Polymorphism simple means one name having multiple forms**.
- There are two types of polymorphism, namely, **compile time polymorphism and run time polymorphism.**
- **Functions** and **operators overloading** are examples of **compile time polymorphism**. The overloaded member functions are selected for invoking by matching arguments both type and number at the compile time. It means that an object is bound to its function call at compile time. **This is called early or static binding or static linking.**
- In **run time polymorphism**, an appropriate member function is selected with the help of virtual functions while the program is running. **It is called late or dynamic binding** because the appropriate function is selected dynamically at run time. Dynamic biding requires use of pointers to objects.

POINTERS

A pointer is a variable that stores the memory address of another variable.

Advantages of using pointers:

- pointers allow direct access to individual bytes in the memory. Thus, data in memory is accessed faster than through ordinary variables. This speeds up the execution of programs.
- Pointers allow direct access to output devices, like the monitor. This speeds up programs that are graphics intensive.
- Pointers allow the program to allocate memory dynamically, only when required, with the help of the new operator. They also allow the program to free the memory when it is no longer required. This is done with the help of the delete operator.

A Pointer Variable:

Every byte in memory is given unique address (location number) by the operating system. The name of the variable is the location name given to the byte(s), in the program. The address of a variable is the location number of the first byte occupied by the variable. This address can never be negative and is usually a very large number.

Pointer to Objects:

A pointer can point to an object created by a class. Object pointers are useful in creating object at run time. We can use the object pointer to access the public members of an object. Consider a class **item** defined as follows:

```
class item
{
        int code;
 public:
        void get( )
        {       cout<<"Enter code : \n";
              cin >> code;
       }
        void show()
        {       cout<<"Code="<<code<<"\n";      }
};
```

Let us declare an **item** variable **x** and a pointer **ptr** to **x** as follows:

```
item  x;
item *ptr = &x;
```

The pointer **ptr** is initialized with the address of **x**. We can refer to the member function of **item** in two ways, one by the ***dot operator*** and the *object*, and another by the ***arrow operator*** and the *object pointer*. The statements

```
x.get( );
x.show( );
```

are equivalent to

```
ptr->get( );
ptr->show( );
```

since ***ptr** is an alias of **x**, we can also use the following method:

```
(*ptr).show( );
```

the parentheses are necessary because the dot operator has higher precedence than the *indirection operator*.*

```
item *ptr = new item;
```

This statement allocates enough memory for the data members in the object structure and assigns the address of the memory space to **ptr**. Then **ptr** can be used to refer to the members as shown below:

```
ptr->show( );
```

Array of pointers to objects:

We can create array of objects using pointers.

```
class item
{
        int code;
public:
        void get( )
        {       cout<<"Enter code : \n";
                cin >> code;
        }
        void show()
        {       cout<<"Code="<<code<<"\n";      }
};
```

```
void main( )
{
        item x;
        item *ptr = new item[5];
                //Array of 5 pointers to item objects
        for(int i = 0; i<5;i++)
        {
            ptr->get( );
            ptr->show( );
        }
}
```

This Pointer:

A **this** pointer refers to an **object that currently invokes** a member function. For example, the function call **a.show( )** will set the pointer '**this'** to the **address of the object a**.

```
Example :       class one
                {
                        int a;
                };
```

The private variable a can be used directly inside a member function, like **a=123**;
We can also use the following statement to access the variable : **this->a=123.**

When a binary operator is overloaded using a member function, we pass only one argument to the function. The other argument is implicitly passed using the pointer **this.**

One important application of the pointer **this** is to return the object it points to. For example, the statement

```
return *this;
```

inside a member function will return the object that invoked the function. This statement assumes importance when we want to compare two or more objects inside a member function and return the invoking object as a result.

```
Example :       person & person :: greater (person & x)
                {
                        if(x.age > age)
                                return x;
                        else
                                return *this;
                }
```

Virtual Functions

Virtual functions are used to achieve run time polymorphism. The function in base class is declared as virtual using the keyword ***virtual*** preceding its normal declaration.

When a function is made ***virtual***, ***C++*** determines which function to use at run time based on the type of object pointed to by the base pointer, rather than the type of the pointer. Thus, by making the base pointer to point to different objects, we can execute different versions of the ***virtual*** function.

