UNIT I

DATA ABSTRACTION & OVERLOADING

Overview of C++ – Structures – Class Scope and Accessing Class Members – Reference Variables – Initialization – Constructors – Destructors – Member Functions and Classes – Friend Function – Dynamic Memory Allocation – Static Class Members – Container Classes and Integrators – Proxy Classes – Overloading: Function overloading and Operator Overloading.

I. INTRODUCTION

Procedure-Oriented Programming

Problem is a sequence of steps to be done, such as *reading, calculating and printing*. A number of functions are written to accomplish this task.

A typical program structure for POP is shown Fig. 1.1.1(a).

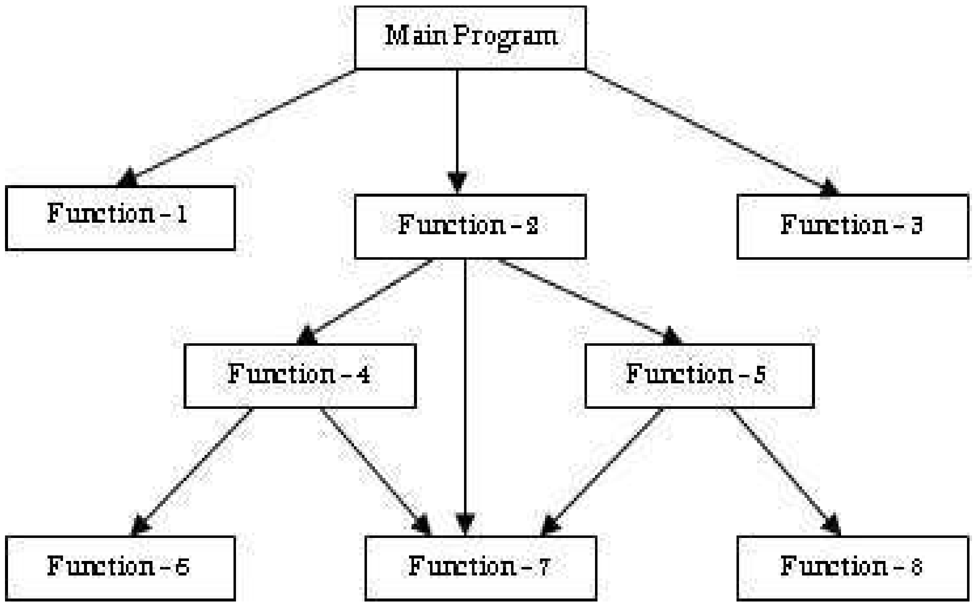


*Fig.1.1.1(a). **T**ypical structure of POP*

POP is an approach in which a list of instructions were written for the computer to follow, and organizing these instructions into groups known as *functions*. It uses a flowchart to organize these actions and to represent the flow of control from one action to another action. In a multi – function program, many important data are placed as global so that they may be accessed by all the functions. Each function has its own *local data*. Fig. 1.1.1(b) shows the relationship of data and functions in a POP.

Drawback of POP

1. When there is a change in the function the global data will change. *Global data* are more vulnerable to an inadvertent change by a function. In a large program it is very difficult to identify the data which is used by a function.
2. When the external data structure is revised it is necessary to revise all functions that access the data.
3. Functions correspond to the elements of the problem as they are action-oriented, they does not model real world problems

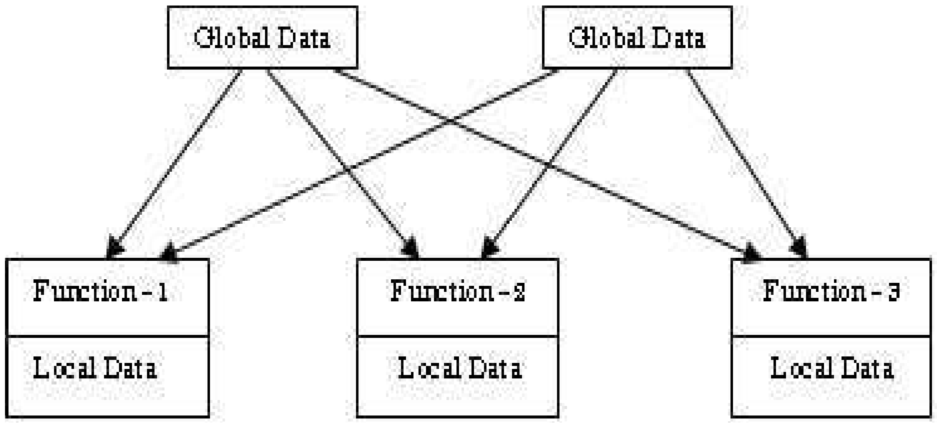


*Fig.1.1.1(b). **R**elationship of data and functions in POP*

OOP Definition

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

Basic concepts of OOP

Important object oriented concepts are namely:

1. Objects
2. Classes
3. Inheritance
4. Data Abstraction

5. Data Encapsulation
6. Polymorphism
7. Dynamic Binding
8. Reusability and Message Passing

Objects

- ✓ Object is the basic unit of object-oriented programming.
- ✓ Objects are identified by its unique name.
- ✓ An object represents a particular instance of a class.
- ✓ There can be more than one instance of an object. Each instance of an object can hold its own relevant data.
- ✓ An Object is a collection of data members and associated member functions. It is also known as methods.

Classes

- ✓ Classes are data types based on which objects are created. Objects with similar properties and methods are grouped together to form a Class. Thus a Class represents a set of individual objects.
- ✓ Characteristics of an object are represented in a class as **Properties (Attributes)**.
- ✓ The actions that can be performed by objects are functions of the class and they are referred to as **Methods (Functions)**. For example if we have a Class of *Cars* under which *Santro Xing*, *Alto* and *WaganR* represents individual Objects. In this context each *Car* Object will have its own, Model, Year of Manufacturing, Colour, Top Speed, Engine Power etc., which form **Properties** of the *Car* class and the associated actions i.e., object. Functions like Start, Move, Stop form the **Methods** of *Car* Class.
- ✓ No memory is allocated when a class is created. Memory is allocated only when an object is created, i.e., when an instance of a class is created.

Inheritance

- ✓ Inheritance is the process of forming a new class from an existing class or *base class*. The base class is also known as *parent class* or *super class*, A new class that is formed from parent class is called *derived class*. Derived class is also known as a *child class* or *sub class*.
- ✓ *Advantage* : Inheritance helps in reducing the overall code size of the program, which is an important concept in object-oriented programming.

Data Abstraction

- ✓ Data Abstraction increases the power of programming language by creating user defined data types.
- ✓ Data Abstraction also represents the needed information in the program without presenting the details.

Data Encapsulation (Data Hiding)

- ✓ Data Encapsulation combines data and functions into a single unit called Class.
- ✓ When using Data Encapsulation, data is not accessed directly; it is only accessible through the functions present inside the class.
- ✓ Data Encapsulation enables the important concept of data hiding possible.

Polymorphism

- ✓ Polymorphism, a Greek term, means the ability to take more than one form.
- ✓ Polymorphism allows the routines to use variables of different types at different times.
- ✓ Polymorphism refers to a single functional or multi-functional operator performing in different ways. (Operator Overloading).

Dynamic Binding

- ✓ Binding refers to the linking of a procedure call to the code which is to be executed in a response to the call.
- ✓ Dynamic binding means that the time of the call at run-time.

Message Passing

- ✓ An OOPS consists of a set of objects that communicate with each other.
- ✓ The process of programming in an object-oriented language, involves the following steps:
  a. Creating classes that define objects and their behavior,
  b. Creating objects from class definitions,
  c. Establishing communication among objects.

Reusability

- ✓ This term refers to the ability for multiple programmers to use the same written and debugged existing class of data. This is a time saving method and adds code efficiency to the language.

Features of OOP

1. Emphasis is on data rather than procedure.

2. Programs are divided into what are known as objects.
3. Data structures are designed such that they characterize the objects.
4. Functions that operate on the data of an object are tied together in the data structure.
5. Data is hidden and cannot be accessed by external functions.
6. Objects may communicate with each other through function.
7. New data and functions can be easily added whenever necessary.
8. Follows bottom up approach in program design.

Benefits of OOP

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

Applications of OOP

1. Real-time Systems
2. Simulation and Modeling
3. Object-Oriented Databases
4. Hypertext, hypermedia and expertext
5. Artificial intelligence and expert system
6. Neural networks and parallel programming
7. Decision support and office automation systems

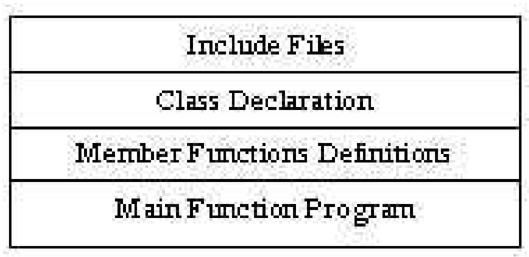
8. CIM/CAM/CAD systems

Difference between C and C++

C	C++
Procedural programming language	Object-oriented programming anguage
Global variable can be declared	It is an error to declare a variable as global
Function prototypes are optional	All functions must be prototyped.
Local variables can be declared only	Local variables can be declared any where the start of a c program in a c++ program.
C makes use of top down approach of problem solving	C makes use of bottom up approach of problem solving
The input and output is done using scanf and printf	The input and output is done using cin and cout
I/O operations are supported by stdio.h header file	I/O operations are supported by iostream.h header file
C does not support inheritance, polymorphism , classes and objects	C supports inheritance, polymorphism , classes and objects
The data type specifier / format specifier (%d, %c, % f) is required in printf and scanf	The data type specifier / format specifier is not required in printf and scanf

1.1. Overview of C++

Structure of C++



Include Files
Class Declaration
Member Functions Definitions
Main Function Program

Tokens

The smallest individual units are known as tokens.

