OOP Note

Feel Free

to Share — to copy, distribute and transmit the work

Special thanks to:

Mr. Sudip Lama
Lecturer and Coordinator at Kathmandu Engineering College

Contributors:

- Ashmita Mishra
- Ayush Maharjan
- Deepak Adhikari
- Kshitiz Tiwari

BCT A
2068 Batch
Kathmandu Engineering College

Report errors/mistakes at **kecbcta2068@gmail.com**

Chapter 1

Introduction to Object Oriented Programming

1.1 Introduction with Procedure Oriented Programming

Limitations of Procedural Oriented Programming:

i. Emphasis on algorithm (or procedure) rather than data. Data takes the backseat with this programming paradigm
ii. Change in a data type being processes ness to propagate to all the functions that use the same data type. This is frustrating and time consuming process
iii. Maintaining and enhancing program code is difficult due to global data
iv. The procedural programming paradigm does not model real world very well

1.2 Basic of Object Oriented Programming

Procedural Programming is the method of programming in which emphasis is given on process. It follows a top-down approach. Procedural Programs are generally decomposed into functions. It does not model real world as it is based on following certain steps to complete a given task. It is generally difficult to debug and modify the code in this approach.
Due to the problem faced in procedural oriented programming, to remove these flaws Object Oriented Programming was introduced. In OOP data is treated, as critical element in program development and it does not allow free flow of data. In OOP, data associate with the function that operates it and protect it from accidental modification from outside functions. OOP, decompose the problem into a number of entities called object and the builds data and functions around these objects. The data of an object can be access though only the function defined in object. And one object can access the function of another object.
OOP is a method of implementation in which program are organized as cooperative collection of objects, each of which represents are instance of some class, and whose classes are all members of a hierarchy of classes united via inheritance relationship.

1.3 Procedure Oriented versus Object Oriented Programming

Procedure Oriented Programming	Object Oriented Programming
1. Emphasis is given on process. 2. Follow top-down approach for program design. 3. Program are decomposed into function. 4. It does not model real world problems. 5. Maintaining and enhancing code is difficult. 6. Code reusability is difficult.	1. Emphasis on data. 2. Follow bottom-up approach for program design. 3. Program are decomposed into objects. 4. It models the real world problems. 5. Maintaining and enhancing code is easy. 6. Code reusability is easy as compare to POP.

1.4.1 Object

Objects are the runtime entities in an object-oriented system. It is also known as identifiable entity with some characteristic and behaviour. For instance, Orange is an object. Its characteristic is : it is a spherical shaped, its color is orange etc. Its behaviour is : it is juicy and it tastes sweet sour. In OOP approach, the characteristic of object means the data and its behaviour is represented by its associated functions.

1.4.2 Class

A class is a group of objects that share common properties and relationships. It is a user defined data type, which defines the set of data and code. Objects are the variable of the type class. Once a class has been defined, we can create any number of objects belonging to that class.

1.4.3 Abstraction

Abstraction refers to the act of representing essential features without including the background details or explanations. Classes use the concept of abstractions are defined as a list of abstract attributes such as size, weight and cost, and functions to operate on these attributes. To understand abstraction, let us consider an example of switch board. We know only to press switches according to our requirements. This is, we know which switch to press for which light or machine but we don't know what is happening inside the switchboard. This is abstraction, we know essential features but we don't know about the background details. This feature also helps in implementing data hiding.

1.4.4 Encapsulation

Encapsulation is known as the mechanism of wrapping up of data and function in a single unit. The only way to access data is through functions that are combined along with data. These functions are called member functions in C++. The data cannot be accessed directly in the object. We can read, write or use the data only through the member function of the object so it prevents from accidental alteration. While encapsulating data and function abstraction are implemented as relevant data are made available and rest are made hidden.

1.4.5 Inheritance

It is the capability of one class to inherit the capabilities or properties from another class. It supports the hierarchical classification. While deriving a new class we can add additional features to derive class without modifying it.
In OOPs, as inheritance is a capabilities to express the inheritance relationships which makes it ensure the closeness with the real-world models. It also give an idea of reusability, it allows the addition of additional features to an existing class without modifying it. Inheritance shows the transitive nature i.e. if B class is derived from class A and class C is derived from B then C inherit the properties of class A

1.4.6 Polymorphism

Polymorphism means, “having different forms”. Polymorphism is the concept that supports the capabilities of an object of a class to behave differently in response to a message or action. An operation may behave differently in different instance.
Example of polymorphism in C++ is operator overloading, function overloading and another type of polymorphism is achieved during runtime is known as late binding.

1.5 Example of Some Object Oriented Languages

- C++
- Java
- LISP
- PHP5
- Python
- Ruby

1.6 Advantages and Disadvantages of OOP

Advantages:

1. High code reusability through inheritance, templates.
2. Real world representation.
3. Data hiding through abstraction so that data is secure.
4. Dividing program into number of object makes software development easy.
5. Maintenance and upgrading of software easy.
6. Improve reliability and flexibility.
7. Existing class can serve as library for further enhancement.
8. Message passing makes the interface easy.

Disadvantages:

1. If the message passing between many object in an application, it is difficult to trace the error.
2. Compile time and execution time is high, as it requires more time for dynamic memory allocation and runtime polymorphism.
3. Require full knowledge of OOP before using it properly.

Chapter 2

Introduction to C++

Bjarne Stroustrup developed the C++ programming language at AT&T Bell Laboratories in the early 1980s. He extended the C with the features from Simula 67. Bjarne Stroustrup called it “C with classes” originally. the name C++ (pronounced C plus plus) was coined by Rick Mascitti where “++” is the increment operator. Even since its birth C++ evolved the cope with problems encountered user and through discussion at AT&T.
During the early 1990’s the language underwent a number of improvements and changes. In november 1997, the ANSI/ISO standards committee standardized these changes and added several new features to the languages specification.
The object oriented features in C++ allows programmers to create large programs with clarity, extensibility and easy to maintenance, incorporating the spirit and efficiency of C.

2.1 Need of C++:

Bjarne Stroustrup said, “C makes it easy to shoot yourself in the foot; C++ makes it harder, but when you do, it blows away your whole leg”. And it is rightly so because, as much as amazing C is, it has its limitations. C cannot handle complexity. In solving problems for larger programs, the developer had tough time grasping the totality. Also data’s that are globally declared can be accessed by functions other than the function declared for. These are just few examples of C++ limitations. The features of C++ gives an elaborate idea of why C++ is needed.
With an intention to solve these problems, Bjarne Stroustrup invented C++ in 1980s. C++ is nothing but C with Classes. It adds object oriented features in C, which means C++ has all features and benefits of C.

2.2 Features of C++:

C++ supports all the main features of Object Oriented Programming. Only some of the important features are mentioned below:

i. Casts: In C++, we can cast an object of fundamentally different in data type using reinterpret_cast, and change the constant by typecasting like const_cast.
ii. Flexible Declaration: In C programming variable must be declared at starting of function definition but in C++ we can declare variable where we need, and it helps to show dynamic initialization.
iii. Function Overloading: In C++ we can define a function with same name and help to remember the name of the function which is logically same by remembering the single name. It also helps in displaying polymorphism in program.
iv. ‘struct’ and ‘union’ tags: In c programming, we have to use the struct and union tags to define the variable of struct or union type.
v. Reference variable: In C++, we can create an alias of predefined variable so that any changes made to it reflected to actual variable. It helps the function to returning multiple values from the functions.

vi. new and delete operator: In C programming for dynamic memory allocation, alloch header file has to be included and we use the functions like malloc(), calloc() and free() to allocate and deallocate the memory. But in C++, we use the new and delete operator to do that.
vii. Class: In C++, declaring class in program helps to bind the data and function in a single unit. And it implements the data abstraction, encapsulation and data hiding.
viii. Inheritance: In C++, it add a capability to class to inherit the properties of another class so it is easy to modify or maintain the previously created program.
ix. Generic programming: In C programming we need to define different code for the same algorithm but with different data types. But in C++, show the generic programming with the helps to define template where we can create single function which can work for different data type.
x. STL: In C++, the data structures, which are needed in programming are already defined with different function associated in a library which is known as Standard Template Library. It implement template to show generic programming.

2.3 C++ Versus C:

C	C++
i. C follows procedural programming paradigm i.e.importance is given to the steps or procedure of the program. ii. C uses the top-down approach. iii. Data is not secure. iv. C is regarded as a low-level language (difficult interpretation & less user friendly). v. Functions are the building blocks of a C program and so it is a function-driven. vi. C does not support function overloading. vii. Structures can not contain functions in C. viii. The NAMESPACE feature is absent in C. ix. C does not allow the use of reference variables.	i. C++ is a multi-paradigm language (procedural as well as object oriented) i.e. C++ focuses on the data rather than the process. ii. C++ uses the bottom-up approach. iii. Data is secure (hidden) because of the OOP feature like data hiding. iv. C++ has feature of both low-level (concentration on what's going on in the machine hardware) & high-level languages (concentration on the program itself) & hence is regarded as a middle-level program. v. C++ is object-driven, as objects are building blocks of a C++ program. vi. C++ enables function overloading with the help of polymorphism (an OOP feature). vii. In C++, functions can be used inside a structure. viii. C++ allows the NAMESPACE feature. ix. C++ allows the use of reference variable.

2.4 History of C++

C++ was developed by Bjarne Stroustrup starting in 1979 at AT&T Bell Labs, it adds object oriented features, such as classes, and other enhancements to the C programming language. Originally named C with Classes, the language was renamed C++ in 1983 by Rick Mascitti, as a pun involving the increment operator. It is an extended version of C with the features of Simula67.
Even since its birth C++ evolved to cope with problems encountered user and through discussion at AT&T.

During the early 1990's the language underwent a number of improvements and changes. In november 1997, the ANSI/ISO standards committee standardized these changes and added several new features to the languages specification.

The object oriented features in C++ allows programmers to create large programs with clarity, extensibility and easy to maintenance, incorporating the spirit and efficiency of C.

Chapter 3

C++ Language Constructs

3.1 C++ Program Structure

Following figure explain the structure of C++ program. C++ commonly organize the program into three sections. FIrst the class declarations is placed with header file and definition of member function is placed in another file This approach helps to separate the class definition from the member function. Finally, the main program is stored in another file, which uses both files, which was created easily.

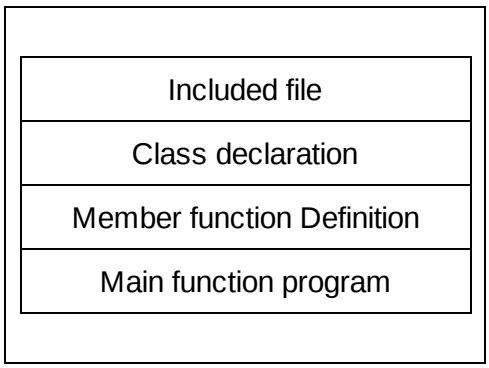


Fig: Structure of C++ program

First the header file needed in programming is included with the required namespace, after that we define the classes with their associated member functions that are require in the program. The important part of program is matin function, a program start and end at main(). In a main() function we write codes that your program need to perform

3.2 Character Set and Tokens

Character Set:
Character set is a set of valid character that a language can recognize. A character represents any letter, digit, or any other sign. The C++ has the following character set.

Digit:	0-9
Letters:	A-Z, a-z
White spaces:	Blank space, Horizontal tab, Carriage return, Newline, Form feed
Special Symbols:	space + - * / \ % ^ ( ) { } [ ] = > < , ‘ “ $ & | ! ~ : ? _ # <= >= ==
Other characters:	C++ can process any of the 256 ASCII character as data or as Literals

Tokens:
Token is the smallest unit of a program. C++ has the following tokens: Keywords, Identifiers, Literals, Punctuators and Operators.