Run time polymorphism is achieved only when a virtual function is accessed through a pointer to the base class.

Rules for Virtual Functions:

When virtual functions are created for implementing late binding, we should observe some basic rules that satisfy the compiler requirements:

- The virtual functions must be members of some class.
- They cannot be static members
- They are accessed by using object pointers
- A virtual function can be a friend of another class
- A virtual function in a base class must be defined, even though it may not be used.
- The prototype of the base class version of a virtual function and all the derived class versions must be identical. If two functions with the same name have different prototype, C++ considers them as overloaded functions, and the virtual function is ignored.
- We cannot have virtual constructors, but we can have virtual destructors.
- If a virtual function is defined in the base class, it need not be necessarily redefined in the next object.

Pure Virtual Functions:

It is normal practice to declare a function virtual inside the base class and redefine it in the derived classes. The function inside the base class is seldom used for performing any task. It only serves as a placeholder. Such functions are called "do nothing" functions.

A "do-nothing" function may be defined as follows:

```
virtual void set( )=0;
```

Such functions are called pure virtual function.

A class containing pure virtual functions cannot be used to declare any objects of its own. As stated earlier, such classes are called **abstract base classes**. The main objective of an abstract base class is to provide some traits to the derived classes and to create a base pointer required for achieving runtime polymorphism.

UNIT – V - WORKING WITH FILES

File : A file is a collection of related data stored in a particular area on the disk.

Stream : It refers to a sequence of bytes.

Text file : It is a file that stores information in ASCII characters. In text files, each line of text is terminated with a special character known as EOL (End of Line) character or delimiter character. When this EOL character is read or written, certain internal translations take place.

Binary file : It is a file that contains information in the same format as it is held in memory. In binary files, no delimiters are used for a line and no translations occur here.

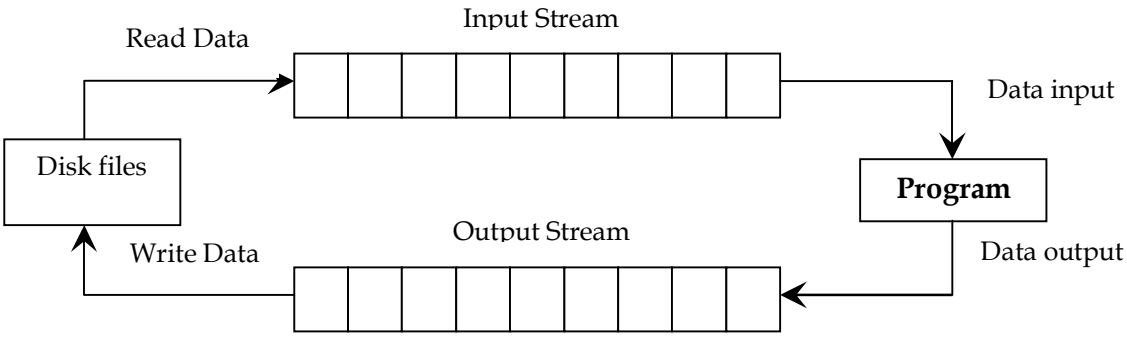
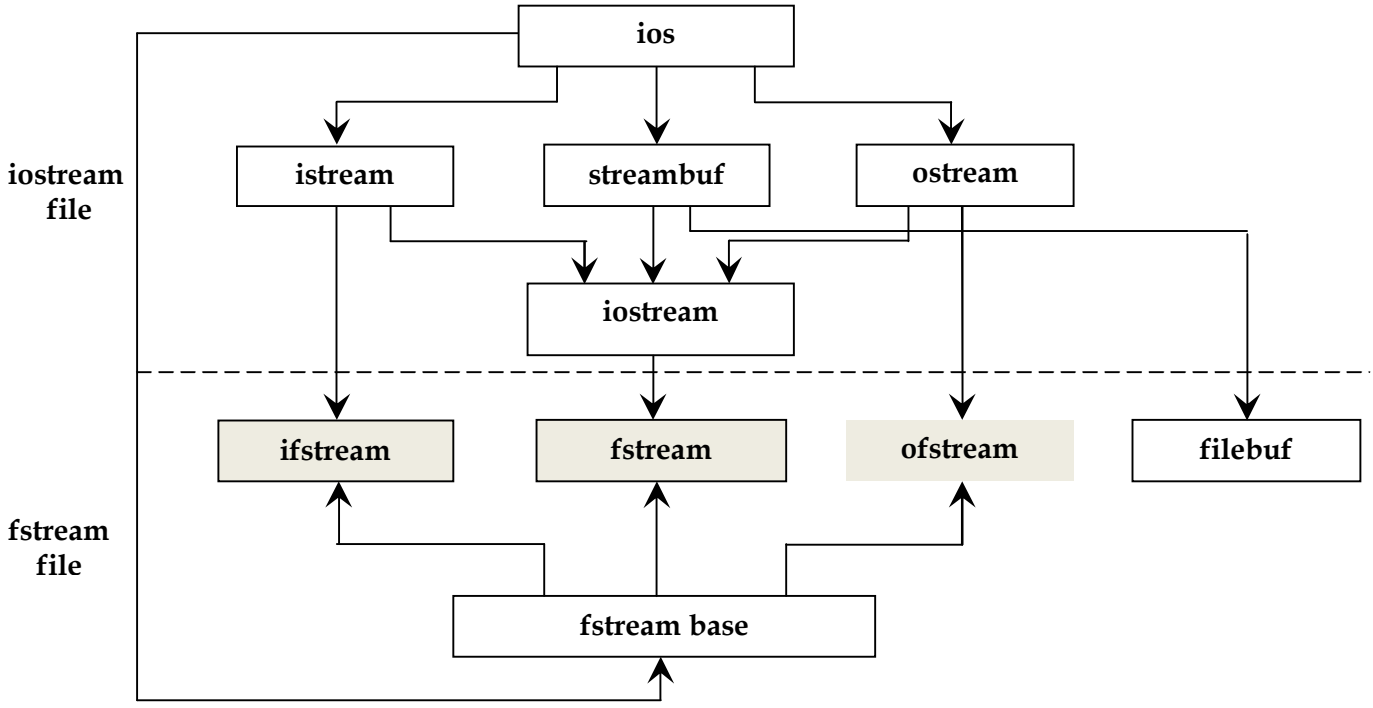


