Object Oriented Programming Using C++

DECAP444

Edited by
Balraj Kumar

LOVELY
PROFESSIONAL
UNIVERSITY

Object Oriented Programming Using C++

Edited By:
Balraj Kumar

CONTENT

Prikshit Kumar Angra, Lovely Professional University

Unit 01: Principles of OOPs and basics of C++

CONTENTS

Objectives

After studying this unit, you will be able to:

- Recognize the basic concept of object-oriented programming
- Describe the OOP languages
- Understand Basics of C++
- Compare the procedural Oriented and object-oriented programming

Introduction

Over the last few decades, programming practices have changed dramatically. As more programmers gained competence, previously undiscovered difficulties began to emerge. The programming community became increasingly worried about the programming philosophy they use and the methodologies they use in software development.

Productivity, reliability, cost effectiveness, reusability, and other factors began to become key concerns. Many conscious attempts were made to comprehend these issues and find potential answers. This is precisely why a growing number of programming languages have been developed and are still being developed. Furthermore, techniques to programme creation have been the subject of extensive research, resulting in the development of several frameworks. The object-oriented programming method, or simply OOP, is one such approach, and it is perhaps the most common.

C++ programming's main goal is to introduce the concept of object orientation to the C programming language.

Inheritance, data binding, polymorphism, and other notions are all part of the Object Oriented Programming paradigm.

1.1 Basic Concept of Object-oriented Programming

Object oriented programming is a type of programming which uses objects and classes its functioning. The object oriented programming is based on real world entities like inheritance, polymorphism, data hiding, etc. It aims at binding together data and function work on these data sets into a single entity to restrict their usage.

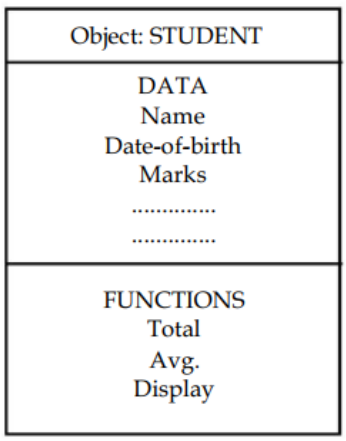
In object-oriented parlance, a problem is viewed in terms of the following concepts:

1. Objects
2. Classes
3. Data abstraction
4. Data encapsulation
5. Inheritance
6. Polymorphism
7. Dynamic binding
8. Message passing

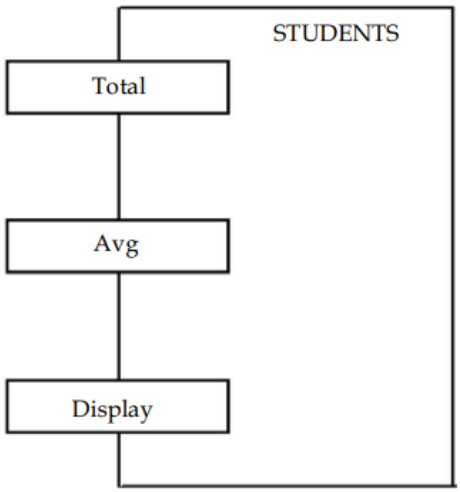
Let us now study the entire concept in detail.

1. **Objects**: Objects are the basic run-time entities in an object-oriented system. They may represent a person, a place, a bank account, a table of data; they may also represent user-defined data such as vectors, time and lists.

They occupy space in memory that keeps its state and is operated on by the defined operations on the object. Each object contains data and code to manipulate the data.



(a)



(b)

Notes: -Objects can interact without having to know details of each other data or code.

2. ***Classes:*** A class represents a set of related objects. The object has some attributes, whose

value consist much of the state of an object. The class of an object defines what attributes an object has. The entire set of data and code of an object can be made a user-defined data type with the help of a class.

Classes are user defined data types that behave like the built-in types of a programming language. Classes have an interface that defines which part of an object of a class can be accessed from outside and how. A class body that implements the operations in the interface, and the instance variables that contain the state of an object of that class.

Notes:- A class is a data-type that has its own members i.e. data members and member functions. It is the blueprint for an object in objects oriented programming language. It is the basic building block of object oriented programming in c++.

The data and the operation of a class can be declared as one of the three types:

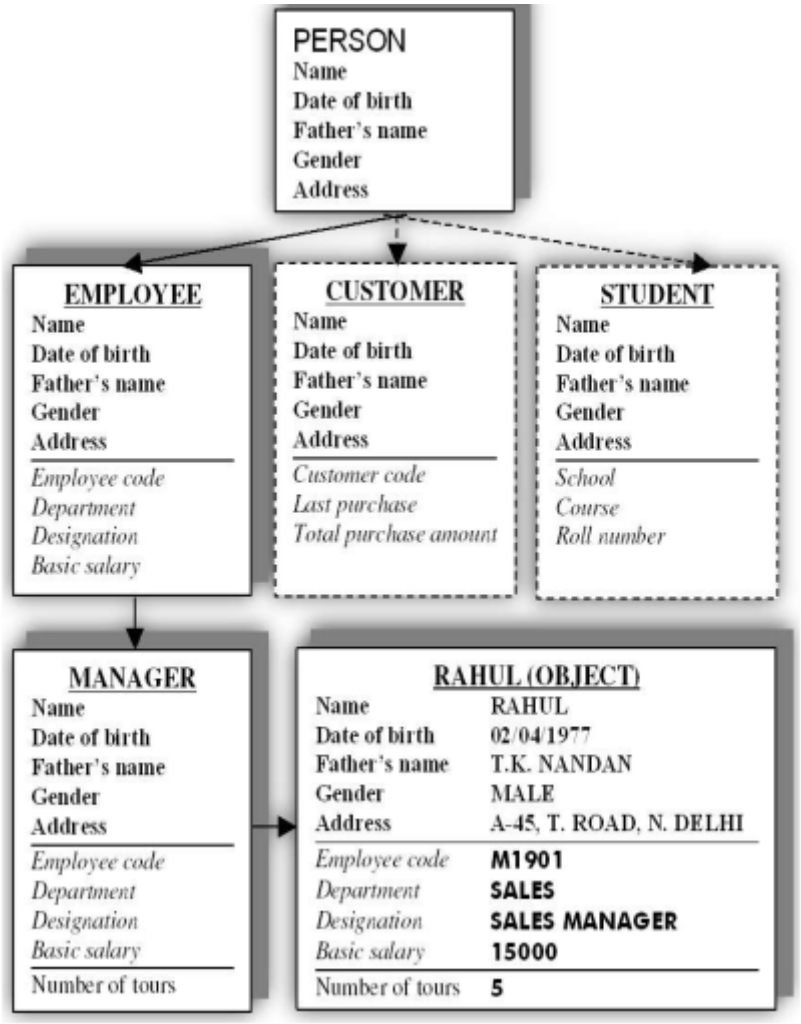
(a) **Public:** These are declarations that are accessible from outside the class to anyone who can access an object of this class.

(b) **Protected**: These are declarations that are accessible from outside the class to anyone who can access an object of this class.

(c) **Private:** These are declarations that are accessible only from within the class itself.

Syntax of class

```
class class_name {
  data_type data_name;
  return_type method_name(parameters);
}
```



3. ***Data abstraction: -*** Data abstraction or Data Hiding is the concept of hiding data and showing only relevant data to the final user. It is also an important part object oriented programing.

   Let's take real life example to understand concept better, when we ride a bike we only know that pressing the brake will stop the bike and rotating the throttle will accelerate but you don't know how it works and it is also not think we should know that's why this is done from the same as a concept data abstraction.

In C plus plus programming language write two ways using which we can accomplish data abstraction −

1. using class
2. using header file

 **Example: -** We can represents essential features without including background details and explanations.

index of text book.

```
class School
{
void sixthc lass();
void seventhclass();
void tenthclass();
}
```

4. ***Data Encapsulation:*** In object oriented programming, encapsulation is the concept of wrapping together of data and information in a single unit. A formal definition of encapsulation would be: encapsulation is binding together the data and related function that can manipulate the data.
5. ***Inheritance:*** A class's capacity to inherit or derive attributes or characteristics from other classes is known as inheritance. It is particularly significant in an object-oriented software since it allows for reusability, i.e. using a method defined in another class via inheritance. Child class or subclass is a class that inherits properties from another class, while base class or parent class is the class from which the properties are inherited.

 **Notes: -** Inheritance allows us to create a new class (derived class) from an existing class (base class).

C plus plus programming language supports the following types of inheritance

- single inheritance
- multiple inheritance
- multi level inheritance
- Hierarchical inheritance
- hybrid inheritance

 **Caution: -** Keep in mind that each subclass defines only those features that are unique to it.

6. ***Polymorphism:-***

Polymorphism means the ability to take more than one form. An operation may exhibit different behavior in different instances. The behavior depends upon the types of the data used in the operation. For example considering the operator plus (+).

16 + 4 = 20

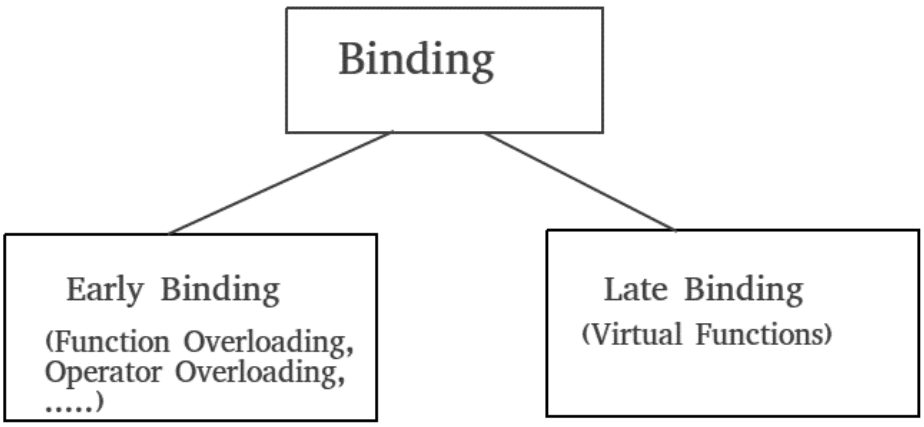
"xyz" + "sqr" = "xyzsqr"

Example: - A person can have more than one behavior depending upon the situation. like a woman a mother, manager and a daughter And this define her behavior. This is from where the concept of polymorphism came from.

In c++ programming language, polymorphism is achieved using two ways. They are operator overloading and function overloading.

7. ***Dynamic Binding:-***

Binding refers to the process of converting identifiers (such as variable and performance names) into addresses. Binding is done for each variable and functions. For functions, it means that matching the call with the right function definition by the compiler. It takes place either at compile time or at runtime.



This is associated with polymorphism and inheritance. A function call associated with a polymorphic reference depends on the dynamic type of that reference. For example in the above figure, by inheritance, every object will have this procedure. Its algorithm is, however, unique to

each object so the procedure will be redefined in each class that defines the objects. At run-time, the code matching the object under reference will be called.

Difference between static and dynamic binding in C++

Did you know?

BASIS FOR COMPARISON	STATIC BINDING	DYNAMIC BINDING
Event Occurrence	Events occur at compile time are "Static Binding".	Event Occurrence
Information	All information needed to call a function is known at compile time.	All information need to call a function come to know at run time.
Advantage	Efficiency.	Flexibility.
Time	Fast execution.	Slow execution.
Alternate name	Early Binding.	Late Binding.

8. ***Message Passing***

Objects communicate with one another by sending and receiving information to each other. A message for an object is a request for execution of a procedure and therefore will invoke a function in the receiving object that generates the desired results. Message passing involves specifying the name of the object, the name of the function and the information to be sent.

Example

```
#include<iostream>
using namespace std;
class demo{
public:
int f_num,s_num;
sum(int a,int b){
cout<<a+b;
}
};
main(){
demo d1;
d1.sum(d1.f_num=10,d1.s_num=39);
return 0;
}
```

1.2 Introduction to OOP languages

Object oriented programming is not the right of any particular language. Although languages like C and Pascal can be used but programming becomes clumsy and may generate confusion when

program grow in size. A language that is specially designed to support the OOP concepts makes it easier to implement them.

To claim that they are object-oriented they should support several concepts of OOP.

Depending upon the features they support, they are classified into the following categories:

1. Object-based programming languages.

2. Object-oriented programming languages.

Characteristics	Simula	Small talk 80	Objective C	C ++	ADA	Object Pascal	Eittel
Binding (early or late)	Both	Late	Both	Both	Early	Late	Early
polymorphism (operator overloading)	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Data hiding	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Concurrency	Yes	Poor	Poor	Poor	Difficult	No	Promised
Inheritance	Yes	Yes	Yes	Yes	No	Yes	Yes
Multiple Inheritance	No	Yes	Yes	Yes	No	Yes	Yes
Garbage Collection	Yes	Yes	Yes	Yes	No	Yes	Yes
Persistence	No	Pormised	No	No	Like 3GL	No	Some support
Genericity	No	No	No	No	Yes	No	Yes
Object lib	Yes	Yes	Yes	No	Not much	Yes	yes

Major features required by object-based programming are:

1. Polymorphism

2. Data encapsulation

3. Data hiding

4. Operator Overloading

5. Inheritance

Languages that support programming with objects are said to be object-based programming languages. These do not support inheritance and dynamic binding ADA is a typical example Object oriented programming incorporates all of objects-based programming features along with two additional feature, namely, inheritance and dynamic binding.

Languages that support these features:-

C++

.NET

Java

PHP

C#

Python

Ruby

1.3 Procedural programming v/s Object-oriented programming

Let's see the comparison between Procedural programming and object-oriented programming. We are comparing both terms on the basis of some characteristics. The difference between both languages are tabulated as follows –

On the basis of	Procedural Programming	Object-oriented programming
Definition	It is a programming language that is derived from structure programming and based upon the concept of calling procedures. It follows a step-by-step approach in order to break down a task into a set of variables and routines via a sequence of instructions.	Object-oriented programming is a computer programming design philosophy or methodology that organizes/ models software design around data or objects rather than functions and logic.
Approach	It follows a top-down approach.	It follows a bottom-up approach.
Importance	It gives importance to functions over data.	It gives importance to data over functions.
Data hiding	There is not any proper way for data hiding.	There is a possibility of data hiding.
Program division	In Procedural programming, a program is divided into small programs that are referred to as functions.	In OOP, a program is divided into small parts that are referred to as objects.

In the modern programming parlance, at least in most of the commercial and business applications areas, programming has been made independent of the target machine. This machine independent characteristic of programming has given rise to a number of different methodologies in which programs can now be developed. We will particularly concern ourselves with two broad programming approaches – or paradigm as they are called in the present context.

1. Procedure-oriented paradigm

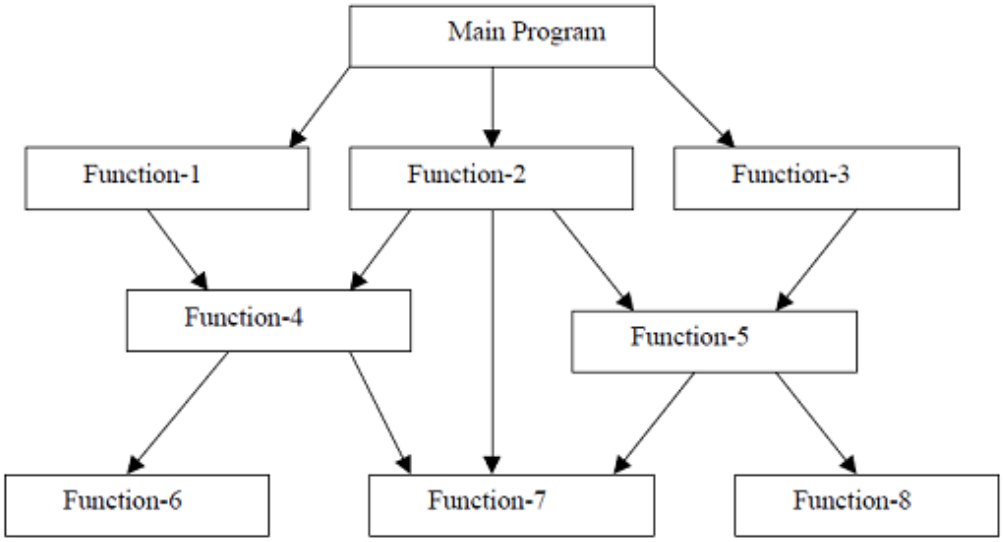
2. Object-oriented paradigm

1.4 Procedure oriented Programming Paradigm

Before you get into OOP, take a look at conventional procedure-oriented programming in a language such as C. Using the procedure-oriented approach; you view a problem as a sequence of things to do such as reading, calculating and printing. Conventional programming using high-level languages is commonly known as procedure-oriented programming.


Example C, COBOL and FORTRAN

You organize the related data items into C structures and write the necessary functions (procedures) to manipulate the data and, in the process, complete the sequence of tasks that solve your problem. Structure of procedure oriented paradigm is shown in following figure.



Many key data items are put as global in a multi-function software so that they can be accessible by all functions. It's possible that each function has its own set of local data. Global data are more vulnerable to a function's unintended alteration. It's difficult to figure out which data is used by which function in a huge software. If we need to make changes to an external data structure, we must likewise make changes to any functions that access the data. Bugs will be able to get in through this opening.

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

Some Characteristics exhibited by procedure-oriented programming are:-

Emphasis is on doing things (algorithms).

Large programs are divided into smaller programs known as functions.

Most of the functions share global data.

Data move openly around the system from function to function.

Functions transform data from one form to another.

1.5 Object-oriented Programming Paradigm

The term Object-oriented Programming (OOP) is widely used, but experts do not seem to agree on its exact definition. However, most experts agree that OOP involves defining Abstract Data Types (ADT) representing complex real-world or abstract objects and organizing programs around the collection of ADTs with an eye toward exploiting their common features. The term data abstraction refers to the process of defining ADTs; inheritance and polymorphism refer to the mechanisms that enable you to take advantage of the common characteristics of the ADTs – the objects in OOP.

Before going any further into OOP, take note of two points. First, OOP is only a method of designing and implementing software. Use of object-oriented techniques does not impart anything to a finished software product that the user can see. However, as a programmer while implementing the software, you can gain significant advantages by using object-oriented methods, especially in large software projects. Because OOP enables you to remain close to the conceptual, higher-level model of the real-world problem you are trying to solve, you can manage the complexity better than with approaches that force you to map the problem to fit the features of the language. You can take advantage of the modularity of objects and implement the program in

relatively independent units that are easier to maintain and extend. You can also share code among objects through inheritance.

Secondly, OOP has nothing to do with any programming language, although a programming language that supports OOP makes it easier to implement the object-oriented techniques. As you will see shortly, with some discipline, you can use objects even in C programs.

1.6 Benefits of OOP

- It is easy to model a real system as real objects are represented by programming objects in OOP. The objects are processed by their member data and functions. It is easy to analyze the user requirements.
- With the help of inheritance, we can reuse the existing class to derive a new class such that the redundant code is eliminated, and the use of existing class is extended. This saves time and cost of program.
- Modular approach is used for write code.
- In OOP, data can be made private to a class such that only member functions of the class can access the data. This principle of data hiding helps the programmer to build a secure program that cannot be invaded by code in other part of the program.
- It is very easy to partition the work in a project based on objects.
- It is possible to map the objects in problem domain to those in the program.
- With the help of polymorphism, the same function or same operator can be used for different purposes. This helps to manage software complexity easily.
- Large problems can be reduced to smaller and more manageable problems. It is easy to partition the work in a project based on objects.

1.7 Applications of Object Oriented Programming

Perhaps the most logical description of the real world is the object-oriented approach. As a result, it may be used in practically every problem-solving circumstance. OOP has been widely accepted in the software industry due to its effectiveness in problem solving and programming. Existing systems will be converted to OOP.

The framework, which spans all phases of system development, is one fundamental component of OOP that is extremely advantageous. Thus, OOA (Object Oriented Analysis), OOD (Object Oriented Design), OOT (Object Oriented Testing), and other object-oriented tools are significantly more appropriate than non-object-oriented ones.

Object oriented programming provides many applications:

- **Client-Server Systems:** Object-oriented client-server systems provide the IT infrastructure, creating Object-Oriented Client-Server Internet (OCSI) applications. Here, infrastructure refers to operating systems, networks, and hardware. OSCI consist of three major technologies:
  - The Client Server
  - Object-Oriented Programming
  - The Internet
- **Real-Time System:** A real time system is a system that give output at given instant and its parameters changes at every time. A real time system is nothing but a dynamic system. Dynamic means the system that changes every moment based on input to the system. OOP approach is very useful for Real time system because code changing is very easy in OOP system and it leads toward dynamic behavior of OOP codes thus more suitable to real time system.

- **Object-Oriented Databases:** They are also called Object Database Management Systems (ODBMS). These databases store objects instead of data, such as real numbers and integers. Objects consist of the following:
    - **Attributes:** Attributes are data that define the traits of an object. This data can be as simple as integers and real numbers. It can also be a reference to a complex object.
    - **Methods:** They define the behavior and are also called functions or procedures.

- **Simulation and modelling:** Another area where OOP approach criteria might be counted is system modelling. Because OOP programmers are easier to grasp, it is preferable to represent a system in a simpler form when using the OOP approach.

- **AI and expert systems:** These are computer applications that are developed to solve complex problems pertaining to a specific domain, which is at a level far beyond the reach of a human brain.

  It has the following characteristics:
    - Reliable
    - Highly responsive
    - Understandable
    - High-performance

- **Neural networks and parallel programming:** It address the problem of prediction and approximation of complex time-varying systems. Firstly, the entire time-varying process is split into several time intervals or slots. Then, neural networks are developed in a particular time interval to disperse the load of various networks. OOP simplifies the entire process by simplifying the approximation and prediction ability of networks.

1.8 C++ Class Member Function

Member function of a class is a function that has its definition or its prototype within the class definition like any other variable. It operates on any object of the class.

A member function is defined outside the class using the::(double colon symbol) scope resolution operator. This is useful when we did not want to define the function within the main program, which makes the program more understandable and easier to maintain.

Syntax

return_type class_name :: member_function

 Example

```
#include<iostream>
using namespace std;
class find_sum{
public:
   int x,y;
int sum();
```

```
};
int find_sum ::sum(){
return x+y;
}
```

1.9 Private Member Function

A private member variable or function cannot be accessed, or even viewed from outside the class. Only the class and friend functions can access private members.

 Example

```
#include <iostream>
using namespace std;
class Box {
  public:
    double length;
    void setWidth( double wid );
    double getWidth( void );
  private:
    double width;
};
double Box::getWidth(void) {
  return width ;
}
```

Summary

- Programming practices have evolved considerably over the past few decades.
- By the end of last decade, millions and millions of lines of codes have been designed and implemented all over the word.
- The main objective is to reuse these lines of codes. More and more software development projects were software crisis.
- It is this immediate crisis that necessitated the development of object-oriented approach which supports reusability of the existing code.
- Software is not manufactured. It is evolved or developed after passing through various developmental phases including study, analysis, design, implementation, and maintenance.
- Conventional programming using high level languages such as COBOL, FORTRAN and C is commonly known as procedures-oriented programming.
- In order to solve a problem, a hierarchical decomposition has been used to specify the tasks to be completed.
- OOP is a method of designing and implementing software.
- Since OOP enables you to remain close to the conceptual, higher-level model of the real world problem, you can manage the complexity better than with approaches that force you to map the problem to fit the features of the language.

- Some essential concepts that make a programming approach object-oriented are objects, classes, Data abstraction, Data encapsulation, Inheritance, Polymorphism, dynamic binding and message passing.
- The data and the operation of a class can be disclosed public, protected or private. OOP provides greater programmer productivity, better quality of software and lesser maintenance cost.

Keywords

Classes: A class represents a set of related objects.

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

Data Encapsulation: The wrapping up to data and functions into a single unit (class) is known as encapsulation.

Design: The term design describes both a final software system and a process by which it is developed.

Dynamic Binding: Binding refers to the linking of a procedure call to the code to be executed in response to the call.

Inheritance: Inheritance is the process by which objects of one class acquire the properties of objects of another class.

Message Passing: Message passing is another feature of object-oriented programming.

Object-oriented Programming Paradigm: The term object-oriented programming (OOP) is widely used, but experts do not seem to agree on its exact definition.

Objects: Objects are the basic run-time entities in an object-oriented system.

Polymorphism: Polymorphism means the ability to take more than one form.

Self Assessment

1. Which feature of OOPS described the reusability of code?

A. Abstraction
B. Encapsulation
C. Polymorphism
D. Inheritance

2. Which of the following is not an OOP?

A. Java
B. C#
C. C++
D. C

3. OOP acronym for

A. Object of Programming
B. Object Original Programming
C. Object Oriented Programming
D. Operating Original Programming

4. Which feature of OOPS derives the class from another class?

A. Inheritance

B. Data hiding
C. Encapsulation
D. Polymorphism

5. Which of the following is correct about class?
A. class can have member functions while structure cannot.
B. class data members are public by default while that of structure are private.
C. Pointer to structure or classes cannot be declared.
D. class data members are private by default while that of structure are public by default.

6. Which of the following is not an access specifier?
A. Public
B. Char
C. Private
D. Protected

7. Which of the following OOP concept is not true for the C++ programming language?
A. A class must have member functions
B. C++ Program can be easily written without the use of classes
C. At least one instance should be declared within the C++ program
D. C++ Program must contain at least one class

8. What is the extra feature in classes which was not in the structures?
A. Member functions
B. Data members
C. Public access specifier
D. Static Data allowed

9. Which operator is used to define a member function outside the class?
A. *
B. ( )
C. +
D. ::

10. Nested member function is
A. A function that call itself again and again.
B. A member function may call another member function within itself.
C. Same as Class in the program
D. Accessed using * operator

11. Which of the following is syntax of C++ class member function?
A. class_name,function_name
B. return_type class_name :: member_function
C. datatype_class_name,function_name
D. class_name_function_name

12. Which among the following feature does not come under the concept of OOPS?
A. Platform independent
B. Data binding

C. Data hiding
D. Message passing

13. The combination of abstraction of the data and code is viewed in_____.

A. Inheritance
B. Class
C. Object
D. Interfaces

14. Which is private member functions access scope?

A. Member functions which can used outside the class
B. Member functions which can only be used within the class
C. Member functions which are accessible in derived class
D. Member functions which can't be accessed inside the class

15. Which syntax among the following shows that a member is private in a class?

A. private::Name(parameters)
B. private: function Name(parameters)
C. private(function Name(parameters))
D. private function Name(parameters)

Answers for Self Assessment

1.	D	2.	D	3.	C	4.	A	5.	D
6.	B	7.	D	8.	A	9.	D	10.	B
11.	B	12.	A	13.	C	14.	B	15.	D

Review Questions

1. Discuss basic concepts of C++ in detail.
2. Inheritance is the process by which objects of one class acquire the properties of objects of another class. Analyze.
3. Examine what are the benefits of OOP?
4. Compare what you look for in the problem description when applying object-oriented approach in contrast to the procedural approach. Illustrate with some practical examples.
5. What is OOP? Explain the applications of object oriented programming in detail.
6. Differentiate procedural programming and object oriented programming.
7. Write programs that demonstrate working of classes and objects.

Further Readings

E. Balagurusamy, Object-oriented Programming through C++, Tata McGraw Hill.

Herbert Schildt, The Complete Reference – C++, Tata Mc Graw Hill.

Robert Lafore, Object-oriented Programming in Turbo C++, Galgotia Publications

Web Links

https://en.wikipedia.org/wiki/Object-oriented_programming

www.web-source.net

www.webopedia.com

Prikshit Kumar Angra, Lovely Professional University

Unit 02: Constructors and Destructors

CONTENTS

Objectives

After studying this unit, you will be able to:

- Recognize the need for constructor and destructors
- Describe the copy constructor
- Explain the dynamic constructor
- Discuss the destructors
- Explain the constructor and destructors with static members

Introduction

When an object is created all the members of the object are allocated memory spaces. Each object has its individual copy of member variables. However, the data members are not initialized automatically. If left uninitialized these members contain garbage values. Therefore, it is important that the data members are initialized to meaningful values at the time of object creation. Conventional methods of initializing data members have lot of limitations. In this unit you will learn alternative and more elegant ways initializing data members to initial values.

When a C++ program runs it invariably creates certain objects in the memory and when the program exits the objects must be destroyed so that the memory could be reclaimed for further use.