1. Keywords

2. Identifiers
3. Constants
4. Strings
5. Operators

1.1.1. Keywords

These are reserved identifiers. The C++ Keywords are,

asm	double	new	Switch	while	long	volatile
auto	else	operator	template	struct	do	static
break	default	inline	sizeof	void	delete	int
case	catch	char	class	cast	continue	enum
extern	float	for	friend	goto	if	private
protected	public	register	return	short	signed	this
export	try	typedef	union	unsigned	virtual	bool
dynamic_cast	throw	typename	const_cast	false	static_cast	using
reinterpret_cast	mutable	true	wchar_t	explicit	namespace	typeid

1.1.2. Identifiers

✓ *Identifiers* refer to the names of variables, functions, arrays, classes, etc. created by the programmer. The Identifier's rules are,

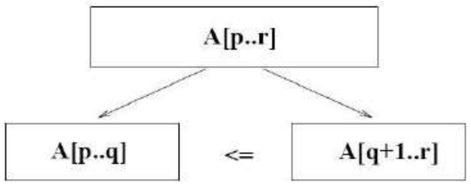
a. Only alphabetic characters, digits and underscores are permitted.
b. The name cannot start with a digit.
c. Uppercase and lowercase letters are distinct
d. A declared keyword cannot be used as a variable name.

1.1.3. Constants

It refers to fixed values that do not change during the execution of a program.

Constants, like variables, contain data type. Integer constants are represented as decimal notation, octal notation, and hexadecimal notation.

- *Decimal notation* is represented with a number.
- *Octal notation* is represented with the



number preceded by a zero character.

- A *hexadecimal number* is preceded with the characters 0x.
- **Example** 80 represent decimal 0115 represent octal 0x167 represent hexadecimal
- By default, the *integer constant* is represented with a number.
- The *unsigned integer* constant is represented with an appended character **u**.
- The *long integer* constant is represented with character or **unsigned**.
- *Signed integers* are used to signify positive and negative number.
- *Unsigned integers* signify only positive numbers or zero.
- **For example** it is declared as unsigned short int a; signed int z; By default, unspecified integers signify a signed integer. For example: int a; is declared a signed integer It is possible to initialize values to variables: data type variable name = value; **Example:** int a=0; int b=5;

1.1.4. Strings

Member Types	Definition
Member types	
value_type	Char
traits_type	char_traits<char>
allocator_type	allocator<char>
Reference	char&
const_reference	const char&
Pointer	char *
const_pointer	const char*
Iterator	a random access iterator to char (convertible to const_iterator)
const_iterator	a random access iterator to const char
reverse_iterator	reverse_iterator<iterator>
const_reverse_iterator	reverse_iterator<const_iterator>
difference_type	ptrdiff_t
size_type	size_t
Member Functions	
Constructor	Construct string object(public member function)

Destructor	String destructor(public member function)
Operator=	String assignment (public member function)
Iterators	
Begin	Return iterator to beginning(public member function)
End	Return iterator to end(public member function)
Rbegin	Return reverse iterator to reverse beginning (public member function)
Rend	Return reverse iterator to reverse end (public member function)
Cbegin	Return const_iterator to beginning (public member function)
Cend	Return const_iterator to end (public member function)
Crbegin	Return const_reverse_iterator to reverse beginning (public member function)
Crend	Return const_reverse_iterator to reverse end (public member function)
Capacity	
Size	Return length of string (public member function)
Length	Return length of string (public member function)
Max_size	Return maximum size of string (public member function)
Resize	Resize string (public member function)
Capacity	Return size of allocated storage (public member function)
Reverse	Request a change in capacity (public member function)
Clear	Clear string (public member function)
Empty	Test if string is empty (public member function)
Shrink_to_fit	Shrink to fit (public member function)
Element Access	
operator[]	Get character of string(public member function)
At	Get character in string(public member function)
Back	Access last character(public member function)
Front	Access first character(public member function)
Modifiers	

Operator+=	Append to string(public member function)
Append	Append to string(public member function)
push_back	Append character to string(public member function)
Assign	Assign content to string(public member function)
Insert	Insert into string(public member function)
Erase	Erase characters from string(public member function)
Replace	Replace portion of string(public member function)
Swap	Swap string values(public member function)
pop_back	Delete last character(public member function)
String operations	
c_str	Get C string equivalent(public member function)
Data	Get string data(public member function)
get_allocator	Get allocator(public member function)
Copy	Copy sequence of characters from string(public member function)
Find	Find content in string(public member function)
Rfind	Find last occurrence of content in string(public member function)
find_first_of	Find character in string(public member function)
find_last_of	Find character in string from the end(public member function)
find_first_not_of	Find absence of character in string(public member function)
find_last_not_of	Find character in string from the end(public member function)
Substr	Find non-matching character in string from the end(public member function)
Compare	Compare strings(public member function)
Member constants	
Npos	Maximum value for size_t(public static member function)
Non-Member function overloads	
operator+	Concatenate strings (function)
relational operators	Relational operators for string(function)
Swap	Exchanges the values of two strings(function)

operator>>	Extract string from stream(function)
operator<<	Insert string into stream(function)
Getline	Get line from stream into string(function)

1.1.5. Operators

Operators	Name of the Operator	Syntax	Purpose of the Operator
::	Scope resolution operator	::variable-name	Access to the global version of a variable
Member Dereferencing Operators			
::*	Pointer-to-member declaratory	expression::*expression	To declare a pointer to the member of a class
.*	Pointer-to-member operator	expression.*expression	return the value of a specific class member for the object specified on the left side of the expression. The right side must specify a member of the class.
->*	Pointer-to-member operator	expression->*expression	
Memory Management Operators			
New	Memory allocation operator	pointer-variable=new data-type	To create object of any type
Delete	Memory release operator	delete[size] pointer-variable	To release the memory space for re-usege
Manipulators			
Endl	Line feed operator	cout<<"m= "<<m<<endl	Used to output statement
setw	Field width	cout<<setw(5)<<sum<<endl	All numbers to be printed

	operator		right-justified
Type Cast Operator			
Cast	Cast operator	type-name(expression)	Type conversion of variables and expressions

1.1.6. Data Types

Types	Bytes	Range	Purpose of Types
i. Basic Data Types			
Char	1	-128 to127	This data type is used to represent a single character
unsigned char	1	0 to255	This data type is used to represent an unsigned character
signed char	1	-128 to127	This data type is used to represent an signed character
Int	2	-32768 to32767	This data type is used to represent integer
unsigned int	2	0 to 65535	This data type is used to represent an unsigned integer
signed int	2	-31768 to 32767	This data type is used to represent an signed integer
short int	2	-31768 to 32767	This data type is used to represent short integer.
unsigned short int	2	0 to 65535	This data type is used to represent unsigned short integer
signed short int	2	-32768 to32767	This data type is used to represent signed short integer
long int	4	-2147483648 to 2147483647	This data type is used to represent long integer
signed long int	4	-2147483648 to	This data type is used to

		2147483647	represent signed long integer
unsigned long int	4	0 to 4294967295	This data type is used to represent unsigned long integer
Float	4	3.4E-38 to 3.4E+38	This data type is used to represent floating point number.
Double	8	1.7E-308 to 1.7E+308	This data type is used to represent double precision floating point number
long double	10	3.4E-4932 to 1.1E+4932	This data type is used to represent double precision floating point number.
Bool		True or False	This data type is used to represent boolean value
ii. User-Defined Data Types			
Structure and Classes : struct, union			
Enumerated Data Types : enum		E.g.enum shape{circle,rectangle};	A way for attaching names to numbers.
iii. Derived Data Types			
Arrays		char string[3]=”xyz”	Collection of similar data-type
Functions			performs a process
Pointers		int *ip;	Returns the address
Type Compatibility		sizeof(‘x’)⇔sizeof(int)	Assigned to one another

1.1.7. Variables

A variable is the storage location in memory that is stored by its value. A variable is identified or denoted by a variable-name. The variable-name is a sequence of one or more letters, digits or underscore, for example: character _

Rules for defining a variable name:

1. A variable name can have one or more letters or digits or underscore for example character _.
2. White space, punctuation symbols or other characters are not permitted to denote variable name.
3. A variable name must begin with a letter.
4. Variable names cannot be keywords or any reserved words of the C++ programming language.
5. C++ is a case-sensitive language. Variable names written in capital letters differ from variable names with the same name but written in small letters. For example, the variable-name A differs from the variable-name a.

- The syntax for *declaring variable names* is,

```
data-type variable-name;
```

- The date type can be int or float or any of the data types listed above. A variable name is given based on the rules for defining variable name (refer above rules). **Example:** int a;
- Dynamic Initialization of variables:
    - C++ permits initialization of variables at run time.
    - E.g.

```
int n=strlen(string);
```

1.1.8. Expressions

Name of Expression	Purpose of an Expression	Example
Constant Expression	Constant values	'x'
Integral Expression	Produces integer results	m*n-5
Float Expression	Produces floating-point results	x+y/10
Pointer Expression	Produces address values	&m
Relational Expression	Produces bool type of results	x<=y
Logical Expression	Combine two or more relational Expression and produces bool results	a>b && a>c
Bitwise Expression	To manipulate data at bit level	x<<3 –shift 3-bit position to the left x>>3 –shift 3-bit position to the right

Special Assignment Expression		
Chained Assignment	Chain format variable assignment	float a=12.34, b=12.34
Embedded Assignment	Embed the variables	y=50; x=y+10;
Compound Assignment	Combination of assignment operator with a binary arithmetic operator	x=x+10; written as, x+=10;

1.1.9. Pointers

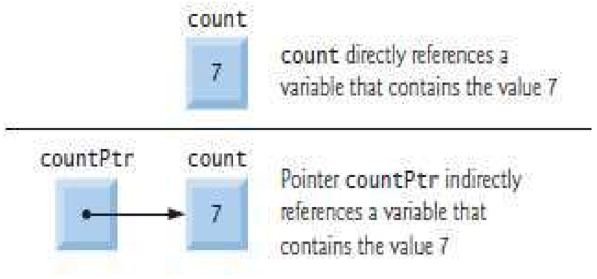
A pointer contains the memory address of a variable In this sense, a variable name directly references a value, and a pointer indirectly references a value. Referencing a value through a c

1.1.10. Reference Variables

- The reference variable provides a kind of a link to the original variable and becomes alias for the original variable..
- Syntax:

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

- E.g. float total=100; float &sum=total;


count
7
count directly references a variable that contains the value 7
countPtr
count
7
Pointer countPtr indirectly references a variable that contains the value 7

1.1.11. Storage Class Specifieres

Four storage class specifiers are,extern, static, register, auto. The first two specifiers result in static storage. Other two specifiers result in automatic storage.