Fig. File input and output streams



Stream classes for file operations (contained in fstream file)

filebuf : Its purpose is to set the file buffers to read and write.
fstreambase : Provides operations common to the file streams. Serves as the base for fstream, ifstream and ofstream class.
ifstream : Provides input operations. Contains open( ) with default input mode
ofstream : Provides output operations. Contains open( ) with default output mode
fstream : Provides support for simultaneous input and output operations. Contains open with default input mode.

OPENING AND CLOSE A FILE

OPENING FILE USING CONSTRUCTOR

```
ofstream fout("results");        //output only
ifstream fin("data");            //input only
```

OPENING FILE USING open()

Syntax : **file-stream-class** *stream-object;*
stream-object.**open(***"filename"***);**

Example :
```
ofstream outfile;
outfile.open("data1");
ifstream infile;
infile.open("data2");
```

MORE OPEN( ) FILE MODES :

The general form of the function **open( )** with two arguments is:

```
stream-object.open("filename", mode);
```

The second argument (called file mode parameter) specifies the purpose for which the file is opened.

File mode parameter	Meaning
ios::app	Append to end of file
ios::ate	go to end of file on opening
ios::binary	file open in binary mode
ios::in	open file for reading only
ios::out	open file for writing only
ios::nocreate	open fails if the file does not exist
ios::noreplace	open fails if the file already exist
ios::trunc	delete the contents of the file if it exist

All these flags can be combined using the bitwise operator OR (|). For example, if we want to open the file example.bin in binary mode to add data we could do it by the following call to member function open():

```
fstream file;
file.open ("example.bin", ios::out | ios::app | ios::binary);
```

CLOSING FILE

```
fout.close();
fin.close();
```

DETECTING END-OF FILE using ifstream Object :

Detecting of the end-of file is necessary for preventing any further attempt to read data from the file.

```
Example :   ifstream fin;
            fin.open("data");
            while ( fin )
            {       fin.getline(line,N);
                    cout << line;
            }
```

An ifstream object, such as fin, returns a value **0** if any error occurs in the file operation including the end-of-file condition. Thus, the while loop terminates when **fin** returns a value **0** on reaching the end-of-file condition. (This loop may terminate due to other failures as well.)