C++ provides mechanisms to cater to the above two necessary activities through constructors and destructors methods.

2.1 Constructor and Destructor

Constructor

Constructors are special class functions which performs initialization of every object. The Compiler calls the Constructor whenever an object is created. Constructors initialize values to object members after storage is allocated to the object. Whereas Destructor on the other hand is used to destroy the class object.

The default constructor method is called automatically at the time of creation of an object and does nothing more than initializing the data variables of the object to valid initial values.


Notes: Constructor has the same name as the class. Constructor is public in the class. Constructor does not have any return type.

While writing a constructor function the following points must be kept in mind:

1. The name of constructor method must be the same as the class name in which it is defined.
2. A constructor method must be a public method.
3. Constructor method does not return any value.
4. A constructor method may or may not have parameters.

Let us examine a few classes for illustration purpose. The class abc as defined below does not have user defined constructor method.

```
classabc
{
}
main()
{
}
intx,y;
abcmyabc;
…;
```

The main function above an object named myabc has been created which belongs to abc class defined above. Since class abc does not have any constructor method, the default constructor method of C++ will be called which will initialize the member variables as:

myabc.x=0 and myabc.y=0.

Let us now redefine myabc class and incorporate an explicit constructor method as shown

below:

classabc

Observed that myabc class has now a constructor defined to except two parameters of integer type. We can now create an object of myabc class passing two integer values for its construction, as listed below:

```
main()
{
```

```
abcmyabc(100,200);
———;
}
```

In the main function myabc object is created value 100 is stored in data variable x and 200 is stored in data variable y. There is another way of creating an object as shown below.

```
main()
{
myabc=abc(100,200);
———;
}
```

Both the syntaxes for creating the class are identical in effect. The choice is left to the programmer. There are other possibilities as well. Consider the following class differentials:

```
classabc
{
intx,y; public:
abc();
}
abc::abc()
{
x=100; y=200;
}
```

In this class constructor has been defined to have no parameter. When an object of this class is created the programmer does not have to pass any parameter and yet the data variables x,y are initialized to 100 and 200 respectively.

Finally, look at the class differentials as given below:

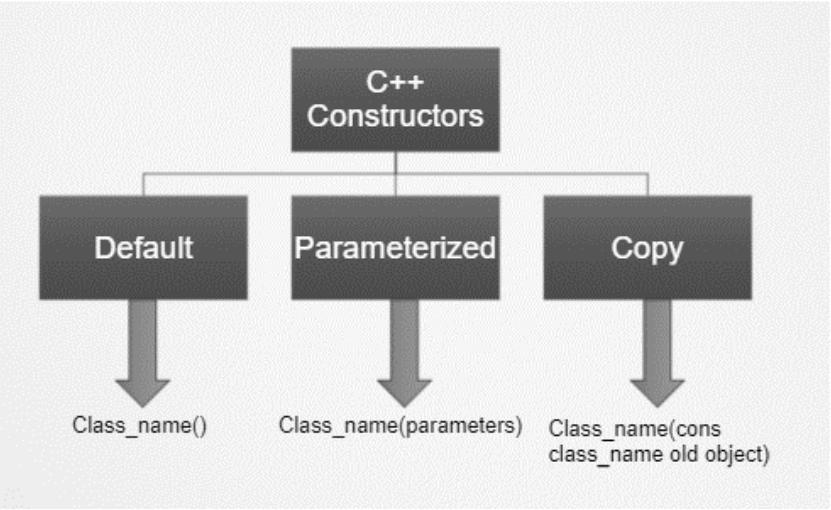
```
classabc

{

intx,y; public:
abc();

abc(int);

abc(int, int);

}

abc::abc()

{
```

```
x=100; y=200;
}

abc::abc(int a)

{

x=a; y=200;
}

abc::abc(int a)

{

x=100;

y=a;

}
```

Class myabc has three constructors having no parameter, one parameter and two parameters respectively. When an object to this class is created depending on number of parameters one of these constructors is selected and is automatically executed.

Types of constructor

1. Default Constructor
2. Parameterized Constructor
3. Copy Constructor



Destructor

Destructors are typically used to de-allocate memory. Also, they are used to clean up for objects and class members when the object gets terminated.

When should you define your own destructor function? In many cases you do not need a destructor function. However, if your class created dynamic objects, then you need to define your own destructor in which you will delete the dynamic objects. This is because dynamic objects cannot be deleted on their own. So, when the object is destroyed, the dynamic objects are deleted by the destructor function you define.

A destructor function has the same name as the class, and does not have a returned value. However you must precede the destructor with the tilde sign, which is ~ .

The following code illustrates the use of a destructor against dynamic objects:

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

class Calculator
{
public:
int *num1;
int *num2;

Calculator(int ident1, int ident2)
{
num1 = new int;
num2 = new int;
*num1 =ident1;
*num2 =ident2;
}
~Calculator()
{
delete num1;
delete num2;
}
int add(){
int sum=*num1+*num2;
return sum;
}
};
int main()
{
Calculator myObject(20,20);
int result = myObject.add();
cout<< result;
```

```
return 0;
}
```

The destructor function is automatically called, without you knowing, when the program no longer needs the object. If you defined a destructor function as in the above code, it will be executed. If you did not define a destructor function, C++ supplies you one, which the program uses unknown to you. However, this default destructor will not destroy dynamic objects.

Notes:-An object is destroyed as it goes out of scope.

Lab Exercise: Program to see how Constructor and Destructor are called.

```
class A
{
 // constructor
A()
 {
cout<< "Constructor called";
 }
// destructor
 ~A()
 {
cout<< "Destructor called";
 }
};
int main()
{
 A obj1; // Constructor Called
int x = 1
if(x)
 {
 A obj2; // Constructor Called
 } // Destructor Called for obj2
} // Destructor called for obj1
```

Output:-

```
Constructor called
Constructor called
Destructor called
Destructor called
```

2.2 Difference between Constructor and Destructor in C++

Constructors	Destructors
The constructor initializes the class and allots the memory to an object.	If the object is no longer required, then destructors demolish the objects.
When the object is created, a constructor is called automatically.	When the program gets terminated, the destructor is called automatically.
It receives arguments.	It does not receive any argument.
A constructor allows an object to initialize some of its value before it is used.	A destructor allows an object to execute some code at the time of its destruction.
It can be overloaded.	It cannot be overloaded.
When it comes to constructors, there can be various constructors in a class.	When it comes to destructors, there is constantly a single destructor in the class.
They are often called in successive order.	They are often called in reverse order of constructor.

2.3 Copy Constructor

A copy constructor method allows an object to be initialized with another object of the same class. It implies that the values stored in data members of an existing object can be copied into the data variables of the object being constructed, provided the objects belong to the same class. A copy constructor has a single parameter of reference type that refers to the class itself as shown below:

```
abc::abc(abc& a)


{
x=a.x;
y=a.y;
}
```

Suppose we create an object myabc1 with two integer parameters as shown below:

```
abc myabc1(1,2);
```

Having created myabc1, we can create another object of abc type, say myabc2 from myabc1, as shown below:

```
myabc2=abc(& myabc1);
```

The data values of myabc1 will be copied into the corresponding data variables of object myabc2. Another way of activating copy constructor is through assignment operator. Copy constructors come into play when an object is assigned another object of the same type, as shown below:

```
abc myabc1(1,2);
abc myabc2;
myabc2=myabc1;
```

Actually assignment operator(=) has been overloaded in C++ so that copy constructor is invoked

whenever an object is assigned another object of the same type.

Did you know?

What is the difference between the copy constructor and the assignment operator?

(a) If a new object has to be created before the copying can occur, the copy constructor isused.

(b) Ifanewobjectdoesnothavetobecreatedbeforethecopyingcanoccur,theassignment operator isused.

2.4 Dynamic Constructor

- Dynamic constructor is used to allocate the memory to the objects at the run time.
- Memory is allocated at run time with the help of 'new' operator.
- By using this constructor, we can dynamically initialize the objects.

Example:-

```
#include <iostream.h>
#include <conio.h>
classdyncons
{
int * p;
public:
dyncons()
 {
  p=new int;
  *p=100;
 }
dyncons(int v)
 {
  p=new int;
  *p=v;
 }
int dis()
 {
return(*p);
 }
};
int main()
{
clrscr();
dyncons o, o1(90);
```

```
cout<<"The value of object o's p is:";
cout<<o.dis();
cout<<"\nThe value of object 01's p is:"<<o1.dis();
return 0;
}
```

Output:

The value of object o's p is:100

The value of object 01's p is:90

2.5 Parameterized Constructors

If it is necessary to initialize the various data elements of different objects with different values when they are created. C++ permits us to achieve this objective by passing arguments to the constructor function when the objects are created. The constructors that can take arguments are called 'Parameterized constructors.' The definition and declaration are as follows:

```
classdist
{
int m, cm; public:
dist(int x, int y);
};
dist::dist(int x, int y)
{
m = x;     n = y ;
}
main()
{
dist d(4,2);
d. show ();
}
```

2.6 Constructors with Default Arguments

This method is used to initialize object with user defined parameters at the time of creation.

Consider the following Program that calculates simple interest. It declares a class interest representing principal, rate and year. The constructor function initializes the objects with principal and number of years. If rate of interest is not passed as an argument to it the Simple Interest is calculated taking the default value of rate of interest.

```
#include <iostream.h>
class interest
{
int principal, rate, year;
float amount;
```

```
public:
interest (int p, int n, int r = 10);
voidcal (void);
};
interest::interest (int p, int n, int r = 10)
{
principal = p;
year = n;
rate = r;
};
void interest::cal (void)
{
cout<< "Principal" <<principal;
cout<< "\ Rate" <<rate;
cout<< "\  Year"  <<year; amount = (float) (p*n*r)/100;
cout<< "\Amount" <<amount;
};
main ( )
{
interest i1(1000,2);
interest i2(1000, 2,15);
il.cal();
i2.cal();
}
```

Two objects are created in main function

```
interest i1(1000,2);
interest i2(1000, 2,15);
```

The data members principal and year of object i1 are initialized to 1000 and 2 respectively at the time when object i1 is created. The data member rate takes the default value 10 whereas when the object 12 is created, principal, year and rate are initialized to 1000, 2 and 15 respectively.

It is necessary to distinguish between the defaults

constructor::construct();

and default argument constructor

construct::construct(int = 0)

The default argument constructor can be called with one or no arguments. When it is invoked with no arguments it becomes a default constructor. But when both these forms are used in a class, it causes ambiguity for a declaration like construct C1;

The ambiguity is whether to invoke construct: : construct ( ) or construct: : construct (int=O)

Lab Exercise

```
//# Program – Default constructor
#include<iostream>
using namespace std;
class constructor{
private:
intx,y;
public:
constructor(){
   x=10;
   y=90;
cout<<"Sum of x and y is :"<<x+y;
   }
   };
int main(){
constructor c;
return 0;
   }
```

Output

```
Sum of x and y is :100
Process returned 0 (0x0)   execution time : 0.014 s
Press any key to continue.
```

Lab Exercise

// Program –Parameterizedconstructor

```
#include<iostream>
using namespace std;
class constructor{
private:
intx,y;
public:
constructor(inta,int b){
   x=a;
y=b;
cout<<"Sum of x and y is :"<<x+y;
   }
   };
int main(){
```

```
constructor c(15,52);
return 0;
  }
```

Output

```
Sum of x and y is :67
Process returned 0 (0x0)   execution time : 0.313 s
Press any key to continue.
```

Lab Exercise

// Program – Copy Constructor

```
#include<iostream>
using namespace std;
classcopyconstructor
{
private:
int x, y;
public:
copyconstructor(int x1, int y1)
  {
    x = x1;
    y = y1;
  }
copyconstructor (constcopyconstructor&sam)
  {
    x = sam.x;
    y = sam.y;
  }

void display()
  {
cout<<x<<" "<<y<<endl;
  }
};
int main()
{
copyconstructor obj1(10, 15);
copyconstructor obj2 = obj1;
cout<<"Constructor : ";
```

```
obj1.display();
cout<<"Copy constructor : ";
obj2.display();
return 0;
}
```

Output

```
Constructor : 10 15
Copy constructor : 10 15

Process returned 0 (0x0)   execution time : 0.105 s
Press any key to continue.
```

Lab Exercise

// Program – Dynamic Constructor

```
#include <iostream>
using namespace std;
class demo {
int* p;
public:
demo()
   {
      p = new int;
       *p = 500;
   }
void display()
{
cout<<"Dynamic constructor"<<endl;
cout<< *p <<endl;
   }
};
int main()
{   demoobj =  demo();
obj.display();
return 0;
}
demo()
   {
      p = new int;
       *p = 500;
```

```
  }


demo(int a)
  {
    p = new int;
     *p = a;
  }
```

Output

```
Dynamic constructor
500

Process returned 0 (0x0)   execution time : 0.110 s
Press any key to continue.
```

2.7 Constructor overloading

- A class can have multiple constructors that assign the fields in different ways.
- Overloaded constructors have the same name as class name but the different number of arguments.
- A constructor is called depending upon the number and type of arguments passed.
- While creating the object, arguments must be passed to let compiler know, which constructor needs to be called.

```
//Program
#include<iostream>
using namespace std;
class demo{
intx,y;
public:
demo(){
x=10;
cout<<"X is ="<<x<<endl;

}
demo(inta,int b){
   x=a;
   y=b;
cout<<"Sum of x and y is"<<x+y;
}
};
```

```
main(){
demo d1,d2(90,59);
}
```

Output

2.8 Destructors

Constructors create an object, allocate memory space to the data members and initialize the data members to appropriate values; at the time of object creation. Another member method called destructor does just the opposite when the program creating an object exits, thereby freeing the memory.

A destructive method has the following characteristics:

1. Name of the destructor method is the same as the name of the class preceded by a tilde (~).
2. The destructor method does not take any argument.
3. It does not return any value.

The following codes snippet shows the class student with the destructor method;

Example

```
#include <iostream>
using namespace std;
class student
 {
public:
student()
     {
cout<<"Constructor Invoked"<<endl;
     }
     ~student()
     {
cout<<"Destructor Invoked"<<endl;
     }
};
int main()
{
```

```
student s1;
return 0;
}
```

Summary

- A constructor is a member function of a class, having the same name as its class and which is called automatically each time an object of that class it created.
- It is used for initializing the member variables with desired initial values. A variable (including structure and array type) in C++ may be initialized with a value at the time of its declaration.
- The responsibility of initialization may be shifted, however, to the compiler by including a member function called constructor.
- A class constructor, if defined, is called whenever a program created an object of that class. Constructors are public member unless otherwise there is a good reason against.
- A constructor may take argument (s). A constructor may take no argument(s) is known as default constructor.
- A constructor may also have parameter (s) or argument (s), which can be provided at the time of creating an object of that class.
- C++ classes are derived data types and so they have constructor (s). Copy constructor is called whenever an instance of same type is assigned to another instance of the same class.
- If a constructor is called with a smaller number of arguments than required an error occurs. Every time an object is created its constructor is invoked.
- The functions that is automatically called when an object is no more required is known as destructor. It is also a member function very much like constructors but with an opposite intent.

Keywords

Constructor: A member function having the same name as its class and that initializes class objects with legal initial values.

Copy Constructor: A constructor that initializes an object with the data values of another object.

Default Constructor: A constructor that takes no arguments.

Destructor: A member function having the same name as its class but preceded by ~ sign and that de initializes an object before it goes out of scope.

Self-Assessment

1. Which of the followings is/are automatically added to every class, if we do not write our own?

A. Copy Constructor
B. Assignment Operator
C. A constructor without any parameter
D. All of the above

2. What will be output of following program?

```
#include<iostream>
using namespace std;
class Point {
   Point() { cout<< "Constructor called"; }
};
int main()
{
  Point t1;
   return 0;
}
```

A. Compiler Error
B. Runtime Error
C. Constructor called
D. None of Above

3. Which of the following gets called when an object is being created?

A. Constructor
B. Virtual Function
C. Destructors
D. Main

4. Can we define a class without creating constructors?

A. True
B. False

5. What will be output of following program?

```
#include<iostream>
using namespace std;
class demo{
public:
intf_num,s_num;
sum(inta,int b){
cout<<a+b;
}
};
main(){
demo d1;
d1.sum(d1.f_num=10,d1.s_num=20);
```

return 0;

}

A. 49
B. 50
C. 30
D. 10

6. Which constructor function is designed to copy object of same class type?
A. Copy constructor
B. Create constructor
C. Object constructor
D. Dynamic constructor

7. Allocation of memory to objects at the time of their construction is known as …….. of objects.
A. Run time construction
B. Dynamic construction
C. Initial construction
D. Memory allocator

8. If new operator is used, then the constructor function is
A. Parameterized constructor
B. Copy constructor
C. Dynamic constructor
D. Default constructor

9. Which among the following best describes constructor overloading?
A. Defining one constructor in each class of a program
B. Defining more than one constructor in single class
C. Defining more than one constructor in single class with different signature
D. Defining destructor with each constructor

10. Does constructor overloading include different return types for constructors to be overloaded?
a. Yes, if return types are different, signature becomes different
b. Yes, because return types can differentiate two functions
c. No, return type can't differentiate two functions
d. No, constructors doesn't have any return type

11. Which among the following is possible way to overload constructor?
a. Define default constructor, 1 parameter constructor and 2 parameter constructor
b. Define default constructor, zero argument constructor and 1 parameter constructor
c. Define default constructor, and 2 other parameterized constructors with same signature
d. Define 2 default constructors

12. Which is executed automatically when the control reaches the end of the class scope?

a. Constructor

b. Destructor

c. Overloading

d. Copy constructor

13. The special character related to destructor is ___

a. +

b. !

c. ?

d. ~

14. A destructor is used to destroy the objects that have been created by a ……..

a. Class

b. Object

c. Constructor

d. Destructor

15. …….. Provides the flexibility of using different format of data at runtime depending upon the situation.

a. Dynamic initialization

b. Run time initialization

c. Static initialization

d. Variable initialization

Answers for Self Assessment

1.	D	2.	A	3.	A	4.	A	5.	C
6.	A	7.	B	8.	C	9.	C	10.	D
11.	A	12.	B	13.	D	14.	C	15.	A

Review Questions

1. Write a program to calculate prime number using constructor.
2. Is there any difference between obj x; and objx();? Explain.
3. Can one constructor of a class call another constructor of the same class to initialize the this object? Justify your answers with an example.
4. Should my constructors use "initialization lists" or "assignment"? Discuss.
5. Explain constructor and different types of constructor with suitable example.
6. Write a program that demonstrate working of copy constructor.
7. What about returning a local variable by value? Does the local exist as a separate object, or does it get optimized away?

Further Readings

- E Balagurusamy; Object Oriented Programming with C++; Tata McGraw-Hill.
- Herbert Schildt; The Complete Reference C++; Tata McGraw Hill. Robert Lafore;
- Object-oriented Programming in Turbo C++; Galgotia.

Web Links

- https://en.wikipedia.org/wiki/Object-oriented_programming
- www.web-source.net
- www.webopedia.com

Unit 03: Functions and Compile Time Polymorphism

CONTENTS

Objectives

After studying this unit, you will be able to:

- Understand the functions
- Describe the function overloading
- Explain the inline functions
- Understand friend function
- Analyze C++ Static Data Members & Functions
- Define polymorphism in C++

Introduction

A function is a code module that only does one thing. Sorting, searching for a specific item, and inverting a square matrix are some instances. After a function is built, it is thoroughly tested. Following that, it is added to the library of functions. A user can use a library function as often as they like. This concept enhances software robustness while simultaneously shortening the time it takes to develop code. System-defined and user-defined functions are the two types of functions.

A function is a group of statements that together perform a task. Every C++ program has at least one function, which is main(), and all the most trivial programs can define additional functions.

A function declaration tells the compiler about a function's name, return type, and parameters. A function definition provides the actual body of the function.

3.1 Why We Need Functions in C++

- Debugging of the code would be easier if you use functions, as errors are easy to be traced.
- Use of functions reduce size of code.
- Duplicate set of statements are replaced by function calls.
- Improve reusability of code, same function can be used in any program.
- Use of function improves readability of code.

3.2 The Main Function

An application written in C++ may have a number of classes. One of these classes must contain one (and only one) method called main method. Although a private main method is permissible in C++ it is seldom used. For all practical purposes the main method should be declared as public method.

The main method can take various forms as listed below: Notes

main()

main(void)

void main()

void main(void)

int main()

int main(void)

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

main(int argc, char *argv[])

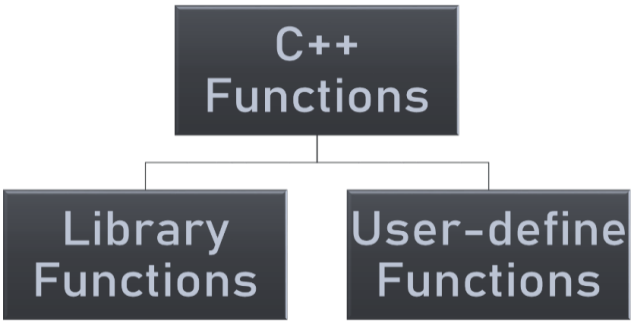
void main(int argc, char *argv[])

As is evident from the above forms, return type of the main method specifies the value returned to the operating system once the program finishes its execution. If your main method does not return any value to the operating system (caller of this method), then return type should be specified as void.

Caution: In case you design your main method in such a way that it must return a value to the caller, then it must return an integer type value and therefore you must specify return type to be int.

3.3 Function Types

We have two types of the functions one is the library function in c++ and user defined functions. Now, we are first going to discuss what are library functions, library functions are functions that are automatically included in the declared and used in the program, but that are not created by the users it is already present inside of the header file, which we always include at the beginning of the program.



Library Functions

- It is already present inside the header file which we always include at the beginning of a program.
- You just have to type the name of a function and use it along with the proper syntax.

User-defined Function

User-defined function is a type of function in which we have to write a body of a function and call the function whenever it's require.

3.4 Function Activities

Function activities are also divided into the function declaration function definition and function call

1. Function declaration
2. Function definition
3. Function call

1. ***Function Declaration***: Function declaration also called a function prototype. We just specify the name of a function that we are going to use in our program like a variable declaration.

Syntax:-

```
return_data_type function_name (data_type arguments);
```

2. **Function Definition:** Function definition function definition means just writing the body of a function means we are just going to write the body of the function what kind of work we can do with the function we are just write the logic they are a body of the function consists of the statements, which are going to be perform a specific task like a sub routine or a sub program, it is also known as the procedure we are writing the some of the lines there we are writing some of the code there that is used to perform some of this specific task.

Function definition means just writing the body of a function. A body of a function consists of statements which are going to perform a specific task. Let's see definition of function is how implemented using code.

Syntax:-

```
int sum(int a,int b)
{
        int c;
        c=a+b;
        return c;
}
```

3. **Function Call:** While creating a C++ function, you give a definition of what the function has to do. To use a function, you will have to call or invoke that function.

Function call has two methods.

1. Call by value
2. Call by reference

1. **Call by value:** This method of passing arguments to a function copies the actual value of an argument into the formal parameter of the function. By default, C++ uses call by value to pass arguments.
2. **Call by reference:** This method copies the reference of an argument into the formal parameter. Inside the function, the reference is used to access the actual argument used in the call. This means that changes made to the parameter affect the argument.


Example

```
#include<iostream>
using namespace std;
int sum(int x,int y)
{
   return x+y;
}
int main(){
int a,b;
cout<<"Enter two numbers";
cin>>a>>b;
cout<<"Sum of entered number using call by value is = "<<sum(a,b);
return 0;
}
```

Output

```
Enter two numbers 35
25
Sum of entered number using call by value is = 60
Process returned 0 (0x0)   execution time : 2.413 s
Press any key to continue.
```

3.5 Inline function

Now we are talking about inline functions c++ inline functions and inline function is a function that is explained in line when it is invoked with us saving the time we are going to save the functions or time to the execution then there are multiple ways we are calling the different functions we know that we are using the multiple functions in one single code. Inline function is used to execute the function but it not makes again the space inside of the memory. This copies the function to the location of the function call in the compile time when we are going to compile our program, it is going to copy the location of the function at the compile time and may take the program execution faster if it is going to save the function location into the compile time.

An inline function is a function that is expanded in line when it is invoked thus saving time. This copies the function to the location of the function call in compile-time and may make the program execution faster. Inline function is request to compiler not a command.

Advantages of Inline Function

1. It also saves the overhead of push/pop variables on the stack when function is called.

2. Inline function may be useful (if it is small) for embedded systems because inline can yield less code than the function call preamble and return.

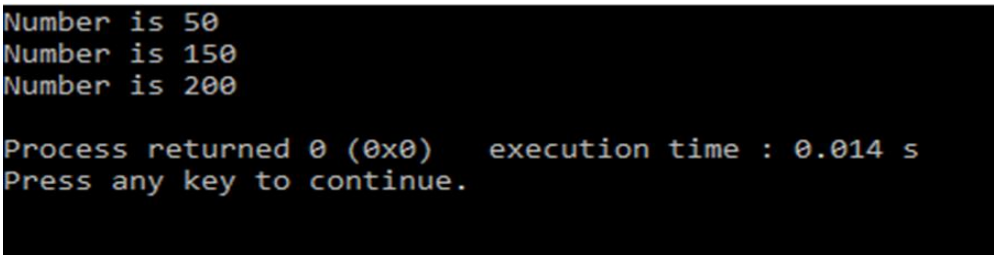
Disadvantages of Inline Function

1. Use too many inline functions then the size of the binary executable file will be large, because of the duplication of same code.
2. Too much inlining can also reduce your instruction cache hit rate.
3. Inline functions may not be useful for many embedded systems. Because in embedded systems code size is more important than speed.

 Example

```
#include <iostream>
using namespace std;
inline int display_number(int n){
cout<<"Number is "<< n<<endl;
}
int main() {
display_number(50);
display_number(150);
display_number(200);
return 0;
}
```

Output



C++ Objects as Function Arguments

We can pass objects to a function in a similar manner as passing regular arguments.

 Example

```
#include <iostream>
using namespace std;
class demo {
public:
  int n=100;
```

```
  char ch='A';
  void disp(demo d){
    cout<<d.n<<endl;
    cout<<d.ch<<endl;
  }
};
int main() {
   cout<<"Passing object to function"<<endl;
  demo obj;
  obj.disp(obj);
  return 0;
}
```

Output

```
Passing object to function
100
A

Process returned 0 (0x0)   execution time : 0.289 s
Press any key to continue.
```

3.6 C++ Friend Functions

A friend function of a class is defined outside that the scope. And it can be accessible all the private and protected members of the class. And we're talking about the friend function in the friend function, when we declared a friend function in class, it can be accessed all of the members of the class which is private or protected and we are using the friend keyword inside the body of the class to declare the friend function.

Syntax

```
class className {
   ... .. ...
   friend returnType functionName(arguments);
   ... .. ...
}
```

3.7 Characteristics of friend function

1. Friend function is not in the scope of the class in which it has been declared as friend.
2. It allows to generate more efficient code.
3. It cannot be called using the object of the class as it is not in the scope of the class.
4. It can be called similar to a normal function without the help of any object.

Example

```
#include <iostream>
using namespace std;
class rectangle{
int a;
public:
  friend void disp(rectangle r);
  void get_length(int l);
};
void rectangle::get_length(int l){
a=l;
}
void disp( rectangle r){
cout<<"Entered length of rectangle is"<<r.a;
}
main(){
rectangle r;
r.get_length(10);
disp(r);
}
```

Output

```
Entered length of rectangle is10
Process returned 0 (0x0)   execution time : 0.006 s
Press any key to continue.
```

3.8 C++ Static Data Members & Functions

Inside a class definition, the keyword static declares members that are not bound to class instances. A static member is shared by all objects of the class. A static member is shared by all of the object of the class. Static member is the one of the member which can be shared with all of the objects in my class. Let us see the syntax of the static member function inside of the class.

Syntax

```
Class class_name{
private:
static data_member;
public:
static return_type function_name()
                    {
```