3.2.1 Keywords

These are the word that conveys a special meaning to the compiler. All keyword have fixed meaning and these meaning cannot be changed, Keywords serve as basic building blocks for program statement. All keywords must be written in lowercase.
The original C++(developed by Stroustrup) contains the following keywords.

and	continue	goto	public	try
and_eq	default	if	register	typedef
asm	delete	inline	reinterpret_cast	typeid
auto	do	int	return	typename
bitand	double	long	short	union
bitor	dynamic_cast	mutable	signed	unsigned
bool	else	namespace	sizeof	using
break	enum	new	static	virtual
case	explicit	not	static_cast	void
catch	export	not_eq	struct	volatile
char	extern	operator	switch	wchar_t
class	false	or	template	while
compl	float	or_eq	this	xor
const	for	private	throw	xor_eq
const_cast	friend	protected	true	

source: http://en.cppreference.com/w/cpp/keyword

3.2.2 Identifiers

An identifier is an arbitrarily long sequence of letter and digits. THese are user-defined names and consist of sequence of letters and digits, with a letter as first character Both uppercase and lowercase letters are permitted, C++ is case sensitive and it treats upper and lower-case characters differently. The underscore character is also permitted in identifiers.
The rules of defining identifier:

i. First character must be an alphabet (or underscore)
ii. Must consist of only letters, digits or underscore
iii. Cannot use a keyword
iv. Must not contain whitespace or special character

The following are the some valid identifiers:
Sagarmatha KEC Data12 _Rise _Do12 _for _foo_bar

3.2.3 Literals

Literals are data items that never change their value during the program run. It is referred as constants. C++ allows several kinds of literals:

i. Integer-constant
ii. Character-constant
iii. Floating-constant
iv. String-constant

3.2.4 Operators and Punctuators

Operators:
Operators are tokens that trigger some computation or action when applied to variables and other objects in an expression. There are mainly following type of operator in C++

i.	Arithmetic operator	+ - * / %
ii.	Increment/Decrement operator	++ --
iii.	Relational operator	< <= > >= == !=
iv.	Logical operator	&& || !
v.	Conditional operator	? : exp1 ? exp2 : exp3 large = (a>b) ? a : b;
vi.	Assignment operator	=
vii.	Bitwise operator	& | ^ << >>
viii.	Other operator	comma (,) sizeof pointer-operator(& and *) selection-operator(. and ->)

Punctuators:
The following characters are used as punctuators (also known as as separators) in C++:
[ ] ( ) { } , ; : * … = #
That enhances a program's readability.

3.8 Namespace

A namespace is the mechanism for expressing logical grouping. That is, if some declaration logically belong together according to some criteria, they can be put in a common namespace to express the fact, The general form of namespace is :

```
namespace namespace_name
{
        //Declaration of variable,function, class etc
}
```

While using the member defined inside the namespace, we use namespace_name::member notation.

For example, consider the width and height and function which calculate the area may be placed in namespace (Rectangle) :

```
namespace Rectangle
{
        int width;
```

```
        int height;
        void area();
}
```

If we want to use the member in a program, we need to access as Rectangle::width = 10;

A width, height variable and area() are inside the scope defined inside the Rectangle so the function area() can use width and height directly without specifying the namespace_name.

A member can be declared inside the namespace definition and defined later anywhere in program. Like in above code we can define the area() as follows:

```
void Rectangle::area()
{
        cout << "\nArea is: " << width*height;
}
```

When a function that is declared inside a namespace is defined outside, it should be qualified. We cannot declare a new member of a namespace outside a namespace definition using the definition syntax. For example :

```
        void Rectangle::sum(); // Invalid
```

This is an error , we cannot define a member outside namespace without declaring inside the namespace.

Following program will demonstrate the use of namespace

```
#include<iostream>
using namespace std;

namespace Rectangle
{
      int width;
      int height;
      void area();
}

int main()
{
      Rectangle::width = 10;
      Rectangle::height = 2;
      Rectangle::area();

return 0;
}

void Rectangle::area(){
      cout << "\nThe Area is: " << width*height << endl;
}
```

```
O/P
```

```
The Area is: 20
```

In above program, we learned to use the member inside the namespace we need to use the

namespace_name like Rectangle::width, Rectangle::height and Rectangle::area().

3.9 User Defined Constant const

We may like the value of certain variables to remain constant during the execution of a program. We can achieve this by declaring the variable with the qualifier const at the time of initialization.

For example : const int size = 30;

const is a new data type qualifier defined by ANSI standard. This tells the compiler that the program must not modify the value of the ini variable size. However, it can be used on the right-hand side of an assignment statement like any other variable.

3.10 Input/Output Streams and Manipulators

The basic data type for input / output is the stream. The most basic stream types are included under the <iostream> header file. The most basic stream types are the standard input.output streams:

istream	cin	built-in input stream variable, by default hooked to keyboard
ostream	cout	built-in output stream variable, by default hooked to console.

C++ also support all the input/output mechanisms that C language include.

Manipulators are the most common way to control output formatting. In C++, the function for formatting output are included under the <iomanip> header file.
Some of the manipulators are discussed below:

endl	Write a newline ('\n') and flush buffer.
setw(n)	Sets minimum field width on output. This sets the minimum size of the field - a larger number will use more columns. To print a column of right justified numbers in a seven column field: cout << setw(7) << n << endl;
setfill(ch)	Only useful after setw(). If a value does not entirely fill a field, the character *ch* will be used to fill in the other characters. Default value is blank. Same effects as cout.fill(ch) For example, to print a number in a 4 character field with leading zeros (eg, 0007): cout << setw(4) << setfill('0') << n << endl;

3.11 Dynamic Memory Allocation with new and delete

Allocation of memory during runtime is known as dynamic memory allocation(DMA). C++ provides two operators for DMA - new and delete.The new operator allocates the memory dynamically and returns a pointer storing the memory address of the allocated memory. The operator delete allocates the memory pointed by the given pointer.

The general form of using new and delete is as follows:

data-type *pointer-variable;
pointer-variable = new data-type; //allocates single variable
pointer-variable = new data-type[size]; // allocates an array of size elements
delete pointer-variable; // if memory was allocated for a single variable
delete [size] pointer-variable; // if memory was allocated for an array of size elements
delete [ ] pointer-variable; // alternative for delete [size] pointer-variable

```
#include <iostream>
#include <stdlib.h>
using namespace std;

int largest (int *p, int s) {
     int l = p[0];
     for ( int i = 1; i < s; ++i)
          if (p[i] > l)
          l = p[i];
return l;
}

int main()
{
     int *ptr, size;
     cout << "Enter the size of array:";
     cin >> size;
     ptr = new int [size]; // memory for array of size elements
allocated
     if (ptr == NULL) {
          cout << "Memory could not be allocated";
          exit(0);
     }
     for ( int i = 0; i < size ; ++i)
          cin >> ptr[i];
     cout << "The largest element is :" << largest(ptr,size);

return 0;
}
```

O/P

```
Enter the size of array:4
3
7
8
2
```

```
The largest element is :8
```

3.13 Functions

3.13.1 Function Syntax:

Function prototype/declaration:

```
return-type function-name(parameter list);
```

Function Definition

```
return-type function-name(parameter list){
     body of the function;
}
```

Function call

```
function-name(list of variable or value);
```

3.13.2 Function Overloading

A signature of a function is its argument type, argument number and argument order and at least one difference must exist in the signature of a function for function overloading.
We can use the same function name to create function that perform a variety of different tasks. This is known as function overloading/ function polymorphism. We can design a function with same function name but with different argument list. The correct function to be invoked is determined by checking the no. and type of argument s but not on the function type.

For example :

```
#include <iostream>
using namespace std;

void sum (int a, int b){
     cout << "Function 1 : The sum is : " << (a+b) << endl;
}

void sum (int a, double b, int c){
     cout << "Function 2 : The sum is : " << (double)(a+b+c) <<
endl;
}

int main()
{
     sum (20, 30);              //Calls Function 1
     sum (20, 30.10 , 40);      //Calls Function 2

return 0;
}
```

O/P

```
Function 1 : The sum is : 50
Function 2 : The sum is : 90.1
```

3.13.3 Inline Functions

Inline Function is a function which expands to a line where it is invoked or called instead of jumping to the function itself. It is declared using the keyword “inline” in the function definition.

Syntax:

```
inline return-type function-name(list-of-arguments) {
      //body of the function
}
```

Inline functions increase the speed of execution of the program but it also increases the size of the program. So it is used only for small functions.

A function cannot be made inline:

- If static variable is declared inside the function
- If a function returns a value where return type is specified
- If function contains loop, switch or goto
- If a function is recursive

The inline keyword is just a request to the compiler to make a function inline function. The compiler may ignore the request if the function definition is too long or complicated and compile the function as a normal function.

Note: Inline functions serve the same function as #define macro (generally used in C) but it provide better type checking and do not require special care for parentheses.

3.13.4 Default arguments

Default arguments are the default values are the default values provided to the arguments of the function. The default values are assigned during function declaration. The default value is used if the value is not passed during function call. Otherwise, it uses the passed value.

```
#include <iostream>
using namespace std;

float interest(float p = 1000; int time = 1; int rate = 0.5){
    return (p * time * rate);
}

int main()
{
    cout << “Interest =” << interest() << endl;
    cout << “Interest =” << interest(1200) << endl;

    cout << “Interest =” << interest(1200, 10) << endl;
    cout << “Interest =” << interest(1200, 10, 0.1) << endl;
```

```
return 0;
}
```

O/P

```
Interest = 750
Interest = 900
Interest = 1800
Interest = 1200
```

A function having N default arguments can be invoked in N+1 different ways as shown by the above example.
Default arguments must be provided from rightmost parameter in the argument list.

```
float interest(float p; int time; int rate = 0.5);
float interest(float p = 1000; int time = 1; int rate);      //error
float interest(float p; int time = 1; int rate);             //error
```

In the function call, any argument in a function cannot have a default value unless all arguments appearing on its right have their default values.

Note: Ambiguity arising when both function overloading and function with default arguments are used must be avoided.

Eg:

```
void function(int x, int y=0);
void function(int x);
```

Ambiguity arises when the function is called using only one integer as argument.

3.13.5 Pass by Reference

Usually when a function is called, the arguments are copied into function and the function works on the copied value. Thus, the original value is not changed. If the original value must be changed then it must be arguments must be passed by reference. It can be achieved in two ways:

- using reference variable
- using pointer

```
#include <iostream>
using namespace std;

//pass by value
void swap1(int a, int b){
    int temp = b;
    b = a;
    a = temp;
}

//pass by reference using reference variable
void swap2(int &a, int &b){
    int temp = b;
    b = a;
    a = temp;
```

```
}

//pass by reference using pointer
void swap3(int *a, int *b){
    int temp = *b;
    *b = *a;
    *a = temp;
}

int main(){
    int a = 10, b = 20;

    swap1(a,b);
    cout << "a=" << a << "," << "b=" << b << endl;

    swap2(a,b);
    cout << "a=" << a << "," << "b=" << b << endl;

    swap3(&a,&b);
    cout << "a=" << a << "," << "b=" << b << endl;

return 0;
}
```

O/P

```
a=10,b=20
a=20,b=10
a=10,b=20
```

In the above example,

- swap1() uses pass by value, i.e., it copies the values to the arguments of function. So, 'a' and 'b' in the main() function and 'a' and 'b' inside swap1() are different variables. Thus, in main() 'a' remains 10 and 'b' remains 20.
- swap2() uses pass by reference using reference variable. So swap2() uses alias names to reference the variables in main(). 'a' is referenced as 'a' and 'b' is referenced as 'b' (they can be reference by different names of course). When 'a' and 'b' are swapped in swap2(), the change is reflected in main() too.
- swap3() uses pass by reference using pointers. swap3() uses pointers to directly access the address of the variables in main() and, hence, directly swaps the values stored in the two memory locations.

Passing by reference using reference variable vs Passing by be rreference using pointer

- using reference variable saves memory as memory is not allocated for pointer variable
- using reference variable is easier as no referencing / dereferencing is required like in pointers.
- reference variable references only a particular address it was first used to reference and hence works like a constant pointer. Using a pointer allows us to use the same pointer to point different addresses when required.

Note: only variables can be passed by reference. We cannot pass constants like swap(2,3). Compiler generates error message if constants are passed.

3.13.6 Return by Reference

A function can also return by reference. Return by reference allows value to be assigned to the variable returned (i.e., the function can be used in the left side of assignment operator '=' !!) as shown in the example below:

```
/*
Program to create a function which takes two arguments and returns
the minimum value. Call the function and assign it to a variable.
Again call this function and assign the value - 100. Trace the output
*/

#include <iostream>
using namespace std;

int &minimum(int &a, int &b){
     return (a<b ? a : b);
}

/*
//this function gives an error as a and b have local scope
int &minimum(int a, int b){
     return (a<b ? a : b);
}
*/

main()
{
     int a, b, min;

     cout << "enter two numbers: ";
     cin >> a >> b;

     min = minimum(a, b);
     cout << "\nvalue of a " << a;
     cout << "\nvalue of b " << b;
     cout << "\nmimimum of a & b = " << min;

     minimum(a,b) = 100;
     cout << "\nvalue of a " << a;
     cout << "\nvalue of b " << b;

return 0;
}
```

O/P

```
enter two numbers: 10 20
value of a 10
value of b 20
minimum of a & b = 10
value of a 100
value of b 20
```

In the above example, minimum returns a variable by reference. The variable to be returned by the function is referenced by an anonymous reference variable in main().

Note:
Scope of the variables should be carefully chosen. A local variable of a function cannot be returned by referenced and generates an error. This is because local variables has a scope within the function only and is destroyed when function execution is completed. Thus, returning by reference will try to reference a variable that does not exist causing the error.