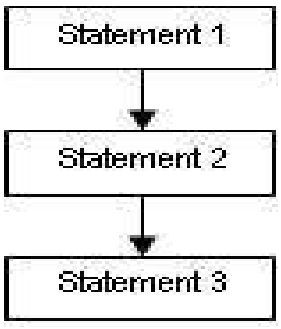
1.1.12. Initialization

The variables should be initialized. Two types of Initialization,

i. Normal Initialization. e.g. int i=0;

ii. Runtime Initialization e.g. get the values from the user through the variables.

1.1.13. Control Structures



It is of three types: *Fig.1.2.13(a)Sequence*

1. Sequence structure : Simple Statements
2. Selection structure : if and switch

3. Loop or iteration or repetition structure : do…while, while, for

Selection Structure

✓ Simple if statement(Fig.1.2.*13*(a))

- The if keyword is used to execute a statement or block only if a condition is fulfilled. Structure
- Its form is,

```
if(condition)
      statement
```

- where condition is the expression that is being evaluated.

✓ if...else statement (Fig.1.2.*13*(b))

- If this condition is true, statement is executed. If it is false, statement is ignored (not executed) and the program continues right after this conditional structure.

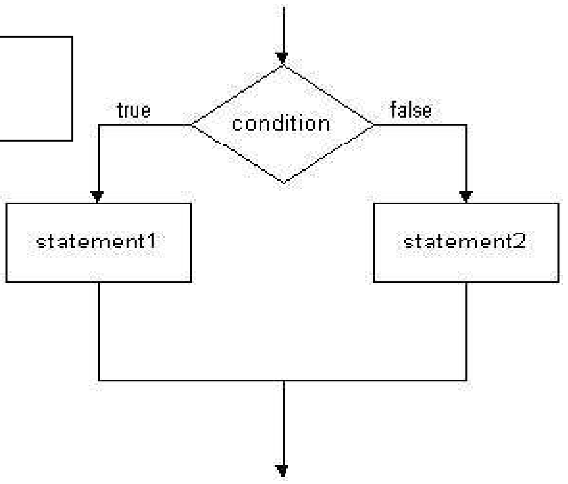


Fig.1.2.5(b)if…else Statement

Fig.1.2.13(b)if...else statement

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

✓ The switch statement (Fig.1.2.*13*(c))

- Multiple-branching statement of if.
- Format is,

```
switch(expression)
{
    case constant1: group of statements 1; break;
    case constant2: group of statements 2; break;...
    default: default group of statements
}
```

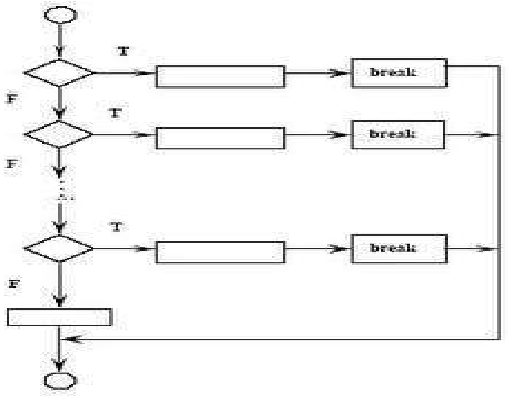


Fig.1.2.13(c) switch statement

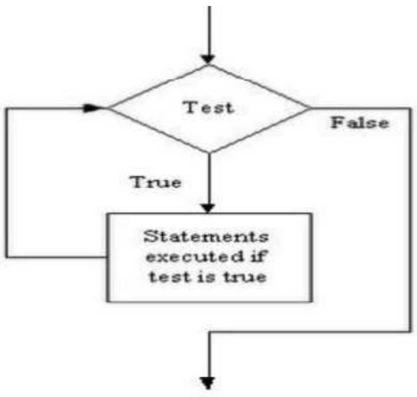


Fig.1.2.14(b) while loop

1.1.14. Loop or Iteration or Repetition Structure

✓ The format of *do⋯while* is (Fig.1.2.14(a))

```
do{}while(condition is true);
```

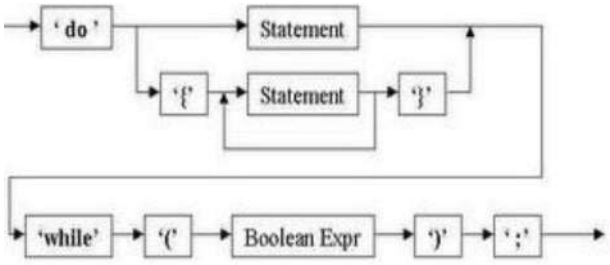


Fig.1.2.14(a) do⋯while loop

✓ The format of *while loop* is (Fig.1.2.14(b))

```
while (expression)
      statement
```

- its functionality is simply to repeat statement while the condition set in expression is true.

✓ The format of *for loop* is (Fig.1.2.14(c))

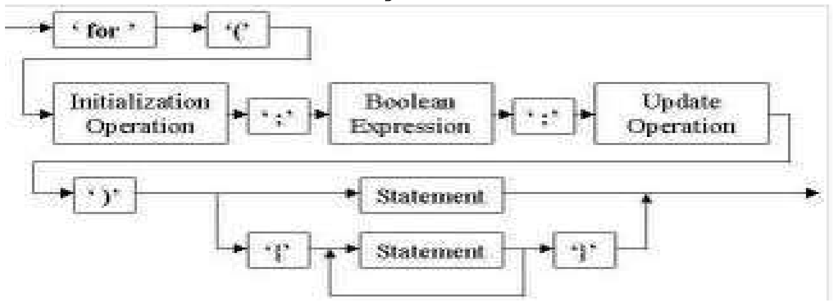


Fig.1.2.14(c) while loop

```
for(initialization;condition;increase or decrease)
```

- It works in the following way:

1. Initialization is executed. Generally it is an initial value setting for a counter variable. This is executed only once.
2. Condition is checked. If it is true the loop continues, otherwise the loop ends and statement is skipped (not executed).
3. Statement is executed. As usual, it can be either a single statement or a block enclosed in braces {}
4. Finally, whatever is specified in the increase field is executed and the loop gets back to step

1.1.15. Classes

Classes represent real world entities that have both data type properties (characteristics) and associated operations (behavior). Class is the collection of objects.

The syntax of a class definition is shown below:

```
class name_of_class
{
  private : variable declaration;//data member
            Function declaration;//Member  Fun.(Method)
   protected: variable declaration;
               Function declaration;
   public   : variable declaration;
              Function declaration;
};
```

The keywords **private, protected** and **public** are called access specifiers.

Program

```
#include<iostream.h>
#include<conio.h>
class student
{
      int roll_no;
```

```
        char name[20];
        char class_st[8];
        int marks[5];
        float percentage;
        float calculate();
        public:
        void readmarks();
        void displaymarks();
};
float student::calculate()
{
        percentage=0;
        for(int i=0;i<5;i++)
                percentage+=marks[i];
        percentage=(percentage/5);
        return percentage;
}
void student::readmarks()
{
        cout<<"Enter the roll no.:";
        cin>>roll_no;
        cout<<"Enter the name:";
        cin>>name;
        cout<<"Enter the class studing in:";
        cin>>class_st;
        cout<<"Enter the marks:"<<endl;
        for(int j=0;j<5;j++)
        {
                cout<<"\tEnter mark "<<j+1<<":";
                cin>>marks[j];
        }
}
```

```
void student::displaymarks()
{
        cout<<"Roll no:"<<roll_no<<endl;
        cout<<"Name:"<<name<<endl;
        cout<<"Class:"<<class_st<<endl;
        cout<<"Percentage:"<<calculate()<<endl;
}
int main()
{
        student s1;
        s1.readmarks();
        s1.displaymarks();
        return 0;
}
```

OUTPUT:

```
Enter the roll no.:12
Enter the name:KARTHIK
Enter the class studing in:12
Enter the marks:
    Enter mark 1:99
    Enter mark 2:95
    Enter mark 3:90
    Enter mark 4:80
    Enter mark 5:99

Roll no:12
Name:KARTHIK
Class:12
Percentage:92.599998
```

1.2. Structures

An array is an aggregate of elements of the same type. A *struct* is an aggregate of elements of (nearly) arbitrary types. For example:

```
struct  address
{
       char *name ; // "Jim Dandy"
       long int number ; // 61
       char *street; / / "South St"
       char *town ; // "New Providence"
       char state[2]; // 'N' 'J'
       long zip ; // 7974
};
```

Variables of type *address* can be declared exactly as other variables, and the individual *members* can be accessed using the. (dot) operator. For example:

```
void f()
{
       address jd ;
       jd.name = "Jim Dandy ";
       jd.number = 61 ;
}
```

The notation used for initializing arrays can also be used for initializing variables of structure types. For example:

```
address  jd  =  {"Jim  Dandy",61,"South  St","New  Providence",
{´N´,´J´}, 7974};
```

Using a constructor is usually better, however. Note that *jd .state* could not be initialized by the string *" NJ "* . Strings are terminated by the character *´\ 0 ´*. Hence, *" NJ "* has three characters – one more than will fit into *jd.state* .

Structure objects are often accessed through pointers using the -> (structure pointer dereference) operator. For example:

```
void print_addr(address *p)
{
       cout<<p->name<<'\n'<<p->number<<p->street<<´\n`<<p->town
       <<´\n´<<p->state[0]<<p->state[1]<<´´<<p>zip<<´\n´;
}
```

When *p* is a pointer, *p->m* is equivalent to (**p*).*m*.

Objects of structure types can be assigned, passed as function arguments, and returned as the result from a function. For example:

```
address current ;
addresss et_current(address next)
{
      address prev = current;
      current = next;
      return prev;
}
```

1.2.1. Type Equivalence

Two structures are different types even when they have the same members. For example,

```
      struct S1{int a;};
      struct S2{int a;};
```
are two different types, so
```
      S1 x ;
      S2 y = x ; // error: type mismatch
```

Structure types are also different from fundamental types, so

```
      S1 x;
      int i = x; // error: type mismatch
```

Every `struct` must have a unique definition in a program.