```
//body
                }
};
```

Example

```
#include <iostream>
using namespace std;
class Demo
{
        private:
                static int a;
        public:
                static void fun()
                {
cout <<"Value of a: " << a << endl;
                }
};
int Demo :: a =500;
int main()
{
        Demo obj;

        obj.fun();

        return 0;
}
```

Output

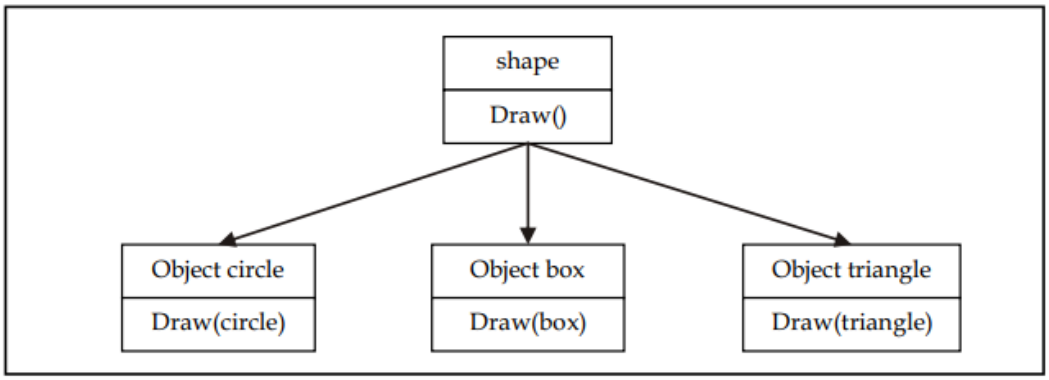
```
Value of a: 500

Process returned 0 (0x0)   execution time : 0.142 s
Press any key to continue.
```

3.9 Polymorphism in C++

Polymorphism is an important OOP concept. Polymorphism means the ability to take more than one form. For example, an operation may exhibit different behavior in different instances. The behavior depends upon the types of data used in the operation. For example, consider the operation of addition. For tow numbers, the operation will generate a sum. If the operands are strings, then the operation will produce a third string by contention. The diagram given below, illustrates that a single function name can be used to handle different number and types of

arguments. This is something similar to a particular word having several different meanings depending on the context.

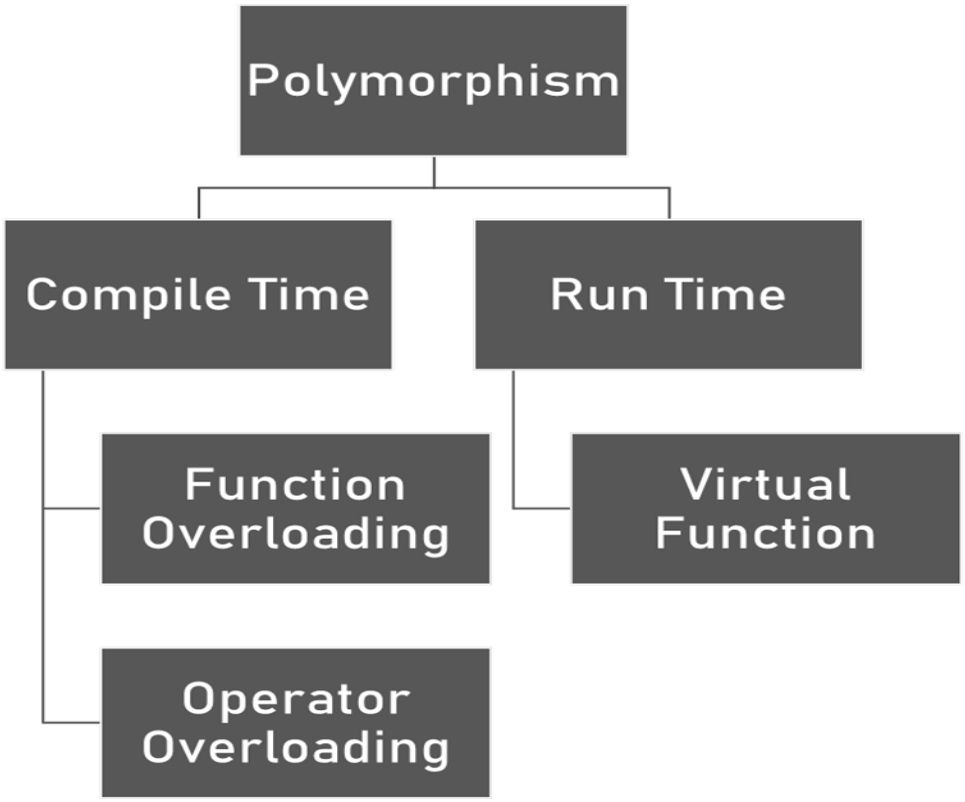
Polymorphism plays an important role in allowing objects having different internal structures to share the same external interface. This means that a general class of operations may be accessed in the same manner even though specific actions associated with each operation may differ. Polymorphism is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



Polymorphism can be implemented using operator and function overloading, where the same operator and function works differently on different arguments producing different results. These polymorphisms are brought into effect at compile time itself, hence is known as early binding, static binding, static linking or compile time polymorphism.

3.10 Types of C++ Polymorphism

1. Compile time Polymorphism
2. Runtime Polymorphism



3.11 Function Overloading

A function may take zero or more arguments when called. The number and type of arguments that a function may take is defined in the function itself. If a function call fails to comply by the number and type of argument(s), the compiler reports the same as error. Suppose we write a function named sum to add two numerical values given as arguments. One can write the function as:

```
int sum(int a, int b)
{
return (a + b);
}
```

Now suppose we want the function to take float type argument then the function definition must be changed as:

```
float sumfloat(float a, float b)
{
return (a + b);
}
```

As a matter of fact the function sum may take so many names as shown below.

```
int sumint(int a, int b)
{
return (a + b);
}
short sumshort(short a, short b)
{
return (a + b);
}
long sumlong(long a, long b)
{
return (a + b);
}
float sumdouble(double a, double b)
{
return (a + b);
}
```

This can be very tiring and extremely difficult to remember all the names. Function overloading is a mechanism that allows a single function name to be used for different functions. The compiler does the rest of the job. It matches the argument numbers and types to determine which functions is being called. Thus we may rewrite the above listed functions using function overloading as:

```
int sum(int a, int b)
{
return (a + b);
}
float sum(float a, float b)
{
```

```
return (a + b);
}
short sum(short a, short b)
{
return (a + b);
}
long sum(long a, long b)
{
return (a + b);
}
float sum(double a, double b)
{
return (a + b);
}
```

Overloaded functions have the same name but different number and type of arguments. They can differ either by number of arguments or type of arguments or both. However, two overloaded function cannot differ only by the return type.

 Example

```
#include<iostream>
using namespace std;
class sample{
   public:
int chk_num(){
int a=10;
cout<<"Value  of a is "<< a<<endl;
}
int chk_num(int a){
if(a%2==0)
   cout<<"Number is even" << a <<endl;
   else
      cout<<"Number is odd" << a <<endl;
      }
float chk_num(float x, float y)
{
cout<<"Sum of floating point number is "<< x+y<<endl;
}

};
main(){
```

```
sample obj;
obj.chk_num();
obj.chk_num(15);
obj.chk_num(15.12,25);

}
```

Output

```
Value  of a is 10
Number is odd15
Sum of floating point number is 40.12

Process returned 0 (0x0)   execution time : 4.936 s
Press any key to continue.
```

Summary

An application written in C++ may have a number of classes. One of these classes must contain one (and only one) method called main method. Although a private main method is permissible in C++ it is seldom used. For all practical purposes the main method should be declared as public method. A function may take zero or more arguments when called. The number and type of arguments that a function may take is defined in the function itself. If a function call fails to comply by the number and type of argument(s), the compiler reports the same as error. When any program is compiled the output of compilation is a set of machine language instructions, which is in executable program. When a program is run, this complied copy of program is put into memory. C++ functions can have arguments having default values. The default values are given in the function prototype declaration. Prototype of a function is the function without its body. The C++ compiler needs to about a function before you call it, and you can let the compiler know about a function is two ways – by defining it before you call it or by specifying the function prototypes before you call it.

Keywords

Function: The best way to develop and maintain a large program is to divide it into several smaller program modules of which are more manageable than the original program. Modules are written in C++ as classes and functions. A function is invoked by a function call. The function call mentions the function by name and provides information that the called function needs to perform its task.

Function Declaration: Statement that declares a function's name, return type, number and type of its arguments.

Function Overloading: In C++, it is possible to define several function with the same name, performing different actions. The functions must only differ in their argument lists. Otherwise, function overloading is the process of using the same name for two or more functions.

Function Prototype: A function prototype declares the return-type of the function that declares the number, the types and order of the parameters, the function expects to receive. The function prototype enables the compiler to verify that functions are called correctly.

Inline Function: A function definition such that each call to the function is, in effect, replaced by the statements that define the function.

Self Assessment

1. Which of the following function / types of function cannot have default parameters?

A. Member function of class
B. Main()
C. Member function of structure
D. None of above

2. Function call type is
A. Call by value
B. Call by reference
C. All of Above
D. None of Above

3. Which of the following is the default return value of functions in C++?
A. int
B. char
C. float
D. void

4. Choose the correct statement for call by value
A. Copy of variable is passed.
B. Original value not modified.
C. Actual arguments remain safe as they cannot be modified accidentally.
D. All of Above

5. What is an inline function?
A. A function that is expanded at each call during execution
B. A function that is called during compile time
C. A function that is not checked for syntax errors
D. A function that is not checked for semantic analysis

6. An inline function is expanded during _____
A. compile-time
B. run-time
C. never expanded
D. end of the program

7. What are the two advantage of function objects than the function call?
A. It contains a state
B. It is a type
C. It contains a state & It is a type
D. It contains a prototype

8. Pick out the correct statement.
A. A friend function may be a member of another class
B. A friend function may not be a member of another class
C. A friend function may or may not be a member of another class
D. None of the mentioned

9. Where does keyword 'friend' should be placed?
A. function declaration
B. function definition
C. main function
D. block function

10. Which keyword is used to declare the friend function?
A. firend
B. friend
C. classfriend
D. myfriend

11. Which keyword should be used to declare static variables?
A. static
B. stat

C. common
D. const

12. Which is the correct syntax for declaring static data member?
A. static mamberName dataType;
B. dataType static memberName;
C. memberName static dataType;
D. static dataType memberName;

13. Overloaded functions in C++ oops are
A. Functions preceding with virtual keywords.
B. Functions inherited from base class to derived class.
C. Two or more functions having same name but different number of parameters or type.
D. None of above

14. Function overloading is _____ in C++.
A. Class
B. Object
C. Compile Time Polymorphism
D. None of above

15. What is the output of this program?

```
#include <iostream>

   using namespace std;

   int Add(int X, int Y, int Z)

   {

      return X + Y;

   }

   double Add(double X, double Y, double Z)

   {

      return X + Y;

   }

   int main()

   {

      cout << Add(5, 6);

      cout << Add(5.5, 6.6);

      return 0;

   }
```

A. 11 12.1
B. 12.1 11
C. 11 12
D. Compile time error

Answers for Self Assessment

1.	B	2.	C	3.	A	4.	D	5.	A
6.	A	7.	C	8.	C	9.	A	10.	B
11.	A	12.	D	13.	C	14.	C	15.	D

Review Questions

1. What is function prototyping? Why is it necessary? When is it not necessary?

2. What is purpose of inline function?

3. Differentiate between the following:

   (a) void main()

   (b) int main()

   (c) int main(int argn, char argv[])

4. In which cases will declare a member function to be friend?

5. Write a program that uses overloaded member functions for converting temperature from Celsius to Kelvin scale.

6. To calculate the area of circle, rectangle and triangle using function overloading.

7. Do inline functions improve performance?

8. How can inline functions help with the tradeoff of safety vs. speed?

9. Write a program that demonstrate working of function overloading.

10. Write a program that demonstrates working of inline functions.

Further Readings

- E Balagurusamy; Object Oriented Programming with C++; Tata Mc Graw-Hill.
- Herbert Schildt; The Complete Reference C++; Tata Mc Graw Hill. Robert Lafore;
- Object-oriented Programming in Turbo C++; Galgotia.

Web Links

- https://en.wikipedia.org/wiki/Object-oriented_programming
- www.web-source.net
- www.webopedia.com

Unit 04: Inheritance

CONTENTS

Objectives

After studying this unit, you will be able to:

- Recognize the inheritance
- Describe the different types of inheritance
- Analyze and Explain making private member inheritable

Introduction

The ability to reuse code is a key element of OOP. The concept of reusability is highly supported in C++. The C++ classes can be reused in a variety of ways. Other programmers can use a class once it has been written and tested. The properties Notes of existing classes can be reused to create new classes.

INHERITANCE is the process of creating a new class from an existing one. Because every object of the defined class "is" also an object of the inherited class type, this is sometimes referred to as a "IS-A" relationship. The previous class is known as the 'BASE' class, whereas the new class is known as the 'DERIVED' class or sub-class.

Notes:

- The capability of a class to derive properties and characteristics from another class is called Inheritance.
- Inheritance is a process in which one object acquires all the properties and behaviours of its parent object automatically

4.1 Defining Derived Class

A derived class is specified by defining its relationship with the base class in addition to its owndetails. The general syntax of defining a derived class is as follows:

Class derived class name : access_specifier base class name

{

……

…… // members of derivedclass

}

The colon (:) indicates that the derivedclassname class is derived from the baseclassname class. The access_specifier or the visibility mode is optional and, if present, may be public, private or protected. By default it is private. Visibility mode describes the accessibility status of derived features. For example,

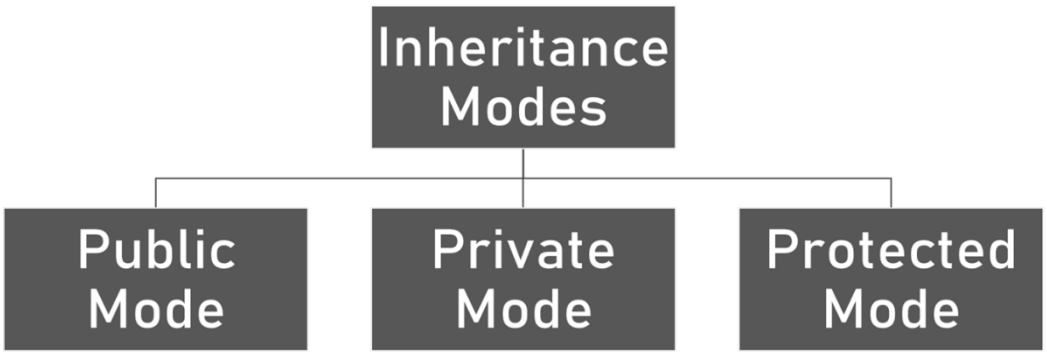
```
class xyz //base class
{
//members of xyz
};
class ABC: public xyz //public derivation
{
//members of ABC
};
class ABC : XYZ //private derivation (by default)
{
//members of ABC
};
```

In inheritance, some of the base class data elements and member functions are inherited into the derived class and some are not. We can add our own data and member functions and thus extend the functionality of the base class.

4.2 Modes of Inheritance

Inheritance Modes:-

1. Public Mode
2. Private Mode
3. Protected Mode



1. **Public Mode:** If we derive a sub class from a public base class. Then the public member of the base class will become public in the derived class and protected members of the base class will become protected in derived class.
2. **Private Mode:** If we derive a sub class from a Private base class. Then both public member and protected members of the base class will become Private in derived class.
3. **Protected Mode:** If we derive a sub class from a protected base class. Then both public member and protected members of the base class will become protected in derived class.

Did you know?

What are the advantages of inheritance?

Inheritance offers the following advantages:

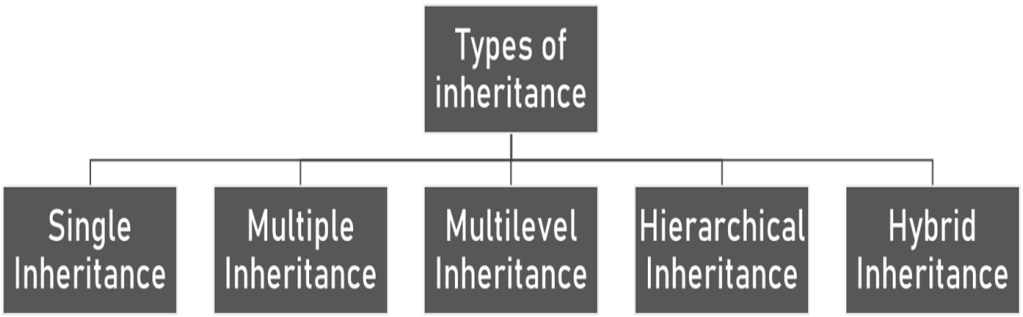
Development model closer to real life object model with hierarchical relationships

Reusability — facility to use public methods of base class without rewriting the same

Extensibility — extending the base class logic as per business logic of the derived class

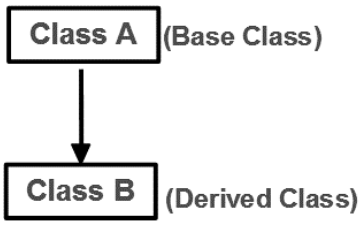
Data hiding — base class can decide to keep some data private so that it cannot be alteredby the derived class.

4.3 Types of Inheritance in C++



1. Single Inheritance

When a class inherits from a single base class, it is referred to as single inheritance.



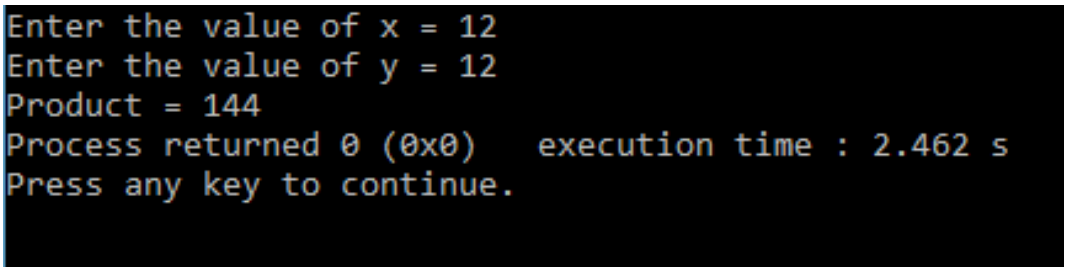
Following program shows working of single inheritance.

Example

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

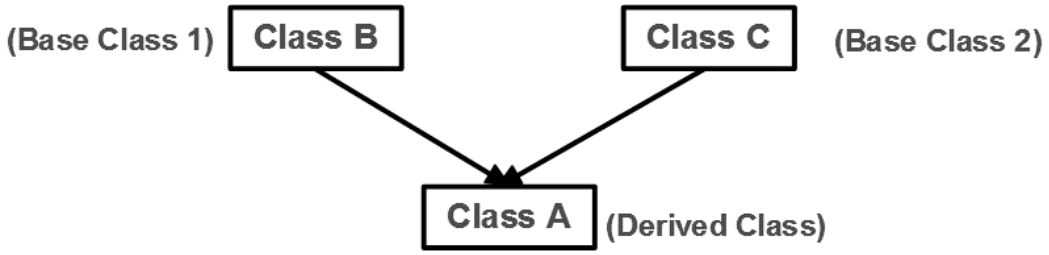
```
{
public:
int x;
voidgetdata()
  {
cout<< "Enter the value of x = "; cin>> x;
  }
 };
class derive : public base
{
private:
int y;
public:
voidreaddata()
  {
cout<< "Enter the value of y = "; cin>> y;
  }
void product()
  {
cout<< "Product = " << x * y;
  }
 };

int main()
 {
derive a;
a.getdata();
a.readdata();
a.product();
return 0;
 }
```

Output



2. Multiple Inheritance

When a class inherits from a two base classes, it is referred to as multiple inheritance.



Following program shows working of multiple inheritance.

Example

```
#include<iostream>
using namespace std;
class A
{
        public:
        int x;
        voidget_data()
  {
cout<< "enter value of x: ";
cin>> x;
  } };
class B
{       public:
        int y;
        void get_data1()
        {
        cout<< "enter value of y: "; cin>> y;
        }   };
class C : public A, public B
{
        public:
        void sum()
        {
        cout<< "Sum = " << x + y;
        }
};
```

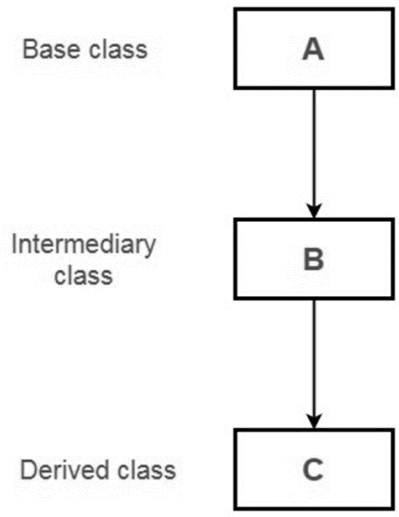
```
int main()
{
        C obj;
      obj.get_data();
       obj.get_data1();
      obj.sum();
      return 0;
}       //end of program
```

Output

```
enter value of x: 25
enter value of y: 35
Sum = 60
Process returned 0 (0x0)   execution time : 2.871 s
Press any key to continue.
```

3. Multilevel Inheritance

If a class is derived from another derived class then it is called multilevel inheritance.



The declaration for the same would be:

```
Class A
{
//body
}
Class B : public A
{
//body
```

```
}
Class C : public B
{
//body
}
```

Following program shows working of multilevel inheritance.

 Example

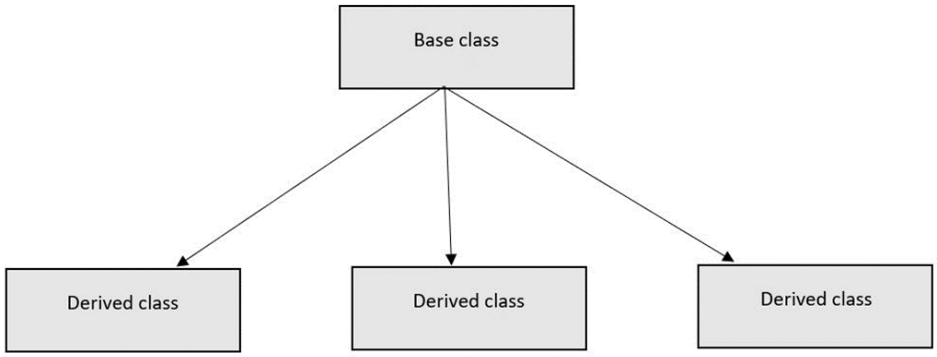
```
#include<iostream>
using namespace std;
class A{
public:
int marks;
voidget_data(){
cout<<"Enter Marks";
cin>>marks;
   }
   };
class B:public A
   {
public:
intshow_data(){
cout<<"Entered Marks: " <<marks;
      }
   };

class C:public B{
   };
main(){
   C obj;
obj.get_data();
obj.show_data();
   }
```

Output

```
Enter Marks50
Entered Marks: 50
Process returned 0 (0x0)   execution time : 2.506 s
Press any key to continue.
```

4. Hierarchical Inheritance

Hierarchical inheritance is a kind of inheritance where more than one class is inherited from a single parent or base class. Especially those features which are common in the parent class is also common with the base class.



Following program shows working of hierarchical inheritance.

Example

```
#include<iostream>
using namespace std;
class A
{
        public:
        intx,y;
        voidget_data()
  {
        cout<< "Enter value of x: ";
        cin>> x;
        cout<<"Enter value of y: ";
        cin>>y;
  }
};
class B:public A
{
```

```
        public:

        voidshow_sum()
        {
        cout<< "Sum of x and y is : "<<x+y<<endl;
        }
};
class C : public A
{
        public:
        voidshow_product()
        {
        cout<< "Product of x and y is : "<<x*y;
        }
};

int main()
{   B obj1;
          C obj2;
         obj1.get_data();
   obj1.show_sum();
         obj2.get_data();
         obj2.show_product();

        return 0;
}
```

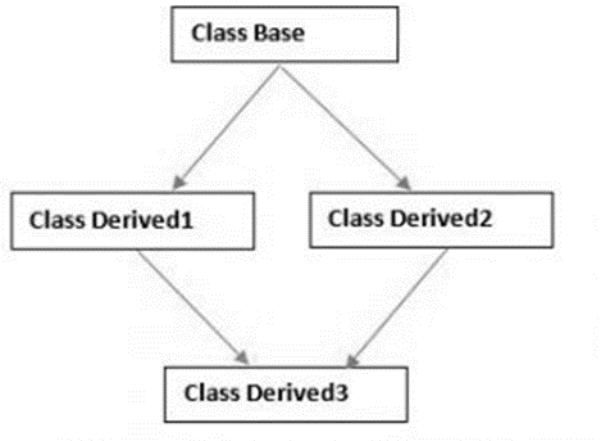
Output

```
Enter value of x: 15
Enter value of y: 2
Sum of x and y is : 17
Enter value of x: 35
Enter value of y: 2
Product of x and y is : 70
Process returned 0 (0x0)   execution time : 5.966 s
Press any key to continue.
```

5. Hybrid Inheritance

There could be situations where we need to apply two or more types of inheritance to design a program. Basically Hybrid Inheritance is the combination of one or more types of the inheritance.

Here is one implementation of hybrid inheritance. Hybrid inheritance is combination of two or more types of inheritance. It is also known as multipath inheritance.



Following program shows working of hybrid inheritance.

Example

```
#include <iostream>
using namespace std;
class A
{
        public:
        int x;
};
class B : public A
{
        public:
        B()
        {
          x = 500;
        }
};
class C
 {
        public:
        int y;
        C()
        {
           y = 100;
     }
};
```

```
class D : public B, public C
{
        public:
        void sum()
        {
        cout<< "Sum= " << x + y;
        }
};

int main()
{
   D obj1;
        obj1.sum();
        return 0;
}
```

Output

```
Sum= 600
Process returned 0 (0x0)   execution time : 0.087 s
Press any key to continue.
```

4.4 Making a Private Member Inheritable

The members of base class which are inherited by the derived class and their accessibility is determined by visibility modes. Visibility modes are:

1. Private: When a base class is privately inherited by a derived class, 'public members' of the base class become private members of the derived class and therefore the publicmembers of the base class can be accessed by its own objects using the dot operator. Theresult is that we have no member of base class that is accessible to the objects of thederived class.

2. Public: On the other hand, when the base class is publicly inherited, 'public members' ofthe base class become 'public members' of derived class and therefore they are accessibleto the objects of the derived class.

3. Protected: C++ provides a third visibility modifier, protected, which serve a little purposein the inheritance. A member declared as protected is accessible by the member functionswithin its class and any class immediately derived from it. It cannot be accessed by functionsoutside these two classes.

Private derivation means that the base class has been inherited privately. Public members and protected members of the base class are private within the derived class.Private members of the base class stay private within the base class.

Syntax

```
classbase_class_name
{
-----
-----
}
```

```
classderived_class_name : private base_class_name
{
----
----
}

//          Program

#include<iostream>
using namespace std;
classemp{
private:
int id;
char name[10];
int salary;
voidget_data(){
cout<<"Enter Id,Name and Salary of  employee"<<endl;
cin>>id>>name>>salary;

   }
public:
voiddisp(){
get_data();
cout<<"Details are"<<endl;
cout<<"Emp ID   "<< id<<endl<<"Emp Name  " <<name  <<endl<<"Emp Salary "<<salary;

   }
};
main(){
empobj;
obj.disp();