3.15 Structure, Union and Enumeration

Structure:

- Structure is by default public
- Public members are accessible from outside class / struct using the object class or structure
- Private or protected members are accessible from outside the class

```
#include<iostream>
using namespace std;

class Time
{
private:
     int hr;
     int min;

public:
     int sec;

void input()
{
     cout<<"\nEnter the hr min sec:";
     cin>>hr>>min>>sec;
}
void display()

{    cout<<"\nThe time:"<<hr<<":"<<min<<":"<<sec<<endl;
}
void add(Time,Time);


};
void Time::add(Time t5,Time t6)
{

     hr=t5.hr+t6.hr;
     min=t5.min+t6.min;
     sec=t5.sec+t6.sec;
     //write conversion rule

}

int main()
```

```
{
    Time t1,t2,t3;
    //t1.hr=9;
    //sec=11;
    t1.input();
    t2.input();
    t3.add(t1,t2);
    t3.display();

return 0;
}
```

O/P

```
Enter the hr min sec:2
3
4

Enter the hr min sec:5
6
7

The time:7:9:11
```

Chapter 4

Objects and Classes

4.1 C++ Classes

A Class is a group of similar objects and describes both characteristics(data members) and behaviour(member functions) of the object. Classes are user defined data types that binds together data types and function.
A class can be defined using the keyword “class” just like structure can be defined using “struct” keyword.

Syntax:

```
class class-name {
      //members of the class are defined here
};
```

Declaration of a class involves four attributes:

- Tag-name/Class-name: the name by which the objects of the class are created
- Data Members: the data types which makes up the class.
- Member Functions/Methods: the function which operate on the data of the class.
- Program Access Levels(private/public/protected): defines where the members of class can be used

A general class construct is show below:

```
class class-name {
        private:
                //private data members and member functions
        public:
                //public data members and member functions
        protected:
                // protected data members and member functions
};
```

4.2 Access Specifiers

The member of a class are categorised into ‘private’, ‘public’ and ‘protected’ to denote how they can be accessed.

- Public Members can be accessed directly by the object of the class
- Private Members can be accessed only through the member functions (and friend functions)
- Protected Members, like private members, can be accessed only through member functions (and friend functions)

Private Members and Protected Members differ during inheritance. Private Members are not inheritable but Protected Members are inheritable.
If access specifier is not specified a class makes it private by default.

4.3 Objects and the Member Access

Objects are the instances of the class just as variables are instances of basic data types. An objects is declared like a normal variable, but using the class-name as data type.
Eg: object o1, o2; // o1 and o2 are objects of class object

The memory allocation takes place only when an object is created, not when the class is defined.
The data members and member functions(in public section) can be accessed just like in structures using dot(.) operator.

Eg:

```
o1.data;          //accessing data member
o1.function();    //accessing member function
```

Note: if pointer of the class is created -> operator is used instead of '.'

4.4 Defining Member Functions

Member functions are declared inside the class but may be defined inside a class or outside the class.

- Defining Member Function inside a class
  A member function can be defined inside a class just like a normal function. A member function defined inside a class is automatically inline.
  Eg:

```
class X {
      int a;
public:
      void input(){
            cout << "Enter value of 'a':";
            cin >> a;
      }
};
```

- Defining Member Function outside a class
  A member function can be defined outside a class using class name and scope resolution operator (::) before the function name as shown in example below:

```
class X {
      int a;
      void input();     // function declaration inside class
};
void X::input(){
      cout <<"Enter value of 'a':";
      cin >> a;
}
```

The scope resolution operator is used to specify the class to which the member function belongs.

4.5 Constructors

Constructor is a member function has same name as class name, it does not have a return type (not even void), and it is invoked or called automatically when the object of that class is created.

4.5.1 Default Constructor

Default Constructor is the constructor with no parameter.

```
#include <iostream>
using namespace std;

class Constructor
{
      int a;

public:
      Constructor(){
            a=10;
      }
      void display(){
            cout << "The value of a is " << a << endl;
      }
};

main()
{
      Constructor C1;
      C1.display();

return 0;
}
```

O/P

```
The value of a is 10
```

```
/*
Program to count number of objects created automatically
*/

#include <iostream>
using namespace std;

class Obj
{
      static int count;
public:
      Obj(){
            count++;
      }
      static void display(){
            cout << "No of object created: " << count << endl;
      }
};
```

```
int Obj::count;

main()
{
     Obj o1, o2, o3;
     Obj::display();

return 0;
}
```

O/P

```
No of object created: 3
```

4.5.2 Parameterized Constructor:

The constructor function that can take arguments are called parameterized constructor.

```
# include <iostream>
using namespace std;
class constructor
{
     int a;
     constructor();
public:
constructor (int c)
{
     a=c;
}
void display()
{
     constructor c1;
          cout << a << endl;
}
};

constructor :: constructor()
{
     a=0;
}

int main()
{
     //constructor c1;
/* since default constructor is not defined inside specific, it can
not be called */
     constructor c2(10);
     constructor c3= constructor(5);
     //c1.display();      /* infinite loop occurs */
     c2.display();
     c3.display();
}
```

O/P

```
10
5
```

4.5.3 Copy Constructor:

It is a member function. It has same name as class name. It is invoked automatically when the object of that class is created. This constructor has an argument of an object of same type or same class as a reference. It is used for initializing an object of a class through another object of same class.

```
#include <iostream>
using namespace std;

class Constructor
{
      int r, m;

public:
      Constructor(){
            cout << "Invoking default constructor" << endl;
            r=m=0;
      }
      Constructor(int x, int y){
            cout << "Invoking parameterized constructor" << endl;
            r=x; m=y;
      }
      Constructor(Constructor &T1){
            cout << "Invoking copy constructor" << endl;
            r = T1.r + 1;
            m = T1.m + 1;
      }
      void display(){
            cout << r << "+" << m << "i" << endl;
      }
};

main()
{
      Constructor C1;
      Constructor C2(1,2);
      Constructor C4(C1);  // implicit
      Constructor C5 = C2; // explicit
      C4.display();
      C5.display();
      C4 = C2;        // this does not invoke the copy constructor
      C4.display();

return 0;
}
```

O/P

```
Invoking default constructor
Invoking parameterized constructor
```

```
Invoking copy constructor
Invoking copy constructor
1+1i
2+3i
1+2i
```

4.6 Destructors:

It is a member function. It has same name as a class name preceded by tilta (~). It does not have any arguments and it does not have any return type (not even void). It is invoked automatically when the object of that class goes out of the scope or flushed from the memory.

```
# include <iostream>
using namespace std;

class demo
{
     int id;
     static int count;
public:
     demo()
     {
          count ++;
          id = count;
               cout <<"\nID" << id << " object created:" ;
     }
     ~demo()
     {
               cout <<"\nID" << id << " object destroyed:" ;
     }
};

int demo:: count = 0;
int main()
{
     {
          demo D1, D2;
     }
return 0;
}
```

O/P

```
ID1 object created:
ID2 object created:
ID2 object destroyed:
ID1 object destroyed:
```

Initialisation List

C++ supports another method of initializing the class objects. This method is known as initialization list in the constructor function.

Syntax:

```
constructor(argument-list) : initialization section {
      // body of constructor
}
```

The part immediately following the colon is known as the initialization section. We use this section to initialize the data members of the class. The initialization section contains a list of initializations separated by commas. This list is known as initialization list.

```
#include <iostream>
using namespace std;

class complex
{
        int imag;
        int real;
        public:
                complex(int x, int y) : real(x), imag(y)
                {       }
                void display() {
                        cout << real << "+i" << imag << endl;
                }
};

int main()
{
        complex c(10,20);
        c.display();
        return 0;
}
```

O/P

```
10+i20
```

When using initialization list to initialize objects, the members are initialized in the order in which they are declared rather than in the order in which they are placed in the Initialization list. Thus, using the data member declared later cannot be used to initialize the members coming earlier. For example in above example,

```
complex(int x):real(imag),imag(x) {     }     // valid
complex(int x):real(x),imag(real) {     }     //invalid
```

Note: In initialization list, members are given value before the constructor even start to execute.

Initialization list is generally used to initialize constant and reference data members. Let us consider the following example

```
class ABC {
      const int x = 10;
      …
};
```

And

```
class ABC {
      int x;
      int &y;
      public:
            ABC(int a, int b) {
                  x = a;
                  y = x;
            }
};
```

Both the above classes produce errors. Constants and Reference data members must be initialized when they are declared (their memory is allocated). The solution to this problem is initialization list. They can be initialized in initialization list.

4.7 Object as Function Argument and Return type

Objects can also be passed as function argument or be returned by a function like normal data type.

```
#include <iostream>
using namespace std;

class complex
{
        public:
        int r, i;
        void input() {
                cout << "Enter real part: ";
                cin >> r;
                cout << "Enter imaginary part: ";
                cin >> i;
        }
        void display() {
                cout << r << "+i" << i << endl;
        }
};

complex add(complex x, complex y) {
        complex t;
        t.r = x.r + y.r;
        t.i = x.i + y.i;
        return t;
}

int main()
{
        complex c1,c2,c3;
        c1.input();
        c2.input();
        c3 = add(c1,c2);
        cout << "The sum is: ";
        c3.display();
        return 0;
}
```

O/P

```
Enter real part: 1
Enter imaginary part: 2
Enter real part: 3
Enter imaginary part: 4
The sum is: 4+i6
```

4.8 Array of Objects

Array of objects is created and used in the same way as normal variables. The data members and member functions in private and protected access specifier cannot be accessed by the objects(members of the array).

An array of objects is created using the following syntax:

class-name array-name[array-size];

The members can be accessed using the following syntax:

array-name[index].datamember; //for data member

array-name[index].function(); //for member functions

4.9 Pointers to objects and Member Access

Similar to pointer of other data type, we can also create pointer type of object of class. This pointer holds address of an object of the class. The general form of declaring the pointer type of object is:

class-name *Pointer-name;

Similar to pointer type variable of structure, the pointer object to class uses the arrow operator(->) to access the members of the class. The general form is

Pointer-object->member

Or,

(*Pointer-object).member

4.10 Dynamic Memory Allocation for Objects and Object Array

Similar to DMA of other data-types, we can dynamically allocate memory for an object or an array of objects.

Syntax:

```
class-name *pointer-object;                 //pointer declared
pointer-object = new class-name;            //for single object
pointer-object = new class-name[size];      //for array of object
```

For deallocation of memory:

```
delete[]pointer-object;
```

4.11 this Pointer:

This operator is an operator which points to a current object. The object which invokes the member
function is called This operator. (->)

```
# include <iostream>
using namespace std;

class Complex
{
     int r;
public:
Complex (int r)
{
     this -> r=r;
}
void input ()
{
     cout << "\n Enter the value of r:";
     cin >> this -> r;
}
void display()
{
     cout <<"\n Value of r:" << r << endl;
}
Complex add( Complex C1, Complex C2)
{
     r= C1.r + C2.r;
     return (* this);
}
};

int main()
{
     Complex a1(1), a2(2), a3(0), a4(0);
     a1.input();
     a2.input();
     a4= a3.add(a1, a2);
     a3.display();
     a4.display();
return 0;
}
```

O/P

```
Enter the value of r:2

 Enter the value of r:3

 Value of r:5

 Value of r:5
```

4.12 static Data Member and static Function

Static data member:
A data member which is defined inside a class as a static is know as static data member. **It is shared by or common to all the object of that class.**

```
#include <iostream>
using namespace std;

class Counter
{
      static int c;

public:
      void display(){
            c++;
            cout << "The call to display function " << c << endl;
      }
};

int Counter::c=0;

main()
{
      Counter C1, C2, C3;
      C1.display();
      C2.display();
      C3.display();

return 0;
}
```

O/P

```
The call to display function 1
The call to display function 2
The call to display function 3
```

```
#include <iostream>
using namespace std;

class Counter
{
      static int c;
      int a;

public:
      void input(){
            cout << "Enter value of a: ";
            cin >> a;
            cout << "Enter value of c: ";
            cin >> c;
      }
      void display(){
            cout << "\nThe value of a is = " << a;
            cout << "\nThe value of c is = " << c;
      }
};

int Counter::c=0;
```

```
main()
{
      Counter C1, C2, C3;
      C1.input();
      C2.input();
      C3.input();
      cout << "\nThe value of variable a & c";
      C1.display();
      C2.display();
      C3.display();

return 0;
}
```

O/P

```
Enter value of a: 2
Enter value of c: 4
Enter value of a: 8
Enter value of c: 16
Enter value of a: 32
Enter value of c: 64

The value of variable a & c
The value of a is = 2
The value of c is = 64
The value of a is = 8
The value of c is = 64
The value of a is = 32
The value of c is = 64
```

```
/*
Program to store the records of 5 students, where all students are of
"2068" batch from "KEC".
*/

#include <iostream>

using namespace std;

class student{

   private:
      string name;               //or char name[15]
      int rollno;
      static int batch;
      static string college;     //or static char * college
   public:
      void input(){
          cout << "Enter name and roll no:";
          cin >> name >> rollno;
      }
      void Display(){
          cout << "Name    :" << name << endl;
          cout << "College :" << college << endl;
```

```
          cout << "Batch   :" << batch << endl;
          cout << "Roll No  :" << rollno <<endl;
     }
};

int student::batch = 2068;
string student::college = "KEC";     //or  char  *  student::college =
“KEC”;

main()
{
   student s[5];
   for (int i = 0; i < 5; ++i)
     s[i].input();
   for (int i = 0; i < 5; ++i)
     s[i].Display();

return 0;
}
```