Program

```
#include<iostream.h>
#include<stdio.h>
struct books
{
    char name[20],author[20];
}a[50];
int main()
{
    int i,n;
    cout<<"No Of Books[less than 50]:";
    cin>>n;
    cout<<"Enter the book details\n";
    cout<<"----------------------\n";
    for(i=0;i<n;i++)
    {
        cout<<"Details of Book No "<<i+1<<"\n";
        cout<<"Book Name :";
        cin>>a[i].name;
        cout<<"Book Author :";
        cin>>a[i].author;
        cout<<"----------------------\n";
    }
    cout<<"================================================\n";
    cout<<" S.No\t| Book Name\t|author\n";

cout<<"=================================================";
    for(i=0;i<n;i++)
    {
        cout<<"\n  "<<i+1<<"\t|"<<a[i].name<<"\t|
      "<<a[i].author;
    }
    cout<<"\n================================================";
    return 0;
}
```

Output

```
No Of Books[less than 50]:2
Enter the book details
----------------------
Details of Book No 1
Book Name :OOP
Book Author :BGS
----------------------
Details of Book No 2
Book Name :DS
Book Author :AW
----------------------
=================================================
 S.No  | Book Name    |author
=================================================
  1   |OOP         | BGS
  2   |DS         | AW
==================================================
```

1.3. Class Scope and Accessing Class Members

- ✓ A class's data members (variables declared in the class definition) and member functions (functions declared in the class definition) belong to that class's scope. Non member functions are defined at *global namespace scope*.
- ✓ Within a class's scope, class members are immediately accessible by all of that class's member functions and can be referenced by name.
- ✓ Outside a class's scope, public class members are referenced through one of the handles on an object—an object name, a reference to an object or a pointer to an object. The type of the object, reference or pointer specifies the interface (i.e., the member functions) accessible to the client.
- ✓ *Member functions of a class can be overloaded, but only by other member functions of that class.* To overload a member function, simply provide in the class definition. This also applies to the class's constructors.
- ✓ Variables declared in a member function have *local scope* and are known only to that function.
- ✓ If a member function defines a variable with the same name as a variable with class scope, the class-scope variable is *hidden* by the block-scope variable in the local scope.
  - A hidden variable can be accessed by preceding the variable name with the class name followed by the scope resolution operator (::).
  - Hidden global variables can be accessed with the scope resolution operator. The dot member selection operator (.) is preceded by an object's name or with a reference to an object to access the object's members.
  - The arrow member selection operator (->) is preceded by a pointer to an object to access the object's members.

```
#include <iostream.h>
class Count
{
      public: // public data is dangerous
      // sets the value of private data member x
      void setX(int value)
      {
            x = value;
      } // end function setX
      // prints the value of private data member x
      void print()
      {
            cout<<x<<endl;
      } // end function print
      private:
            int x;
}; // end class Count
int main()
{
      cout<<"Set x to 1 and print using the object's name:";
      cout<<"Set x to 2 and print using a reference to an
   object:";
      cout<<"Set x to 3 and print using a pointer to an
   object:";
} // end main
```

Output

```
Set x to 1 and print using the object's name: 1
Set x to 2 and print using a reference to an object: 2
Set x to 3 and print using a pointer to an object: 3
```

Fig. 1.4 Accessing an object’s member functions through each type of object handle—the object’s name, a reference to the object and a pointer to the object.

1.4. Reference Variables

- A reference is an *alias*, or an *alternate name* to an existing variable. For example, suppose you make peter a reference (alias) to paul, you can refer to the person as either peter or paul.
- The main use of references is acting as function formal parameters to support pass-by-reference. In an reference variable is passed into a function, the function works on the original copy (instead of a clone copy in pass-by-value). Changes inside the function are reflected outside the function.
- A reference is similar to a pointer. In many cases, a reference can be used as an alternative to pointer, in particular, for the function parameter.

1.4.1. Pointer Variables

A *pointer variable* (or *pointer* in short) is basically the same as the other variables, which can store a piece of data. Unlike normal variable which stores a value (such as an int, a double, a char), a *pointer stores a memory address.*

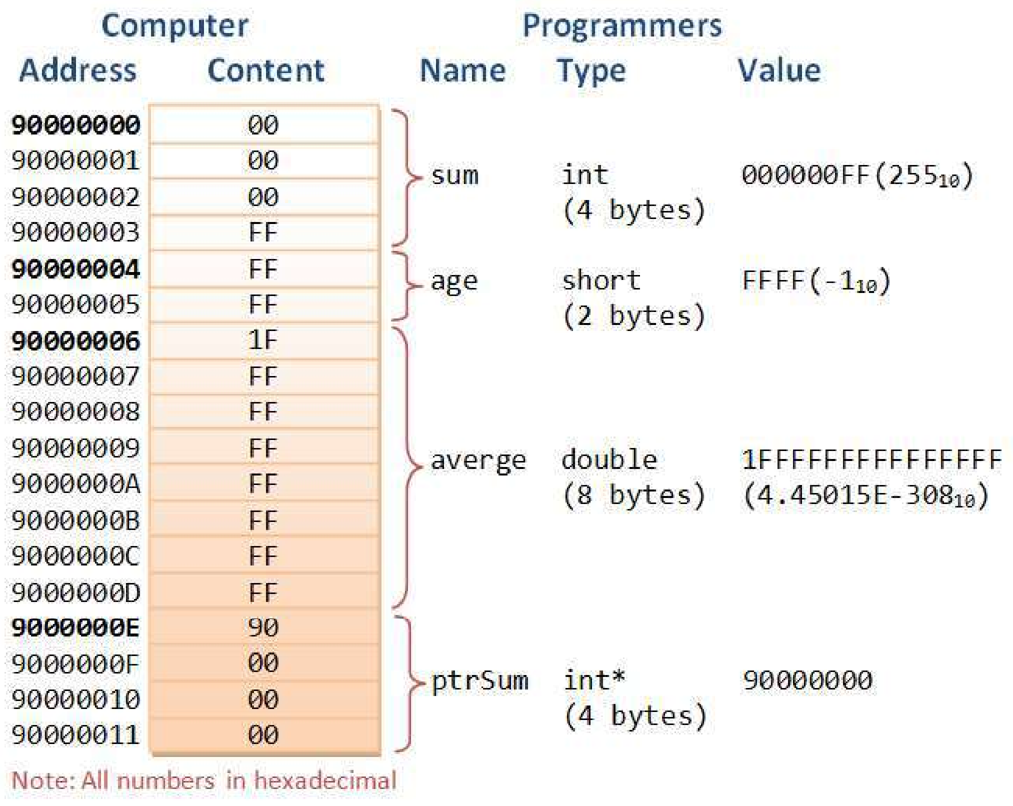
A *computer memory location* has an *address* and holds content. The *address* is a numerical number (often expressed in hexadecimal), which is hard for programmers to use directly. Typically, each address location holds 8-bit (i.e., 1-byte) of data.

A variable is a *named* location that can store a *value* of a particular *type*. Instead of numerical addresses, names (or identifiers) are attached to certain addresses.

1.4.1.1. Declaring Pointers

*The syntax of declaring a pointer is to place a * in front of the name.*

```
type *ptr;//Declare a pointer variable called ptr
as a pointer of type  (or)
type* ptr;
```



Computer		Programmers		
Address	Content	Name	Type	Value
90000000	00	sum	int (4 bytes)	000000FF(255_{10})
90000001	00			
90000002	00			
90000003	FF			
90000004	FF	age	short (2 bytes)	FFFF(-1_{10})
90000005	FF			
90000006	1F	averge	double (8 bytes)	1FFFFFFFFFFFFFFF ($4.45015E-308_{10}$)
90000007	FF			
90000008	FF			
90000009	FF			
9000000A	FF			
9000000B	FF			
9000000C	FF			
9000000D	FF			
9000000E	90	ptrSum	int* (4 bytes)	90000000
9000000F	00			
90000010	00			
90000011	00			

Note: All numbers in hexadecimal

1.4.1.2. Initializing Pointers via the Address-Of Operator (&)

The *address-of operator* (&) operates on a variable, and returns the address of the variable. For example, if number is an int variable, &number returns the address of the variable number.

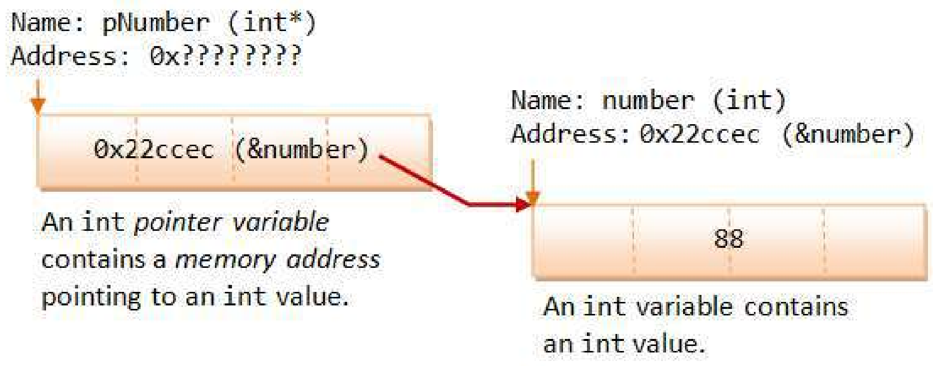
For example,

int number = 88; // An int variable with a value

int * pNumber; // Declare a pointer variable called pNumber pointing to an int (or int pointer)

pNumber = &number; // Assign the address of the variable number to pointer pNumber

int * pAnother = &number; // Declare another int pointer and init to address of the variable number


Name: pNumber (int*)
Address: 0x????????
0x22ccec (&number)
An int pointer variable contains a memory address pointing to an int value.
Name: number (int)
Address: 0x22ccec (&number)
88
An int variable contains an int value.

1.4.2. References (or Aliases) (&)

To denote the *address-of* operator in an expression using &. An additional meaning of & in declaration to declare a reference variable.

Difference between the meaning of symbol & is in an expression and in a declaration.

When it is used in an expression, & denotes the address-of operator, which returns the address of a variable, e.g., if number is an int variable, &number returns the address of the variable number (this has been described in the above section).