}
```

Output

```
Enter Id,Name and Salary of  employee
12
John
15000
Details are
Emp ID   12
Emp Name  John
Emp Salary 15000
Process returned 0 (0x0)   execution time : 7.479 s
Press any key to continue.
```

Summary

Inheritance is the capability of one class to inherit properties from another class.

It supports reusability of code and is able to simulate the transitive nature of real lifeobjects. Inheritance has many forms: Single inheritance, multiple inheritance, hierarchicalinheritance, multilevel inheritance and hybrid inheritance.

A subclass can derive itself publicly, privately or protected. The derived class constructoris responsible for invoking the base class constructor, the derived class can directly accessonly the public and protected members of the base class.

When a class inherits from more than one base class, this is called multiple inheritance.

A class may contain objects of another class inside it. This situation is called nesting of objects and in such a situation, the contained objects areconstructed first before constructing the objects of the enclosing class.

Single Inheritance: Where a class inherits from a single base class, it is known as singleinheritance.

Multilevel Inheritance: When the inheritance is such that the class. A serves as a base classfor a derived class B which is turn serves as a base class for the derived class C. This typeof inheritance is called 'Multilevel Inheritance.

Multiple Inheritance: A class inherit the attributes of two or more classes. This mechanism is known as Multiple Inheritance.'

Hybrid Inheritance: The combination of one or more types of the inheritance.

Keywords

Abstract Class: A class serving only a base class for other classes and no objects of which are created.

Base class: A class from which another class inherits. (Also called super class) Containership: The relationship of two classes such that the objects of a class are enclosed within the other class.

Derived class: A class inheriting properties from another class. (also called sub class).

Inheritance: Capability of one class to inherit properties from another class.

Inheritance Graph: The chain depicting relationship between a base class and derived class.

Visibility Mode: The public, private or protected specifier that controls the visibility and availability of a member in a class.

Self Assessment

1. Inheritance allow in C++ program?

A. Class Re-usability
B. Creating a hierarchy of classes
C. Extendibility
D. All of above

2. Functions that can be inherited from base class in C++ program.

A. Constructor
B. Destructor
C. Static function
D. None of above

3. A class can inherit properties of another class which is known as …….. Inheritance.

A. Single
B. Multiple
C. Multilevel
D. Hierarchical

4. What is the syntax of inheritance of class?
A. class name
B. class name: access specifer
C. class name : access specifer class name
D. None of above

5. Members which are not intended to be inherited are declared as_____
A. Public members
B. Protected members
C. Private Members
D. Private or protected members

6. While inheriting a class, if no access mode is specified, then which among the following is true in C++?
A. It gets inherited publicly by default
B. It gets inherited protected by default
C. It gets inherited privately by default
D. It is not possible.

7. How can you make the private members inheritable?
A. By making their visibility mode as public only
B. By making their visibility mode as protected only
C. By making their visibility mode as private in derived class
D. Can be done both by making the visibility mode public or protected

8. What is meant by multiple inheritance?
A. Deriving a base class from derived class
B. Deriving a derived class from base class
C. Deriving a deriving class from more than one base class
D. None of above

9. Which symbol is used to create multiple inheritance?
A. Dot
B. Comma
C. Dollar
D. None of the above

10. Which among the following best define multilevel inheritance?
A. A class derived from another derived class
B. Classes being derived from another derived class
C. Continuing single level inheritance
D. Class which have more than one parent

11. All the classes must have all the members declared private to implement multilevel inheritance.
A. True
B. False
C. Sometimes true, sometimes false
D. Always false

12. Which among the following is best to defined hierarchical inheritance?
A. More than one classes being derived from one class
B. More than two classes being derived from single base class
C. At most two classes being derived from single base class
D. At most 1 class derived from another class

13. How many classes must be there to implement hierarchical inheritance?
A. Exactly 3
B. At least 3
C. At most 3
D. At least 1

14. Which type of inheritance must be used so that the resultant is hybrid?
A. Multiple
B. Hierarchical
C. Multilevel
D. None of the above

15. What is the minimum number of classes to be there in a program implemented hybrid inheritance?
A. 2
B. 3
C. 4
D. No limit

Review Questions

1. What do you mean by inheritance? Explain different types of inheritance with suitable example.
2. Consider a situation where three kinds of inheritance are involved. Explain this situation with an example.
3. What is the difference between protected and private members?
4. Scrutinize the major use of multilevel inheritance.
5. Discuss a situation in which the private derivation will be more appropriate as compared to public derivation.
6. Write a C++ program to read and display information about employees and managers. Employee is a class that contains employee number, name, address and department. Manager class and a list of employees working under a manager.
7. Differentiate between public and private inheritances with suitable examples.
8. Explain how a sub-class may inherit from multiple classes.
9. What is the purpose of virtual base classes?
10. Write a C++ program that demonstrate working of hybrid inheritance.

Answers for Self Assessment

1.	D	2.	D	3.	A	4.	C	5.	C
6.	C	7.	D	8.	C	9.	B	10.	B
11.	B	12.	A	13.	B	14.	D	15.	D

Further Readings

E Balagurusamy; Object-oriented Programming with C++; Tata McGraw-Hill.

Herbert Schildt; The Complete Reference C++; Tata McGraw Hill.

Robert Lafore; Object-oriented Programming in Turbo C++; Galgotia.

web Links

http://msdn.microsoft.com/en-us/library/wcz57btd(v=vs.80).aspx

http://www.learncpp.com/cpp-tutorial/117-multiple-inheritance/

Unit 05: Operator Overloading

CONTENT

Objectives

After studying this unit, you will be able to:

- Recognize the operator overloading
- Describe the rules for operator overloading
- Explain the overloading of unary operators
- Discuss the various binary operators with friend function and member function

Introduction

C++ comes with a large number of operators. Many of these operators have previously been defined and used in earlier units. Operator overloading is one of C++'s unique features. In a programming language that supports object-oriented features, this characteristic is required.

Overloading an operator simply means attaching additional meaning and semantics to an operator. It enables an operator to exhibit more than one operation polymorphically, as illustrated below:

You're probably aware that the addition operator (+) is primarily a numeric operator, requiring two number operands. It returns a numeric number equal to the product of the two operands. Obviously, this can't be used to join two strings together. We may, however, extend the addition operator's operation to include string concatenation. As a result, the addition operator would work like this:

"PROG"+"RAM"

Should produce a single string

"PROGRAM"

This act of redefining the effect of an operator is called operator overloading. The original meaning and action of the operator however remains as it is. Only an additional meaning is added to it.

Function overloading allows different functions with different argument list having the same name. Similarly an operator can be redefined to perform additional tasks.

Operator overloading is accomplished using a special function, which can be a member function or friend function. The general syntax of operator overloading is:

<return_type> operator <operator_being_overloaded>(<argument list>);

To overload the addition operator (+) to concatenate two characters, the following declaration, which could be either member or friend function, would be needed:

char * operator + (char *s2);

Notes

1. The purpose of operator overloading is to provide a special meaning of an operator for a user-defined data type.
2. Using operator overloading, we can redefine the majority of C++ operator.
3. Using operator overloading, we can specify more than one meaning for an operator in one scope.

5.1 Defining Operator Overloading

The overloading principle applies to both functions and operators in C++. That is, operators can be expanded to deal with classes as well as built-in types. By overloading the built-in operator to execute a specific computation when the operator is used on objects of that class, a programmer can provide his or her own operator to a class. Is it true that operator overloading is useful in real-world applications? It most certainly can, making writing code that seems natural a breeze.

Operator overloading, on the other hand, like any advanced C++ feature, adds to the language's complexity. Furthermore, because operators have a much-defined meaning and most programmers don't expect them to do a lot of work, overloading operators can be exploited to make code incomprehensible. But we're not going to do it.

Did you know?

What is the advantage of operator overloading?

By using operator overloading we can easily access the objects to perform any operations.

An Example of Operator Overloading

Complex a (1.2, 1.3); //this class is used to represent complex numbers

Complex b (2.1, 3); //notice the construction taking 2 parameters for the real and imaginary part

Complex c = a+b; //for this to work the addition operator must be overloaded

The addition without having overloaded operator + could look like this:

Complex c = a.Add(b);

This piece of code is not as readable as the first example though—we're dealing with numbers, so doing addition should be natural. (In contrast to cases when programmers abuse this technique, when the concept represented by the class is not related to the operator—like using + and - to add and remove elements from a data structure. In this cases operator overloading is a bad idea, creating confusion.)

In order to allow operations like Complex c = a+b, in above code we overload the "+" operator. The overloading syntax is quite simple, similar to function overloading, the keyword operator must be followed by the operator we want to overload:

```
class Complex
{
public:
Complex(double re,doubleim)
:real(re),imag(im)
{};
Complex operator+(const Complex& other);
Complex operator=(const Complex& other);
private:
double real;
doubleimag;
};
Complex Complex::operator+(const Complex& other)
{
doubleresult_real = real + other.real;
doubleresult_imaginary = imag + other.imag;
return Complex( result_real, result_imaginary );
}
```

The assignment operator can be overloaded similarly. Notice that we did not have to call any accessor functions in order to get the real and imaginary parts from the parameter other since the overloaded operator is a member of the class and has full access to all private data.

Alternatively, we could have defined the addition operator globally and called a member to do the actual work. In that case, we'd also have to make the method a friend of the class, or use anaccessor method to get at the private data:

```
friend Complex operator+(Complex);
Complex operator+(const Complex &num1, const Complex &num2)
{
doubleresult_real = num1.real + num2.real;
doubleresult_imaginary = num1.imag + num2.imag;
return Complex( result_real, result_imaginary );
}
```

Why would you do this? when the operator is a class member, the first object in the expression must be of that particular type. It's as if you were writing:

```
Complex a( 1, 2 );
Complex a( 2, 2 );
Complex c = a.operator=( b );
```

When it's a global function, the implicit or user-defined conversion can allow the operator to act even if the first operand is not exactly of the same type:

```
Complex c = 2+b; //if the integer 2 can be converted by the Complex
class, this expression is valid
```

By the way, the number of operands to a function is fixed; that is, a binary operator takes two operands, a unary only one, and you can't change it. The same is true for the precedence of operators too; for example the multiplication operator is called before addition. There are some

operators that need the first operand to be assignable, such as : operator=, operator(), operator[] and operator->, so their use is restricted just as member functions (non-static), they can't be overloaded globally. The operator=, operator& and operator, (sequencing) have already defined meanings by default for all objects, but their meanings can be changed by overloading or erased by making them private.

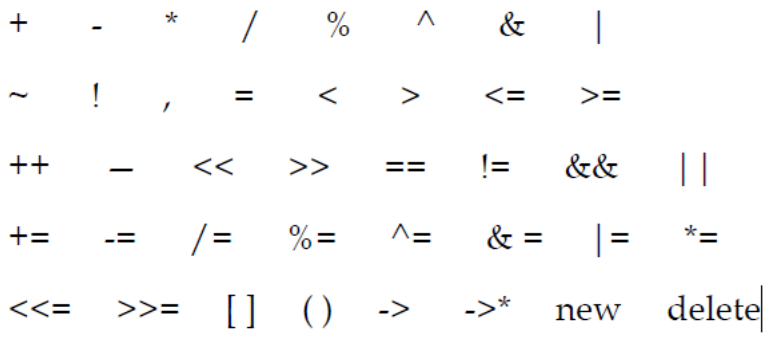
Another intuitive meaning of the "+" operator from the STL string class which is overloaded to do concatenation:

string prefix("de");

string word("composed");

string composed = prefix+word;

Using "+" to concatenate is also allowed in Java, but note that this is not extensible to other classes, and it's not a user defined behavior. Almost all operators can be overloaded in C++:



The only operators that can't be overloaded are the operators for scope resolution (::), member selection (.), and member selection through a pointer to a function(.*). Overloading assumes you specify a behavior for an operator that acts on a user defined type and it can't be used just with general pointers. The standard behavior of operators for built-in (primitive) types cannot be changed by overloading, that is, you can't overload operator+(int,int).

will not evaluate all three operations and will stop after a false one is found. This behavior does not apply to operators that are overloaded by the programmer.

Even the simplest C++ application, like a "hello world" program, is using overloaded operators. This is due to the use of this technique almost everywhere in the standard library (STL). Actually the most basic operations in C++ are done with overloaded operators, the IO(input/output) operators are overloaded versions of shift operators(<<, >>). Their use comes naturally to many beginning programmers, but their implementation is not straightforward. However a general format for overloading the input/output operators must be known by any C++ developer. We will apply this general form to manage the input/output for our Complex class:

```
friendostream&operator<<(ostream&out, Complex c) //output
{
out<<"real part: "<<real<<"\n";
out<<"imag part: "<<imag<<"\n";
return out;
}
friendistream&operator>>(istream&in, Complex &c) //input
{
cout<<"enter real part:\n";
in>>c.real;
cout<<"enter imag part: \n";
in>>c.imag;
```

return in;

}

A important trick that can be seen in this general way of overloading IO is the returning reference for istream/ostream which is needed in order to use them in a recursive manner:

Complex a(2,3);

Complex b(5.3,6);

cout<<a<<b;

5.2 Rules for Overloading Operator

To overload any operator, we must first comprehend the rules that apply. Let's go over some of the ones that have already been discussed.

Notes

1. For it to work, at least one operand must be a user-defined class object.
2. We can only overload existing operators. We can't overload new operators.
3. Some operators cannot be overloaded using a friend function. However, such operators can be overloaded using member function.

Following are the operators that cannot be overloaded.

Operator	Purpose
.	Class member access operator
.*	Class member access operator
::	Scope Resolution Operator
?:	Conditional Operator
Sizeof	Size in bytes operator
#	Preprocessor Directive
=	Assignment operator
0	Function call operator
0	Subscripting operator
->	Class member access operator

1. Operators already predefined in the C++ compiler can be only overloaded. Operator cannot change operator templates that is for example the increment operator ++ is used only as unary operator. It cannot be used as binary operator.

2. Overloading an operator does not change its basic meaning. For example assume the + operator can be overloaded to subtract two objects. But the code becomes unreachable.

class integer

{intx, y;

public:

int operator + ();

}

int integer: : operator + ( )

{

return (x-y);

}

3. Unary operators, overloaded by means of a member function, take no explicit argument and return no explicit values. But, those overloaded by means of a friend function take one reference argument (the object of the relevant class).

4. Binary operators overloaded through a member function take one explicit argument and those which are overloaded through a friend function take two explicit arguments.

Operator to Overload	Arguments passed to the Member Function	Argument passed to the Friend Function
Unary Operator	No	1
Binary Operator	1	2

5. Overloaded operators must either be a non-static class member function or a global function. A global function that needs access to private or protected class members must be declared as a friend of that class. A global function must take at least one argument that is of class or enumerated type or that is a reference to a class or enumerated type.

5.3 Overloading Unary Operators

When a unary operator is overloaded with a member function, no arguments are provided to the function, but a friend function requires only one input.

Did you know? **Unary operators operate on a single operand.**

Example

- **The increment (++) and decrement (--) operators.**
- **The unary minus (-) operator.**
- **The logical not (!) operator.**

// Program

```
#include<iostream>
using namespace std;
class sample{
private:
int x;
public:
sample(){
x=0;
}
void operator ++ (){
++x;
```

```
}
voiddisp(){
cout<<"Value of X is"<< x <<endl;
}
};
main(){
sampleobj;
obj.disp();
++obj;
obj.disp();
}
```

Output

```
Value of X is0
Value of X is1

Process returned 0 (0x0)   execution time : 0.052 s
Press any key to continue.
```

5.4 Overloading Binary Operator

Those operators which operate on two operands or data are called binary operators.Binary operators can be overloaded using C++.This will be better understood by means of the following program.

//Program

```
#include<iostream>
using namespace std;
class Complex{
inta,b;
public:
voidget_data(){

cout<<"Enter the value of complex numbers a and b";
cin>>a>>b;
  }
Complex operator +(Complex ob){
Complex c;
c.a=a+ob.a;
c.b=b+ob.b;
return (c);
}
void display()
```

```
{
cout<< a << "+" << b << "i" << "\n";
}
};
main(){
Complex obj1,obj2,result;
obj1.get_data();
obj1.display();
obj2.get_data();
result=obj1+obj2;
result.display();
}
```

Output

```
Enter the value of complex numbers a and b 12 2
12+2i
Enter the value of complex numbers a and b 12 2
24+4i

Process returned 0 (0x0)   execution time : 6.215 s
Press any key to continue.
```

5.5 Using Friend Function

- Friend function using operator overloading offers better flexibility to the class.
- These functions are not a members of the class and they do not have 'this' pointer.
- Overload a unary operator you have to pass one argument.
- Overload a binary operator you have to pass two arguments.
- Friend function can access private members of a class directly.

Now we are going to discuss what is a friend function in C++a friend function of a class is defined outside, and all of the private members we can access usingthe friend function outside the class prototype for friend functions appear in the class definition.If we are going for the return type of the function, we are going to use a friend keyword thereand after that we are writing the return type of the function, and friend function can access all theprivate members of the class also outside the class using the keyword friend compiler knows thegiven function is friend function by when we are using to the declared keyword there, itrepresents to the compiler that we are using the friend function in our code. Now we have someof the using friend functions in C++ why we are using the friend function and why friendfunctions are used to access the members in C++.

Notes

A friend function of a class is defined outside that class' scope but it has the right to access all private and protected members of the class.

```
//Program
#include <iostream>
```

```
using namespace std;
class sample
{
private:
int number;
public:
sample(): number(0) { }
friendintprintNumber(sample);
};
intprintNumber(sample s)
{
s.number += 100;
returns.number;
}
int main()
{
sample s;
cout<<"Entered Number is : "<<printNumber(s)<<endl;
return 0;
}
```

Output

```
Entered Number is : 100

Process returned 0 (0x0)   execution time : 0.013 s
Press any key to continue.
```

Task: Analyze the uses of friend function in operator overloading.

5.6 Using Member Function

You learned in the unit on overloading the arithmetic operators that it's ideal to implement the overloaded operator as a friend function of the class when it doesn't affect its operands. We usually overload the operator with a class member function for operators that modify their operands.

Overloading operators using a member function is very similar to overloading operators usinga friend function. When overloading an operator using a member function:

1. The leftmost operand of the overloaded operator must be an object of the class type.

2. The leftmost operand becomes the implicit *this parameter. All other operands becomefunction parameters.

Most operators can actually be overloaded either way, however there are a few exception cases:

3. If the leftmost operand is not a member of the class type, such as when overloadingoperator+(int, YourClass), or operator<<(ostream&, YourClass), the operator must beoverloaded as a friend.

4. The assignment (=), subscript ([]), call (()), and member selection (->) operators must beoverloaded as member functions.

Overloading the unary negative (–) operator

The negative operator is a unary operator that can be implemented using either method. Beforee show you how to overload the operator using a member function, here's a reminder of howwe overloaded it using a friend function:

```
class Cents
{
private:
intm_nCents;
public:
Cents(intnCents) { m_nCents = nCents; }
// Overload -cCents
friend Cents operator-(const Cents &cCents);
};
// note: this function is not a member function!
Cents operator-(const Cents &cCents)
{
return Cents(-cCents.m_nCents);
}
```

Now let's overload the same operator using a member function instead:

```
class Cents
{
private:
intm_nCents;
public:
Cents(intnCents) { m_nCents = nCents; }
// Overload -cCents
Cents operator-();
};
// note: this function is a member function!
Cents Cents::operator-()
{
return Cents(-m_nCents);
}
```

 Caution: Remember that when C++ sees the function prototype Cents Cents::operator-();,the compiler internally converts this to Cents operator-(const Cents *this), which you willnote is almost identical to our friend version Cents operator-(const Cents &cCents)!

You'll notice that this procedure is very similar to the previous one. The member function version of operator, on the other hand, does not take any parameters! What happened to the parameter? You learnt that a member function has an implicit *this pointer that always points to the class object the member function is working on in the course on the hidden this pointer. In the member function version, the parameter we had to explicitly list in the friend function version (which doesn't have a *this pointer) becomes the implicit *this parameter.

Overloading the binary addition (+) operator

Let's take a look at an example of a binary operator overloaded both ways. First, overloadingoperator+ using the friend function:

```
class Cents
{
private:
intm_nCents;
public:
Cents(intnCents) { m_nCents = nCents; }
// Overload cCents + int
friend Cents operator+(Cents &cCents, intnCents);
intGetCents() { return m_nCents; }
};
// note: this function is not a member function!
Cents operator+(Cents &cCents, intnCents)
{
return Cents(cCents.m_nCents + nCents);
}
```

Now, the same operator overloaded using the member function method:

```
class Cents
{
private:
intm_nCents;
public:
Cents(intnCents) { m_nCents = nCents; }
// Overload cCents + int
Cents operator+(intnCents);
intGetCents() { return m_nCents; }
};
// note: this function is a member function!
Cents Cents::operator+(intnCents)
{
return Cents(m_nCents + nCents);
}
```

Our two-parameter friend function becomes a one-parameter member function, because theleftmost parameter (cCents) becomes the implicit *this parameter in the member function version.

Most programmers find the friend function version easier to read than the member functionversion, because the parameters are listed explicitly. Furthermore, the friend function versioncan be used to overload some things the member function version cannot. For example, friendoperator+(int, cCents) cannot be converted into a member function because the leftmost parameteris not a class object.

However, when dealing with operands that modify the class itself (eg. operators =, +=, -=, ++,-, etc…) the member function method is typically used because C++ programmers are used towriting member functions (such as access functions) to modify private member variables. Writingfriend functions that modify private member variables of a class is generally not consideredgood coding style, as it violates encapsulation.

5.7 Manipulation of Strings using Operator Overloading

ANSI C implements strings using character assays, pointers and string functions. There are nooperators for manipulating the strings. There is no direct operator that could act upon thestrings. Although, these limitations exist in C++ as well, it permits us to create our own definitionsof operators that can be used to manipulate the strings very much similar to the decimal numbersfor e.g. we shall be able to use the statement like

String3 = string1 + string2;

The following program to overload the '+' operator to append one string to another

```
//Program
#include<iostream.h>
#include<conio.h>
classstr
{
char *name;
public:
str();
str(char *);
void get();
void show();
str operator + (str);
};

str::str() Notes
{
name = new char[1];
}
str::str(char *a)
{
inti = strlen(a);
name = new char[i + 1];
strcpy(name, a);
}
```

```
strstr::operator + (str s)
{
inti = strlen(name) + strlen(s.name);
strtmp;
tmp.name = new char[i+1];
strcpy(tmp.name, name);
strcat(tmp.name, s.name);
returntmp;
}
voidstr::get()
{
cin>> name;
}
voidstr::show()
{
cout<< name;
}
void main()
{
clrscr();
str S3;
str S1("hello");
str S2("monty");
S3 = S1 + S2;
S3.show();
}
```

Summary

- In this unit, we have seen how the normal C++ operators can be given new meaningswhen applied to user-defined data types.
- The keyword operator is used to overload an operator, and the resulting operator willadopt the meaning supplied by the programmer.
- Closely related to operator overloading is the issue of type conversion. Some conversionstake place between user defined types and basic types.
- We have seen that how friend function is working with different situations.
- Two approaches are used in such conversion: A one argument constructor changes a basictype to a user defined type, and a conversion operator converts a user-defined type to abasic type.
- When one user-defined type is converted to another, either approach can be used.

Keywords

Operator Overloading: Attaching additional meaning and semantics to an operator. It enablesto exhibit more than one operations polymorphically.

Strings: The C++ strings library provides the definitions of the basic_string class, which is a classtemplate specifically designed to manipulate strings of characters of any character type.

Unary Operators: Unary operators operate on one operand (variable or constant). There are twotypes of unary operators- increment and decrement.

Self Assessment

1. Which is the correct statement about operator overloading in C++?

A. Only arithmetic operators can be overloaded

B. Associativity and precedence of operators does not change

C. Precedence of operators are changed after overloading

D. Only non-arithmetic operators can be overloaded

2. Which of the following operators cannot be overloaded?

A. .* (Pointer-to-member Operator)

B. :: (Scope Resolution Operator)

C. .* (Pointer-to-member Operator)

D. All of the above

3. Operator overloading is also called …….. polymorphism.

A. Run time

B. Initial time

C. Compile time

D. Completion time

4. Which one is not unary operator

A. @

B. ++

C. –

D. - , !

5. …… overloaded by means of a member function, take no explicit arguments and return no explicit values.

A. Unary operators

B. Binary operators

C. Arithmetic operators

D. Function operator

6. Those operators which operate on two operands or data are called binary operators.

A. True

B. False

7. we can define a binary operator as :

A. The operator that performs its action on two operand
B. The operator that performs its action on three operand
C. The operator that performs its action on any number of operands
D. The operator that performs its action on a single operand

8. Correct example of a binary operator is _____.

A. –
B. +
C. ++
D. Dereferencing operator (*)

9. Overloading the addition (+) operator is correct function name of :

A. operator (+).
B. operator_+.
C. operator+.
D. operator:+.

10. What is the syntax of overloading operator + for class A?

A. A operator+(argument_list){}
B. A operator[+](argument_list){}
C. int +(argument_list){}
D. int [+](argument_list){}

11. What is a friend function in C++?

A. A function which can access all the private, protected and public members of a class
B. A function which is not allowed to access any member of any class
C. A function which is allowed to access public and protected members of a class
D. A function which is allowed to access only public members of a class

12. Pick the correct statement.

A. Friend functions are in the scope of a class
B. Friend functions can be called using class objects
C. Friend functions can be invoked as a normal function
D. Friend functions can access only protected members not the private members

13. Which keyword is used to represent a friend function?

A. friend
B. Friend
C. friend_func
D. Friend_func

14. What is the syntax of friend function?

A. friend class1 Class2;
B. friend class;
C. friend class
D. friend class()

15. Which operator is overload in following program

```
#include <iostream>
using namespace std;
class sample {
  int x, y;
public:
  sample() {}
  sample(intsx, intsy) {
    x = sx;
    y = sy;
  } void
show() {
    cout<< x << " ";
    cout<< y << "\n";
  }
friend sample operator+(sample ob1, sample ob2);
};
sample operator+(sample ob1, sample ob2)
{
  sample temp;

  temp.x = ob1.x + ob2.x;
  temp.y = ob1.y + ob2.y;

  return temp;
}
int main()
{
  sample ob1(10, 20), ob2( 5, 30);
  ob1 = ob1 + ob2;
  ob1.show();
  return 0;
}
```

A. Overloading function
B. Binary operator overloading using friend function
C. Unary operator overloading
D. None of above

Answers for Self Assessment

1.	B	2.	D	3.	C	4.	A	5.	A
6.	A	7.	B	8.	B	9.	A	10.	A

11. A 12. C 13. A 14. A 15. B

Review Questions

1. Overload the addition operator (+) to assign binary addition. The following operationshould be supported by +.

110010 + 011101 = 1001111

2. Write a program that demonstrate working of binary operator overloading.

3. Which operators are not allowed to be overloaded?

4. What are the differences between overloading a unary operator and that of a binaryoperator? Illustrate with suitable examples.

5. Why is it necessary to convert one data type to another? Illustrate with suitable examples.

6. How many arguments are required in the definition of an overloaded unary operator?

7. When used in prefix form, what does the overloaded + + operator do differently fromwhat it does in postfix form?

8. Explain in detail manipulation of strings using operator overloading.

9. Write a note on unary operators.

10. What are the various rules for overloading operators?

Further Readings

E Balagurusamy; Object-oriented Programming with C++; Tata McGraw-Hill.

Herbert Schildt; The Complete Reference C++; Tata McGraw Hill.

Robert Lafore; Object-oriented Programming in Turbo C++; Galgotia.

Web Links

http://msdn.microsoft.com/en-us/library/wcz57btd(v=vs.80).aspx

http://www.learncpp.com/cpp-tutorial/117-multiple-inheritance/

https://www.codecademy.com/learn/learn-c-plus-plus

Unit 06: Type Conversion

CONTENTS

Objectives

After studying this unit, you will be able to:

- Recognize the type conversions
- Describe the basic type to class type
- Explain the class type to basic type
- Discuss the class type to another type

Introduction

It is the transformation of one type into another. Type casting, in other terms, is the process of transforming an expression of one type into another.

Conversion of variables from one type to another are known as type conversion. Type conversions ultimate aim is to make variables of one data type work with variables of another data type.

Type conversions can be used to force the correct type of mathematical computation needed to be performed.

Depending on whether the type conversion is ordered by the programmer or the compiler, it can be explicit or implicit. When a programmer wants to go around the compiler's typing system, he or she uses explicit type conversions (casts); however, the programmer must use them appropriately to succeed. Problems that the compiler avoids may develop, such as when the processor requires data of a certain type to be stored at specific addresses or when data is truncated because a data type on a given platform does not have the same size as the original type. Explicit type conversions between objects of different kinds result in difficult-to-read code at best.

6.1 Type Conversion

Constants and variables in a mixed expression are of distinct data kinds. According to certain rules, assignment operations cause automatic type conversion between the operands.

The data type to the right of an assignment operator is transformed to the data type of the variable on the left automatically.

Example

int a = 45;

float b= 1253.25;

a=b;

This converts float variable b to an integer before its value assigned to a. The type conversion is automatic as far as data types involved are built in types. We can also use the assignment operator in case of objects to copy values of all data members of right hand object to the object on left hand. The objects in this case are of same data type.

Different types of Type Conversion

1. Implicit type conversion
2. Explicit type conversion

Implicit type conversion

Done by the compiler on its own, without any external trigger from the user. Generally takes place when in an expression more than one data type is present. In such condition type conversion takes place to avoid loss of data.

Lab Excersise

```
//Program
#include <iostream>
using namespace std;
int main()
{
        int x = 100;
        char y = 'a';
        x = x + y;
float z = x + 1.0;
        cout << "x = " << x << endl
                << "y = " << y << endl
                << "z = " << z << endl;
        return 0;
}
```

Output

```
x = 197
y = a
z = 198

Process returned 0 (0x0)   execution time : 0.030 s
Press any key to continue.
```

Explicit type conversion

This process is user defined, also called type casting.

Lab Exercise

```
//Program
#include <iostream>
using namespace std;
int main()
{
   double x = 25.5;
   int sum = (int)x + 1;
  cout << "Sum = " << sum;
   return 0;
}
```

Output

```
Sum = 26
Process returned 0 (0x0)   execution time : 0.099 s
Press any key to continue.
```

There are three types of situations that arise where data conversion are between incompatible types. Possible Type Conversions in C++

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

6.2 Basic Type to Class Type

A constructor was used to build a matrix object from an int type array. Similarly, we used another constructor to build a string type object from a char* type variable. In these examples constructors performed a defect type conversion from the argument's type to the constructor's class type

Consider the following constructor:

```
string::string(char*a)
{
length = strlen(a);
name=new char[len+1];
strcpy(name,a);
}
```

This constructor builds a string type object from a char* type variable a. The variables length and name are data members of the class string. Once you define the constructor in the class string, it can be used for conversion from char* type to string type.

Example

```
string s1, s2;
```

char* name1 = "Good Morning";

char* name2 = "STUDENTS" ;

s1 = string(name1);

s2 = name2;

The program statement

s1 = string (name1);

first converts name1 from char* type to string type and then assigns the string type values to the object s1. The statement

s2 = name2;

performs the same job by invoking the constructor implicitly.

Consider the following example

class time

{

int hours;

int minutes;

public:

time (int t) II constructor

{

hours = t / 60; //t is inputted in minutes

minutes = t % 60; .

}

};

In the following conversion statements:

time T1; //object T1 created

int period = 160;

T1 = period; //int to class type

The object T1 is created. The variable period of data type integer is converted into class type time by invoking the constructor. After this conversion, the data member hours of T1 will have value 2 arid minutes will have a value of 40 denoting 2 hours and 40 minutes.

In both the examples, the left-hand operand of = operator is always a class object. Hence, we can also accomplish this conversion using an overloaded = operator.

 Lab exercise

// Program - Using Constructor

#include<iostream>

using namespace std;

class Time

{

int hrs,min;

```
public:
Time(int t)
{
cout<<"Basic Type to Class Type Conversion...\n";
hrs=t/60;
min=t%60;
}
void show();
};
void Time::show()
{
cout<<hrs<< ": Hours(s)" <<endl;
cout<<min<< " Minutes" <<endl;
}
int main()
{
int duration;
cout<<"\nEnter time duration in minutes";
cin>>duration;
Time t1(duration);
t1.show();
return 0;
}
```

Output

```
Enter time duration in minutes850
Basic Type to Class Type Conversion...
14: Hours(s)
10 Minutes

Process returned 0 (0x0)   execution time : 1.778 s
Press any key to continue.
```

Lab exercise

```
// Program - Using Operator
#include<iostream>
using namespace std;
class Time
{
```

```
int hrs,min;
public:
void display()
{
cout<<hrs<< ": Hour(s)\n";
cout<<min<<": Minutes\n";
}
void operator =(int t)
{
cout<<"\nBasic Type to Class Type Conversion...\n";
hrs=t/60;
min=t%60;
}
};
int main()
{
Time t1;
int duration;
cout<<"Enter time duration in minutes";
cin>>duration;
cout<<"object t1 overloaded assignment..."<<endl;
t1=duration;
t1.display();
cout<<"object t1 assignment operator 2nd method..."<<endl;
t1.operator=(duration);
t1.display();
return 0;
}
```

Output

```
Enter time duration in minutes521
object t1 overloaded assignment...