Static member function:

A member function which is defined as a static can access only static data member and it can be invoked or called using name or object of that class.

```
#include<iostream>
using namespace std;

class Counter
{
int a;
static int c;
public:
     void input(){
          cout<<"\n Enter the value of a:";
          cin>>a;
          cout<<"\n Enter the value of c:";
          cin>>c;
     }
     static void display(){
          //cout<<"\nThe value of a is:"<<a;
          cout<<"\nThe value of c is:"<<c<<endl;
     }
};

int Counter::c=0;

int main()
{
     Counter C1,C2,C3;
     C1.input();
     C2.input();
     C3.input();
     cout<<"\nThe value of variable c is:";
     C1.display();
     Counter::display();
```

```
     C3.display();

return 0;
}
```

O/P

```
 Enter the value of a:2

 Enter the value of c:3

 Enter the value of a:4

 Enter the value of c:5

 Enter the value of a:5

 Enter the value of c:54

The value of variable c is:
The value of c is:54

The value of c is:54

The value of c is:54
```

4.13 Constant Member Functions and Constant Objects

Constant member function is a function that can't modify the data member of class but can use that data member. A member function is declared as constant member function using keyword const. For example.

void large(int, int) const;
float interest(void) const;

The qualifier const appears both in member function declaration and definitions. Once a member function declared as const, if cannot alter the data values of the class. The compiler will generate an error message if such functions try to alter the data values.

```
#include<iostream>
#include<cmath>
using namespace std;
class Coordinate
{
     int x;
     int y;
public:
     void input()
     {
          cout << "Enter X and Y coordinates : ";
          cin >> x >> y;
```

```
     }
     void display() const;
};

void Coordinate::display() const
{
     float sum;
     cout << "\nX Co-ordinate : " << x;
     cout << "\nY Co-ordinate : " << y;
     sum = sqrt(pow(x,2) + pow(y,2));
     cout << "\nMagnitude : " << sum;

     // x++ ; Error : Cannot change the value of variable inside the
constant member function
}

int main()
{
     Coordinate C1;
     C1.input();
     C1.display();
return 0;
}
```

```
O/P
Enter X and Y coor
4

X Co-ordinate : 3
Y Co-ordinate : 4
Magnitude : 5
```

Note: A constant member function of a class can only invoke other constant member functions of the same class i.e. In above program, we cannot call input() from display().

Constant Objects:

Just like constant variables, a constant object is an object of a class that cannot modified. A constant object can call only const member functions because they are the only ones that guarantees not to modify the object.

```
#include <iostream>
using namespace std;

class Coordinate
{
     int x;
     int y;
```

```
public:
     Coordinate(int a, int b) {
          x = a;
          y = b;
     }
     void get_coordinate() {
          cout << "Enter the x-coordinate:";
          cin >> x;
          cout << "Enter the y-coordinate:";
          cin >> y;
     }
     void show_coordinate() const {       //constant member function
          cout << "\nx-coordinate:" << x;
          cout << "\ny-coordinate:" << y;
     }
     void display() {
          cout << "\nx-coordinate:" << x;
          cout << "\ny-coordinate:" << y;
     }
};

int main()
{
     Coordinate c1(0,0);
     c1.get_coordinate();
     c1.show_coordinate();
     c1.display();

     const Coordinate c2(10,20);          //constant member function
     //c2.get_coordinate()
    //Invalid:get_coordinate() modifies the member of Coordinate
class
     c2.show_coordinate();
    //Valid: show_coordinate() is constant member function
     //c2.display();
    //Invalid: constant object can only invoke constant member
function

return 0;
}
```

In the above program, we have defined two member functions which use the data members but do not change it, i.e., show_coordinate() const and display(). The show_coordinate() is defined as const member function whereas display() is defined as normal member function. get_coordinate() is another function which modifies the data members of the class. An ordinary object c1 can invoke all three member functions. A const object c2 with initial value (10,20) can invoke only show_coordinate() as it is a const member function.

Mutable Keyword

As discussed above, const objects can only invoke const member functions and this const member functions cannot change the data member defined in a class. However, a situation may arise when we want to create const object but we would like to modify a particular data item only. In such situation, we can make it possible by defining the data item as mutable.

```
#include <iostream>

using namespace std;

class Student
{
      char *name;
      mutable char *address;
public:
      Student(char *n, char *ad) {
            name = n;

            address = ad;
      }
      void change_name(char *new_name) {
            name = new_name;
      }
      void change_address(char *new_address) const {
            address = new_address;
      }
      void display() const {
            cout << "Name:" << name << endl;
            cout << "Address:" << address << endl;
      }
};

int main()
{
      const Student s1("ABC","PlanetEarth");
      //s1.change_name("XYZ");
      s1.change_address("Nepal");
      s1.display();
return 0;
}
```

In above program, we have two data members, address defined as mutable and name as normal variable. In main(), we create a const object s1 with inital value to name and address. When we try to invoke change_name(), it encounters an error as it is not const member function so constant object s1 cannot invoke it. When s1 invokes change_address() is const member function and it changes the data member address value. All this was possible due to address member being defined as mutable.

Note: the const function changing the value of mutable data cannot have statements that try to change the value of other ordinary data.

4.14 Friend Function and Friend Classes

A friend function is a function that is not a member of a class but has the access to the class's private and protected members. Friend functions are not considered class members; they are normal external functions that are given special access privileges.
Friends are not in the class's scope, and they are not called using the member-selection operators(. or ->).
Syntax:

```
class class-name
{
      …
      friend return-type function(arguments);
};
return-type function(arguments) {
      //body of function
}
```

Friend declaration can be placed anywhere in the class and the access specifier does not matter.

Friend class is a class whose member functions can access another class's private and protected members.

```
/*
WAP to make global function which returns the average of 10 number list stored at a member
class
*/

#include <iostream>
using namespace std;
const unsigned int SIZE = 10;

class Num
{
        int n[SIZE];
public:
        void input(){
                cout << "Enter the elements:";
                for(int i=0; i<SIZE; i++)
                        cin >> n[i];
        }
        friend float average(Num);
};
float average(Num n1){
        float sum=0.0;
        for(int i=0; i<SIZE; i++)
```

```
                sum += n1.n[i];
        return sum/SIZE;
}

main()
{
        Num n1;
        float avg;

        n1.input();
        avg = average(n1);

        cout << "average = " << avg << endl;

return 0;
}
```

O/P

```
Enter the elements:1
2
3
4
5
6
7
8
9
0
average = 4.5
```

Friend as bridge function

Friend functions can be used as a bridge between two classes. It can be used to access the private and protected members of two or more classes. This can be accomplished by declaring the same function as the friend of the classes it is required to link.

```
/*
WAP to swap the private data of two different class
*/

#include <iostream>
using namespace std;

class ObjB;

class ObjA
{
      int x;
public:
      void input(){
            cout << "Enter value of x: ";
            cin >> x;
      }
      void display(){
            cout << "Value of x is " << x << endl;
```

```
      }
      friend void swap(ObjA&, ObjB&);
};

class ObjB
{
      int a;
public:
      void input(){
            cout << "Enter value of a: ";
            cin >> a;
      }
      void display(){
            cout << "Value of a is " << a << endl;
      }
      friend void swap(ObjA&, ObjB&);
};

void swap(ObjA& a, ObjB& b){
      int tmp = a.x;
      a.x = b.a;
      b.a = tmp;
}

main()
{
      ObjA a1;
      ObjB b1;

      a1.input();
      b1.input();

      swap(a1, b1);
      a1.display();
      b1.display();

return 0;
}
```

O/P

```
Enter value of x: 1
Enter value of a: 2
Value of x is 2
Value of a is 1
```

Friend function of a class as a friend of another

The member functions of a class can be friend functions of another class. In such case their declaration as friend in another class use their qualified name (full name).

```
/*
WAP to make a mult() function of class A as a friend of B and display
the proper output
```

```
*/

#include <iostream>

using namespace std;

class B;

class A
{
      int a;
   public:
      void input(){
          cout << "Enter the value of a :";
          cin >> a ;
      }
      void mult(B t);
};

class B
{
      int b;
   public:
      void input(){
          cout << "Enter the value of b:";
          cin >> b;
      }
      friend void A::mult(B t);
};

void A::mult(B t){
      cout << a << "*" << t.b << "=" << a*t.b << endl;
}

main()
{
   A a;
   B b;
   a.input();
   b.input();
   a.mult(b);
return 0;
}
```

O/P

```
Enter the value of a :1
Enter the value of b:2
1*2=2
```

Friend Class

There may be a situation when all the member functions of a class have to be declared as friend of another class. In such situations, instead of making the functions friend separately, we can make the whole class a friend of another class.

```
#include <iostream>
using namespace std;
class ABC;
class XYZ
{
        int x;
        public:
                friend class ABC;
};
class ABC
{
        int a;
        public:
                void getdata(XYZ &o1) {
                        cout << "Enter the value of a:";
                        cin >> a;
                        cout << "Enter the value of x:";
                        cin >> o1.x;
                }
                void sum(XYZ &o1) {
                        cout << "The sum is:" << a+o1.x << endl;
                }
                void product(XYZ &o1) {
                        cout << "The product is:" << a * o1.x << endl;
                }
};

int main()
{
        ABC obj1;
        XYZ obj2;
        obj1.getdata(obj2);
        obj1.sum(obj2);
        obj1.product(obj2);
return 0;
}
```

O/P

```
Enter the value of a:2
Enter the value of x:1
The sum is:3
The product is:2
```

Chapter 5

Operator Overloading

5.1 Overloadable Operators

C++ permits us to make user defined data type behave like the built-in types by allowing the overloading of operators. Defining the meaning of various operators for user defined data types is known as operator overloading. In C++, all operators can be overloaded except the following:

- sizeof sizeof operator
- . Member operator
- .* Pointer to member operator
- :: Scope resolution operator
- ?: Conditional Operator

5.2 Syntax of Operator Overloading:

Syntax:

```
return type classname::operator op(arglist)
{
      //function body       task defined
}
```

5.3 Rules of Operator Overloading

a. Only existing operators can be overloaded. New operators cannot be created.
b. The overloaded operator must have at least one operand that is that is of user-defined type.
c. Overloaded operators follow the syntax rules of the original operators. They cannot be overridden.
d. The operator that cannot be overloaded is:
   sizeof sizeof operator
   . Member operator
   .* Pointer to member operator
   :: Scope resolution operator
   ?: Conditional Operator
   The operator that cannot be overloaded using Friend Function is :
   = Assignment operator
   ( ) Function call operator
   [ ] Subscripting operator
   -> Class member access operator
e. In binary operator overloading using friend function, we can specify *left-hand operand* as class object or *built-in type* same as in *right-hand operand.* But in

binary operator overloading using member function, *left-hand operand* must be the *object of class which contains operator function* with *right-hand operand* of *class or built in type* as an argument to it.