However, when & is used in a *declaration* (including *function formal parameters*), it is part of the type identifier and is used to declare a *reference variable* (or *reference* or *alias* or *alternate name*). It is used to provide *another name*, or *another reference*, or *alias* to an existing variable.

The syntax is as follow:

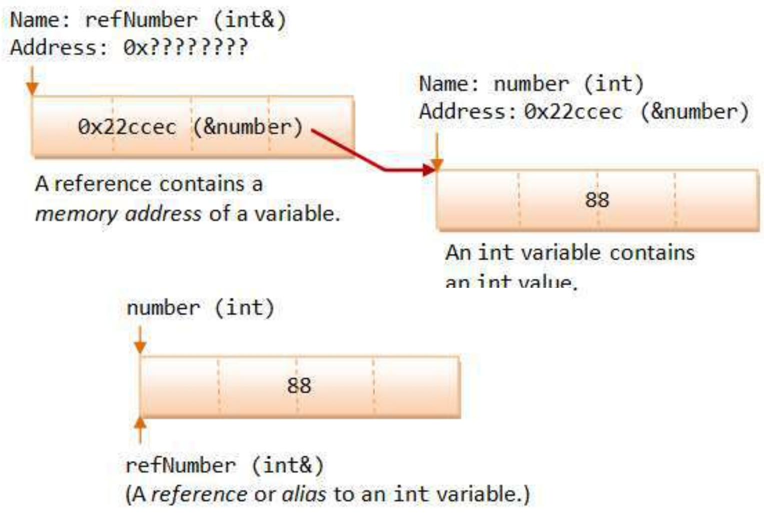
```
type &newName = existingName;// or
type& newName = existingName;
```

For example,

```
/* TestReferenceDeclaration.cpp */
#include <iostream.h>
int main()
{
      int number=88;   // Declare an int variable called number
      int & refNumber=number; // Declare a reference (alias) to
      the variable number Both refNumber and number refer to the
      same value
      cout<<number<<endl; // Print value of variable number (88)
      cout<<refNumber<<endl; // Print value of reference (88)
      refNumber =99;       // Re-assign a new value to refNumber
      cout<<refNumber<<endl;
      cout<<number<<endl;   // Value of number also changes (99)
      number=55;               // Re-assign a new value to number
      cout<<number<<endl;
    cout<<refNumber<<endl;//Value of refNumber also changes (55)
}
```

Output

```
88
88
99
99
55
55
```


Name: refNumber (int&)
Address: 0x????????
0x22ccec (&number)
A reference contains a memory address of a variable.
Name: number (int)
Address: 0x22ccec (&number)
88
An int variable contains an int value.
number (int)
88
refNumber (int&)
(A reference or alias to an int variable.)

1.4.3. How References Work?

A reference works as a pointer. A reference is declared as an alias of a variable. It stores the address of the variable, as illustrated:

1.4.4. References vs. Pointers

Pointers and references are equivalent, except:

1. A reference is a *name constant for an address*. You need to initialize the reference during declaration.

   int & iRef; // Error: 'iRef' declared as reference but not initialized

   Once a reference is established to a variable, you cannot change the reference to reference another variable.

2. To get the value pointed to by a pointer, you need to use the dereferencing operator * (e.g., if pNumber is a int pointer, *pNumber returns the value pointed to by pNumber. It

is called *dereferencing* or *indirection*). To assign an address of a variable into a pointer, you need to use the address-of operator & (e.g., pNumber = &number). On the other hand, referencing and dereferencing are done on the references implicitly. For example, if refNumber is a reference (alias) to another int variable, refNumber returns the value of the variable. No explicit dereferencing operator * should be used. Furthermore, to assign an address of a variable to a reference variable, no address-of operator & is needed.

For example,

```
#include <iostream.h>
int main()
{
      int number1 = 88, number2 = 22;
      //create a pointer pointing to number1
       int * pNumber1 = &number1;  //Explicit referencing
       *pNumber1 = 99;             //Explicit dereferencing
      cout<<*pNumber1<<endl;  //99
      cout<<&number1<<endl;   //0x22ff18
      cout<<pNumber1 << endl;//0x22ff18 (content of the pointer
variable - same as above)
      cout<<&pNumber1<<endl;//0x22ff10(address  of  the  pointer
variable)
      pNumber1 = &number2; //Pointer can be reassigned to store
another address//Create a reference (alias) to number1
      int  &  refNumber1  =  number1;//Implicit  referencing(NOT
&number1)
      refNumber1 = 11;//Implicit dereferencing (NOT *refNumber1)
      cout << refNumber1 << endl;  //11
      cout << &number1 << endl;    //0x22ff18
      cout << &refNumber1 << endl; //0x22ff18
      refNumber1 = number2; //refNumber1 is still an alias to
number1.
      //Assign value of number2 (22) to refNumber1 (and number1)
      number2++;
      cout << refNumber1 << endl;  //22
      cout << number1 << endl;     //22
      cout << number2 << endl;     //23
}
```

output

```
99
ox8fc3fff4
ox8fc3fff4
ox8fc3fff0
11
ox8fc3fff4
ox8fc3fff4
22
22
23
```

1.4.5. Passing the Function's Return Value

Passing the Return-value as Reference

```
#include <iostream.h>
#include<conio.h>
int & squareRef(int &);
int * squarePtr(int *);
int main()
 {
      clrscr();
      int number1 = 8;
      cout <<  "In main() &number1: " << &number1 << endl;  //
      0x22ff14
      int & result = squareRef(number1);
      cout <<  "In main() &result: " << &result << endl;  //
      0x22ff14
      cout << result << endl;   // 64
      cout << number1 << endl;  // 64
```

```
        int number2 = 9;
         cout <<  "In main() &number2: " << &number2 << endl;  //
        0x22ff10
        int * pResult = squarePtr(&number2);
        cout <<  "In main() pResult: " << pResult << endl;  //
        0x22ff10
        cout << *pResult << endl;    // 81
        cout << number2 << endl;     // 81
        getch();
}
int & squareRef(int & rNumber)
{
        cout <<  "In squareRef(): " << &rNumber << endl;  //
        0x22ff14
        rNumber *= rNumber;
         return rNumber;
}
int * squarePtr(int * pNumber)
{
    cout <<  "In squarePtr(): " << pNumber << endl;  // 0x22ff10
    *pNumber *= *pNumber;
    return pNumber;
}
```

Output

```
In main() &number1: 0x8fc9fff4
In squareRef(): 0x8fc9fff4
In main() &result: 0x8fc9fff4
64
64
In main() &number2: 0x8fc9fff2
In squarePtr(): 0x8fc9fff2
In main() pResult: 0x8fc9fff2
```

1.5. Initialization

If an initializer is specified for an object, that initializer determines the initial value of an object. If no initializer is specified, a global, namespace or local static object (collectively called *static objects*) is initialized to *0* of the appropriate type. For example,

```
int a ; //means‘‘int a = 0;’’
double d ; //means‘‘double d = 0.0;’’
```

Local variables (sometimes called *automatic objects*) and objects created on the free store (sometimes

called *dynamic objects* or *heap objects*) are not initialized by default. For example:

```
void f()
{
    int x ; //x does not have a welldefinedvalue
    // ...
}
```

Members of arrays and structures are default initialized or not depending on whether the array or structure is static. User Defined Types may have default initialization defined. More complicated objects require more than one value as an initializer. This is handled by initialize lists delimited by { and } for C style initialization of arrays and structures. For User Defined Types with constructors, function style argument lists are used. For example:

```
    int a[]={1,2}; //array initializer
    Point z(1,2); // function style initialize (initialization
    by constructor)
    int f(); // function declaration
```

1.6. Constructors

The main use of constructors is to initialize objects.

General Syntax of Constructor

Constructor is a special member function that takes the same name as the class name. The syntax generally is as given below:

```
class name()
{
        argumentlist
}
```

The default constructor for a class X has the form **X::X()** In the above example the arguments is optional. The constructor is automatically invoked when an object is created.

Declaration and Definition of a Constructor

It is defined like other member functions of the class, i.e., either inside the class definition or outside the class definition.

Example:

```
//To demonstrate a constructor
#include <iostream.h>
#include <conio.h>
class rectangle
{
      private :
        float length, breadth;
      public:
        rectangle()//constructor definition
        {
          //displayed whenever an object is created
          cout<<"I am in the constructor";
          length=10.0; breadth=20.5;
        }
        float area()
        {
              return (length*breadth);
        }
  };
  void main()
  {
        clrscr();
        rectangle rect; //object declared
      cout<<"\nThe area of the rectangle with default
parameters is:"<<rect.area()<<"sq.units\n";
      getch();
}
```

Output

```
I am in the constructor
The area of the rectangle with default parameters
is:205sq.units
```

➢ **Special Characteristics of Constructors**

1. These are called automatically when the objects are created.
2. All objects of the class having a constructor are initialized before some use.
3. These should be declared in the public section for availability to all the functions.
4. Return type (not even **void**) cannot be specified for constructors.
5. These cannot be inherited, but a **derived** class can call the base class constructor.
6. These cannot be static.
7. Default and copy constructors are generated by the compiler wherever required. Generated constructors are public.
8. These can have default arguments as other C++ functions.
9. A constructor can call member functions of its class. \
10. An object of a class with a constructor cannot be used as a member of a **union.**
11. A constructor can call member functions of its class.
12. We can use a constructor to create new objects of its class type by using the syntax.

1.6.1. Types of Constructors

1. **Default constructors** : This constructor has no arguments in it. Default Constructor is also called as *no argument constructor*.
2. **Parameterized constructors** : A parameterized constructor is just one that has parameters specified in it.
3. **Copy constructors** : The purpose of the copy constructor is to initialize a new object with data copied from another object of the same class.