Basic Type to Class Type Conversion...
8: Hour(s)
41: Minutes
object t1 assignment operator 2nd method...

Basic Type to Class Type Conversion...
8: Hour(s)
41: Minutes

Process returned 0 (0x0)   execution time : 4.971 s
Press any key to continue.
```

Notes

- In this conversion source type is basic type and the destination type is class type.
- Basic data type is converted into class type.

6.3 Class Type to Basic Type

The constructor functions do not support conversion from a class to basic type. C++ allows us to define a overloaded casting operator that convert a class type data to basic type. The general form of an overloaded casting operator function, also referred to as a conversion function, is:

```
operator typename()
{
//Program statement
}
```

This function converts a class type data to type name. For example, the operator double() converts a class object to type double, in the following conversion function:

```
vector:: operator double()
{
double sum = 0;
for(int I = 0; i<size; i++)
sum = sum + v[i] * v[i ]; //scalar magnitude
return sqrt(sum);
}
```

Did you know?

Conversion function must be a class member.

Conversion function must not specify the return value even though it returns the value.

Conversion function must not have any argument.

In the string example discussed earlier, we can convert the object string to char* as follows:

```
string::operator char*()
{
retum(str);
}
```

Did you know?

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

Lab exercise

```
// Program

#include<iostream>
using namespace std;
class Time
{
int h,m;
public:
Time(int a,int b)
{
h=a;
m=b;
}
operator int()
{
cout<<"\nClass Type to Basic Type Conversion...";
return(h*60+m);
}
~Time()
{
cout<<"\nDestructor called..."<<endl;
}
};
int main()
{
int h,m,duration;
cout<<"\nEnter Hours ";
cin>>h;
cout<<"\nEnter Minutes ";
cin>>m;
Time t(h,m);
duration = t;
cout<<"\nTotal Minutes are "<<duration;
cout<<"\n2nd method operator overloading ";
duration = t.operator int();
cout<<"\nTotal Minutes are "<<duration;
return 0;
}
```

Output

```
Enter Hours 850

Enter Minutes 1200

Class Type to Basic Type Conversion...
Total Minutes are 52200
2nd method operator overloading
Class Type to Basic Type Conversion...
Total Minutes are 52200
Destructor called...

Process returned 0 (0x0)   execution time : 4.923 s
Press any key to continue.
```

Notes

Class type to basic type conversion requires special casting operator function for class type to basic type conversion. This is known as the conversion function.

6.4 Class Type to another Class Type

We have just seen data conversion techniques from a basic to class type and a class to basic type. But sometimes we would like to convert one class data type to another class type.

Example

Obj1 = Obj2 ; //Obj1 and Obj2 are objects of different classes.

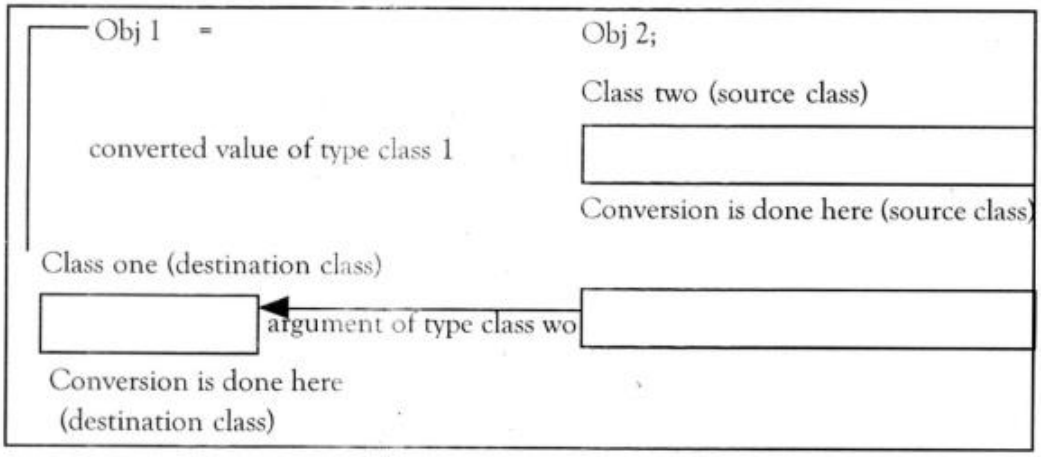
Obj1 is an object of class one and Obj2 is an object of class two. The class two type data is converted to class one type data and the converted value is assigned to the Obj1. Since the conversion takes place from class two to class one, two is known as the source and one is known as the destination class.

Such conversion between objects of different classes can be carried out by either a constructor or a conversion function. Which form to use, depends upon where we want the type-conversion function to be located, whether in the source class or in the destination class.

We studied that the casting operator function

Operator typename()

The following Figure illustrates the above two approaches.



The following Table summarizes all the three conversions. It shows that the conversion from a class to any other type (or any other class) makes use of a casting operator in the source class. To perform the conversion from any other type or class to a class type, a constructor is used in the destination class.

Conversion	Conversion takes place in	
	Source class	Destination class
Basic → class	Not applicable	Constructor
Class → Basic	Casting operator	Not applicable
Class → class	Casting operator	Constructor

When a conversion using a constructor is performed in the destination class, we must be able to access the data members of the object sent (by the source class) as an argument.

Lab Exercise

```
// Program – Using Constructor
#include<iostream>
using namespace std;
class Time
{
int hrs,min;
public:
Time(int h,int m)
{
hrs=h;
min=m;
```

```
}
Time()
{
cout<<"\n Time's Object Created";
}

int getMinutes()
{
int tot_min = ( hrs * 60 ) + min ;
return tot_min;
}

void display()
{
cout<<"Hours: "<<hrs<<"\n";
cout<<" Minutes : "<<min <<"\n";
}
};
class Minute
{
int min;
public:
Minute()
{
min = 0;
}
void operator=(Time T)
{
min=T.getMinutes();
}
void display()
{
cout<<"\n Total Minutes : " <<min<<"\n";
}
};
int  main()
{
Time t1(1,20);
t1.display();
Minute m1;
m1.display();
```

```
m1 = t1;
t1.display();
m1.display();
return 0;
}
```

Output

```
Hours: 1
 Minutes : 20

 Total Minutes : 0
Hours: 1
 Minutes : 20

 Total Minutes : 80

Process returned 0 (0x0)   execution time : 0.015 s
Press any key to continue.
```

 Lab Exercise

```
//Program – Using conversion function
#include<iostream>
using namespace std;
class inventory1
{
int ino,qty;
float rate;
public:
inventory1(int n,int q,float r)
{
ino=n;
qty=q;
rate=r;
}
int getino()
{
return(ino);
}
float getamt()
{
return(qty*rate);
}
```

```
void display()
{
cout<<"\nino = "<<ino<<" qty = "<<qty<<" rate = "<<rate;
}
};
class inventory2
{
int ino;
float amount;
public:
void operator=(inventory1 I)
{
ino=I.getino();
amount=I.getamt();
}
void display()
{
cout<<"\nino = "<<ino<<" amount = "<<amount;
}
};
int main()
{
inventory1 I1(101,50,45);
inventory2 I2;
I2=I1;
I1.display();
I2.display();
}
```

```
ino = 101 qty = 50 rate = 45
ino = 101 amount = 2250
Process returned 0 (0x0)   execution time : 0.224 s
Press any key to continue.
```

Summary

- A type conversion may either be explicit or implicit, depending on whether it is ordered by the programmer or by the compiler. Explicit type conversions (casts) are used when a programmer want to get around the compiler's typing system; for success in this endeavour, the programmer must use them correctly.
- Used another constructor to build a string type object from a char* type variable.

- The general form of an overloaded casting operator function, also referred to as a conversion function, is:

```
operator typename()
{
//Program statement
}
```

- Which form to use, depends upon where we want the type-conversion function to be located, whether in the source class or in the destination class.

Keywords

Implicit Conversion: An implicit conversion sequence is the sequence of conversions required to convert an argument in a function call to the type of the corresponding parameter in a function declaration.

Explicit Conversion:

Operator Typename(): Converts the class object of which it is a member to typename.

Self Assessment

1. Following program is an example of _____ conversion.

```
#include <iostream>
using namespace std;
int main()
{
    int x = 100;
    char y = 'a';
    x = x + y;
float z = x + 1.0;
    cout << "x = " << x << endl
      << "y = " << y << endl
      << "z = " << z << endl;
    return 0;
}
```

A. Implicit
B. Explicit
C. Both
D. None of Above

2. What is type casting?

A. Converting one function into another
B. Converting one data type into another
C. Converting operator type to another type

D. None of them

3. Choose the correct syntax for explicit conversion.

A. Explicit (type)

B. (type) expression;

C. Expression (explicit)

D. None of Above

4. Who carries out implicit type casting?

A. The Micro Controller

B. The Compiler

C. The Programmer

D. The User

5. Who initiates explicit type casting?

A. The Micro Controller

B. The Compiler

C. The Programmer

D. The User

6. What will be the data type of the result of the following operation?

(float)a * (int)b / (long)c * (double)d

A. int

B. long

C. float

D. double

7. When double is converted to float, the value is?

A. Truncated

B. Rounded

C. Depends on the compiler

D. Depends on the standard

8. Which of the following type conversion is not possible in C++?

A. Basic to Class type

B. Class to Basic type

C. One Class to another class type

D. Inheritance to inheritance

9. Which of the following is correct statement for class to basic type conversion?

A. Class type to basic type conversion never performed

B. In this conversion source type is class type and the destination type is basic type.

C. Class type to basic type conversion acts like data type

D. None of above

10. Conversion function _____.

A. must be a class member

B. must not have any argument

C. All of above

D. None of above

11. Conversion function must not specify the return value even though it returns the value.

A. True

B. False

12. To convert from a user defined class to a basic type, you would most likely use.

A. Built-in conversion function

B. A one-argument constructor

C. A conversion function that's a member of the class

D. An overloaded '='operator

13. How many ways to perform conversion from one class to another class can perform?

A. 4

B. 2

C. 3

D. 1

14. ____ refers to the process of changing the data type of the value stored in a variable.

A. Type char

B. Type int

C. Type float

D. Type cast

15. Which of the following type-casting have chances for wrap around?

A. From int to float

B. From int to char

C. From char to short

D. From char to int

Answers for Self Assessment

1.	A	2.	B	3.	B	4.	B	5.	C
6.	D	7.	C	8.	D	9.	B	10.	C
11.	A	12.	C	13.	B	14.	D	15.	B

Review Questions

1. What do you mean by type casting? Explain the difference between implicit and explicit type casting in detail.
2. How type conversion occurs in a program. Explain with suitable example.
3. Write a program that demonstrate working of explicit type conversion.

4. The assignment operations cause automatic type conversion between the operand as per certain rules. Describe.
5. Write a program that demonstrate working of implicit type conversion.
6. What is class to another class type conversion?
7. List the situations in which we need class type to basic type conversion.
8. How to convert one data type to another data type in C++. Explain in detail.
9. Write a program which the conversion of class type to basic type conversion.
10. There are three types of situations that arise where data conversion are between incompatible types. What are three situations explain briefly.

Further Readings

E Balagurusamy; Object-oriented Programming with C++; Tata Mc Graw-Hill.

Herbert Schildt; The Complete Reference C++; Tata Mc Graw Hill.

Robert Lafore; Object-oriented Programming in Turbo C++; Galgotia.

Web Links

http://msdn.microsoft.com/en-us/library/wcz57btd(v=vs.80).aspx

http://www.learncpp.com/cpp-tutorial/117-multiple-inheritance/

https://www.codecademy.com/learn/learn-c-plus-plus

Unit 07: Run-time Polymorphism

CONTENTS

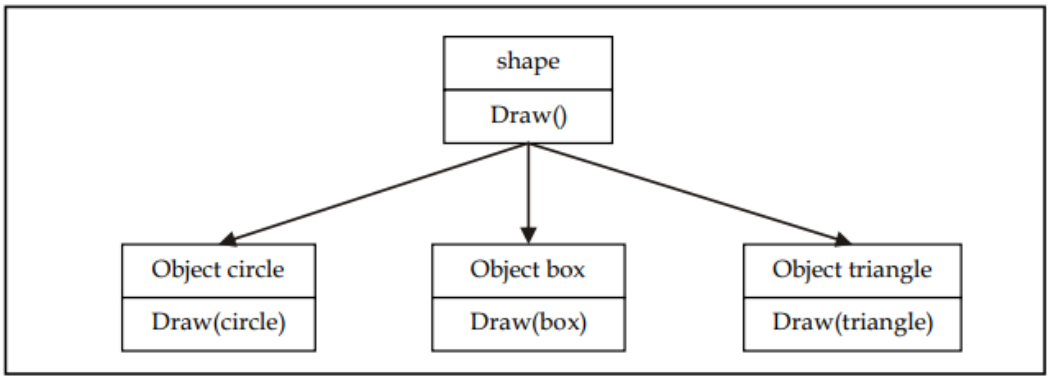
Objectives

After studying this unit, you will be able to:

- Understand run time polymorphism
- Describe the virtual base class, abstract class
- Recognize pointer to object
- Explain this pointer
- Analyze pointer to derived class

Introduction

Polymorphism is a key concept in OOP. Polymorphism refers to the ability to take on multiple identities. An operation, for example, may behave differently in different situations. The behavior is determined by the data types utilized in the operation. Take, for example, the addition operation. For tow numbers, the operation will generate a sum. If the operands are strings, then the operation will produce a third string by contention. The diagram given below, illustrates that a single function name can be used to handle different number and types of arguments. This is something similar to a particular word having several different meanings depending on the context. Polymorphism is crucial in allowing things with disparate internal structures to share the same external interface. This means that even while the precise actions associated with each operation may differ, a generic class of operations can be accessed in the same way. As demonstrated here, polymorphism is widely used to implement inheritance.



Polymorphism can be implemented using operator and function overloading, where the same operator and function works differently on different arguments producing different results. These polymorphisms are brought into effect at compile time itself, hence is known as early binding, static binding, static linking or compile time polymorphism.

However, uncertainty arises when a function with the same name exists in both the base and derived classes. Consider the following code snippet as an example.

```
Class ab
{
Int a;
Public:
Void display();    {……}                    //        display in base class
};
Class ba: public ab
{
Int b;
Public:
Void display();    {…….}                   //        display in derived class
};
```

There is no overloading because the functions ab.display() and ba.display() are the same but in distinct classes, therefore early binding does not apply. Run time polymorphism selects the right function at run time.

7.1 Run-time Polymorphism

C++ supports run-time polymorphism by a mechanism called virtual function. It exhibits late binding or dynamic linking.

Caution: What do you mean by function overriding?

Function overriding occurs when a child class declares a method that already exists in the parent class; in this case, the child class overrides the parent class. Function overriding is an example of Runtime polymorphism.

Example

```
// Program of runtime polymorphism


#include <iostream>
```

```
using namespace std;
class A {
public:
  void disp(){
    cout<<"Display in base class"<<endl;
  }
};
class B: public A{
public:
  void disp(){
    cout<<"Display in derived class";
  }
};
int main() {
  //Parent class object
  A obj;
  obj.disp();
  //Child class object
  B obj2;
  obj2.disp();
  return 0;
}
```

Output

```
Display in base class
Display in derived class
Process returned 0 (0x0)   execution time : 0.356 s
Press any key to continue.
```

7.2 Virtual Base Class

Virtual classes are primarily used during multiple inheritance. Virtual base class is used in situation where a derived have multiple copies of base class. When two or more objects are derived from a common base class, we can prevent multiple copies of the base class being present in an object derived from those objects by declaring the base class as virtual when it is being inherited.

Virtual base class means that child classes can access the base classes virtually from a long distance in the parent-child hierarchy.

 Lab Exercise

```
#include<iostream>
using namespace std;
class A{
```

```
public:
   int x;
};
class B:virtual public A{
public:
   int y;
};
class C: virtual public A{
public:
   int z;
};
class D:public C,public B{
public:
   int x1;
};
main(){
D obj;
obj.x=100;
obj.y=20;
obj.z=30;
obj.x1=200;
cout<< "\n X : "<< obj.x;
            cout<< "\n Y : "<< obj.y;
        cout<< "\n Z : "<< obj.z;
        cout<< "\n X1 : "<<              obj.x1;
            }
```

Output

```
 X : 100
 Y : 20
 Z : 30
 X1 : 200
Process returned 0 (0x0)   execution time : 0.015 s
Press any key to continue.
```

7.3 Abstract Class

An abstract class in C++ has at least one pure virtual function by definition. In other words, a function that has no definition. The abstract class's descendants must define the pure virtual function; otherwise, the subclass would become an abstract class in its own right.

Abstract classes are used to describe general notions that can be used to create more concrete classes. It is not possible to generate an abstract class type object. However, pointers and references can be used to abstract class types. When building an abstract class, make sure to include at least one pure virtual member feature. A virtual function is declared using the pure specifier (= 0) syntax.

Notes

- A class that contains a pure virtual function is known as an abstract class.
- Abstract classes are essential to providing an abstraction to the code to make it reusable and extendable.

Characteristics of Abstract Class

1. Abstract class cannot be instantiated, but pointers and references of Abstract class type can be created.
2. Abstract class can have normal functions and variables along with a pure virtual function.
3. Abstract classes are mainly used for Upcasting, so that its derived classes can use its interface.

Properties of Abstract Class

1. Abstract classes cannot have objects.
2. To be an abstract class, it must have a presence of at least one virtual class.
3. We can use pointers and references to abstract class type.
4. We can create constructors of an abstract class.

Did you know?

Pure virtual Functions are virtual functions with no definition. They start with virtual keyword and ends with = 0.

Lab Exercise

```
// Program
#include <iostream>
using namespace std;
class A
{
public:
  virtual void test() = 0;
};
```

```
class B : public A
{
public:
  void test()
   {
cout << " Hello !  Virtual function " << endl;
  }
};
int main(void)
{
  B obj;
 obj.test();
 return 0;
}
```

Output

```
 Hello !  Virtual function

Process returned 0 (0x0)   execution time : 0.131 s
Press any key to continue.
```

Task: Write a Program to demonstrate working of abstract class in C++.

Restriction of Abstract Class

Abstract classes cannot be used for the following scenarios:

- Member data or variables
- Forms of debate
- Types of function output
- Conversions that are made explicitly

7.4 Pointer to object in C++

In C++ a pointer to a class is created in the same way as a pointer to a structure, and you use the member access operator -> to access members of a pointer to a class, just as you do with pointers to structures. You must also initialize the pointer before using it, as with all pointers.

Lab Exercise

```
//Program
#include <iostream>
using namespace std;
class sum
```

```
{
int b;
public:
void get_data(int a ){
b=a;
}
int display(){
return b;
}
};
main(){
sum obj;
sum* p;
p=&obj;
p->get_data(10);
cout<<" Value of a "<<p->display();
}
```

Output

```
 Value of a 10
Process returned 0 (0x0)   execution time : 0.177 s
Press any key to continue.
```

Caution: C++ class is done exactly the same way as a pointer to a structure and to access members of a pointer to a class access operator (arrow operator) -> is used.

7.5 The 'this' pointer

The 'this' pointer is also another concept here in which in C++ programming, this is a keyword that refers to the current instance of the class. If we are going to execute the current instance of the class, that is the object of my class there and if we have the privilege there using this pointer, we can add the data into there and we can store some of the information according to that use of this keyword, it can be used to pass current object as the parameter to the another method, and it can be used to refer the current class instance variable like we have a class members and their member functions if we want to use the this pointer there we can access the current members of member functions with this pointer it can be used to declare the indexes.

Uses of 'this' keyword

- It can be used to pass current object as a parameter to another method.
- It can be used to refer current class instance variable.
- It can be used to declare indexers.

Consider the following example

Class ABC

{

Int a;

…….

…….

};

The private variable 'a' can be used directly inside a member function, like

a=100;

We can also use the following statement to do the same job:

this->a=100;

Since C++ permits the use of shorthand form a 100, we have not been using the pointer this explicitly SO far. However, we have been implicitly using the pointer this when overloading the operators using member function.

 Lab Exercise

```
//Program
#include <iostream>
using namespace std;
class employee
{
int id;
string name;
int salary;
public:
employee(int e_id,string  e_name,int e_salary ){
this->id=e_id;
this->name=e_name;
this->salary=e_salary;
}
void display(){
cout<<id<<name<<salary<<endl;
}
};
main(){
employee e1=employee(10,"john",25000);
e1.display();
}
```

Output

```
10john25000

Process returned 0 (0x0)   execution time : 0.049 s
Press any key to continue.
```

Task:

- Write a program to demonstrate working of pointer to object in C++.
- Write a program to demonstrate working of 'this' pointer in C++.

7.6 Pointer to Derived Class

We can use pointers not only to the base objects but also to the objects of derived classes. Pointers to objects of a base class are type-compatible with pointers to objects of a derived class. Therefore, a single minter variable ran he made to paint to objects belonging to different classes. For example, if **B** is a base class and **D** is a derived class from B, then a pointer declared as a pointer to **B** can also be a pointer to **D**. Consider the following declarations:

```
B *cptr;                    //Pointer to class B type variable
B b;                        //Base object
D d;                        //Derived object
Cptr=&b;                    //Cptr points to object b
```

We can make cptr to point to the object d as follows:

```
Cptr=&d;                    //cptr points to object d
```

This is perfectly valid with C++ because d is an object derived from the class B.

However, there is a problem in using cptr to access the public members of the derived class D. Using cptr, we can access only those members which are inherited from B and not the members that originally belong to D. In case a member of D has the same name as one of the members of B, then any reference to that member by cptr will always access the base class member.

Although C++ permits a base pointer to point to any object derived from that base, the pointer cannot be directly used to access all the members of the derived class. We may have to use another pointer declared as pointer to the derived type.

Lab Exercise

```
//Program
#include<iostream>
using namespace std;
class base
{
   public:
    void show()
   {
      cout<<"Base class"<<endl;
```

```
    }
};
class derive : public base
{
    public:
     void display()
    {
 cout<<"Derived Class";
    }
};
int main()
{
   base b;
   base *bptr;
  bptr->show();
  derive d;
bptr=&d;
((derive *)bptr)->display();
   return 0;
}
```

Output

```
Base class
Derived Class
Process returned 0 (0x0)   execution time : 0.141 s
Press any key to continue.
```

Summary

- Polymorphism simply means one name having multiple forms.
- There are two types of polymorphism, namely, compile time polymorphism and run time polymorphism.
- Functions and operators overloading are examples of compile time polymorphism. The overloaded member functions are selected for invoking by matching arguments, both type and number. The compiler knows this information at the compile time and, therefore, compiler is able to select the appropriate function for a particular call at the compile time itself. This is called early or static binding or static linking. It means that an object is bound to its function call at compile time.
- In run time polymorphism, an appropriate member function is selected while the program is running. C++ supports run time polymorphism with the help of virtual functions. It is called late or dynamic binding because the appropriate function is selected dynamically at

run time. Dynamic binding requires use of pointers to objects and is one of the powerful features of C++.

- Object pointers are useful in creating objects at run time. It can be used to access the public members of an object, along with an arrow operator.
- A this pointer refers to an object that currently invokes a member function. For example, the function call ashow0 will set the pointer 'this' to the address of the object 'a'.
- Pointers to objects of a base class type are compatible with pointers to objects of a derived class. Therefore, we can use .1 single pointer variable to point to objects of base class as well as derived classes.
- When a function is made virtual, C-+ determines which function to use at run time based on the type of object pointed to by the base pointer, rather than the type of the pointer. By making the base pointer to point to different objects, we can execute different versions of the virtual function.
- Run time polymorphism is achieved only when a virtual function is accessed through a pointer to the base class. It cannot be achieved using object name along with the dot operator to access virtual function.

Keywords

This Pointer: The pointer is an implicit parameter to all member functions. Therefore, inside a member function, this may be used to refer to the invoking object.

Abstract Class: A class that contains a pure virtual function is known as an abstract class.

Function Pointer: A function may return a reference or a pointer variable also. A pointer to a function is the address where the code for the function resides. Pointer to functions can be passed to functions, returned from functions, stored in arrays and assigned to other pointers.

Memory location: A container that can store a binary number.

Virtual Base Class: Virtual base class is used in situation where a derived have multiple copies of base class.

Pointer: A variable holding a memory address.

Alias: A different name for a variable of C++ data type. Base Address: Starting address of a memory location holding array elements.

Reference: An alias for a pointer that does not require de-referencing to use.

Self Assessment

1. Which class is used to design the base class?

A. abstract class

B. derived class

C. base class

D. derived & base class

2. Which is also called as abstract class?

A. virtual function

B. pure virtual function

C. derived class

D. base class

3. What will be the output of the following C++ code?