5.4 Unary Operator Overloading:

Unary operator overloading through member function:
Syntax:

```
return-type class-name::operator OP(){
     // body of operator function
}
```

```
# include <iostream>
using namespace std;

class Stud
{
     int a;
public:
void input(){
     cout<<"\n Enter the value of a:";
     cin>>a;
}
void display(){
     cout<<"\n The value of a is:" << a << endl;
}
void operator -();
};
void Stud::operator -()
{
     a=-a;
}

int main()
{
     Stud S1,S2;
     S1.input();
     S1.display();
     -S1; // S1-operator -();
     S1.display();
return 0;
}
```

O/P

```
 Enter the value of a:1

 The value of a is:1

 The value of a is:-1
```

Note: It is not necessary that - operator always does -. Its meaning is being changing. But also we don't have to define new operator.

```
# include <iostream>
using namespace std;
class Stud
{
     int a;
public:
void input()
{
     cout <<"\n Enter the value of a:";
     cin >> a;
}
void display()
{
     cout<<"\n The value of a is:" << a << endl;
}
Stud operator-();
};

Stud Stud::operator-()
{
     Stud temp;
     temp.a= -a;
     return temp;
}

int main()
{
     Stud S1, S2;
     S1.input();
     S1.display();
     S2=-S1;    // S2=S1.Operator -();
     S1.display();
     S2.display();
return 0;
}
```

O/P

```
Enter the value of a:3

 The value of a is:3

 The value of a is:3

 The value of a is:-3
```

Prefix operator overloading using friend function:

```
#include <iostream>
using namespace std;

class Complex
{
     int r, m;
public:
     void input(){
          cout << "Enter real and imaginary part " << endl;
          cin >> r >> m;
     }
     void display(){
          cout << r << "+" << m << "i" << endl;
     }
     friend Complex operator -- (Complex&);
};

Complex operator -- (Complex& c){
     Complex temp;
     temp.r = c.r-1;
     temp.m = c.m-1;
     return temp;
}

main()
{
     Complex c1, c2;
     c1.input();
     c2 = --c1;
     c2.display();

return 0;
}
```

O/P

```
Enter real and imaginary part
4
5
3+4i
```

5.5 Binary Operator Overloading

Binary operator overloading using member function
Syntax:

```
return_type classname::operator OP(argument){
     statements;
}
```

```
#include <iostream>
```

```
using namespace std;

class Complex
{
      int r, m;
public:
      void input(){
            cout << "Enter real and imaginary part " << endl;
            cin >> r >> m;
      }
      void display(){
            cout << r << "+" << m << "i" << endl;
      }
      Complex operator + (int);
};

Complex Complex::operator + (int a){
      Complex temp;
      temp.r = r+a;
      temp.m = m+a;
      return temp;
}

main()
{
      Complex c1, c2;
      c1.input();
      c2 = c1 + 2;    // c1.operator+(2)
      c2.display();

return 0;
}
```

O/P

```
Enter real and imaginary part
3
4
5+6i
```

Binary operator using friend function

In binary operator using friend function we can specify the order of left and right operand. The first argument implies left operand and second argument implies right operand.

```
#include <iostream>
using namespace std;

class Complex
{
      int r, m;
public:
      void input(){
            cout << "Enter real and imaginary part " << endl;
            cin >> r >> m;
      }
      void display(){
            cout << r << "+" << m << "i" << endl;
      }
      friend Complex operator + (Complex, int);
};

Complex operator + (Complex c, int a){
      Complex temp;
      temp.r = c.r+a;
      temp.m = c.m+a;
      return temp;
}

main()
{
      Complex c1, c2;
      c1.input();
      c2 = c1 + 2;    // operator+(2)
      c2.display();

return 0;
}
```

O/P

```
Enter real and imaginary part
4
5
6+7i
```

Note:
In binary operator overloading using friend function we can specify left hand operand as class object or built in data type. Same as in right hand operand. But in binary operator overloading using member function the left and right hand operand must be the object of class which contains operator function with right hand operand of class or built in data type as argument to it.

```
/*
WAP to compare the magnitude of a complex number by overloading <, >
and ==
*/

#include <iostream>
#include <math.h>
using namespace std;
enum Bool { FALSE, TRUE };

class Complex
{
     int r;
     int i;
public:
     void input(){
          cout << "Enter real and imaginary part" << endl;
          cin >> r >> i;
     }
     void display(){
          cout << r << "+" << i << "i" << endl;
     }
     Bool operator < (Complex C){
          float m1 = sqrt(r*r + i*i);
          float m2 = sqrt(C.r*C.r + C.i*C.i);
          return (m1 < m2 ? TRUE : FALSE);
     }
     Bool operator > (Complex C){
          float m1 = sqrt(r*r + i*i);
          float m2 = sqrt(C.r*C.r + C.i*C.i);
          return (m1 > m2 ? TRUE : FALSE);
     }
     Bool operator == (Complex C){
          float m1 = sqrt(r*r + i*i);
          float m2 = sqrt(C.r*C.r + C.i*C.i);
          return (m1 ==  m2 ? TRUE : FALSE);
     }
};

main()
{
     Complex c1, c2;
     c1.input();
     c2.input();

     if(c1<c2)
          cout  <<  "1st  complex  number  is  less  than  2nd  complex
number" << endl;
     else if(c1>c2)
          cout  <<  "1st complex number is greater than 2nd complex
number" << endl;
     else if(c1==c2)
          cout  <<  "1st  complex  number  is  equal  to  2nd  complex
number" << endl;
     c1.display();
```

```
     c2.display();

return 0;
}
```

O/P

```
Enter real and imaginary part
4
5
Enter real and imaginary part
6
7
1st complex number is less than 2nd complex number
4+5i
6+7i
```

5.6 Operator Overloading with Member and Non-Member function

An Operator may be overloaded with a member function and non-member function as shown in examples of binary operator overloading.

5.7 Data Conversion: Basic-User Defined and User Defined-User Defined

Conversion between user defined type and built in type cannot be performed implicitly by the compiler but c++ allows type conversion between them from after the rules for the type conversion have been defined. Three types of situations arise in the data conversion between incompatible types:

- Conversion from basic type to class type
- Conversion from class type to basic type
- Conversion from one class type to another class type

Basic to Class type Conversion

A basic to class conversion can be performed through a constructor with argument of basic type. The constructor must have only one argument.

Syntax:

```
constructor (basic type) {
     // conversion steps;
}
```

```
#include<iostream>
using namespace std;

class Complex
{
     int r;
     int m;
public:
     Complex(){
          r=0;
          m=0;
```

```
      }
      Complex(int a)       // Constructor for Basic-Class conversion
      {
            r=a;
            m=0;
      }
      void display(){
            cout<<"\nThe real is:"<<r;
            cout<<"\nThe imag is:"<<m;
      }
};
int main()
{
      Complex c1;
      c1=9;
      c1.display();

return 0;
}
```

O/P

```
The real is:9
```

Class to Basic type conversion

Class to Basic type conversion can be achieved using operator function. C++ allows us to define an overloaded casting operator that could be used to convert a class type data to a basic type.

Syntax:

```
operator typename()  {
      // conversion statements
}
```

This function is defined inside a class. The object of this class is converted by the statements in the body of function and returns a variable of type name.

```
#include<iostream>
#include<cmath>
using namespace std;

class Complex
{
      int r;
      int m;
public:
      void input()
      {
            cout<<"\nEnter the value of r n m";
            cin>>r>>m;
      }
      void display()
      {
            cout<<"\nReal:"<<r;
            cout<<"\nImag:"<<m;
      }
```

```
      operator float()
      {
            float m1;
            m1=sqrt(r*r+m*m);
            return m1;
}
};
int main()
{
      Complex c1;
      c1.input();
      float magnitude=c1;
      cout<<"\nThe magnitude is :"<<magnitude;

return 0;
}
```

O/P

```
Enter the value of r n m3

4
The magnitude is :5
```

Class to Class type Conversion

Class to class conversion requires identification of source class and destination class. The right-hand operand of the assignment operator acts as destination class operand and left-hand sided operand is source class operand.

For example

obj1 = obj2;

If obj1 is an object of class A and obj2 is an object of class B, then class A is the destination class and class B is the source class.

Class to class Conversion can be performed in two ways:

- using constructor
- using operator function

When using constructor, the constructor is defined inside the destination class and the object of source class type is the argument of the constructor.

```
#include<iostream>
using namespace std;

class Grade
{
      float d;
public:
      void input()    {
            cout<<"\nEnter Grade:";
            cin>>d;
      }
      float getGrade(){
            return d;
      }
```

```
};

class Radian
{
      float r;
public:
      Radian(){
            r=0.0;
      }
      Radian(Grade);
      void display(){
            cout<<"\nThe radian is:"<<r;
      }
};

Radian::Radian(Grade G){
      r=(G.getGrade()*3.14)/200;
}

int main()
{
      Grade G1;
      Radian R1;
      G1.input();
      R1=G1;
      cout << "In radian:";
      R1.display();

return 0;
}
```

O/P

```
Enter Grade:50
In radian:
The radian is:0.785
```

When using operator function, the function is defined inside source class with a return of destination class object.

```
#include<iostream>
using namespace std;
class Radian;

class Grade
{
      float d;
public:
      void input()    {
            cout<<"\nEnter Grade:";
            cin>>d;
      }
      Grade(){
            d=0.0;
```

```
      }
      operator Radian();

};

class Radian
{
      float r;
public:
      Radian(){
            r=0.0;

      }
      void display(){
            cout<<"\nThe radian is:"<<r;
      }

      void setRadian(float r){
            this->r=r;
      }
};

Grade:: operator Radian(){
      Radian R;
      R.setRadian(d*3.14/200);
      return R;
}

int main()
{
      Grade G1;
      cout<<"\nGrade to Radian ";
      G1.input();

      Radian R1;
      R1=G1;
      R1.display();

return 0;
}
```

O/P

```
Grade to Radian
Enter Grade:100

The radian is:1.57
```

Chapter 6

Inheritance

Inheritance is the capability of one class to inherit properties from another class. It also defines a mechanism of driving a new class from an old one.
The most significant advantage of inheritance is code reusability. Once a base class is written and debugged, it can be used in various situations without having to redefine or rewrite it. Already tested and saved previous code can be reused. Reusing existing code saves time, money and efforts and increases a program's reliability. Inheritance allows addition of new properties to derived class and redefine an inherited class member functions without redefining the old class.

Need for inheritance:

1. Capability of expressing the inheritance relationship that ensures closeness with the real world models.
2. Allows reusability of code,i.e., addition of additional features to an existing class without modifying it.
3. Transitive nature of inheritance allows derived class to automatically inherit properties of base class

6.1 Base and Derived Class

A *Derived class* is the class which inherits the property of another class. The class from which the properties are inherited is known as *base class*. Derived class is also known as *subclass* or *child class*. Base class is also known as *parent class*.

6.2 Protected Access Specifier

Private access specifiers cannot be inherited and public members are inheritable but directly accessible through objects. Protected access specifier is used when the data members or member functions are required to be inheritable but inaccessible through objects. Like private members, they can be accessed only through functions.

6.3 Derived Class Declaration

Syntax:

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

The classname is followed by colon(:), visibility mode and base class name respectively. Visibility mode may be *private, public* or *protected*.

Note: By default the visibility mode is private.

For inheritance from multiple class, commas are used as shown below:

```
     class derived-name : visibility base1-name, visibility base2-name
{
          // members
     };
```

Visibility mode

Visibility mode basically controls the access-specifier to which inherited members of base-class are placed in derived class.

Visibility Mode	Private Member of Base	Public Members of Base	Protected Members of Base
public	Not Inherited	public	protected
private	Not Inherited	private	private
protected	Not Inherited	protected	protected

Private Members of Base class is never inherited. In public visibility mode, the public members are inherited as public members of derived class and protected members are inherited as protected members,i.e., members are inherited in the same access specifier. Private visibility mode inherits both public and protected members in private access specifier. Protected visibility mode inherits both public and protected members into protected section.

```
/* single inheritance */

#include<iostream>
using namespace std;

class Base{
     protected:
          int a;
     public:
          void inputBase(){
               cout<<"enter the value of a:";
               cin>>a;
          }
          void displayBase(){
               cout<<"the value of a is:"<< a<< endl;
          }
};

class Derived: public Base{
     int b;
     public:
     void inputDerived(){
```

```
          inputBase();
           cout<<"enter value of b:";
           cin>>b;
     }
     void displayDerived(){
           displayBase();
           cout<<"the value of b is:"<< b << endl;
           cout <<"the sum ="<< a+b << endl;
     }
};

int main(){
     Derived D1;
     D1.inputDerived();
     cout<<"Displaying the taken value:"<< endl;
     D1.displayBase();
     D1.displayDerived();
return 0;
}
```

```
output:

enter value of b:5
enter the value of a:
3
Displaying the taken value:
the value of a is:3
the value of a is:3
the value of b is:5
the sum =8
```

In the above example, 'Derived' class is inherited from 'Base' class with public visibility mode. 'a' is a member of 'Base' and 'b' is a member of 'Derived'. Input and display functions are defined for both classes. Input and display of base class is called inside the input and display of derived class. Thus, invoking input and display functions of input and display of derived class invokes input and display of base class as well. As the visibility mode is public, input and display can be called in main() as well.

6.4 Member function Overriding

Redeclaration of member functions in derived class which is already defined inside visible sections (private and public) of base is known as function overriding.Whe a derived class defines a function which is already defined inside a base class in inheritance then this is known as function overriding.

Ambiguity in multiple inheritance
When a derived class is inherited from multiple base classes,i.e., two or more base classes, and

the base classes have the same functions, ambiguity arises. This ambiguity can be removed using scope resolution operator (::).

```
/* ambiguity multiple inheritance */

#include<iostream>
using namespace std;

class Base1{
protected:
     int x;
public:
     void input(){
          cout<<"enter value to x of Base1:" << endl;
          cin >> x;
     }
     void display(){
          cout <<"the x of Base 1 is:" << x << endl;
     }
};
class Base2{
protected:
     int x;
public:
     void input(){
          cout<<"enter value of x for Base 2:" << endl;
          cin >> x;
     }
     void display(){
          cout<<"the x of Base2 is: " << x << endl;
     }
};
class Derived:public Base1, public Base2{
//protected:
     int c;
public:
     void inputDerived(){
          cout<<"enter the value of c:" << endl;
          cin >> c;
          Base1::input();
          Base2::input();
     }
     void displayDerived(){
          Base1::display();
          Base2::display();
          cout<<"the value of c is:" << c << endl;
          cout<<"the sum is:" << Base1::x + c + Base2::x << endl;
     }
};
```

```
int main(){
      Derived D1;
      D1.inputDerived();
      cout<<"displaying the taken value:" << endl;
      D1.displayDerived();
return 0;
}
```

```
output:
```

```
enter the value of c:
5
enter value to x of Base1:
4
enter value of x for Base 2:
3
displaying the taken value:
the x of Base 1 is:4
the x of Base2 is: 3
the value of c is:5
the sum is:12
```

In the above example, 'Derived' class is inherited from 'Base1' and 'Base2'. Both 'Base1' and 'Base2' have same functions input() and display(). When inherited into 'Derived', ambiguity arises as it is unclear whether the function of 'Base1' or 'Base2' is to be called. So, scope resolution is used to resolve this ambiguity.