DETECTING END-OF FILE using the member function eof( ):

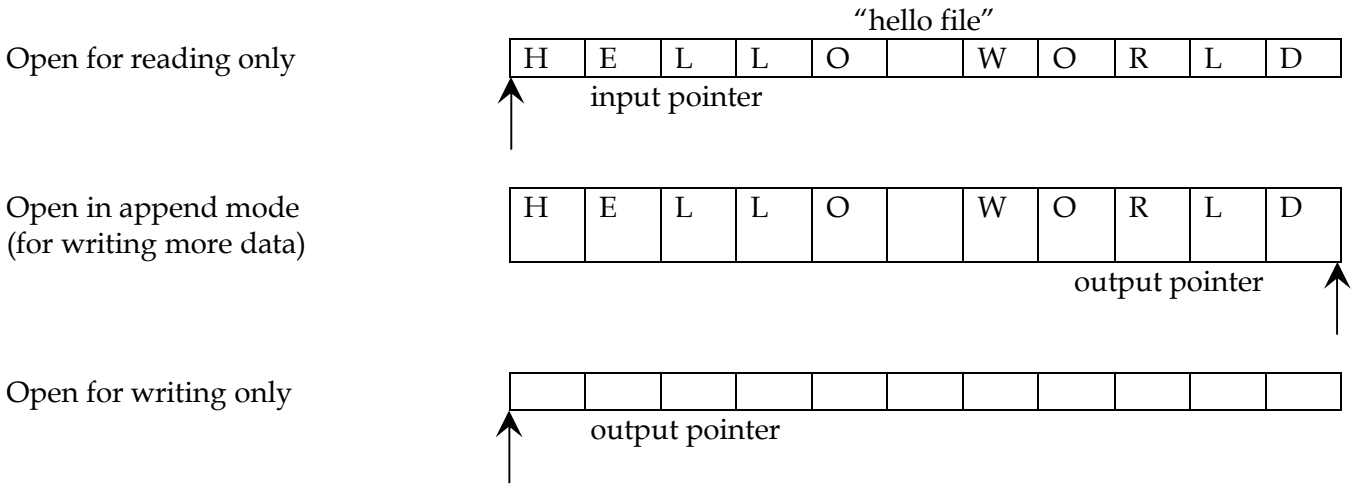
```
Example :   ifstream fin;
            fin.open("data");
            while ( fin.eof( )!=0 )
            {     exit(1);     }
```

The **eof( )** is a member function of **ios** class. It returns a non-zero value if the end-of-file(**EOF**) condition is encountered, and a zero, otherwise. Therefore, the above statement terminates the program on reaching the end of file.

FILE POINTERS AND THEIR MANIPULATION

- Each file has associated pointers known as **file pointers**.
- One of them is called the **input pointer** (or get pointer) used for reading the contents of a given file location.
- The other is called the **output pointer** (or put pointer) used for the writing to a given file location.
- We can use these pointers to move through the files while reading or writing.

Default Actions



Action on file pointers while opening a file

FUNCTIONS FOR MANIPULATION OF FILE POINTERS :

The input and output file pointers can be manipulated using the following member functions:

seekg()	moves get pointer(input) to a specified location
seekp()	moves put pointer (output) to a specified location
tellg()	gives the current position of the get pointer
tellp()	gives the current position of the put pointer

The prototype for these functions is:

Syntax: **seekg(** offset, refposition **);**
seekp(offset, refposition **);**

The parameter offset represents the number of bytes the file pointer is to be moved from the location specified by the parameter refposition.

The refposition takes one of the following three constants defined in the ios class.

ios::beg	start of the file
ios::cur	current position of the pointer
ios::end	end of the file

Example:

***file*.seekg(-10, ios::cur);**

Pointer offset calls	
fout.seekg(0, ios::beg);	Go to start
fout.seekg(0, ios::cur);	Stay at current position
fout.seekg(0, ios::end);	Go to the end of file
fout.seekg(m, ios::beg);	Move to (m+1)th byte in the file
fout.seekg(m, ios::cur);	Go forward by m bytes from the current position
fout.seekg(-m, ios::cur);	Go backward by m bytes from the current position
fout.seekg(-m, ios::end);	Go backward by m bytes from the end.