```
#include <iostream>
using namespace std;
class MyInterface
{
   public:
   virtual void Display() = 0;
};
class Class1 : public MyInterface
{
   public:
   void Display()
   {
      int  a = 5;
      cout << a;
   }
};
class Class2 : public MyInterface
{
   public:
   void Display()
   {
      cout <<" 5" << endl;
   }
};
int main()
{
   Class1 obj1;
   obj1.Display();
   Class2 obj2;
   obj2.Display();
   return 0;
}
```

A. 5
B. 10
C. 15
D. 5 5

4. What is meant by pure virtual function?
a. Function which does not have definition of its own
b. Function which does have definition of its own
c. Function which does not have any return type
d. Function which does not have any return type & own definition

5. Where the abstract class does is used?
A. base class only
B. derived class
C. both derived & base class
D. virtual class

6. Which of the following is the correct way to declare a pointer?
A. int *ptr
B. int ptr
C. int &ptr
D. All of the above

7. Which is the pointer which denotes the object calling the member function?
A. Variable pointer
B. This pointer
C. Null pointer
D. Zero pointer

8. The this pointer is accessible _____
A. Within all the member functions of the class
B. Only within functions returning void
C. Only within non-static functions
D. Within the member functions with zero arguments

9. What is the use of this pointer?
A. When local variable's name is same as member's name, we can access member using this pointer.
B. To return reference to the calling object
C. Can be used for chained function calls on an object
D. All of the above

10. This pointer can be used directly to _____
A. To manipulate self-referential data structures
B. To manipulate any reference to pointers to member functions
C. To manipulate class references

D. To manipulate and disable any use of pointers

11. Which members can never be accessed in derived class from the base class?

A. Private

B. Protected

C. Public

D. All except private

12. Which of the following is true for pointer to derived class?

A. We can use pointers not only to a base objects but also to the objects of derived classes.

B. We cannot use pointers in the classes.

C. Using pointer in class is form of inheritance

D. All of above.

13. We can access those members of derived class which are inherited from base class by base class pointer.

A. True

B. False

14. What will be the output of following code?

```
#include<iostream>
using namespace std;
class base
{
    public:
     void show()
    {
       cout<<"Base class"<<endl;
    }
};
class derive : public base
{
    public:
     void display()
    {
  cout<<"Derived Class";
    }
};
int main()
{
   base b;
```

```
 base *bptr;
 bptr->show();
 derive d;
bptr=&d;
((derive *)bptr)->display();
 return 0;
}
```

A. Base class Derived Class
B. Base Class
C. Derived Class
D. All of Above

15. If same message is passed to objects of several different classes and all of those can respond in a different way, what is this feature called?

A. Inheritance
B. Overloading
C. Polymorphism
D. Overriding

Answers for Self Assessment

1.	A	2.	B	3.	D	4.	A	5.	A
6.	A	7.	B	8.	C	9.	D	10.	A
11.	D	12.	A	13.	A	14.	A	15.	C

Review Questions

1. What does polymorphism mean in C++ language?
2. How is polymorphism achieved at run time? Explain with suitable example.
3. What does this pointer point to?
4. What are the applications of this pointer?
5. Write a program that demonstrate working of virtual base class.
6. Why do we need abstract class?
7. What is run time polymorphism?
8. Explain pointer to derived class with suitable example.
9. How does pointer variable differ from simple variable?
10. Write a program that demonstrate the working of abstract class.

Further Readings

E Balagurusamy; Object-oriented Programming with C++; Tata Mc Graw-Hill.

Herbert Schildt; The Complete Reference C++; Tata Mc Graw Hill.

Robert Lafore; Object-oriented Programming in Turbo C++; Galgotia.

Web Links

http://msdn.microsoft.com/en-us/library/wcz57btd(v=vs.80).aspx

http://www.learncpp.com/cpp-tutorial/117-multiple-inheritance/

https://www.codecademy.com/learn/learn-c-plus-plus

Unit 08: Virtual Functions

CONTENTS

Objectives

After studying this unit, you will be able to:

- Understand concept of binding in C++
- Understand early binding and late binding
- Demonstrate the virtual functions
- Recognize the pure virtual function

Introduction

At compile time, C++ defaults to matching a function call with the correct function definition. This is referred to as static binding. Dynamic binding allows you to tell the compiler to match a function call with the correct function definition at runtime. If you use the keyword virtual to declare a function, the compiler will use dynamic binding for that function.

8.1 Virtual functions

In C++ polymorphism refers to the property by which objects belonging to different classes are able to respond to the same message, but in different forms. An essential requirement of polymorphism is therefore the ability to refer to objects without any regard to their classes. This necessitates the use of a single pointer variable to refer to the objects of different classes. Here, we use the pointer to base class to refer to all the derived objects. But, we just discovered that a base pointer, even when it is made to contain the address of a derived class, always executes the function in the base class. The compiler simply ignores the contents of the pointer and chooses the member function that matches the type of the pointer. How do we then achieve polymorphism? It is achieved using what is known as 'virtual' functions.

When we use the same function name in both the base and derived classes, the function in base class is declared as virtual using the keyword virtual preceding its normal declaration. When a function is made virtual, C++ determines which function to use at run time based on the type of

object pointed to by the base pointer, rather than the type of the pointer. Thus, by making the base pointer to point to different objects, we can execute different versions of the virtual function.

Notes

- A virtual function is a member function which is declared within a base class and is re-defined (Overridden) by a derived class.
- Virtual functions ensure that the correct function is called for an object, regardless of the type of reference (or pointer) used for function call.
- Virtual functions mainly used to achieve run time polymorphism.
- Functions are declared with a virtual keyword in base class.

8.2 Rules for Virtual Functions

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

- The virtual functions must be members of some class.
- They cannot be static members.
- They are accessed by using object pointers.
- A virtual function can be a friend of another class.
- A virtual function in a base class must be defined, even though it may not be used.

Lab Exercise

```
//Program
#include <iostream>
using namespace std;
class Base {
  public:
   virtual void print() {
      cout << "Base Function" << endl;
   }
};
class Derived : public Base {
  public:
   void print() {
      cout << "Derived Function" << endl;
   }
};
int main() {
   Derived d1;
  Base* bptr = &d1;
bptr->print();
   return 0;
}
```

Output

```
Derived Function

Process returned 0 (0x0)   execution time : 0.013 s
Press any key to continue.
```

8.3 Late Binding

Late binding refers to the act of selecting functions while the programme is running. Although late binding adds to the overhead, it gives you more power and flexibility. Due to the fact that late binding is implemented using virtual functions, we must specify a class object as either a pointer to a class or a reference to a class.

Did you know?

What is binding?

Binding refers to the process that is used to convert identifiers (such as variable and function names) into machine language addresses. Although binding is used for both variables and functions, but we have to focus on function binding.

For example the following shows how a late binding or run time binding can be carried out with the help of a virtual function.

```
class base {
private :
int x;
float y;
public:
virtual void display ( );
int sum ( );
};
class derivedD : public baseA
{
private:
int x;
float y;
public:
void display (); //virtual
int sum ( );
};
void main ( )
{
baseA *ptr;
```

```
derivedD objd;
ptr = &objd;
Other Program statements
ptr->display (); //run time binding
ptr->sum ( ); //compile time binding
}
```

Note that the keyword virtual is be followed by the return type of a member function if a run time is to be bound. Otherwise, the compile time binding will be effected as usual. In the above program segment, only the display ( ) function has been declared as virtual in the base class, whereas the sum () is non-virtual. Even though the message is given from the pointer of the base class to the objects of the derived class, it will not access the sum ( ) function of the derived class as it has been declared as non-virtual. The sum ( ) function compiles only the static binding.

The following program demonstrates the run time binding of the member functions of a class. The same message is given to access the derived class member functions from the array of pointers. As function are declared as virtual, the C++ compiler invokes the dynamic binding.

```
#include <iostream.h>
#include <conio.h>
class baseA {
public:
virtual void display () {
cout<< "One \n";
}
};
class derivedB : public baseA
{
public:
virtual void display 0 {
cout<< "Two\n"; }
};
class derivedC: public derivedB
{
public:
virtual void display ( ) {
cout<< "Three \n"; }
};
void main ( ) {
//define three objects
baseA obja;
derivedB objb;
derivedC objc;
base A *ptr [3]; //define an array of pointers to baseA
ptr [0] = &obja;
ptr [1] = &objb;
```

```
ptr [2] = &objc;
for (int i = 0; i <=2; i ++ )
ptr [i]->display (); //same message for all objects
getche ( );
}
```

Output

One

Two

Three

The following programme demonstrates the static binding of a class's member functions. There are two classes in the programme: student and academic. The term "class academic" comes from the term "class student." Both classes have the member functions getdata and display specified. *obj is a variable for the class student, whose address is recorded in the class academic's object.

```
# include <iostream.h>
Using namespace std;
class student {
private:
int rollno;
char name [20];
public:
void getdata ( );
void display ( );
};
class academic: public student {
private:
char stream;
public:
void getdata ( );
void display ( );
} ;
void student:: getdata ( )
{
cout<< "enterrollno\n";
cin>> rollno;
cout <<"enter name \n";
cin > > name;
}
void student:: display ( )
{
cout<< "the student's roll number is "<<rollno<<"and name is"
<<name <<"Stream "<<stream;
```

```
cout<< endl;
}
void academic :: getdata ()
{
cout<< "enter stream of a student? \n";
cin >>stream;
}
void academic :: display () {
cout<< "students stream \n";
cout <<stream<< end!;
}
void main ( )
{
student *ptr;
academic obj;
ptr=&obj;
ptr->getdata ();
ptr->display ();
}
```

Output

enter rollno

1

enter name

ajay

enter stream of a student?

IT

the student's roll number is 1 and name is ajay and Stream IT

 Did you know?

What is dynamic binding?

You can specify the compiler match a function call with the correct function definition at run time; this is called dynamic binding.

 **Lab Exercise**

```
#include <iostream>
using namespace std;
class Base
{
public:
virtual void print()
```

```
{
cout << "This is parent class" << endl;
}
};
class Derived : public Base
{
public:
void print()
{
cout << "This is child class" << endl;
}
};
int main()
{
Base *bptr;
Derived d;
bptr= &d;
bptr-> print();
return 0;
}
```

Output

```
This is child class

Process returned 0 (0x0)   execution time : 0.068 s
Press any key to continue.
```

8.4 Early Bindings

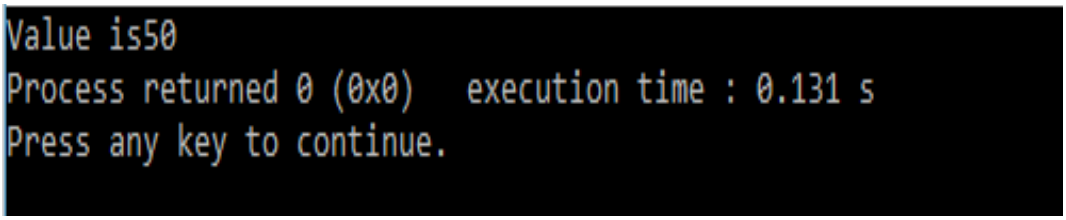
- Most of the function calls the compiler encounters will be direct function calls. A direct function call is a statement that directly calls a function.
- It is also known as Static Binding or Compile-time Binding.
- Direct function calls can be resolved using a process known as early binding.
- Early binding (also called static binding) means the compiler is able to directly associate the identifier name (such as a function or variable name) with a machine address.
- All functions have a unique machine address. So when the compiler encounters a function call, it replaces the function call with a machine language instruction that tells the CPU to jump to the address of the function.

Example

```
// Program
#include <iostream>
```

```
using namespace std;
void displayValue(int val){
cout<<"Value is" << val;
}
int main(){
displayValue(50);   // Direct Function call
return 0;
}
```

Output



8.5 Difference between Early Binding and Late Binding?

Let's see the difference according to multiple parameters between bindings

Early Binding vs Late Binding	
The process of using the class information to resolve method calling that occurs at compile time is called Early Binding.	The process of using the object to resolve method calling that occurs at run time is called the Late Binding.
Time of Binding	
Early Binding happens at compile time.	Late Binding happens at run time.
Functionality	
Early Binding uses the class information to resolve method calling.	Late Binding uses the object to resolve method calling.
Synonyms	
Early Binding is also known as static binding..	Late Binding is also known as dynamic binding.
Occurrence	
Overloading methods are bonded using early binding.	Overridden methods are bonded using late binding.
Execution Speed	
Execution speed is faster in early binding.	Execution speed is lower in late binding.

Task

- Explain the difference between early binding and late binding with suitable programming examples using C++.

8.6 Pure Virtual Function

It is normal practice to declare a function virtual inside the base class and redefine it in the derived classes. The function inside the base class is seldom used for performing any task. It only serves as a placeholder. For example, we have not defined any object of class media and therefore the function display() in the base class has been defined 'empty'. Such functions are called "do-nothing" functions.

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

virtual void display()=0;

Such functions are called pure virtual functions. A pure virtual function is a function declared in a base class that has no definition relative to the base class. In such cases, the compiler requires each derived class to either define the function or re-declare it as a pure virtual function. Remember that a class containing pure virtual functions cannot be used to declare any objects of its own. As stated earlier, such classes are called abstract base classes. The main objective of an abstract base class is to provide some traits to the derived classes and to create a base pointer required for achieving run time polymorphism.

Lab Exercise

```
#include <iostream>
using namespace std;
class A
{
public:
  virtual void test() = 0;
};
class B : public A
{
public:
  void test()
   {
cout << " Hello ! Pure Virtual function " << endl;
  }
};
int main(void)
{
  B obj;
 obj.test();
 return 0;
}
```

Output

```
Hello ! Pure Virtual function

Process returned 0 (0x0)   execution time : 0.102 s
Press any key to continue.
```

8.7 Virtual Function vs. Pure Virtual Function

Virtual function	Pure virtual function
A virtual function is a member function of base class which can be redefined by derived class.	A pure virtual function is a member function of base class whose only declaration is provided in base class and should be defined in derived class otherwise derived class also becomes abstract.
Classes having virtual functions are not abstract.	Base class containing pure virtual function becomes abstract.
Definition is given in base class.	No definition is given in base class.
Base class having virtual function can be instantiated i.e. its object can be made.	Base class having pure virtual function becomes abstract i.e. it cannot be instantiated.
If derived class do not redefine virtual function of base class, then it does not affect compilation.	If derived class do not redefine virtual function of base class, then no compilation error but derived class also becomes abstract just like the base class.
All derived class may or may not redefine virtual function of base class.	All derived class must redefine pure virtual function of base class otherwise derived class also becomes abstract just like base class.

Summary

- OOP is used commonly for software development. One major pillar of OOP is polymorphism. Early Binding and Late Binding are related to that. Early Binding occurs at compile time while Late Binding occurs at runtime. In method overloading, the bonding happens using the early binding. In method overriding, the bonding happens using the late binding. The difference between Early and Late Binding is that Early Binding uses the class information to resolve method calling while Late Binding uses the object to resolve method calling.
- When a function is made virtual, C-+ determines which function to use at run time based on the type of object pointed to by the base pointer, rather than the type of the pointer. By making the base pointer to point to different objects, we can execute different versions of the virtual function.
- If a virtual function is defined in the base class, it need not be necessarily redefined in the derived class. In such cases, the respective calls will invoke the base class function.
- A virtual function, equated to zero is called a pure virtual function. It is a function declared in a base class that has no definition relative to the base class. A class containing such pure function is called an abstract class.
- Early binding refers to the events that occur at compile time while late binding means selecting function during the execution. The late binding is implemented through virtual function.

Keywords

Late Binding: Selecting functions during the execution. Though late binding requires some overhead it provides increased power and flexibility.

Virtual Function: Virtual functions, one of advanced features of OOP is one that does not really exist but it appears real in some parts of a program.

Compile time polymorphism: Compile-time polymorphism is achieved through method overloading.

Run time polymorphism: The runtime polymorphism can be achieved by method overriding.

Do nothing function: The Do-Nothing method is used to arrange that a call to a method or property should be ignored. When using Do-Nothing, all the logic inside the arranged method or property body is skipped and nothing happens when you call it.

Self Assessment

1. Which is the correct declaration of pure virtual function in C++

A. virtual void func = 0;
B. virtual void func() = 0;
C. virtual void func(){0};
D. void func() = 0;

2. In a class, pure virtual functions in C++ is used

A. To create an interface
B. To make a class abstract
C. To force derived class to implement the pure virtual function
D. All the above

3. Virtual functions mainly used to achieve run time polymorphism.

A. True
B. False

4. The prototype of virtual functions should be the same in the base as well as derived class.

A. True
B. False

5. If a …… is defined in the base class, it need not be necessarily redefined in the derived class.

A. member function
B. virtual function
C. static function
D. real function

6. ……. is a function declared in a base class that has no definition relative to the base class.

A. member function
B. virtual function
C. pure virtual function
D. pure function

6. State, whether the following statements about virtual functions are True.

i) The virtual function must be a member of some class

ii) virtual functions cannot be static members

iii) A virtual function cannot be a friend of another class.

A. i and ii only
B. ii and iii only
C. i and iii only
D. All i, ii and iii

7. State whether the following statements about virtual functions are True or False.

i) A virtual function, equated to zero is called pure virtual function.

ii) A class containing pure virtual function is called an abstract class

A. True, True
B. True, False
C. False, True
D. False, False

8. In compile-time polymorphism, a compiler is able to select the appropriate function for a particular call at the compile time itself, which is known as ……
A. early binding
B. static binding
C. static linking
D. All of the above

9. ……. binding means that an object is bound to its function call at compile time.
A. late
B. static
C. dynamic
D. fixed

10. Which of the following concepts means waiting until runtime to determine which function to call?
A. Data hiding
B. Dynamic loading
C. Dynamic binding
D. Data casting

11. C++ supports run time polymorphism with the help of virtual functions, which is called …….. binding.
A. dynamic
B. run time
C. early binding
D. static

12. Static binding is same as …….

A. Run Time Binding
B. Compile Time Binding
C. All of above
D. None of above

13. Use of virtual functions implies
A. overloading
B. overriding
C. static binding
D. dynamic binding

14. What would be the output of following code?

```
#include <iostream>
using namespace std;
int Add(int x,int y){
return x+y;
}
int main(){
int (*fptr)(int,int)=Add;
cout<<fptr(10,40);
return 0;
}
```

A. 10
B. 40
C. 50
D. Compile time error

Answers for Self Assessment

1.	B	2.	D	3.	A	4.	A	5.	B
6.	C	7.	A	8.	A	9.	D	10.	B
11.	C	12.	A	13.	B	14.	D	15.	C

Review Questions

1. How can C++ achieve dynamic binding? Explain with suitable example.
2. What do you mean by binding in C++?
3. Differentiate between early binding and late binding.
4. Write a program using C++ that demonstrate working of dynamic binding.
5. Differentiate between virtual function and pure virtual function.
6. Pure virtual functions force the programmer to redefine the virtual function inside the derived class. Comment the statement.
7. Write a program using C++ that demonstrate working of pure virtual function.

8. Write a program using C++ that demonstrate working of static binding.
9. When do we make a virtual function "pure"? What are the implications of making a function a pure virtual function?
10. Why do we need virtual functions?

Further Readings

E Balagurusamy; Object-oriented Programming with C++; Tata Mc Graw-Hill.

Herbert Schildt; The Complete Reference C++; Tata Mc Graw Hill.

Robert Lafore; Object-oriented Programming in Turbo C++; Galgotia.

Web Links

http://www.learncpp.com/cpp-tutorial/117-multiple-inheritance/

https://www.codecademy.com/learn/learn-c-plus-plus

http://www.artima.com/cppsource/pure_virtual.html

http://publib.boulder.ibm.com/infocenter/comphelp/v8v101 index.jsp?topic=%2Fcom.ibm.xlcpp8a.doc%2Flanguage%2Fref%2Fcplr142.htm

Prikshit Kumar Angra, Lovely Professional University

Unit 09: Working with Streams and Files

CONTENTS

Objectives

After studying this unit, you will be able to:

- Demonstrate the opening a file
- Understand C++ Stream classes
- Explain the processing and closing a file
- Discuss the detection of end of file

Introduction

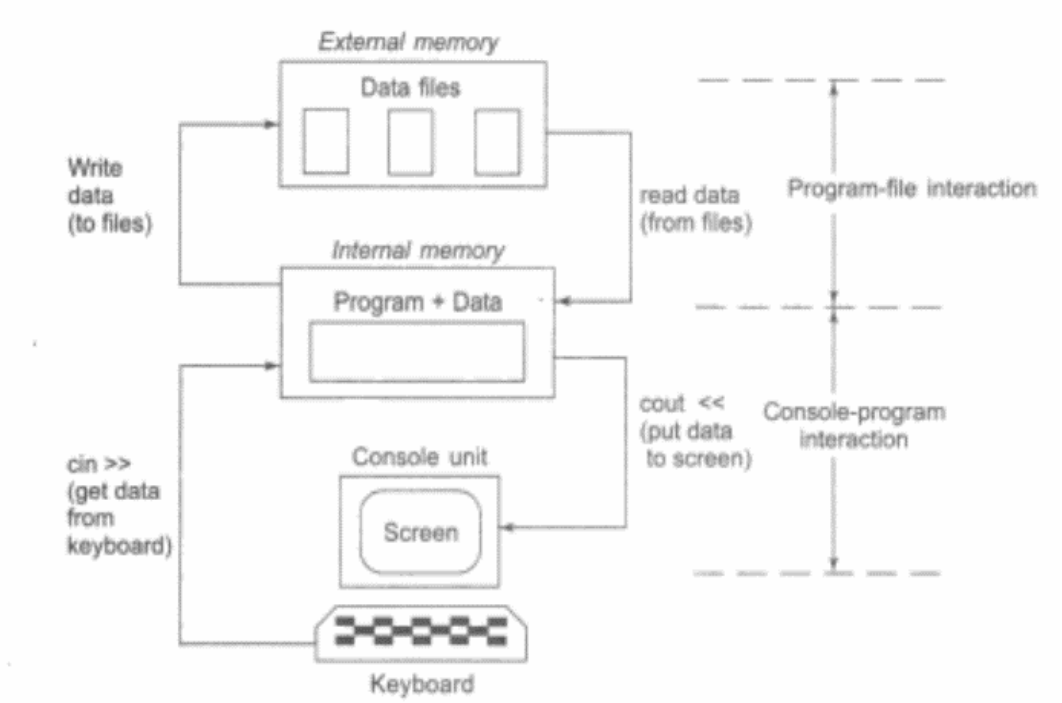
Many real-life problems handle large volumes of data and, in such situations, we need to use some devices such as floppy disk or hard disk to store the data. The data is stored in these devices using the concept of files. A file is a collection of related data stored in a particular area on the disk. Programs can be designed to perform the read and write operations on these files.

A program typically involves either or both of the following kinds of data communication:

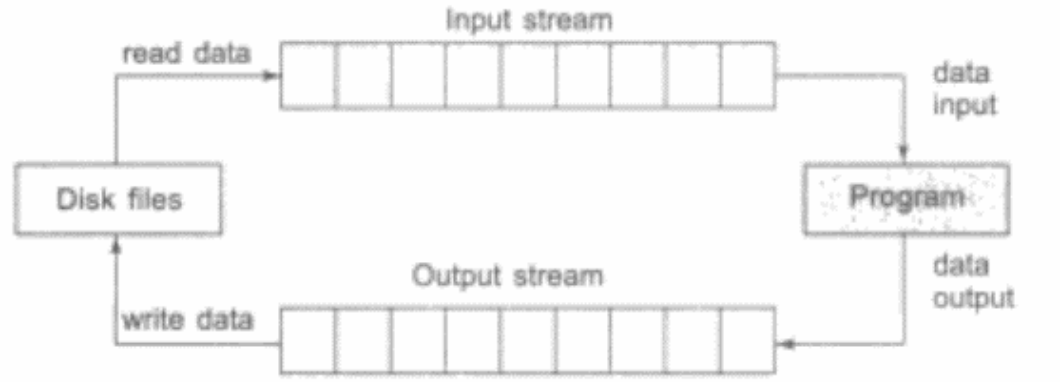
- Data transfer between control unit and the program.
- Data transfer between the program and a disk file.

We have already discussed the technique of handling data communication between the console unit and the program. In this chapter, we will discuss various methods available for storing and retrieving the data from files.

This is illustrated in figure.



C++'s I/O system deals with file operations that are quite similar to console input and output activities. As an interface between the applications and the files, it employs file streams. The input stream is the one that provides data to the programme, while the output stream is the one that receives data from the programme. To put it another way, the input stream pulls (or reads) data from the file, whereas the output stream writes data to the file. Figure shows how this works.



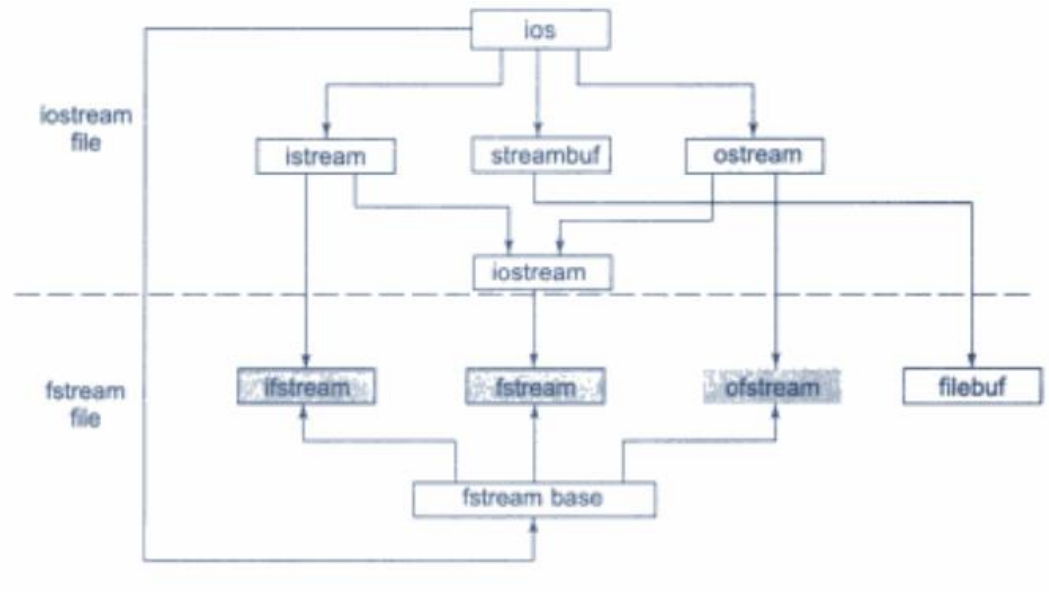
The input operation entails establishing an input stream and connecting it to the programme and the input file. Similarly, setting up an output stream with the appropriate linkages to the programme and the output file is part of the output procedure.

Notes

- Stream classes in C++ are used to input and output operations on files and i/ o devices. These classes have specific features and to handle input and output of the program.
- The **iostream.h** library holds all the stream classes in the C++ programming language.

9.1 Classes for File Stream Operations

The file handling techniques are defined by a set of classes in C++'s I/O system. Ifstream, ofstream, and (stream) are a few examples. As illustrated in Figure, these classes are generated from fstreambase and the matching tostream class. These classes, designed to manage disc files, are declared in (stream, and as a result, every programme that uses files must include this file.



Details of file stream classes listed in following table.

Did you know?

What is Stream in C++?

- A stream is nothing but a flow of data.
- In the object-oriented programming, the streams are controlled using the classes. The operations with the files mainly consist of two types. They are read and write.
- C++ provides various classes, to perform these operations.
- Each stream is associated with a particular class, which contains member functions and definitions for dealing with that particular kind of data flow.
- The stream that supplies data to the program in known as an input stream. It reads the data from the file and hands it over to the program.
- The stream that receives data from the program is known as an output stream. It writes the received data to the file.

Benefits of Stream Classes

- The ios class: The ios class is responsible for providing all input and output facilities to all other stream classes.
- The istream class: This class is responsible for handling input stream. It provides number of function for handling chars, strings and objects such as get, getline, read, ignore, putback etc.

Classes for File Stream Operation

- Files are dealt mainly by using three classes fstream, ifstream, ofstream.

ofstream: This Stream class signifies the output file stream and is applied to create files for writing information to files

ifstream: This Stream class signifies the input file stream and is applied for reading information from files

fstream: This Stream class can be used for both read and write from/to files.

File Stream Classes

Class	Purpose
filebuf	Its purpose is to set the file buffers to read and write. Contains **Openprot** constant used in the **open()** of file stream classes. Also contains **close()** and **open()** as members.
fstreambase	Provides operations common to file streams Serves as a base for fstream, ifstream and ofstream class. Contains open() and close() functions.
ifstream	Provides input operations. Contains open° with default input mode. Inherits the functions get(), getline(),read(), seekg(), tellg() functions from istream.
ofstream	Provides output operations. Contains open() with default output mode. Inherits put(), seekp(), tellp() and write() functions from onstream.
fstream	Provides support for simultaneous input and output operations. Contains open with default input mode. Inherits all the functions from istream and ostream classes through iostream.

9.2 Creating A File

We create a file by specifying new path of the file and mode of operation. Operations can be reading, writing, appending and truncating.

Syntax

FilePointer.open("Path",ios::mode);

Example

```
#include<iostream>
#include <fstream>
using namespace std;
int main()
{
   fstream st;
   st.open("test.txt",ios::out);
   if(!st)
   {
      cout<<"File creation failed";
   }
   else
   {
```

```
    cout<<"New file created";
    st.close();
  }
  return 0;
}
```

Output

```
New file created
Process returned 0 (0x0)   execution time : 0.013 s
Press any key to continue.
```

9.3 Opening a File

The example programs listed above are indeed very simple. A data file can be opened in a program in many ways. These methods are described below.

ifstream filename("filename <with path>"); Or ofstream filename("filename <with path>");

This is the form of opening an input file stream and attaching the file "filename with path" in one single step. This statement accomplishes a number of actions in one go:

1. creates an input or output file stream

2. Looks for the specified file in the file system

3. Attaches the file to the stream if the specified file is found otherwise returns a NULL value.

The pointer is set to the initial location if the file was successfully joined to the stream. The object's name can be used to access the created file stream. If a path is supplied, the requested file is searched in that directory; otherwise, the file is only searched in the current directory. If the file isn't found, a new one is generated in case it's being used for output. If the requested file is identified while opening a file for output, it is truncated before being opened, resulting in the loss of everything in the file previously. It is important to guarantee that the application does not mistakenly overwrite a file. If the file is not discovered, a NULL value is given, which we can check to make sure we aren't reading a file that isn't there. If we try to read a file that isn't there, we'll get an error in the application.

ifstream filename;

filename.open("file name <with path>");

In this approach the input stream - filename - is created but no specific file is attached to the stream just created. Once the stream has been created a file can be attached to the stream using open() member function of the class ifstream or ofstream as is exemplified by the following program snippet which defines a function to read an input file.