   For example to invoke a copy constructor the programmer writes:

```
Exforsys e3(e2); or Exforsys
```

Example

```
#include <iostream.h>
#include <conio.h>
class employee
{
      int empl_no;
      float salary;
      public:
            employee() //default constructor
            {}
            //constructor with arguments(Parameterized
      Constructor)
            employee(int empno,float s)
            {
                  empl_no=empno;
                  salary=s;
            }
            employee(employee &emp) //copy constructor
            {
                  cout<<"\ncopy constructor working\n";
                  empl_no=emp.empl_no;
                  salary=emp.salary;
            }
            void display (void)
            {
                  cout<<"\nEmp.No:"<<empl_no<<"salary:"<<salary<
            <endl;
            }
};
void main()
{
      int eno;
      float sal;
```

```
        clrscr();
        cout<<"Enter the employee number and salary\n";
        cin>>eno>>sal;
        employee obj1(eno,sal);//dynamic initialization of object
        cout<<"\nEnter the employee number and salary\n";
        cin>eno>>sal;
        employee obj2(eno,sal);//dynamic initialization of object
        obj1.display();//function called
        employee obj3=obj2;//copy constructor called
        obj3.display();
        getch();
}
```

Output

```
Enter the employee number and salary
101
20000
Enter the employee number and salary
102
30000
Emp.No:101salary:20000
copy constructor working
Emp.No:102salary:30000
```

4. Overloaded Constructors

It performing the role of member data initialization, constructors are no different from other functions. This included overloading also.

For example, consider the following program with overloaded constructors for the rectangle class

```
#include<iostream.h>
#include<conio.h>
class rectangle
{
    int length,breadth;
    public:
     rectangle()
     {
           length=breadth=0;
           cout<<"Constructor with zero argument
     called"<<endl<<endl
     }
     rectangle(int a)
     {
           length=breadth=a;
           cout<<"Constructor with one argument
     called"<<endl<<endl;
     }
     rectangle(int l,int b)
     {
           length=l;
           breadth=b;
           cout<<"Constructor
     withtwoargumentscalled"<<endl<<endl;
     }
     int area()
     {
           return(length*breadth);
     }
};
void main()
{
    clrscr();
    rectangle r1;
    cout<<"\tArea  of  rectangle  zero  arguments  is:"<<r1.area()
<<endl;
```

```
    rectangle r2(5);
    cout<<"\tArea    of    rectangle    with    one    argument
is:"<<r2.area();
    rectangle r3(11,12);
    cout<<"\tArea    of    rectangle    with    two    arguments
is:"<<r3.area();
    getch();
}
```

Output

```
Constructor with zero argument called
        Area of rectangle zero arguments is:0
Constructor with one argument called
        Area of rectangle with one argument is:25
Constructor with two arguments called
        Area of rectangle with two arguments is:132
```

5. Dynamic Constructor

The constructor can be allocate the memory for the objects using new operator

```
#include<iostream.h>
#include<conio.h>
#include<string.h>
class string
{
      char *mystr;
      int len;
      public:
            string()
            {
                  len=0;
                  mystr=new char[len+1];
                  strcpy(mystr,""); // Init by empty string
            }
            string(const char *s1)
            {
                  len=strlen(s1);
                  mystr=new char[len+1];
                  strcpy(mystr,s);
            }
            void disply()
            {
                  cout<<"mystr";
            }
};
int main()
{
      string s2;
      string s3("ponjesly");
      s2.disply();
      cout<<"The copied string is";
      s3.disply();
      return 0;
}
```

1.6.2. Dynamic Initialization of Objects

Objects can be initialized at run time (dynamically).

1.7. Destructors

The main use of destructors is to release dynamic allocated memory. Destructors are used to free memory, release resources and to perform other clean up. Destructors are automatically called when an object is destroyed.

`General Syntax of Destructors`

```
~classname();
```

- In the above, the symbol tilda ~ represents a destructor which precedes the name of the class.

➢ **Special Characteristics of Destructors**

1. These are called automatically when the objects are destroyed.

2. Destructor functions follow the usual access rules as other member functions.
3. These **de-initialize** each object before the object goes out of scope.
4. No argument and return type (even void) permitted with destructors.
5. These cannot be inherited.
6. **Static** destructors are not allowed.
7. Address of a destructor cannot be taken.
8. A destructor can call member functions of its class.
9. An object of a class having a destructor cannot be a member of a union.

For example

```
//Illustration of the working of Destructor function
#include<iostream.h>
#include<conio.h>
class add
{
      private:
            int num1,num2,num3;
      public:
            add(int=0,int=0);//default argument constructor
            //to reduce the number of constructors
            void sum();
            ~add(void); //Destructor
};
//Destructor definition ~add()
//destructor called automatically at end of program
add::~add(void)
{
      num1=num2=num3=0;
      cout<<"\nAfter the final execution, me, the object has
entered  in the"<<"\ndestructor to destroy myself\n";
}
//Constructor definition add()
add::add(int n1,int n2)
```

```
{
      num1=n1;
      num2=n2;
      num3=0;
}
//function definition sum()
void add::sum()
{
      num3=num1+num2;
      cout<<"\nThe sum of two numbers is "<<num3<<endl;

}
void main()
{
      //objects created and initialized
      clrscr();add obj1,obj2(5),obj3(10,20);
      clrscr();
      cout<<"\nUsing obj1\n";
      obj1.sum(); //function call
      cout<<"\nUsing obj2\n";
      obj2.sum();
      cout<<"\nUsing obj3\n";
      Obj3.sum();
      getch();
}
```

output

```
Using obj1  The sum of two numbers is 0
Using obj2 The sum of two numbers is 5
Using obj3 The sum of two numbers is 30
```

- **When Constructors and Destructors Are Called?**

Constructors and destructors are called *implicitly* by the compiler. The order in which these function calls occur depends on the order in which execution enters and leaves the scopes where the objects are instantiated. Generally, destructor calls are made in the *reverse order* of the corresponding constructor calls, but the storage classes of objects can alter the order in which destructors are called.

1.8. Member Functions and Classes

Classes represent real world entities that have both data type properties (characteristics) and associated operations (behavior). Class is the collection of objects.

Classes are data types based on which objects are created. Objects with similar properties and methods are grouped together to form a Class. Thus a Class represents a set of individual objects. Characteristics of an object are represented in a class as **Properties (Attributes)**.

For example, a class that represents a bank account might contain one member function to *deposit* money to an account, another to *withdraw* money from an account and a third to *inquire* what the account's current balance is.

The syntax of a class definition is shown below :

The keywords **private, protected** and **public** are called access specifiers.

```
class name_of_class
{
   private   :variable declaration; //data member
              Function declaration; //Member Fun.(Method)
   protected: variable declaration;
             Function declaration;
   public    : variable declaration;
              Function declaration;
};
```

1.8.1. Member Function Definition

The class specification can be done in two part :

i. **Class definition.** It describes both data members and member functions.
ii. **Class method definitions.** It describes how certain class member functions are coded.

The member functions can be coded in two ways :

i. Inside class definition
ii. Outside class definition using scope resolution operator (::),

- In this case the function's full name (qualified_name) is written as shown:

```
Name_of_the_class::function_name()
```

The syntax for a member function definition outside the class definition is :

```
return_type name_of_the_class::function_name(argument list)
{
      //body of function
}
```

Declaration of Objects as Instances of a Class

The objects of a class are declared after the class definition. Object Name creation is Class name followed by object name. i.e

```
class-name object-name;
```

1.8.2. Data Members, *set* Functions (obtain the values of) and *get* Functions (assign values to)

- Variables declared in a function definition's body are known as *local variables* and can be used only from the line of their declaration in the function to closing right brace (}) of the block in which they're declared.
- A local variable must be declared before it can be used in a function. A local variable cannot be accessed outside the function in which it's declared.
- When a function terminates, the values of its local variables are lost.
- A class normally consists of one or more member functions that manipulate the attributes that belong to a particular object of the class. Attributes are represented as variables in a class definition. Such variables are called data members and are declared *inside* a class definition but *outside* the bodies of the class's member-function definitions.
- Each object of a class maintains its own copy of its attributes in memory. These attributes exist throughout the life of the object.
- *set* functions are sometimes called mutators and *get* functions are also called accessors.

Accessing Members from Object(s)

After defining a class and creating class variables i.e., object can access the data members and member functions of the class. Example:

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

```
class student
{
   int regno;
   char name[20];
   public:
         void setdata(int r, char*n);
         void putdata()
         {
               cout<<"reg.no:"<<regno;
               cout<<"Name:"<<name;
         }
};
void student::setdata(int r, char *n)
{
      regno=r;
      strcpy(name,n);
}
int main()
{
      clrscr();
      student s1,s2;
      s1.setdata(100,"ajay");
      s2.setdata(101,"vijay");
      s1.putdata();
      s2.putdata();
      getch();
      return(0);
}
```

1.9. Friend Function

Need for Friend Function

When a data is declared as private inside a class, then it is not accessible from outside the class.

A function that is not a member or an external class will not be able to access the private data. A programmer may have a situation where he or she would need to access private data from non-member functions and external classes. For handling such cases, the concept of Friend functions is a useful tool.

What is a Friend Function?

- A friend function of a class is defined outside that class's scope, yet has the right to access the non-public (and public) members of the class. Standalone functions, entire classes or member functions of other classes may be declared to be friends of another class.
- A friend declaration can appear anywhere in the class.
- To declare a function as a friend of a class, precede the function prototype in the class definition with keyword friend.

How to define and use Friend Function in C++

✓ The friend function is written as any other normal function, except the function declaration of these functions is preceded with the keyword friend.

✓ The friend function must have the class to which it is declared as friend passed to it in argument.

✓ General Syntax for friend function

```
friend return-type function-name(class-name object-name);
```

➢ **Some important points to note while using friend functions in C++:**

1. The keyword friend is placed only in the function declaration of the friend function and not in the function definition. .
2. It is possible to declare a function as friend in any number of classes. .
3. When a class is declared as a friend, the friend class has access to the private data of the class that made this a friend.
4. A friend function, even though it is not a member function, would have the rights to access the private members of the class. .
5. It is possible to declare the friend function as either private or public. .
6. The function can be invoked without the use of an object. The friend function has its argument as objects, seen in example below.