Example of function overriding

```
/* function overriding*/

#include<iostream>
using namespace std;

class Base1{
protected:
      int a;
public:
      void input(){
            cout<<"enter value to a of Base1:" <<endl;
            cin >> a;
      }
      void display(){
            cout<<"the a of Base1 is:" << a << endl;
      }
};
class Base2{
protected:
```

```
      int a;
public:
      void input(){
            cout<<"enter value to a of Base2" << endl;
            cin >> a;
      }
      void display(){
            cout<<"the a of Base2 is:" << a << endl;
      }
};

class Derived: public Base1, public Base2{
      int c;
public:
      void input(){
            cout<<"enter the value of c:" << endl;
            cin >> c;
      }
      void display(){
            Base1::display();
            Base2::display();
            cout<<"the value of c is:" << c << endl;
            cout<<"the sum is:" << Base1::a + c + Base2::a << endl;
      }
};

int main(){
      Derived D1;           // D1.Derrived::input();
      D1.Base1::input();
      D1.Base2::input();
      D1.input();
      cout<<"displaying the taken value:" << endl;

      D1.display();
return 0;
}
```

output:

```
enter value to a of Base1:
3
enter value to a of Base2:
4
enter the value of c:
5
displaying the taken value:
the a of Base1 is:3
the a of Base2 is:4
```

```
the value of c is:5
the sum is:12
```

In the above example, 'Derived' class is inherited from 'Base1' and 'Base2'. All the classes have input() and display() function. This is function overriding as derived class has the same functions as base class. In main(), when input() and display() are invoked, the functions of derived class are called.

Note: The overridden functions of base class can be invoked in two ways:

1. From the member function of derived class
2. From the object of derived class by using scope resolution operator.(eg: obj.base::display();)

6.4 Forms of Inheritance: Single, Multiple, Multilevel, Hierarchical, Hybrid, Multipath

Single Inheritance
This is the simplest form of inheritance. One derived class is inherited from one base class.

The example of single inheritance is given above (as an example of inheritance).

Multiple Inheritance
When a derived class is inherited from two or more base classes it is known as multiple inheritance.

```
/* multiple inheritance */

#include<iostream>
using namespace std;

class Base1{
protected:
     int a;
public:
     void inputBase1(){
          cout<<" enter the value of a:" << endl;
          cin >> a;
     }
     void displayBase1(){
          cout<<"the value of a is:"<< a << endl;
     }
};
```

```
class Base2{
protected:
      int b;
public:
      void inputBase2(){
            cout<<"enter the value of b:" << endl;
            cin >> b;
      }
      void displayBase2(){
            cout<<"the value of b is:" << b << endl;
      }
};

class Derived:public Base1, public Base2 {
      int c;
public:
      void inputDerived(){
            cout<<"enter the value of c:" << endl;
            cin >> c;

            inputBase1();
            inputBase2();
      }
      void displayDerived(){
            displayBase1();
            displayBase2();
            cout<<"the value of c is:" << c << endl;
            cout<<"the sum is:" << a+b+c << endl;
      }
};

int main(){
      Derived D1;
      D1.inputDerived();
            cout<<"displaying the taken value: "<< endl;
            D1.displayDerived();
return 0;
}
```

output:

```
enter the value of c:
5
enter the value of a:
4
enter the value of b:
3
displaying the taken value:
```

```
the value of a is:4
the value of b is:3
the value of c is:5
the sum is:12
```

```
/* ambiguity multiple inheritance */


#include<iostream>
using namespace std;

class Base1{
protected:
     int x;
public:
     void input(){
          cout<<"enter value to x of Base1:" << endl;
          cin >> x;
     }
     void display(){
          cout <<"the x of Base 1 is:" << x << endl;
     }
};
class Base2{
protected:
     int x;
public:
     void input(){
          cout<<"enter value of x for Base 2:" << endl;
          cin >> x;
     }
     void display(){
          cout<<"the x of Base2 is: " << x << endl;
     }
};
class Derived:public Base1, public Base2{
     int c;
public:
     void inputDerived(){
          cout<<"enter the value of c:" << endl;
          cin >> c;
          Base1::input();
          Base2::input();
     }
     void displayDerived(){
          Base1::display();
```

```
            Base2::display();
            cout<<"the value of c is:" << c << endl;
            cout<<" the sum is:" << Base1::x + c + Base2::x << endl;
      }
};

int main(){
      Derived D1;
      D1.inputDerived();
      cout<<"displaying the taken value:" << endl;
      D1.displayDerived();
return 0;
}
```

output:

```
enter the value of c:
3
enter value to x of Base1:
5
enter value of x for Base 2:
6
displaying the taken value:
the x of Base 1 is:5
the x of Base2 is: 6
the value of c is:3

the sum is:14
```

Multilevel Inheritance

When a derived class acts as a base class for another class, it is known as multilevel inheritance.

```
/*Multilevel Inheritance */

#include <iostream>
using namespace std;

class A {
protected:
        int a;
public:
        void inputA() {
                cout << "Enter the value of a:";
                cin >> a;
        }
        void displayA() {
```

```
                cout << "a = " << a << endl;
        }
};

class B: public A{
protected:
        int b;
public:
        void inputB() {
                inputA();
                cout << "Enter the value of b:";
                cin >> b;
        }
        void displayB() {
                displayA();
                cout << "b = " << b << endl;
        }
};

class C: public B{
        int c;
public:

        void inputC() {
                inputB();
                cout << "Enter the value of c:";
                cin >> c;
        }
        void displayC() {
                displayB();
                cout << "c = " << c << endl;
        }
};

int main()
{
        C obj;
        obj.inputC();
        obj.displayC();
return 0;
}
```

O/P

```
Enter the value of a:1
Enter the value of b:2
Enter the value of c:3
a = 1
b = 2
```

```
c = 3
```

Hierarchical Inheritance

When two or more classes from one base class, it is known as hierarchical inheritance.

Fig: Hierarchical Inheritance with two derived classes

```
#include <iostream>
using namespace std;

class A {
protected:
	int a;
public:
	void inputA() {
		cout << "Enter the value of a:";
		cin >> a;
	}
	void displayA() {
		cout << "a = " << a << endl;
	}
};

class B: public A{
protected:
	int b;
public:
	void inputB() {
		inputA();
		cout << "Enter the value of b:";
		cin >> b;
	}
	void displayB() {
		displayA();
		cout << "b = " << b << endl;
	}
};

class C: public A{
	int c;
public:
	void inputC() {
		inputA();
		cout << "Enter the value of c:";
		cin >> c;
	}
	void displayC() {
		displayA();
		cout << "c = " << c << endl;
```

```
        }
};

int main()
{
        cout << "Class B" << endl;
        B obj1;
        obj1.inputB();
        obj1.displayB();

        cout << "Class C" << endl;
        C obj2;
        obj2.inputC();
        obj2.displayC();
return 0;
}
```

O/P

```
Class B
Enter the value of a:1
Enter the value of b:2
a = 1
b = 2
Class C
Enter the value of a:3
Enter the value of c:4
a = 3
c = 4
```

Hybrid or Multipath Inheritance

Hybrid Inheritance is a combination of more than one of the above inheritance types(multiple, multilevel, hierarchical). It is also called multipath inheritance.

Examples:

6.6 Multipath Inheritance and Virtual Base Class

```
/* virtual base class or multiple path inheritance */

#include<iostream>
using namespace std;

class Aclass{
protected:
     int a;
public:
```

```
      void inputA(){
            cout<<"enter the value of a:" << endl;
            cin >> a;
      }
};
class Bclass: virtual public Aclass{
protected:
      int b;
public:
      void inputB(){
            cout<<"enter value of b:" << endl;
            cin >> b;
      }
};
class Cclass:public virtual Aclass{
protected:
      int c;
public:
      void inputC(){
            cout<<"enter value of c:" << endl;
            cin >> c;
      }
};
class Derived: public Bclass, public Cclass{
protected:
      int d;
public:
      void inputD(){
            cout<<"enter value of d:" << endl;
            cin >> d;
      }
      void display(){
            cout<<"the value of a is:" << a << endl;
            cout<<"the value of b is:" << b << endl;
            cout<<"the value of c is:" << c << endl;
            cout<<"the value of d is:" << d << endl;
            cout<<"the sum is:" << ((a+b+c)+d) << endl;
      }
};

int main(){
      Derived D1;
      D1.inputA();
      D1.inputB();
      D1.inputC();
      D1.inputD();
      D1.display();
return 0;
```

```
}
```

```
output:
```

```
enter the value of a:
1
enter value of b:
3
enter value of c:
4
enter value of d:
5
the value of a is:1
the value of b is:3
the value of c is:4
the value of d is:5
the sum is:13
```

6.7 Constructor Invocation in Single and Multiple Inhertances

Note:
If a base class has a parameter constructor then there must be a constructor inside a derived class which passes a value to a base class parameters constructor through initialization list.

```
/* parameterized constructor through initialization list*/

#include<iostream>
using namespace std;
class Base1{
     int a;
public:
     Base1(int a){
          cout<<"Base1 constructor:" << endl;
          this ->a = a;
     }
     ~Base1(){
          cout<<" Base1 destructor:" << a << endl;
     }
};

class Base2{
     int c;
public:
     Base2(int c){
          this ->c = c;
               cout<<"Base2 constructor:" << endl;
     }
```

```
     ~Base2(){
          cout<<"Base2 destructor:" << c << endl;
     }
};
class Derived: public Base2, public Base1{
     int d;
public:
     Derived(): Base1(10), Base2(20){
          d=0;
          cout<<"Derived constructor:" << endl;
     }
     Derived( int x, int y, int z): Base1(x), Base2(y){
          d=z;
          cout<<"Derived constructor:" << endl;
     }
     ~Derived(){
          cout<<"Derived destructor:" << d << endl;
     }
};
int main(){
     {
          Derived D1;
          Derived D2(1,2,3);
     }
return 0;
}
```

output:

```
Base2 constructor:
Base1 constructor:
Derived constructor:
Base2 constructor:
Base1 constructor:
Derived constructor:
Derived destructor:3
Base1 destructor:1
Base2 destructor:2
Derived destructor:0
Base1 destructor:10
Base2 destructor:20
```

```
/* constructor in simple inheritance */

#include<iostream>
```

```
using namespace std;
class Base1{
     int x;
public:
     Base1(){
          cout<<"Base constructor" << endl;
     }
     ~Base1(){
          cout<<"Base destructor" << endl;
     }
};
class Derived: public Base1{
     int c;
public:
     Derived(){
          cout<<"Derived constructor" << endl;
     }
     ~Derived(){
          cout<<"Derived destructor" << endl;
     }
};
int main(){
     {
          Derived D1;
     }
return 0;
}
```

output:

```
Base constructor
Derived constructor
Derived destructor
Base destructor
```

```
/* access private of base class in derived class*/


#include<iostream>
using namespace std;

class Base1{
     int x;
```

```
public:
      void inputB(){
            cout<<"enter value to x of Base1:" << endl;
            cin >> x;
      }
      void display(){
            cout<<"the x of Base1 is:" << x;
      }
      int returnx(){
            return x;
      }
};
class Derived: public Base1{
      int c;
public:
      void inputD(){
            cout<<"enter the value of c:" << endl;
            cin >> c;
      }
      void displayD(){
            display();
            cout<<" the value of c is:" << c << endl;
            cout<<" the sum is:" << returnx() + c << endl;
      }
};

int main(){
      Derived D1;
            cout<<"the num of data element inside derived class:" <<
sizeof(D1)/sizeof(int) << endl;
      D1.inputB();
      D1.inputD();
            cout<<"Displaying the taken value:" << endl;
      D1.displayD();
return 0;
}
```

output:

```
enter value to x of Base1:
5
enter the value of c:
3
Displaying the taken value:
the x of Base1 is:5 the value of c is:3
the sum is:8
```

6.8 Destructor Invocation in Single and Multiple Inheritances

Chapter 9

Templates

Templates are a relatively new addition to C++. They allow you to write generic classes and functions that work for several different data types. The result is that you can write generic code once and then use it over again for many different uses. C++ uses the templates to enable generic techniques. Generic programming is about generalizing software components so that they can be easily reused in a wide variety of situations. In C++, class and function templates are particulary effective mechanisms for generic programming because they make the generalization possible without sacrificing efficiency.