```
#include <fstream.h>
void read(ifstream &ifstr)                    // file streams can be passed to functions
{
char ch;
while(!ifstr.eof())
{
ifstr.get(ch);
cout << ch;
```

```
}
cout << endl << " – – – –" << endl; Notes
}
void main()
{
ifstream filename("data1.dat");
read(filename);
filename.close();
filename.open("data2.dat");
read(filename);
filename.close();
}
```

ifstream filename(char *fname, int open_mode);

The ifstream function Object() { [native code] } accepts two parameters in this form: a filename and the mode in which the input file should be read. C++ has a variety of input file opening modes, each of which provides distinct forms of reading control over the opened file. In C++, the file opening modes are represented by an enumerated type called ios. Below is a list of the various file opening modes.

The ifstream function Object() { [native code] } accepts two parameters in this form: a filename and the mode in which the input file should be read. C++ has a variety of input file opening modes, each of which provides distinct forms of reading control over the opened file. In C++, the file opening modes are represented by an enumerated type called ios. Below is a list of the various file opening modes.

Opening Mode	Description
ios::in	Open file in input mode for reading
ios::out	Open file in output mode for writing
ios::app	Open file in output mode for writing the new content at the end of the file without removing the previous contents of the file.
ios::ate	Open file in output mode for writing the new content at the current position of the pointer without removing the previous contents of the file.
ios::trunc	Open file in output mode for writing the new content at the beginning of the file removing the previous contents of the file.
ios::nocreate	The file is not created. The operation takes place on existing file. If the file is not found an error occurs.
ios::noreplace	The existing file is not overwritten. The operation takes place on existing file. If the file is not found an error occurs.
ios::binary	Opens the file in binary mode reading not a character but reading/writing whatever binary value is stored in the file.

9.4 Opening and Closing File

If we want to use a disk file, we need to decide the following things about the file and its intended use:

1. Suitable name for the file
2. Data type and structure
3. Purpose

4. Opening Method

The filename is a string of characters that make up a valid filename for the operating system. It may contain two parts, a primary, name and an optional period with extension. Examples:

Input.data

Test.doc

INVENT.ORY

student

salary

OUTPUT

As stated earlier, for opening a file, we must first create a file stream and then link it to the filename. A file stream can be defined using the classes ifstream, ofstream, and factream that are contained in the header file fttream. The class to be used depends upon the purpose, that is, whether we want to read data from the file or write data to it. A file can be opened in two ways:

As stated earlier, for opening a file, we must first create a file stream and then link it to the filename. A file stream can be defined using the classes ifstream, ofstream, and fstream that are contained in the header file fstream. The class to be used depends upon the purpose, that is, whether we want to read data from the file or write data to it. A file can be opened in two ways:

1. Using the constructor function of the class.
2. Using the member function open() of the class.

The first method is useful when we use only one file in the stream. The second method is used when we want to manage multiple files using one stream.

Opening File Using Constructor

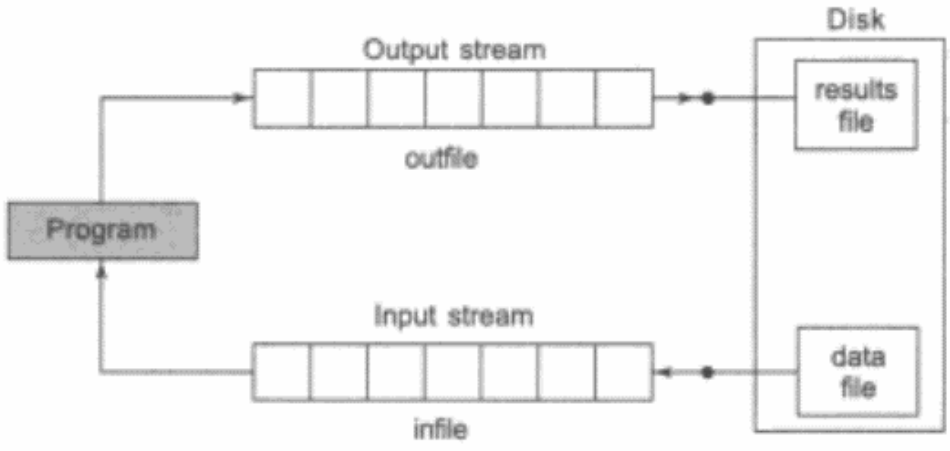
We know that a constructor is used to initialize an object while it is being created. Here, a filename is used to initialize the file stream object. This involves the following steps:

1. Create a file stream object to manage the stream using the appropriate class. That is to say, the class ofstream is used to create the output stream and the class ifstream to create the input stream.
2. Initialize the file object with the desired filename.

For example, the following statement opens a file named "results" for output:

ofstream outfile("results"); // output only

This creates outfile as an ofstream object that manages the output stream. This object can be any valid C++ name such as o_file, myfile or Pout. This statement also opens the file results and attaches it to the output stream outfile. This is illustrated in Figure.



Similarly, the following statement declares infile as an ifstream object and attaches it to the file data for reading (input).

ifstream infile("data"); // input only

The program may contain statements like:

outfile « "TOTAL.;

outfile « sum;

infile » number;

infile >> string;

Lab Exercise

```
#include <iostream>
#include <fstream>
using namespace std;
int main()
{
   ofstream of("demo.txt");
   of<< "Writing to file using fstream constructor!" << endl;
   of.close ();
   return 0;
}
```

Notes: The constructors of stream classes (ifstream, ofstream, or fstream) are used to initialize file stream objects with the filenames passed to them.

Opening File Using open()

As stated earlier, the function open() can be used to open multiple files that use the same stream object. For example, we may want to process a set of files sequentially. In such cases, we may create a single stream object and use it to open each file in turn. This is done as follows:

```
file-stream-class stream-object;
stream-object.open ("filename");
```

Example:

```
ofstream outfile;           // Create stream (for output)
outfile.open("DATA1");      // Connect stream to DATA1
.....
.....
outfile.close();            // Disconnect stream from DATA1
outfile.open("DATA2);       // Connect stream to DATA2
.....
.....
outfile.close();            // Disconnect stream from DATA2
.....
.....
```


Notes

- If the situation requires simultaneous processing of two files, then you need to create a separate stream for each file.


Lab Exercise

```
#include <iostream>
#include <fstream>
using namespace std;
int main()
{
string text;
ifstream ReadFile("a.txt");
while (getline (ReadFile, text)) {
 cout << text;
}
ReadFile.close();
return 0;
}
```

9.5 Detection end-of-file

Detection of the end-of-file condition is necessary for preventing any further attempt to read data from the file. This was illustrated in Program by using the statement.

while(fin)

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

There is another approach to detect the end-of-file condition. Note that we have used the following statement in Program :

if(finl.eof() !. 0) (exit(1);)

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

Lab Exercise

```
#include <iostream.h>
#include <fstream.h>
#include <stdlib.h>            // for exit() function

int main()
{
          const  int SIZE = 80;
          char line[SIZE];

          ifstream fin1, fin2;        // create two input streams
          fin1.open("country");
          fin2.open("capital");

          for(int i=1; i<=10; i++)
          {
                 if(fin1.eof() != 0)
                 {
                        cout << "Exit from country \n";
                        exit(1);
                 }
          fin1.getline(line, SIZE);
          cout << "Capital of "<< line ;

          if(fin2.eof() != 0)
          {
                 cout << "Exit from capital\n";
                 exit(1);
          }

          fin2.getline(line,SIZE);
          cout << line << "\n";
    }
    return 0;
}
```

Lab Exercise

```
#include <iostream>
#include <fstream>
using namespace std;
int main () {
ifstream is("demo.txt");
  char c;
  while (is.get(c))
  cout << c;
  if (is.eof())
  cout << "EoF reached";
  else
  cout << "error reading";
  is.close();
  return 0;
}
```

Output

```
New file created
Process returned 0 (0x0)   execution time : 0.013 s
Press any key to continue.
```

Task

Write a program using C++ to demonstrate process of reading from file in C++.

Write a program using C++ to demonstrate process of writing in file.

Summary

- The C++ I/O system contains classes such as ifstream, ofstream and fatream to deal with file handling. These classes are derived from fstreambase class and are declared in a header file iostream.
- A file can be opened in two ways by using the constructor function of the class and using the member function open() of the class. While opening the file using constructor, we need to pass the desired filename as a parameter to the constructor.
- The open() function can be used to open multiple files that use the same stream object. The second argument of the open() function called file mode, specifies the purpose for which the file is opened.
- If we do not specify the second argument of the open() function, the default values specified in the prototype of these class member functions are used while opening the file. The default values are as follows:
  ios :: in – for ifstream functions, meaning-open for reading only.
  ios :: out – for ofstream functions, meaning-open for writing only.
- When a file is opened for writing only, a new file is created only if there is no file of that name. If a file by that name already exists, then its contents are deleted and the file is presented as a clean file.
- To open an existing file for updating without losing its original contents, we need to open it in an append mode.
- The (stream class does not provide a mode by default and therefore we must provide the mode explicitly when using an object of (stream class. We can specify more than one file modes using bitwise OR operator while opening a file.

Keywords

ofstream: This Stream class signifies the output file stream and is applied to create files for writing information to files

ifstream: This Stream class signifies the input file stream and is applied for reading information from files

fstream: This Stream class can be used for both read and write from/to files.

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

eof(): It returns non-zero when the end of file has been reached, otherwise it returns zero.

Self Assessment

1. C++ uses <iostream.h> directive because
A. C++ is an object oriented language
B. C++ is a markup language
C. C++ does not have any input/output facility
D. All of the above

2. The unformatted input functions are handled by
A. ostream class
B. instream class
C. istream class
D. bufStream class

3. The istream class defines the
A. Cin objects
B. Stream extraction operator for formatted input
C. Cout objects
D. Both Cin and Cout objects

4. iostream is a subclass of
A. istream
B. instream
C. ostream
D. Both istream and ostream

5. The class fstream is used for
A. High level stream processing
B. Low level stream processing
C. File stream Processing
D. All above

6. Which stream class is to only write on a file?
A. ofstream
B. ifstream
C. fstream
D. iostream

7. Which is correct syntax?
A. Myfile:open(“example.bin”,ios::out);
B. Myfile.open(“example.bin”,ios::out);
C. Myfile::open(“example.bin”,ios::out);
D. Myfile.open(“example.bin”,ios:out);

8. Which operator is used to insert the data into a file?

A. @

B. >

C. <<

D. None of above

9. Which of the following is the default mode of the opening using the ofstream class?

A. ios::in

B. ios::trunk

C. ios::out

D. ios::app

10. Which of the following is the default mode of the opening using the fstream class?

A. ios::in | ios::out

B. ios::trunk

C. ios::out

D. ios::in

11. "ios::app" causes all output to that file to be appended to the end. This value can be used only with file capable of output.

A. True

B. False

12. The close() function is used to close a file,closed by disconnecting with its streaming.

A. True

B. False

13. Which among following is correct syntax of closing a file?

A. myfile@close();

B. myfile.close();

C. myfile:close();

D. myfile$close();

14. Detection of end of file not possible in C++.

A. True

B. False

15. Which of these is the correct statement about eof() ?

A. Returns true if a file opens for reading has reached the next character.

B. Returns true if a file opens for reading has reached the next word.

C. Returns true if a file opens for reading has reached the end.

D. Returns true if a file opens for reading has reached the middle.

Answers for Self Assessment

1.	C	2.	C	3.	D	4.	D	5.	C
6.	A	7.	B	8.	C	9.	C	10.	A
11.	A	12.	A	13.	B	14.	A	15.	C

Review Questions

1. What do you mean by C++ streams?
2. What are the uses of files in computer system and how data can be write using C++.
3. What are the steps involved in using a file in a C++ program.
4. What is a file mode? Describe the various file mode options available.
5. Describe the various approaches by which we can detect end of file condition successfully.
6. Write a C++ program to demonstrate working of detection of end of file in C++.
7. What are the advantages of files?
8. How can we open a file? Explain with suitable example.
9. Write full process with suitable C++ program for create a new file.
10. Explain file opening process in C++.

Further Readings

E Balagurusamy; Object-oriented Programming with C++; Tata Mc Graw-Hill.

Herbert Schildt; The Complete Reference C++; Tata Mc Graw Hill.

Robert Lafore; Object-oriented Programming in Turbo C++; Galgotia.

Web Links

https://study.com/academy/lesson/practical-application-for-c-plus-plus-programming-working-with-files.html

https://www.codecademy.com/learn/learn-c-plus-plus

https://www.guru99.com/cpp-file-read-write-open.html

Unit 10: More on Files

CONTENTS

Objectives

After studying this unit, you will be able to:

- Recognize the concepts of streams
- Describe the hierarchy of console stream classes
- Explain the unformatted I/O operations
- Discuss the managing output with manipulators
- Describe the appending in a file
- Discuss the different types of files
- Discuss the command line arguments

Introduction

One of the most essential features of interactive programming is its ability to interact with the users through operator console usually comprising keyboard and monitor. Accordingly, every computer language (and compiler) provides standard input/output functions and/or methods to facilitate console operations.

C++ accomplishes input/output operations using concept of stream. A stream is a series of bytes whose value depends on the variable in which it is stored. This way, C++ is able to treat all the input and output operations in a uniform manner. Thus, whether it is reading from a file or from the keyboard, for a C++ program it is simply a stream.

Programs frequently read data from a data file and write the result to another (or the same) data file. This section will cover concerns with using a C++ programme to access data from a data file.

10.1 Concept of Streams

A stream is a source of sequence of bytes. A stream abstracts for input/output devices. It can be tied up with any I/O device and I/O can be performed in a uniform way. The C++ iostream library is an object-oriented implementation of this abstraction. It has a source (producer) of flow of bytes and a sink (consumer) of the bytes. The required classes for the stream I/O are defined in different library header files.

To use the I/O streams in a C++ program, one must include iostream.h header file in the program. This file defines the required classes and provides the buffering. Instead of functions, the library provides operators to carry out the I/O. Two of the Stream Operators are:

<< : Stream insertion for output.

>> : Stream extraction for input.

The following streams are created and opened automatically:

cin : Standard console input (keyboard).

cout : Standard console output (screen).

cprn : Standard printer (LPT1).

cerr : Standard error output (screen).

clog : Standard log (screen).

caux : Standard auxiliary (screen).


Example

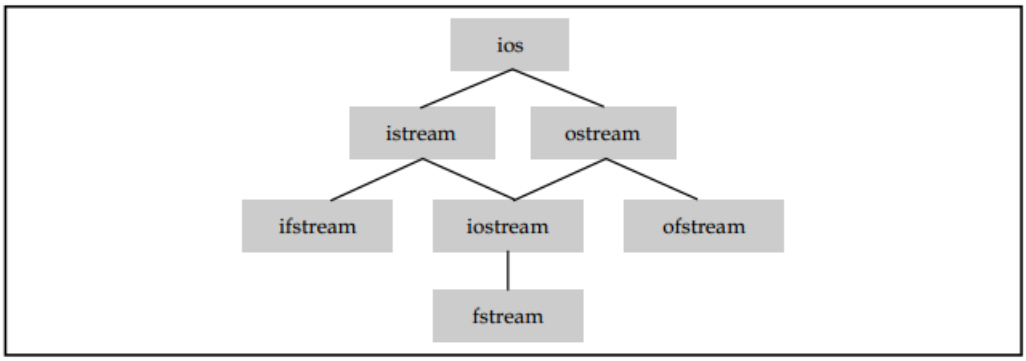
```
#include <iostream>          // Header for stream I/O.
int main(void)
{
int x;                       // variable to hold the input integer
cout << "Enter an integer: ";
cin >> x;
cout << "\n You have entered " <<x;
}
```

10.2 Hierarchy of Console Stream Classes

Data files can also be linked to streams. fstream.h and/or strstream.h define the essential stream classes for file I/O.

fstream : File I/O class.

ifstream	Input file class.
istrstream	Input string class.
ofstream	Output file class.
ostrstream	Output string class.
strstream	String I/O class.



There are some special functions that can alter the state the stream. These functions are called manipulators. Stream manipulators are defined in iomanip.h.

dec	Sets base 10 integers.
endl	Sends a new line character.
ends	Sends a null (end of string) character.
flush	Flushes an output stream.
fixed	Sets fixed real number notation.
oct Sets base 8 integers.	oct Sets base 8 integers.
ws	Discard white space on input.
setbase(int)	Sets integer conversion base (0, 8, 10 or 16 where 0 sets base 10).
setfill(int)	Sets fill character.
setprecision(int)	Sets precision.
setw(int)	Sets field width.
resetiosflags(long)	Clears format state as specified by argument.

A file may be opened for a number of file operations. The corresponding stream must be set with the intended operation. The different file stream modes are indicated by File Access Flags as listed below:

Ios::app	Open in append mode.
Ios::ate	Open and seek to end of file.
Ios::in	Open in input mode.
Ios::nocreate	Fail if file doesn't already exist.
Ios::noreplace	Fail if file already exists.
Ios::out	Open in output mode.
Ios::trunc	Open and truncate to zero length.
Ios::binary	Open as a binary stream.

10.3 Reading/Writing a Character from/into a File

Reading and writing a character in a data file has been dealt with in the previous sections in detail. The general procedure of reading a file one character at a time is listed below:

1. Create an input file stream from <fstream.h> header file:

ifstream name_of_input_stream;

2. Open the data file by passing the file name (optionally full name) to this input stream:

name_of_input_stream.open("data.dat");

Both the above statements can be combined in the following:

ifstream name_of_input_stream("data.dat");

3. Set up a character type variable to hold the read character.

char ch;

4. Read a character from the opened file using get() function:

name_of_input_stream.get(ch);

This way you can read the entire file in a loop stopping condition of the loop being the end of file:

```
while(!filename.eof())
{
name_of_input_stream.get(ch);
//process the read character
}
```

5. When finished close the file using close() function:

name_of_input_stream.close();

The if stream class is defined in fstream.h header file. Therefore you must include this file in your program. The complete program is listed below.

```
//reading a file one character at a time
#include <fstream.h>
void main() //the program starts here
{

ifstream filename("c:\cppio.dat");
char ch;
```

```
while(!filename.eof())
{
filename.get(ch);
cout << ch;
}
filename.close();
}
```

The general procedure of writing one character at atime in a file is listed below:

1. Create an output file stream from <fstream.h> header file:

ofstream name_of_output_stream;

2. Open the data file by passing the file name (optionally full name) to this output stream:

name_of_output_stream.open("data.dat");

Both the above statements can be combined in the following:

ofstream name_of_output_stream("data.dat");

3. Write a character in the opened file using << operator:

name_of_output_stream << 'A';

4. When finished close the file using close() function:

name_of_output_stream.close();

 Lab Exercise

```
//Program to write in file
#include <iostream>
#include <fstream>
using namespace std;
int main()
{
   ofstream of("demo.txt");
   of<< "Writing to file using fstream constructor!" << endl;
   of.close ();
   return 0;
}
```

 Lab Exercise

```
//Program to read from file
#include <iostream>
#include <fstream>
using namespace std;
int main()
```

```
{
string text;
ifstream ReadFile("a.txt");
while (getline (ReadFile, text)) {
  cout << text;
}
ReadFile.close();
return 0;
}
```

Note: - Create **a.text** file and write something in the file before execute above program.

10.4 File pointer & manipulator

Each file object has two integer values associated with it:

get pointer and put pointer

These values specify the byte number in the file where reading or writing will take place.

By default reading pointer is set at the beginning and writing pointer is set at the end (when file open in ios::app mode)

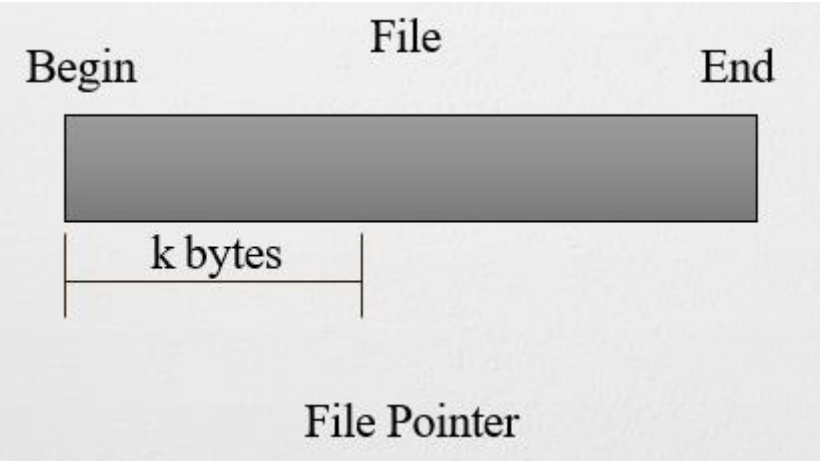
Functions associated with file pointers:

- The **seekg()** and **tellg()** functions allow you to set and examine the get pointer.
- The **seekp()** and **tellp()** functions allow you to set and examine the put pointer.

seekg() function

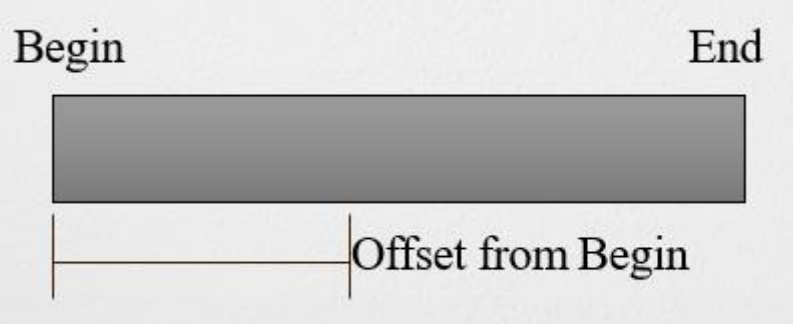
With one argument:

seekg(k) where k is absolute position from the beginning. The start of the file is byte 0.



With two argument:

- The first argument represents an offset from a particular location in the file.
- The second specifies the location from which the offset is measured.



tellg() function

- The tellg() function is used with input streams and returns the current "get" position of the pointer in the stream.

Lab Exercise

```
//Program
#include <iostream>
#include <fstream>
using namespace std;

int main()
{
   fstream file;
   file.open("details.txt",ios::out);
   if(!file){
      cout<<"Error";
      }
     file<<"OOP using C++";
    cout<<"current position in file"<<file.tellp()<<endl;
file.close();
file.open("details.txt",ios::in);
if(!file){
   cout<"error";
}

   cout<<"After open again file position is "<<file.tellg()<<endl;
char ch;
```

```
while(!file.eof()){
   cout<<"Position"<<file.tellg();
   file>>ch;
   cout<<" Character \" "<<ch<<"\""<<endl;

}
file.close();

    }
```

Output

```
current position in file13
After open again file position is 0
Position0 Character " O"
Position1 Character " O"
Position2 Character " P"
Position3 Character " u"
Position5 Character " s"
Position6 Character " i"
Position7 Character " n"
Position8 Character " g"
Position9 Character " C"
Position11 Character " +"
Position12 Character " +"
Position13 Character " +"

Process returned 0 (0x0)   execution time : 0.136 s
Press any key to continue.
```

tellp() and seekp() function

- tellp() function returns the position of pointer then using seekp() function the pointer is shift back from n position.

 Lab Exercise

```
//Program
#include <iostream>
#include<fstream>
using namespace std;
main(){
ofstream file("details.txt",ios::in);
if(!file){
   cout<<"Error";
}
else{
   cout<<file.tellp()<<endl;
```

```
   file<<"Good Programming";
   file.seekp(150);
  cout<<file.tellp()<<endl;
   file<<"Good Programming";
   file.close();
}
}
```

 Task

Write a program to demonstrate working of file pointers in C++.

ignore()

This function is used when reading a file to ignore certain number of characters. You can use seekg() as well for this purpose just to move the pointer up in the file. However, ignore() function has one advantage over seekg() function. The prototype is of ignore() function is given below.

fstream& ignore(int, char);

Caution

You can specify a delimiter character in ignore() function whence it ignores all the characters up to the first occurrence of the specified delimiter.

Where int is the count of characters to be ignored and delimiter is the character up to which you would like to ignore as demonstrated in the following program.

```
//demonstration of ignore() function
#include <fstream.h>
void main()
{
//Assume that the text contained in data.dat file is "Welcome
to C++"
ifstream myfile("data.dat");
static char Carray[10];
//go on ignoring all the characters in the input up to 10th
character unless an 'm' is found
myfile.ignore(10,'m');
myfile.read(Carray,10);
cout << Carray << endl; //it should display "me to C++"
myfile.close();
}
```

getline()

This function is used to read one line at a time from an input stream until some specified criterion is met. The prototype is as follows:

getline(Array,Array_size,delimiter);

The stopping criterion can be either specified number of characters (Array_size) or the first occurrence of a delimiter (delimiter) else the entire line (up to newline character '\n') is read. If you wish to stop reading until one of the following happens:

1. You have read 10 characters

2. You met the letter 'm'

3. There is new line

Then the function will be called as follows:

getline(Carray,10,'m');

The use of getline() function is demonstrated in the following example.

```
//Demonstration of getline() function to read a file line-wise
#include <fstream.h>
void main()
{
//Assume that the text contained in data.dat file is "Welcome to
C++"
ifstream myfile("data.dat");
static char Carray[10];
myfile.getline(Carray,10,'m');
cout << Carray << endl; //the output should be "Welco"
myfile.close();
}
```

peek()

This function returns the ASCII code of the current character from an input file stream very much like get() function, however, without moving the pointer to the next character. Therefore, any number of successive call to peek() function will return the ASCII code of same character each time. To convert the ASCII code (as returned by peek() function use char type cast) as demonstrated in the following code program.