Example to understand the friend function:

```
#include<iostream.h>
#include<conio.h>
class Matrix
{
      private:
            int a[3][3];
      public:
            void getMatrix();
            void printMatrix();
            friend Matrix Multiply(Matrix A,Matrix B);
};
void Matrix::getMatrix()
{
      cout<<"Enter the Matrix one by one\n";
      for(int i=0;i<3;i++)
      {
            for(int j=0;j<3;j++)
            {
                  cin>>a[i][j];
            }
      }
}
void Matrix::printMatrix()
{
      cout<<"Given Matrix is\n";
      for(int i=0;i<3;i++)
      {
            for(int j=0;j<3;j++)
            {
                  cout<<a[i][j]<<"";
            }
            cout<<"\n";
      }
}
Matrix Multiply(Matrix A, Matrix B)
{
      Matrix T;
      for(int i=0;i<3;i++)
      {
            for(int j=0;j<3;j++)
            {
                  T.a[i][j]=0;
                  for(int k=0;k<3;k++)
                  {
                        T.a[i][j]=T.a[i][j]+A.a[i][k]*B.a[k][j];
                  }//end of k loop
            }//end of j loop
      }//end of i loop
      return T;
}
void main()
{
      Matrix P,Q,R;
      P.getMatrix();
      Q.getMatrix();
      R=Multiply(P,Q);
      R.printMatrix();
      getch();
}
```

Output:

```
Enter the Matrix one by one
111
111
111
```

```
Enter the Matrix one by one
222
222
222
Given Matrix is
666
666
666
```

Friend Classes

In C++ , a class can be made a friend to another class. For example,

```
class TWO; // forward declaration of the class TWO
class ONE
{
        ..........................
        public:
                ......................
        //class TWO declared as friend of class ONE
        friend class TWO;
};
```

1.10. Dynamic Memory Allocation

The dynamic memory is allocation is done using an operator new. The syntax is,

```
variable-name=new data-type
```

1.11. Static Class Members

Need for static class members

Each object of a class has its own copy of all the data members of the class. In certain cases, only *one* copy of a variable should be *shared* by *all* objects of a class.

What is a Static Class Members

- A static data member represents “class-wide” information (i.e., a property of the class shared by all instances, not a property of a specific object of the class).
- Static data members have class scope and can be declared public, private or protected.
- A class’s static members exist even when no objects of that class exist.

- To access a public static class member when no objects of the class exist, simply prefix the class name and the scope resolution operator (::) to the name of the data member.
- A member function should be declared static (p. 438) if it does not access non-static data members or non-static member functions of the class. Unlike non-static member functions, a static member function does not have a this pointer, because ***static data members and static member functions*** exist independently of any objects of a class.

For Example,

```
#include<iostream.h>
#include<conio.h>
class test
{
        int code;
        static int count;
        public :
                void setcode(void)
                {
                        code= ++count;
                }
                void showcode(void)
                {
                        cout<<"object number"<<code<<endl;
                }
                static void showcount(void)
                {
                        cout<<"count"<<count<<endl;
                }
};
int test::count;
int main()
{
        clrscr();
        test t1,t2;
        t1.setcode();
        t2.setcode();
        test::showcount();
        test t3;
        t3.setcode();
        test::showcount();
        t1.showcode();
        t2.showcode();
        t3.showcode();
        getch();
        return(1);
}
```

OUTPUT

```
Count 2
Count 3
Object number 1
Object number 2
Object number 3
```

The static data member is defined outside the class as :

```
   int student::count; //definition outside class
```

We can also initialize the static data member at the time of its definition as:

```
   int student::count=0;
```

Static Member Function

A static member function can access only the static members of a class.

1.12. Container Classes and Iterators

1.12.1. Containers

- A class with the main purpose of holding objects is commonly called a *container*. Providing suitable containers for a given task and supporting them with useful fundamental operations are important steps in the construction of any program.
- Much computing involves creating collections of various forms of objects and then manipulating such collections. Reading characters into a string and printing out the string is a simple example.

- The containers are divided into three major categories—
    1. Sequence Containers – Vector, Deque, list
    2. Associative Containers – set, multiset, map. multimap
    3. Container Adapters – stack, queue, priority_queue

1.12.1.1. Vector

A built in array of (name, number) pairs would seem to be a suitable starting point:

```
struct Entry
{
      string name;
      int number;
};
Entryphone_book[1000];
void print_entry(int i) //simple use
{
      cout<<phone_book[i].name<<´´<<phone_book[i].number<<
´\n´;
}
```

However, a builtin array has a fixed size. If we choose a large size, we waste space; if we choose a smaller size, the array will overflow. In either case, we will have to write low level memorymanagement code. The standard library provides a *vector* that takes care of that:

```
vector<Entry>phone_book(1000);
void print_entry(int i) //simple use, exactly as for array
{
      cout<<phone_book[i].name<<´´<<phone_book[i].number<<
´\n´;
}
void add_entries(int n) //increase size by n
{
      phone_book.resize(phone_book.size()+n);
}
```

The *vector* member function *size*() gives the number of elements.

Note the use of parentheses in the definition of *phone_book* . We made a single object of type *vector* <*Entry*> and supplied its initial size as an initializer. This is very different from declaring a builtin array:

```
vector<Entry>book(1000); //vector of 1000 elements
vector<Entry>books[1000]; //1000 empty vectors
```

1.12.1.2. Range Checking

The standard library *vector* does not provide range checking by default. For example:

```
template <class T > class Vec : public vector <T >
{
      public:
            Vec() : vector <T >()
            { }
            Vec (int s ) : vector <T >(s) { }
            T& operator[](int i)
            {
                  return at(i);
            } // rangechecked
            const T& operator[](int i) const
            {
                  return at(i);
            } // rangechecked
};
```

Here, a *Vec* is like a *vector* , except that it throws an exception of type *out_of_range* if a subscript is out of range.

The *at*() operation is a *vector* subscript operation that throws an exception of type *out_of_range* if its argument is out of the *vector*'s range.

```
Vec <Entry> phone_book (1000 );
void print_entry(int i) // simple use, exactly as for
vector
{
      cout<<phone_book[i].name<< ´´<<phone_book[i].number;
}
```

1.12.1.3. List

Insertion and deletion of phone book entries could be common. Therefore, a list could be more appropriate than a vector for representing a simple phone book. For example:

```
list<Entry>phone_book ;
void print_entry(const string &s)
{
      typedef list<Entry>::const_iterator LI ;
      for(LI i=phone_book.begin();i!=phone_book.end();++i)
      {
            Entry &e = *i ; //reference used as shorthand
            if(s==e.name)
                  cout<<e.name<<´´<<e.number<<´\n´;
      }
}
```

Adding elements to a *l i s t* is easy:

```
void add_entry(Entry & e,list<Entry>::iterator i)
{
      phone_book.push_front(e); //add at beginning
      phone_book.push_back(e); //add at end
      phone_book.insert(i,e); //add before the element
}
```

1.12.1.4. Map

A *map* is a container of pairs of values. For example:

```
map<string,int>phone_book;
```

In other contexts, a *map* is known as an associative array or a dictionary.

When indexed by a value of its first type (called the *key*) a *map* returns the corresponding value of the second type (called the *value* or the *mapped type*). For example:

```
void print_entry(const string &s)
{
      if(int i = phone_book[s])
            cout<<s<<´´<<i<<´\n´;
}
```

If no match was found for the key *s* , a default value is returned from the *phone_book* . The default
value for an integer type in a *map* is *0* .

1.12.1.5. Standard Containers

A *ma p*, a *list* , and a *vector* can each be used to represent a phone book. However, each has strengths and weaknesses. For example, subscripting a *vector* is cheap and easy. On the other hand, inserting an element between two elements tends to be expensive. A *list* has exactly the opposite properties. A *map* resembles a *list* of (key,value) pairs except that it is optimized for finding values based on keys.

The standard library provides some of the most general and useful container types to allow the programmer to select a container that best serves the needs of an application:

Container	Standard
vector <T>	A variablesized Vector
list<T>	A doublylinked List
queue<T>	A queue
stack<T>	A stack
deque <T>	A double ended Queue
priority_queue<T>	A queue sorted by value
set<T>	A set
multiset<T>	A set in which a value can occur many times
map<key,val >	An associative array
multimap< key,val>	A map in which a key can occur many times

1.12.2. Iterators

- *Iterators* are used to point to first-class container elements.
- Iterators hold state information sensitive to the particular containers on which they operate; thus, iterators are implemented appropriately for each type of container.
- For example, the *dereferencing operator (*)* dereferences an iterator so that you can use the element to which it points. The ++ *operation on an iterator* moves it to the container's *next element* (much as incrementing a pointer into an array aims the pointer at the next array element).

- An object of type *iterator* refers to a container element that can be modified. An object of type *const_iterator* refers to a container element that *cannot* be modified.
- Using *istream_iterator* for *Input* and *ostream_iterator* for *Output*

```
#include<iostream.h>
#include<iterator.h>
int main()
{
//create istream_iterator for reading int values from cin
      cout<<"Enter two integers: ";
      istream_iterator<int>inputInt(cin);
// read int from standard input
      int number1 = *inputInt;
// move iterator to next input value
      ++inputInt;
// read int from standard input
      int number2 = *inputInt;
// create ostream_iterator for writing int values to cout
      ostream_iterator<int> outputInt(cout);
      *outputInt = number1 + number2
      cout << "The sum is: "<< endl;
} // end main
```

Output

```
Enter two integers: 12 25
The sum is: 37
```

Fig. 1.12.2 | Input and output stream iterators.

1.12.2.1. Iterator Categories and Iterator Category Hierarchy

Figure 1.12.2.1(a) shows the categories of STL iterators. Each category provides a specific set of functionality. Figure 1.12.2.1(b) illustrates the hierarchy of iterator categories.

Category	Description
Input	Used to read an element from a container
Output	Used to write an element to a container.

Forward	Combines the capabilities of *input and output iterators* and retains their position in the container
Bidirectional	Combines the capabilities of a *forward iterator* with the ability to move in the backward direction
random access	Combines the capabilities of a *bidirectional iterator* with the ability to directly access any element of the container

Figure 1.12.2.1(a). Categories of STL iterators

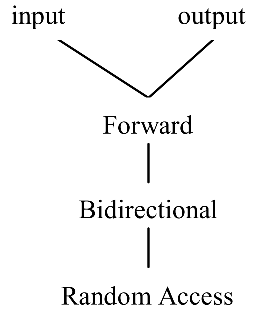


Fig. 1.12.2.1(b) Hierarchy of iterator categories

Container	***Type of Iterator Support***
Sequence containers (first class)	
vector	random access
deque	random access
list	bidirectional
Associative containers (first class)	
set	bidirectional
multiset	bidirectional
map	bidirectional
multimap	bidirectional
Container adapters	
stack	no iterators supported
queue	no iterators supported
priority_queue	no iterators supported

Fig. 1.12.2.1(c) Iterator types supported by each container.