9.1 Function Template

In C++, function overloading allows to define multiple functions with the same names but with different arguments. In a situation, where we need to define functions with the same body but of different data type we can use function overloading. But while overloading a function, different data types will need different functions, and hence the function needs to be rewritten several times for each data type. This is time consuming and space consuming.
So C++ introduced the new concept of function template. Function template are special functions that can operate with generic types. It is the function which is written just for once, and have it worked for many different data type. The key innovation in function templates is to represent the data type used by the function not as a specific type such as int, float, etc., but as a name that can stand for any type.

Syntax:

```
template <class T>
returntype function-name(T type argument){
      // body
}
```

The template keyword signals the compiler that we're about to define a function template. The keyword class, with angle brackets, acts as the type. We can define our own data type using classes, so there's no distinction between types and classes. The variable following keyword class ('T' in above syntax) is called the template argument.

```
#include <iostream>

using namespace std;

template <class T1>
T1 sum(T1 a, T1 b){

}     return (a+b);
```

```
main()
{
      cout << sum(10, 20) << endl;
      cout << sum(11.5, 13.1) << endl;

return 0;
}
```

O/P

```
30
24.6
```

In the above example, sum() is a template function. It takes two arguments of T1 type and returns a T1 type data equal to the sum of two arguments. Defining a function as template doesn't generate code for different type, it is just a message to the compiler that the function can be called with different data type. Thus, in main(), when sum(10,20) is called, the template generates a function by substituting template argument (T1) with int. This is called instantiating the function template, and each instantiating version of the function is called template function. When sum(11.5,13.1) is invoked, instantiating of the template by float takes place and another template function of float type is created.

Note: The compiler generates only one version of function for each data type

```
/*
Write a function template to find largest element in an array. Write
main to call the template function with integer and float array
*/

#include <iostream>
using namespace std;

template <class T1>
void large(T1 a[], int s, T1 &large){
      T1 l = a[0];
      for(int i=1; i<s; i++)
            if(a[i] > l)
                  l = a [i];
      large = l;
}

main()
{
      int a1[] = {1, 23,4,5,5,353};

      float a2[] = {12.2, 32.3, 53.42, 3.1};
      int large1;
      float large2;

      large(a1, sizeof(a1)/sizeof(a1[0]), large1);
      large(a2, sizeof(a2)/sizeof(a2[0]), large2);

      cout << "the largest number of integer array = "
```

```
          << large1
          << endl;

     cout << "the largest number of float array = "
          << large2
          << endl;

return 0;
}
```

O/P

```
the largest number of integer array = 353
the largest number of float array = 53.42
```

The above example illustrates the function template which finds the largest element of an array of type that is passed into the function. The program displays largest values for two arrays, an integer array and a float array.

Function Template with Multiple Arguments

We can use more than one generic type in a funciton template. They are declared as a comma-separated list within the template specification.

Syntax:

```
template <class T1, class T2, ...>
returntype function-name(arguments of type T1, T2, T3...){
     // body
}
```

```
/*
Write a function template to find average element in an array. Write
main to call the template function with integer and float array
*/

#include <iostream>
using namespace std;

template <class T1, class T2>
void average(T1 a[], int s, T2 &avg){
     T2 sum=0;
     for(int i=0; i<s; i++)
          sum +=a [i];
     avg = sum/s;
}

main()
{
     int a1[] = {1, 23,4,5};
     float a2[] = {12.2, 32.3, 53.42, 3.1};
     float avg1;
     double avg2;

     average(a1, 4, avg1);
     average(a2, 4, avg2);
```

```
     cout << "the average of integer array = "
          << avg1
          << endl;

     cout << "the average of double array = "
          << avg2
          << endl;

return 0;
}
```

O/P

```
the average of integer array = 8.25
the average of double array = 25.255
```

In the above example, average() is a template function with multiple arguments. It uses an array of type T1 and avg as a reference of type T2. The first average() called in main() uses integer array and avg of float type and the second average() uses float array and avg of double type.

9.2 Overloading Function Template

Defining a template function along with other non template function or a template function of a same name is known as overloading of function template.

9.2.1 Overloading with Functions

```
/*
overloading function template with normal function
*/

#include <iostream>
using namespace std;

template <class T1>
T1 sum(T1 a, T1 b){
     cout << "template function" << endl;
     return (a+b);
}
int sum(int c, int d){     // priority
     cout << "normal function" << endl;
     return (c+d);
}

main()
{
     cout << sum(10, 20) << endl;
     cout << sum(11.3, 13.3) << endl;

return 0;
```

```
}
```

O/P

```
normal function
30
template function
24.6
```

In the above example, the sum() function is overloaded. Two functions of same name and same arguments are declared, one as a function template and other of type int. The function template generates functions for every data type but non-template functions take precedence over template function. Thus, sum(10,20) invokes normal function and sum(11.3,13.3) invokes template function.

9.2.2 Overloading with other Template

Template functions can also be overloaded with other template function. Compiler will generate functions for every data type but will choose the definition that is most specialized or match.

```
/*
overloading function template with other template
*/

#include <iostream>

using namespace std;

template <class T1>
T1 sum(T1 a, T1 b){
      cout << "template 1 function" << endl;
      return (a+b);
}

template <class T2>
T2 sum(T2 c, T2 d, T2 e){
      cout << "template 2 function" << endl;
      return (c+d+e);
}

main()
{
      cout << sum(10, 20, 5) << endl;
      cout << sum(11.3, 13.3) << endl;

return 0;
}
```

O/P

```
template 2 function
35
template 1 function
24.6
```

The above example has two template functions that are overloaded. The sum() function chosen

depends on number of arguments as the two functions are distinguished by the number of arguments. Thus sum(10,20,5) uses the second template function whereas sum(11.3,13.3) uses the first template function.

Note: Template function may be overloaded with non-template function or another template function of its name. The overloading resolution is accomplished as follows:

1. Call an ordinary(non-template) function of exact match.
2. Call a template function that can be created with exact match.
3. Try normal overloading resolution to ordinary functions and call the one that matches

An error is generated if no match can be generated.

9.3 Class Template

Class template is a template used to generate template classes. Class templates are used to construct classes having the same functionality for different data types. A class template provides specification for generating classes based on parameters.

Syntax:

```
template <class T>
class class-name
{
/*class member specification with anonymous type T wherever appropriate*/
};
```

Template class is an instance of a class template as a template function is an instance of a function template.

Syntax(for defining the object of template class):

classname<type> objectname(arglist);

Note: Class Templates are commonly used to implement containers like stacks, queues, etc.

9.3.1 Function Definition of Class Template

If the functions are defined outside the template class body, they should always be defined with the full template definition. They must be written like normal functions with scope resolution operator.

Syntax:

```
template <class T1>
return-type classname <T1>::function-name(arglist)
{
      // body of function
}
```

```
/*
write a class template to sort numbers
*/

#include <iostream>
using namespace std;

template <class T>
class Array
{
      T a[5];
public:
      void input(){
            cout << "enter five numbers " << endl;
            for(int i=0; i<5; i++)
                  cin >> a[i];
      }
      void display(){
            cout << "numbers in ascending order" << endl;
            for(int i=0; i<5; i++)
                  cout << a[i] << endl;
            cout << endl;
      }
      void sort();
};

template <class T>
void Array<T>::sort(){
      for(int i=5-1; i>=0; i--){
            for(int j=0; j<=i-1; j++){
                  if(a[j] > a[j+1]){
                        T tmp = a[j];
                        a[j] = a[j+1];
                        a[j+1] = tmp;
                  }
            }
      }
}

main()
{
      Array<int>a1;
      a1.input();
      a1.sort();
      a1.display();

      Array<float>a2;
      a2.input();
      a2.sort();
      a2.display();

return 0;
}
```

O/P

```
enter five numbers
```

```
23
53
03
93
932
numbers in ascending order
3
23
53
93
932

enter five numbers
93.3
23.23
93.23
5.32
.34
numbers in ascending order
0.34
5.32
23.23
93.23

93.3
```

Class Template with multiple arguments

Like in function template, we can use more than one generic type in class template. Each type are declared in Template specification, and they are separated by comma operator.

Syntax (for defining class template):

```
template <class T1, class T2, … >
class class-name
{
        //body of class
};
```

Syntax (for defining object of the class):

```
template <type1, type2, … > object-name;
```

Syntax (for defining member function outside class):

```
template <class T1, class T2, … >
return-type class-name<T1, T2, …>::function-name (argument-list) {
        // body of function
}
```

```
#include <iostream>

using namespace std;

template<class T1, class T2>
class Record
{
        T1 a;
        T2 b;
```

```
   public:
        Record(T1 x, T2 y) {
                a = x;
                b = y;
        }
        void show();
};

template<class T1, class T2>
void Record<T1,T2>::show() {
        cout << a << " and " << b << endl;
}

int main()
{
        Record<int, char> Obj1(1,'A');
        Record<float, char> Obj2(12.3,'B');
        Obj1.show();
        Obj2.show();
return 0;
}
```

O/P

```
1 and A
12.3 and B
```

The class 'Record' can take two generic types to store the records of the two types. Obj1 stores an int and a char and Obj2 stores a float and a char.

9.3.2 Non-Template Type Arguments

A non-type template argument provided within a template argument list is an expression whose value can be determined at compile time. Such arguments must be constant expressions. addresses of functions or objects with external linkage, or addresses of static class members. Non-template arguments are normally used to initialise a class or to specify the sizes of class members.

Example:

```
template <class T, int size>
class array
{
      T a[size];
      …
      ….. .
}
```

```
/*
write a class template to sort numbers also defining size of array
*/
```

```
#include <iostream>
using namespace std;

template <class T, int s>
class Array
{
     T a[s];
public:
     void input(){
          cout << "enter numbers " << endl;
          for(int i=0; i<s; i++)
               cin >> a[i];
     }
     void display(){
          cout << "sorted numbers " << endl;
          for(int i=0; i<s; i++)
               cout << a[i] << endl;
          cout << endl;
     }
     void sort();
};

template <class T, int s>
void Array<T, s>::sort(){
     for(int i=s-1; i>=0; i--){
          for(int j=0; j<=i-1; j++){
               if(a[j] > a[j+1]){
                    T tmp = a[j];
                    a[j] = a[j+1];
                    a[j+1] = tmp;
               }
          }
     }
}

main()
{
     Array<int,4>a1;
     a1.input();
     a1.sort();
     a1.display();

     Array<float,3>a2;
     a2.input();
     a2.sort();
     a2.display();

return 0;
}
```

O/P

```
enter numbers
12
32
124
```

```
23
sorted numbers
12
23
32
124

enter numbers
2.3
123.2

1 2
sorted numbers
1.3
2.3
123.2
```

9.3.3 Default Arguments with Class

Template parameters may have default argument. the set of default template arguments accumulates over all declarations of a given template. the following example:

```
/*
WAP to show the implementation of default argument with class
template
*/
#include <iostream>
using namespace std;
template <class T=float, int s=2>
class Array
{
     T a[s];
public:
     void input();
     void sum();
};

template <class T, int s>
void Array<T,s>::input(){
          cout << "enter numbers " << endl;
          for(int i=0; i<s; i++)
               cin >> a[i];
     }

template <class T, int s>
void Array<T, s>::sum(){
     T sum = 0;
     for(int i=0; i<s; i++)
          sum += a[i];
     cout << "The sum is : " << sum << endl;
}
```

```
main()
{
      cout << "For integer array of size 5" << endl;
      Array<int,5>a1;
      a1.input();
      a1.sum();

      cout << "For default values" << endl;
      Array<>a2;
      a2.input();
      a2.sum();

return 0;
}
```

O/P

```
enter numbers
1
2
3
4
5
The sum is : 15
For default values
enter numbers
1.1
2.2
The sum is : 3.3
```

9.4 Derived Class Template

Templates and derivation are mechanisms for building new types out of existing ones, and generally for writing useful codes that exploit various forms of commonality. there may be many situations where the base can be or the derived class can be template class. Some of them are discussed below:

A base class can be template class of the derived class is non template class, thenwhile creating the derived class we need to specify the template argument of the base class. For instance consider the following program:

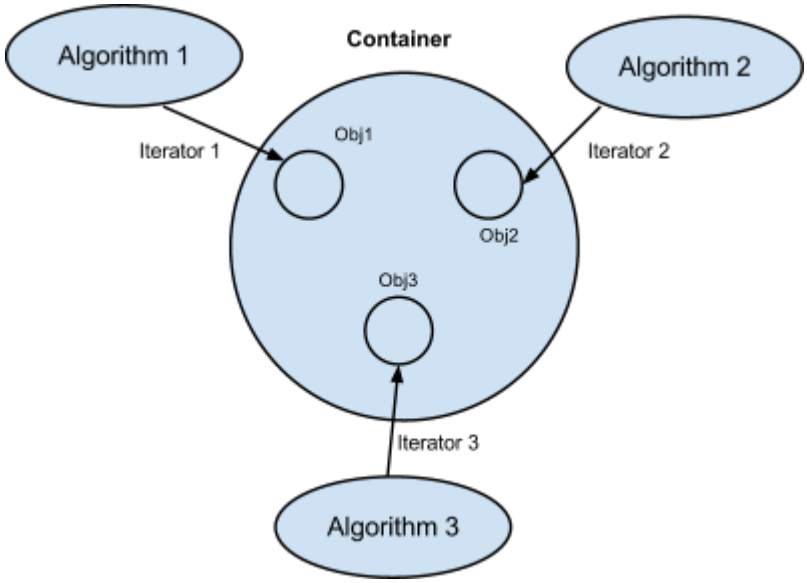
NOT AVAILABLE

9.5 Introduction to Standard Template Library

We have learned about generic programming in early section from the use of template class and template function. During the standardization of C++, standard Template Library was included. It provide general purpose, templatized class and functions that implement many popular and commonly used algorithms and data structures, for example vector, stack queue, list map, etc. As the STL is constructed from template classes, the algorithm and data structure can be applied for any type of data.