SEQUENTIAL INPUT AND OUTPUT OPERATIONS ON FILES

put() and get() function

The function put() writes a single character to the associated stream.

The function get() reads a single character form the associated stream.

Example:
```
char ch;
file.get(ch);
file.put(ch);
```

write() and read() function :

write() and read() functions write and read blocks of binary data.

Example:
```
infile.read((char *) & obj, sizeof(obj));
outfile.write((char *) & obj, sizeof(obj));
```

UPDATING A FILE : RANDOM ACCESS

Updating is a routine task in the maintenance of any data file. The updating would include one or more of the following tasks:

- Displaying the contents of a file.
- Modifying an existing item.
- Adding a new item.
- Deleting an existing item.

These actions require the file pointers to move to a particular location. This can be done if we have a collection of items/objects of equal lengths.

First, we should find the size of each object by using the statement

```
int object_length = sizeof(object);
```

Then, the location of a desired object, say the mth object, may be obtained as follows:

```
int location = m * object_length;
```

The **location** gives the byte number of the first byte of the mth object. Now, we can set the file pointer to reach this byte with the help of seekg( ) or seekp( ).

We can also find out the total number of objects in a file using the object_lenth as follows:

```
int n = file_size / object_lenth;
```

The file_size can be obtained using the function tellg( ) and tellp( ).

ERROR HANDLING DURING FILE OPERATIONS

One of the following things may happen when dealing with the files:

1. Opening a file for reading does not exist
2. Filename given for a new file already exist
3. Reading past the end-of-file (End-file condition encountered)
4. There is not enough space on disk for storing a file.
5. An invalid filename given. (Filename with special characters)
6. Reading a file in output mode and Writing a file in input mode. (Opening a file with wrong open modes)

Error handling functions

FUNCTION	RETURN VALUE AND MEANING
eof( )	returns true (non zero) if end of file is encountered while reading; otherwise return false (zero)
fail( )	return true when an input or output operation has failed
bad( )	returns true if an invalid operation is attempted or any unrecoverable error has occurred.
good( )	returns true if no error has occurred.

```
Example :   ifstream  infile;
            infile.open("ABC");
            while(infile.fail( ))
            {       ……
                    ( process the file )
            }
            if( infile.eof( ) )
            {       ……
                    ( terminate program normally
            )
            }
```

```
else
        if( infile.bad( ) )
        {
                …… ( report fatal error )
        }
                else
                {
                infile.clear( );   // clear error state
        ……
        }
        ……
```

COMMAND-LINE ARGUMENTS

Command line arguments are optional string arguments that a user can give to a program upon execution. These arguments are passed by the operating system to the program, and the program can use them as input.

File names may be supplied as arguments to the main( ) function at the time of invoking the program. These arguments are known as **command-line arguments**.

The main( ) function may take two arguments as shown below:

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

- The first argument **argc** (known as argument counter) represents the number of arguments in the command line.
- The second argument **argv** (known an argument vector) is an array of **char** type pointers that points to the command line arguments.
- The size of this array will be equal to the value of argc.

```
Example :   C > exam data results
            argc = 3                    argv[0] ---> exam
                                        argv[1] ---> data
                                        argv[2] ---> results
```

```
/* Lab 5 : Program for create, read and write in files */
#include <iostream.h>
#include <fstream.h>
#include <stdlib.h>
#include <conio.h>
void main( int argc, char * argv[ ] )
{
        int number[9] = { 11,12,13,14,15,16,17,18,19 };
        clrscr( );
        if ( argc != 2)
        {
                cout << "argc = " << argc << "\n";
                cout << "Error in arguments \n";
                exit(1);
        }
        ofstream fout;
        fout.open(argv[1]);
        if ( fout.fail( ) )
        {
                cout << "Could not open file "
                     << argv[1] << "\n";
                exit(1);
        }
        for ( int i = 0; i<9 ; i++)
        {
                fout << number[i] << " ";
        }

        fout.close( );

        ifstream fin;
        fin.open(argv[1]);
        char ch;

        while(fin.eof()==0)
        {
                fin.get(ch);
                cout << ch;
        }

        cout << "\n \n";
        fin.close( );
        getch( );

}
```