1.12.2.2. Predefined Iterator typedefs

Figure 1.12.2.2(a) shows the predefined iterator typedefs that are found in the class definitions of the STL containers. Figure 1.12.2.2(b) shows some Iterator Operations

Predefined typedefs for iterator types	Direction of ++	Capability
iterator	forward	read/write
const_iterator	forward	read
reverse_iterator	backward	read/write
const_reverse_iterator	backward	read

Fig. 1.12.2.2(a) Iterator typedefs.

Iterator operation	Description
All iterators	
++p p++	Preincrement an iterator, Postincrement an iterator
Input iterators	
*p p = p1 p == p1 p != p1	Dereference an iterator Assign one iterator to another Compare iterators for equality Compare iterators for inequality
Output iterators	
*p p = p1	Dereference an iterator Assign one iterator to another
Forward iterators	
Forward iterators	provide all the functionality of both input iterators and output iterators.
Bidirectional iterators	
--p p—	Predecrement an iterator Postdecrement an iterator
Random-access iterators	
p +=i	Increment the iterator p by i positions

p -= i	Decrement the iterator p by i positions
p + i *or* i + p	Expression value is an iterator positioned at p incremented by i
p – i	positions
p - p1	Expression value is an iterator positioned at p decremented by i
	positions
p[i]	Expression value is an integer representing the distance between
p < p1	two elements in the same container.
	Return a reference to the element offset from p by i positions
p <= p1	Return true if iterator p is less than iterator p1 (i.e., iterator p is
	before
	iterator p1 in the container); otherwise, return false.
p > p1	Return true if iterator p is less than or equal to iterator p1 (i.e.,
	iterator p
p >= p1	is *before* iterator p1 or *at the same location* as iterator p1 in the
	container); otherwise, return false.
	Return true if iterator p is greater than iterator p1 (i.e., iterator p
	is *after*
	iterator p1 in the container); otherwise, return false.
	Return true if iterator p is greater than or equal to iterator p1
	(i.e., iterator
	p is *after* iterator p1 or *at the same location* as iterator p1 in the
	container);otherwise, return false.

Fig. 1.12.2.2(b) Iterator operations for each type of iterator.

1.13. Proxy Classes

- A proxy class that allows you to hide even the private data of a class.
- Providing clients of your class with a proxy class that knows only the public interface to your class enables the clients to use your class's services without giving the clients access to your class's implementation details, such as its private data.
- When a class definition uses only a pointer or reference to an object of another class, the class header for that other class (which would ordinarily reveal the private data of that class) is not required to be included with #include. You can simply declare that other class as a data type with a forward class declaration before the type is used in the file.

- The implementation file containing the member functions for a proxy class is the only file that includes the header for the class whose private data we would like to hide.
- The implementation file containing the member functions for the proxy class is provided to the client as a precompiled object code file along with the header that includes the function prototypes of the services provided by the proxy class. Example,
- Creation of Proxy classes
  1. Myclass.h
  2. MySInterface.h
  3. MyInterface.cpp
  4. Test.cpp

```
Example,
```

MyClass.h

```
class MyClass
{
      private :
            int val;
            MyClass(int v):value(v)
            {}
            void setValue(int v)
            {
                  val=v;
            }
            int getValue()
            {
                  return val;
            }
};
```

MyInterface.h

```
class Myclass;
class MyInterface
{
      private:
            Myclass *ptr;
      public:
            MyInterface(int);
            void setValue(int);
            int getValue();
            ~MyInterface();
};
```

MyInterface.cpp

```
#include"d:\MyInterface.h"
#include"d:\Myclass.h"
MyInterface::MyInterface(int v):ptr(new Myclass(v))
{}
void MyInterface::setValue(int v)
{
      ptr->setValue(v);
}
int MyInterface::getValue()
{
      return pt->getValue();
}
MyInterface::~MyInterface()
{
      delete ptr;
}
```

Test.cpp

```
#include<iostream.h>
#include"d:\MyInterface.h"
void main()
{
      MyInterface obj(10);
      cout<<"Interface Contains(Before):"<<obj.getValue();
      obj.setValues(20);
```

```
        cout<<"Interface Contains(After):"<<obj.getValue();
}
```

OUTPUT

```
Interface Contains(Before):10
Interface Contains(After):20
```

1.14. Overloading: Function overloading and Operator Overloading.

1.14.1. Function Overloading

A function is overloaded when same name is given to different function. However, the two functions with the same name will differ at least in one of the following:

a. The number of parameters
b. The data type of parameters
c. The order of appearance

These three together are referred to as the *function signature.*

For example if we have two functions :

void foo(int i,char a); void boo(int j,char b);

- Their signature is the same **(int ,char)** but a function **void moo(int i,int j) ;** has a signature **(int, int)** which is different.

While overloading a function, the return types of the functions need to be the same. In general functions are overloaded when :

1. Functions differ in function signature.
2. Return type of the functions is the same.

```
#include<iostream.h>
#include<conio.h>
int volume(int);
double volume(double,int);
long volume(long,int,int);
int main()
{
        clrscr();
        cout<<”volume of cube=”;
        cout<<volume(10)<<"\n";
        cout<<"volume of cylinder=";
        cout<<volume(2.5,8)<<"\n";
        cout<<"volume of cubiod=";
        cout<<volume(100,75,15)<<"\n";
        return 0;
        getch();
}
int volume(int s)
{
        return(s*s*s);
}
double volume(double r,int h)
{
        return(3.14*r*r*h);
}
long volume(long l,int b,int h)
{
        return(l*b*h);
}
```

OUTPUT

```
volume of cube = 1000
volume of cylinder = 15700
volume of cubiod = 112500
```

1.14.2. Operator Overloading

To create new definitions to existing operators. In other words a single operator can perform several functions as desired by programmers.

The general syntax for defining an operator overloading is as follows:

```
return_type classname::operator operator-symbol(argument)
{
          …………..
          statements;
}
```

Example

```
#include<iostream.h>
#include<string.h>
#include<conio.h>
const int bufsize=50;
class string
{
        private:
                char str[bufsize];
        public:
        string()
        {
                strcpy(str," ");
        }
        string(char*mystr)
        {
            strcpy(str,mystr);
        }
        void echo()
        {
            cout<<str;
        }
        string operator+(string s)
        {
            string temp=str;
            strcat(temp.str,s.str);
            return temp;
        }
};
void main()
{
        clrscr();
        string str1=" stud";
        string str2=" ent";
        string str3;
        cout<<"\n before str3=str1=str2;.";
        cout<<"\n str1=";
        str1.echo();
        cout<<"\nstr2=";
        str2.echo();
        cout<<"\n str3=";
        str3.echo();
        str3=str1+str2;
        cout<<"\n\n After str3=str1+str2:..";
        cout<<"\n str1=";
        str1.echo();
        cout<<"\n str2=";
        str2.echo();
        cout<<"\n str3=";
        str3.echo();
        getch();
}
```

OUTPUT

```
Before str3=str1=str2;
str1=stud
str2=ent
str3=
After str3=str1+str2:
str1=stud
str2=ent
str3=student
```

1.14.2.1. Types of Operators

i. Unary Operators

ii. Binary Operator

i. Unary Operator Overloading

As the name implies takes operate on only one operand. Some unary operators are namely

1.	++	- Increment operator
2.	--	- Decrement Operator
3.	!	- Not operator
4.	-	- unary minus.

The important steps involved in defining an operator overloading in case of unary operators are namely:

1. Inside the class the operator overloaded member function is defined with the return data type as member function or a friend function.
2. If the function is a member function then the number of arguments taken by the operator member function is none.
3. If the function defined for the operator overloading is a friend function then it takes one argument.

Example

```
#include<iostream.h>
#include<conio.h>
class unary
{
        private:
        int x,y,z;
        public:
         unary(void)
         {
                cout<<"Enter Any Three Integer Nos.:";
             cin>>x>>y>>z;
         }
         void display(void)
         {
             cout<<"The Three Nos. are:"<<x<<" ,"<< y<<","<<z;
```

```
            }
            void operator--()
            {
                        x=--x;
                  y=--y;
                  z=--z;
            }
            void operator++()
            {
                        x=++x;
                  y=++y;
                  z=++z;
            }
};
void main()
{
            clrscr();
      unary s;
      s.display();
      --s;
      cout<<endl"The Decremented Values"<<endl;
      s.display();
      ++s;
      cout<< endl<<"The Incremented Values"<<endl;
      s.display();
      getch();
}
```

Output:

```
Enter Any Three Integer Nos. :
4 7 9
The Three Nos. Are : 4 , 7 , 9
The Decremented Values
The Three Nos. Are : 3 , 6 , 8
The Incremented Values
The Three Nos. Are : 4 , 7 , 9
```

ii. Binary Operator Overloading

The arithmetic operators, comparison operators, and arithmetic assignment operators come under this category.

Example

```
#include <iostream.h>
#include<conio.h>
class complex
{
      float x;
      float y;
      public:
            complex(){}
            complex(float real,float image)
            {
                  x=real;
                y=image;
            }
            complex operator +(complex);
            void display(void);
};
complex complex::operator+(complex c)
{
            complex temp;
      temp.x=x+c.x;
      temp.y=y+c.y;
      return(temp);
}
void complex::display(void)
{
            cout<<x<<"+j"<<y;
}
int main()
{
      clrscr();
      complex c1,c2,c3;
      c1=complex(2.5,3.5);
      c2=complex(1.6,2.7);
      c3=c1+c2;
      cout<<"c1 =";
      c1.display();
      cout<<"c2 =";
      c2.display();
      cout<<"c3 =";
      c3.display();
      getch();
      return 0;
}
```

OUTPUT

```
c1=2.5+j3.5
c2=1.6+j2.7
c3=4.1+j6.2
```