STL is large and complex and it is difficult to discuss all the features. STL components that are now part of the Standard C++ library are defined in the namespace std. So we should use this directive to use STL.

The fundamental components of STL are containers, algorithms and iterators. Their relationship is shown by the figure below



9.5.1 Containers

Containers are objects that hold other objects. It is a way in which data is organized in memory. The STL containers are implemented by template classes and therefore can be easily customized to hold different types of data. The STL defines ten containers that are grouped into three categories:

1. Sequence Containers
2. Associative Containers
3. Derived Containers

Sequence Containers

Sequence Containers store elements in a linear list. They are catagorised in following types:

1. Vector
2. List
3. Deque

Elements in all these containers can be accessed using the iterator. The difference between them is related to only their performances.

Container	Random Access	Insertion and deletion in middle
Vector	Fast	Slow
List	Slow	Fast
Deque	Fast	Slow

Associative Containers

Associative Containers are containers which allows efficient retrieval of value based on keys. They are not sequential. There are four types of associative containers:

1. set
2. multiset
3. map

4. multipmap

All these containers store data in a structure called tree that faciliates fast searching, deletion, and insertion. Each element in this container has key value. However these are very slow for random access and inefficient for sorting.

Containers set and multiset can store number of items containing keys. The main difference between a set and a multiset is that multiset allows duplicate items while a set does not.

Containers map and multimap are used to store a pair of value one called the key and the other called the value. we can manipulate the value using the keys associated with them. The main difference between map and multimap is that a map allows only one key for a given value to be stored while multimap allows a number of values with the same key.

Derived Containers

These containers are derived from sequential container. These are also known as container adaptors. The STL provides three derived containers namely, stack, queue and priority_queue.

Stack is a derived container where elements are inserted and removed in LIFO. The element last entered is the first to be removed.

Queue is a derived container that follows FIFO. The elements are added to the back of queue and are removed from the front of the queue.

By default, Stack and Queue are of deque type.

Priority Queue is a derived container where insertion, removal and traversal is performed on the top elements where it guarantees that the top element is the element with the highest priority queue. By default, it is of vector type.

Stacks, queues and priority-queues can be created from different sequence containers. The derived containers do not support iterators and therefore we cannot use them for data manipulation. However, they support two member functions pop() and push() for implementing deleting and inserting operations.

9.5.2 Algorithms

An algorithm is a procedure that is used to process the data contained in the containers. STL provides more than sixty standard algorithms to support more extended or complex operations. Standard algorithms also permit us to work with two different types of containers at the same time. STL algorithms are not member functions or friends of containers. They are standalone template function. STL algorithms are not member functions or friends of containers. They are standalone template functions. STL algorithm based on the nature of operations they perform may be categorized as:

1. Mutating algorithms
2. Sorting algorithms
3. Set algorithms
4. Relational algorithms
5. Retrieve or non-mutating algorithms

9.5.3 Iterators

Iterators behave like pointers and are used to access individual elements in container. They are often used to traverse from one element to another, a process known as iterating through the container. That means if the iterator points to one element in range then it is possible to increase or decrease iterator so that we can access next element in the range. Iterators connect algorithms and play a key role in the manipulation of data stored in the containers.

There are five types of iteraotrs

Iterator	Access Method	Direction of movement	I/O capability	Remark
Input	Linear	Forward only	Read only	Cannot be saved
Output	Linear	Forward only	Write only	Cannot be saved
Forward	Linear	Forward only	Read/Write	Can be saved
Bidirectional	Linear	Forward and Backwards	Read/Write	Can be saved
Random	Random	Forward and Backwards	Read/Write	Can be saved

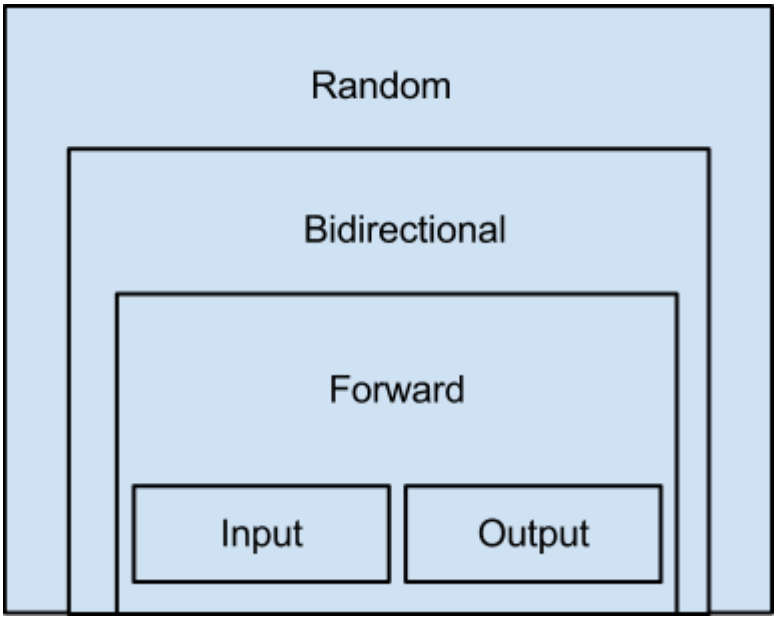


Fig: Relationship between types of iterators

Only sequential and associative containers are transferrable with iterators. The input and output support the least functions. They can be used only to traverse in a container.

Chapter 10

Exception Handling

Exception handling is the process of responding to the occurrence, during computation, of exceptions – anomalous or exceptional events requiring special processing – often changing the normal flow of program execution.
In C++, we usr keyword try, catch and throw. Try keyword is used to a block of statements surrounded by braces which may generate exception. When a exception is detected, it is thrown using throw statement is try block, then the program control leaves the try block and enter the catch block comparing the catch argument try and the exception thrown type. If the type of the object matches the argument type in the catch statement, then catch block is executed for handling the exception. If they don't match, the program is aborted with the help of the abort() function which is invoked by default. When no exception is detected and thrown, the control goes to the statement immediately after the catch block i.e. catch block is skipped. In handling exception we need to create a new kind of entity called an exception class.

10.2 Exception Handling Constructs(try, catch, throw)

```
#include<iostream>
#include<cmath>

using namespace std;

int main()
{
      int a;
      cout << "Enter the number : ";
      cin >> a;

      try{
            if(a < 0)
                  throw a;

            cout << "Square root : " << sqrt(a) << endl;
      }

      catch (int a){
            cout << "Error in input " << endl;
      }
}
```

O/P (Without Exception)

```
Enter the number : 5
Square root : 2.23607
```

O/P (With Exception)

```
Enter the number : -10
Error in input
```

10.4 Multiple Exception Handling

Multiple catch with basic data type
We can make different catch for different exception, to display different error message.

```
#include<iostream>
using namespace std;

void test(int a, int b)
{
     try{
          if (b < 0)
               throw b;

          if(b == 0)
               throw 1.1;

          cout << "The Quotient = " << (a/b) << endl;
     }

     catch(int c){
          cout << "\nSecond Operand is less than zero" << endl;
     }

     catch(double c){
          cout << "\nSecond operand is equal to zero" << endl;
     }
}

int main()
{
     test(12,3);
     test(12, -3);
     test(12,0);
return 0;
}
```

O/P

```
The Quotient = 4

Second Operand is less than zero

Second operand is equal to zero
```

```
#include<iostream>
using namespace std;
void Calculate(int a,int b){
     try{
          cout<<"\nInside inner try block";
          if(b==0)
               throw b;
          cout<<"\nQuotation is:"<<a/b<<endl;
     }
     catch(int)
     {
          cout<<"\nInside inner catch";
          throw;
     }
}

int main()
{
     int c,d;
     cout<<"\nEnter the value of c and d:";
     cin>>c>>d;
     try
     {
          cout<<"\nInside outer try block";
          Calculate(c,d);
     }
     catch(int)
     {
          cout<<"\nInside outer Catch";
          cout<<"\nException due to 0"<<endl;
     }
return 0;
}
```

O/P

```
Enter the value of c and d:88
5

Inside outer try block
Inside inner try block

Quotation is:17
```

10.6 Catching All Exceptions

In some situations, we may not be able to anticipate all the possible types of exceptions and therefore may not be able to design independent catch handlers to catch them. In such circumstances, we can force a catch statement to catch all exceptions instead of a certain type. To define this type of catch we need to put **...** in place of arguments.

Syntax:

```
catch(...)
{
```

```
      //Statements
}
```

Example :

```
#include<iostream>
using namespace std;

int main()
{
      int a,b;
      cout << "Enter the value of a and b : ";
      cin >> a >> b;

      try{
            if(b == 0)
                  throw a;

            if(b < 0)
                  throw 1.0;

            cout << "Quotient = " << a/b;
      }

      catch(int){
            cout << "Division by zero" << endl;
      }

      catch(...){
            cout << "Other Exceptions" << endl;
      }
return 0;
}
```

O/P

```
Enter the value of a and b : 10 0
Division by zero
```

```
Enter the value of a and b : 10 -2
Other Exceptions
```

```
Enter the value of a and b : 10 2
Quotient = 5
```

10.7 Exception with Arguments

There may be situation, where we need more information about the cause of exception. Let us consider there may be more than one function that throws the same exception in a program. It would be nice if we know which function threw the exception and the cause to throw an exception. This can be accomplished by defining the object in the exception handler. This means, adding data members to the exception class which can be retrieved by exception handler to know the cause. While throwing, throw statement throws exception class object with some initial value,

which is used by exception handler.

Example:

```
#include<iostream>
using namespace std;

class argument
{
      int a, b;
public:
      char *msg;
      argument(){
            a= b= 0;
      }

      argument(char *name){
            msg = name;
      }
      void input(){
            cout << "Enter the value of a and b : ";
            cin >> a >> b;
      }

      void calculate(){
                  if(b == 0)
                        throw argument("Divided by Zero.");

                  if(b < 0)
                        throw argument("Divided by Negative number");

                  cout << "Quotient = " << a/b << endl;
      }
};

int main()
{
      argument A1;
      A1.input();
      try{
            A1.calculate();
      }

      catch(argument A){
            cout << "Exception : " << A.msg << endl;
      }
return 0;
}
```

O/P

```
Enter the value of a and b : 10 0
Exception : Divided by Zero.
```

```
Enter the value of a and b : 10 -1
Exception : Divided by Negative number
```

```
Enter the value of a and b : 10 5
Quotient = 2
```

10.8 Exceptions Specifications for Function

Throwing or catching an exception affects the way a function relates to other functions. It can therefore be worthwhile to specify the set of exceptions that might be thrown as part of the function declaration. To restrict a function to throw only certain specified exceptions, we do this by adding a throw list clause to the function definition.

Syntax:

```
returntype function_name(arguments-list) throw(type-list)
{
      //function body
}
```

Example:

```
#include<iostream>
using namespace std;

void display(){
      cout<<"\nProgram is about to terminate:";
      //system("pause");
}

void calculate(int a,int b) throw(int){

      if(b==0)
            throw b;
      if(b<0)
            throw 1;
      cout<<"\nQuotation is :"<<a/b;
}

int main()
{
      int c,d;

            set_unexpected(display);
      set_terminate(display);

      cout<<"\nEnter the value of c n d:";
      cin>>c>>d;

      try
      {
            calculate(c,d);
      }
      catch(double)
      {
            cout<<"\nException as Second operand is -ve";
      }
return 0;
}
```

O/P

```
Enter the value of c n d:50
8
```

```
Quotation is :6
```

O/P

```
Enter the value of c n d:10
0

Program is about to terminate:Aborted
```

O/P

```
Enter the value of c n d:80
-2

Program is about to terminate:
Program is about to terminate:Aborted
```