```
//Demonstration of peek() function
#include <fstream.h>
void main()
{
// Assume that the text contained in data.dat file is "Welcome to
C++"
ifstream myfile("data.dat");
char ch;
myfile.get(ch);
cout << ch << endl; //the output should be 'W' and the pointer
will move to point 'e'
cout << char(myfile.peek()) << endl; //should display "e"
cout << char(myfile.peek()) << endl; //should display "e" again
cout << myfile.peek() << endl; //should display 101
myfile.get(ch);
```

```
cout << ch << endl; //will display "e" again and move the
pointer to 'l'
myfile.close();
}
```

putback()

This function returns the last read character, and moves the pointer back. In other words, if you use get() to read a char and move the pointer to next character, then use putback(), it will show you the same character, but it will set the pointer to previous character, so the next time you call get() again, it will again show you the same character as shown in the following program.

```
//Program demonstrating use of putback() function
#include <fstream.h>
void main()
{
// Assume that the text contained in data.dat file is "Welcome to
C++"
ifstream myfile("data.dat");
char ch;
myfile.get(ch);
cout << ch << endl; //output will be 'W'
myfile.putback(ch);
cout << ch << endl; //output will again be 'W'
myfile.get(ch);
cout << ch << endl; // output will again be 'W'
myfile.close();
}
```

flush()

I/O streams are created and maintained in the RAM. Therefore, when dealing with the output file stream, the data is not saved in the file as the program enters them. A buffer in the memory holds the data until the time you close the file or the buffer is full. When you close the file the data is actually saved in the designated file on the disk. Once the data has been written to the disk the buffer becomes empty again.

In case you want to force the data be saved even though the buffer is not full without closing the file you can use the flush() function. A call to flush() function forces the data held in the buffer tobe saved in the file on the disk and get the buffer empty.

10.5 Sequential Input and Output Operations

The file stream classes support a number of member functions for performing the input and output operations on files.

The get() and put() functions are capable of handling a single character at a time.

The getline() function lets you handle multiple characters at a time.

Sequential File Structure in C++

- C++ imposes no structure on the data stored in files, they typically are stored in an unordered format.
- If we want to set any form of order in the file data, we must do so programmatically.

Example

```
//Program
#include<iostream>
#include<fstream>
using namespace std;
 int main()
{
        char fname[20], ch;
ifstream fin;
   cout<<"Enter the name of the file: ";
        cin.get(fname, 20);
        cin.get(ch);
        fin.open(fname, ios::in);
        if(!fin)
        {
                cout<<"Error ";
                }
        while(fin)
       {
                fin.get(ch);
                cout<<ch;
        }
                fin.close();
        return 0;
}
```

Output

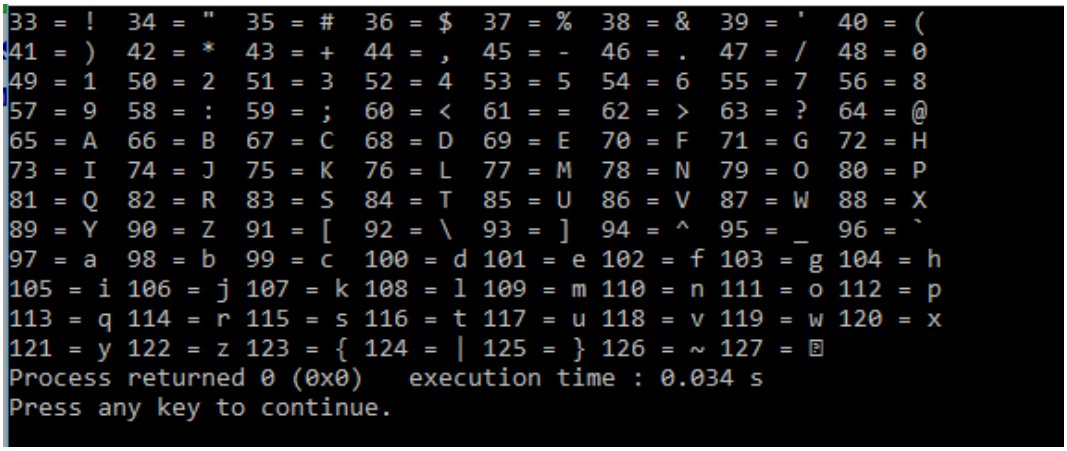
```
Enter the name of the file: details.txt
Helloo
Press any key to exit...

Process returned 0 (0x0)   execution time : 7.680 s
Press any key to continue.
```

 Lab Exercise

```
#include<iostream>
#include<fstream>
#include<stdlib.h>
using namespace std;
int main()
{
        ofstream fout;
        char ch;
        int line=0;
        int i;
        fout.open("sample.txt", ios::app) ;
        if(!fout)
        {
                cout<<"Error"<<endl;
                }
        for(i=33; i<128; i++)
        {
                fout.put((char)(i));
        }
        fout.close();
        ifstream fin;
        fin.open("sample.txt", ios::in);
        fin.seekg(0);
        for(i=33; i<128; i++)
        {
                fin.get(ch);
                cout<<i<<" = ";
                cout.put((char)(i));
                cout<<"\t";
                if(!(i%8))
                {
                        cout<<"\n";
                        line++;
                }
                if(line>22)
                {
                        system("PAUSE");
                        line = 0;
                }
```

```
        }
  return 0;
}
```

Output



10.6 Update a File

Updating a file is process of changing values in one or more records of a file.

Random Access

Now we have a file open for reading. By default, the file is open and the cursor is sitting at the beginning of the file. But now we want to move the file pointer that is the read/write location inside your file. Note that we told C++ to open the file in append mode. This means our pointer is now at the END of the file.

Once you have a file open for processing, you can navigate to different parts of the file. This is often referred to as random access of files. It really means that you aren't at the beginning or the end. Two functions are available:

istream: seekg() (or seek and get)

ostream: seekp() (seek and put)

Did you know?

Types of Files

There are three basic types of files:

Regular	Stores data (text, binary, and executable).
directory	Contains information used to access other files.
special	Defines a FIFO (first-in, first-out) pipe file or a physical device.

All file types recognized by the system fall into one of these categories. However, the operating system uses many variations of these basic types.

Regular Files

Regular files are the most common files. Another name for regular files is ordinary files. Regular files contain data.

Text Files

Text files are regular files that contain information readable by the user. This information is stored in ASCII. You can display and print these files. The lines of a text file must not contain NUL characters, and none can exceed {LINE_MAX} bytes in length, including the new-line character.

The term text file does not prevent the inclusion of control or other nonprintable characters (other than NUL). Therefore, standard utilities that list text files as inputs or outputs are either able to process the special characters gracefully or they explicitly describe their limitations within their individual sections.

Binary Files

Binary files are regular files that contain information readable by the computer. Binary files may be executable files that instruct the system to accomplish a job. Commands and programs are stored in executable, binary files. Special compiling programs translate ASCII text into binary code.

The only difference between text and binary files is that text files have lines of less than {LINE_MAX} bytes, with no NUL characters, each terminated by a new-line character.

10.7 Different Types of Files

A file is a collection of letters, numbers, and special characters that can be used to create a programme, a database, a dissertation, a reading list, or a simple letter, among other things. You can sometimes import a file from another location, such as another computer. You'll need to create a file if you wish to insert your own text or data. You'll need to return to a file later to update its contents, whether you copied it from somewhere else or generated it yourself.

The most common file systems rely on an underlying data storage device that provides access to a set of fixed-size blocks, referred to as sectors,that are typically 512 bytes in size. The file system software is in charge of grouping these sectors into files and directories, as well as keeping track of which sectors belong to which files and which aren't. Most file systems handle data in "clusters" or "blocks," which are fixed-sized units containing a specific number of disc sectors (usually 1-64). This is the minimum amount of logical disc space that can be assigned to a file.

However, file systems need not make use of a storage device at all. A file system can be used to organize and represent access to any data, whether it be stored or dynamically generated (e.g, from a network connection).

Whether the file system has an underlying storage device or not, file systems typically have directories which associate file names with files, usually by connecting the file name to an index into a file allocation table of some sort, such as the FAT in an MS-DOS file system, or an inode in a Unix-like file system. Directory structures may be flat, or allow hierarchies where directories may contain subdirectories. In some file systems, file names are structured, with special syntax for filename extensions and version numbers. In others, file names are simple strings, andper-file metadata is stored elsewhere.

Other bookkeeping information is typically associated with each file within a file system. The length of the data contained in a file may be stored as the number of blocks allocated for the file or as an exact byte count. The time that the file was last modified may be stored as the file's timestamp. Some file systems also store the file creation time, the time it was last accessed, and the time that the file's meta-data was changed. (Note that many early PC operating systems did not keep track of file times.) Other information can include the file's device type (e.g., block,character, socket, subdirectory, etc.), its owner user-ID and group-ID, and its access permission settings (e.g., whether the file is read-only, executable, etc.).

The hierarchical file system was an early research interest of Dennis Ritchie of Unix fame; previous implementations were restricted to only a few levels, notably the IBM implementations, even of

their early databases like IMS. After the success of Unix, Ritchie extended the file system concept to every object in his later operating system developments, such as Plan 9 and Inferno.

Traditional file systems offer facilities to create, move and delete both files and directories. They lack facilities to create additional links to a directory (hard links in Unix), rename parent links (".." in Unix-like OS), and create bidirectional links to files.

Traditional file systems also offer facilities to truncate, append to, create, move, delete and inplace modify files. They do not offer facilities to prepend to or truncate from the beginning of a file, let alone arbitrary insertion into or deletion from a file. The operations provided are highly asymmetric and lack the generality to be useful in unexpected contexts. For example, interprocess pipes in Unix have to be implemented outside of the file system because the pipes concept does not offer truncation from the beginning of files.

Secure access to basic file system operations can be based on a scheme of access control lists or capabilities. Research has shown access control lists to be difficult to secure properly, which is why research operating systems tend to use capabilities. Commercial file systems still use access control lists.

A file system is a method for storing and organizing computer files and the data they contain to make it easy to find and access them. File systems may use a data storage device such as a hard disk or CD-ROM and involve maintaining the physical location of the files, they might provide access to data on a file server by acting as clients for a network protocol (e.g., NFS, SMB, or 9P clients), or they may be virtual and exist only as an access method for virtual data. More formally, a file system is a set of abstract data types that are implemented for the storage, hierarchical organization, manipulation, navigation, access, and retrieval of data.

10.8 Types of File Systems

File system types can be classified into disk file systems, network file systems and special purpose file systems.

1. **Disk file systems:** A disk file system is a file system designed for the storage of files on a data storage device, most commonly a disk drive, which might be directly or indirectly connected to the computer. Examples of disk file systems include FAT, FAT32, NTFS, HFS and HFS+, ext2, ext3, ISO 9660, ODS-5, and UDF. Some disk file systems are journaling file systems or versioning file systems.
2. **Flash file systems:** A flash file system is a file system designed for storing files on flash memory devices. These are becoming more prevalent as the number of mobile devices is increasing, and the capacity of flash memories catches up with hard drives. While a block device layer can emulate a disk drive so that a disk file system can be used on a flash device, this is suboptimal for several reasons:

   (a) **Erasing blocks:** Flash memory blocks have to be explicitly erased before they can be written to. The time taken to erase blocks can be significant, thus it is beneficial to erase unused blocks while the device is idle.

   (b) **Random access:** Disk file systems are optimized to avoid disk seeks whenever possible, due to the high cost of seeking. Flash memory devices impose no seek latency.

   (c) **Wear leveling:** Flash memory devices tend to wear out when a single block is repeatedly overwritten; flash file systems are designed to spread out writes evenly.

Log-structured file systems have all the desirable properties for a flash file system. Such file systems include JFFS2 and YAFFS.

3. Database file systems: A new concept for file management is the concept of a databasebased file system. Instead of, or in addition to, hierarchical structured management, files are identified by their characteristics, like type of file, topic, author, or similar metadata. Example: dbfs.

4. **Transactional file systems:** Each disk operation may involve changes to a number of different files and disk structures. In many cases, these changes are related, meaning that it is important that they all be executed at the same time. Take for example a bank sending another bank some money electronically. The bank's computer will "send" the transfer instruction to the other bank and also update its own records to indicate the transfer has occurred. If for some reason the computer crashes before it has had a chance to update its own records, then on reset, there will be no record of the transfer but the bank will be missing some money.

   Transaction processing introduces the guarantee that at any point while it is running, a transaction can either be finished completely or reverted completely (though not necessarily both at any given point). This means that if there is a crash or power failure, after recovery, the stored state will be consistent. (Either the money will be transferred or it will not be transferred, but it won't ever go missing "in transit".)

5. **Network file systems:** A network file system is a file system that acts as a client for a remote file access protocol, providing access to files on a server. Examples of network file systems include clients for the NFS, SMB protocols, and file-system-like clients for FTP and WebDAV.
6. **Special purpose file systems:** A special purpose file system is basically any file system that is not a disk file system or network file system. This includes systems where the files are arranged dynamically by software, intended for such purposes as communication between computer processes or temporary file space.

   Special purpose file systems are most commonly used by file-centric operating systems such as Unix. Examples include the procfs (/proc) file system used by some Unix variants, which grants access to information about processes and other operating system features.

10.9 Binary Files

In binary files, to input and output data with the extraction and insertion operators (<< and >>) and functions likegetline is not efficient, since we do not need to format any data, and data may not use the separation codes used by text files to separate elements (like space, newline, etc...). File streams include two member functions specifically designed to input and output binary data sequentially: write and read. The first one (write) is a member function of ostream inherited by ofstream. And read is a member function of istream that is inherited by ifstream. Objects of class fstream have both members. Their prototypes are:

write (memory_block, size);

read (memory_block, size);

Where memory_block is of type "pointer to char" (char*), and represents the address of an array of bytes where the read data elements are stored or from where the data elements to be written are taken. The size parameter is an integer value that specifies the number of characters to be read or written from/to the memory block.

 Example

```
// reading a complete binary file
#include <iostream>
#include <fstream>
using namespace std;
ifstream::pos_type size;
char * memblock;
int main () {
```

```
ifstream file ("example.bin", ios::in|ios::binary|ios::ate);
if (file.is_open())
{
size = file.tellg();
memblock = new char [size];
file.seekg (0, ios::beg);
file.read (memblock, size);
file.close();
cout << "the complete file content is in memory";
delete[] memblock;
}
else cout << "Unable to open file";
return 0;
} the complete file content is in memory
```

10.10 Command Line Argument in C++

Any input value is passed through command prompt at the time of running of program is known as command line argument.

- They facilitate the use of your program in batch files
- They give a professional appearance to your program

A command-line argument is the information that follows the name of the program on the command line of the operating system.

C++ defines two built-in parameters to main()

- They receive the command line arguments
- Their names are argc and argv

The names of the parameters are arbitrary. However, argc and argv have been used by convention for several years.

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

- argc is an integer Holds the number of arguments on the command line Since the name of the program always corresponds to the first argument, it is always at least 1

argv is a pointer to an array of character pointers.

Each character pointer in the argv array corresponds a string containing a command-line argument

argv[0] points the name of the program, argv[1] points to the first argument, argv[2] points to the second argument and so in

Note: Each command-line argument is a string

If you want to pass numerical information to your program, your program should convert the corresponding argument into its numerical equivalent

Note: Each command-line argument must be separated by spaces or tabs

Commas, semicolons, and the like are not valid argument separators

Passing numeric command-line arguments

All command-line arguments are passed to the program as strings

- Program should convert them into their proper internal format

C++ supports standard library functions

- **atof()** : converts a string to a double and returns the result
- **atoi() :** converts a string to a int and returns the result
- **atol() :** converts a string to a long int and returns the result

Properties of Command Line Arguments

- They are passed to main() function.
- They are parameters/arguments supplied to the program when it is invoked.
- They are used to control program from outside instead of hard coding those values inside the code.
- argv[argc] is a NULL pointer.
- argv[0] holds the name of the program.
- argv[1] points to the first command line argument and argv[n] points last argument.

Operating System parses command line arguments

An Operating System (OS) is an interface between a computer user and computer hardware.

Operating System

- Command Line Interface
- Graphical User Interface

Command Line Interface

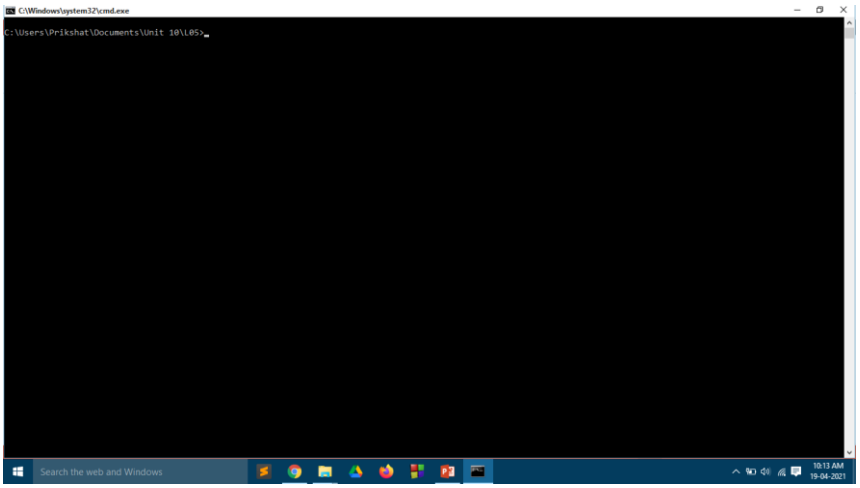
A command-line interface (CLI) processes commands to a computer program in the form of lines of text.

Example

LINUX

UNIX

DOS



Graphical user Interface

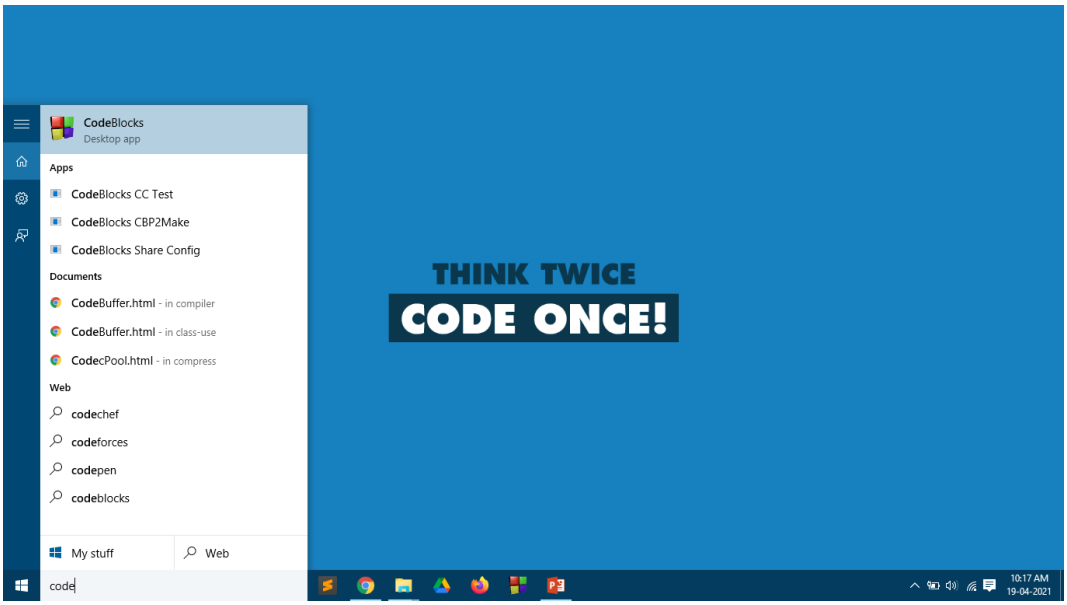
A GUI (graphical user interface) is a system of interactive visual components for computer software.

Example

WINDOWS OS

MAC OS

UBUNTU (LINUX)



Lab Exercise

```
#include<iostream>
using namespace std;
int main(int argc, char* argv[])
{
int i;
cout<<"Total number of arguments: "<<argc;
for(i=0;i< argc;i++)
{
cout<<endl<< i<<"argument: "<<argv[i];
}
return 0;
}
```

Output

No argument passed

```
Total number of arguments: 1
0argument: C:\Users\Prikshat\Desktop\a.exe
Process returned 0 (0x0)   execution time : 0.095 s
Press any key to continue.
```

Command line arguments passed

```
C:\Users\Prikshat\Desktop>a.exe 12 21 35 25 36
Total number of arguments: 6
0argument: a.exe
1argument: 12
2argument: 21
3argument: 35
4argument: 25
5argument: 36
C:\Users\Prikshat\Desktop>
```

Summary

- Fields in C++ are interpreted as a sequence of or stream of bytes stored on some storage media'. Member functions of these or base classes are used to perform I/O operations.
- The read( ) and write( ) functions work in binary mode. The ifstream class is used for input, ofstream for output and itstream for both input and output.
- A data of a file is stored in the form of readable and printable characters then the file is known as text file. A file contains non-readable characters in binary code then the file is called binary file.
- The function get( ) read and write data respectively. The read( ) and write( ) function read and write block of binary data. The close( ) function close the stream.
- C++ treats each source of input and output uniformly. The abstraction of a data source and data sink is what is termed as stream. A stream is a data abstraction for input/output of data to and fro the program.
- C++ library provides prefabricated classes for data streaming activities. In C++, the file stream classes are designed with the idea that a file should simply be viewed as a stream or array or sequence of bytes.

Keywords

Command Line Parameters: The main functions may be defined not to have any parameter. In some cases, though, the program is provided with some input values at the line of execution. These values are known as command line parameter.

File: A storage unit that contains data. A file can be stored either on tape or disk.

Input Stream: The stream that suppliers date to the program is known as input stream.

Output Stream: The stream that receives data from the program is known as output stream.

Stream: A stream is a general name given to a flow of data.

Self Assessment

1. Which header file is required to use file I/O operations?

A. <ifstream>

B. <ostream>

C. <fstream>

D. <iostream>

2. By default, all the files in C++ are opened in _____ mode.

A. Text

B. Binary

C. ISCII

D. VTC

3. Which among following is used to open a file in binary mode ?

A. ios:app

B. ios::out

C. ios::in

D. ios::binary

4. Which function is used in C++ to get the current position of file pointer in a file?

A. tell_p()

B. get_pos()

C. get_p()

D. tell_pos()

5. The tellg() function is used with input streams and returns the current "get" position of the pointer in the stream.

A. True

B. False

6. By default reading pointer is set at the end when file open in ios::app mode.

A. True

B. False

7. The get() and put() functions are capable of handling a single character at a time.

A. True

B. False

8. Which of the following is the correct syntax to read the single character to console in the C++ language?

A. Read ch()

B. Getline vh()

C. get(ch)

D. Scanf(ch)

9. Which of the following is the correct syntax to read the single character to console in the C++ language?

A. Print ch()

B. Printtline vh()

C. put(ch)

D. Scanf(ch)

10. Updating of the file means

A. adding new data

B. deleting any existing data

C. changing the existing record items

D. all of above

11. For a direct access file :

A. there are restrictions on the order of reading and writing

B. there are no restrictions on the order of reading and writing

C. access is restricted permission wise

D. access is not restricted permission wise

12. The command line arguments are handled using?

A. void()

B. main()

C. header files

D. macros

13. argc refers to the?

A. number of arguments passed

B. a pointer array

C. Both number of argument passed and pointer array

D. None of the above

14. What will be output for the following code?

```
#include <stdio.h>

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

{

   printf(""%d"", argc);

   return 0;

}
```

A. 0

B. 1

C. error

D. Depends on the compiler

15. What does the second parameter of the main function represent?

A. Number of command line arguments

B. List of command line arguments

C. Dictionary of command line arguments

D. Stack of command line arguments

Review Questions

1. How is C++ able to treat all the input and output operation uniformly?
2. What do you mean by file pointers? Explain in detail.
3. Write a note on read() and write() functions.
4. What is a file? Explain different types of files.
5. How can you achieve random access in C++?
6. Write a program in C++ to create a file.
7. Write a program in C++ that creates a text file, which is an exact copy of a given file.

Answers for Self Assessment

1.	C	2.	A	3.	D	4.	A	5.	A
6.	B	7.	A	8.	C	9.	C	10.	D
11.	B	12.	B	13.	A	14.	B	15.	B

Further Readings

E Balagurusamy; Object-oriented Programming with C++; Tata Mc Graw-Hill.

Herbert Schildt; The Complete Reference C++; Tata Mc Graw Hill.

Robert Lafore; Object-oriented Programming in Turbo C++; Galgotia.

Web Links

https://study.com/academy/lesson/practical-application-for-c-plus-plus-programming-working-with-files.html

https://www.codecademy.com/learn/learn-c-plus-plus

https://www.guru99.com/cpp-file-read-write-open.html

Prikshit Kumar Angra, Lovely Professional University

Unit 11: Generic Programming with Templates

CONTENTS

Objectives

After studying this unit, you will be able to:

- Describe the function of templates
- Describe the classes of templates
- Understand generic programming

Introduction

One of C++'s most advanced and powerful features is the template. Despite the fact that it was not included in the original C++ specification, it was introduced several years ago and is now supported by all recent C++ compilers. It is possible to create generic functions and classes using templates. The kind of data on which a generic function or class is based. This allows the function or class to work with a variety of data types without having to explicitly recode specialised versions for each data type.

A template can be considered as a kind of macro. When an object of a specific type is defined for actual use, the template definition for that class is substituted with the required data type. Since a template is defined with a parameter that would be replaced by a specified data type at the time of actual use of the class or function, the templates are sometimes called parameterized classes or functions.

Did you know?

What is a template parameter?

A template parameter is a special kind of parameter that can be used to pass a type as argument: just like regular function parameters can be used to pass values to a function, template parameters allow to pass values and also types to a function.

11.1 Class Template

- Like function templates, we can also create class templates for generic class operations.
- A class implementation that is same for all classes, only the data types used are different.
- Generally, we need to create a different class for each data type OR create different member variables and functions within a single class.

Create object of class template

To create a class template object, you need to define the data type inside a < > when creation.

Syntax:-

```
className<dataType> classObject;
```


Example:

```
className<int> classObject;
className<double> classObject;
```


Lab Exercise

```
#include<iostream>
using namespace std;
template<class T>
class sum{
T num1,num2;
public:
   sum(T a, T b){
num1=a;
num2=b;
   }
   void disp(){
   cout<<"Numbers are "<<num1<< "  "<<num2<<endl;
   cout<<"sum is"<<add()<<endl;
   }
T add(){return num1+num2;};
};
main(){
sum<int> sumint(50,50);
sum<float> sumfl(50.25,15.225);
cout<<"Int Result" <<endl;
sumint.disp();
cout<<"Float Result"<<endl;
sumfl.disp();
}
```

Output

```
Int Result
Numbers are 50  50
sum is100
Float Result
Numbers are 50.25  15.225
sum is65.475

Process returned 0 (0x0)   execution time : 0.144 s
Press any key to continue.
```

11.2 Class Templates with Multiple Parameter

We can use more than one generic data type in a class template. They are declared as a comma-separated list within the template specification as shown below:

```
template<class T1, class T2, ...>
class classname
{
   .....
   .....    (Body of the class)
   .....
};
```

Lab Exercise

```
#include <iostream>

using namespace std;

template<class T1, class T2>
class Test
{
        T1 a;
        T2 b;
  public:
        Test(T1 x, T2 y)
        {
                a = x;
                b = y;
        }
        void show()
        {
                cout << a << " and " << b << "\n";
        }
};

int main()
{
        Test <float,int> test1 (1.23,123);
        Test <int,char> test2 (100,'W');

        test1.show();
        test2.show();

        return 0;
};
```

Output

1.23 and 123

100 and W

11.3 Function Template

In C ++ when a function is overloaded, many copies of it have to be created, one for each data type it acts on. In the example of the max ( ) function, which returns the greater of the two values passed to it this function would have to be coded for every data type being used. Thus, you will end up coding the same function for each of the types, like int, float, char, and double. A few versions of max ( ) are:

```
Int max ( int x, int y)
{
return x > y ? x : y
}
char max ( char x, char y )
{
return x > y ? > x : y
}
double max (double x , double y)
{
return x > y ? x : y
}
float max ( float x, float y)
{
return x > y ? x : y
}
```

Here you can see, the body of each version of the function is identical. The same code has to be repeated to carry out the same function on different data types. This is a waste of time and effort, which can be avoided using the template utility provided by C ++.

"A template function may be defined as an unbounded functions "all the possible parameters to the function are not known in advance and a copy of the function has to be created as and when necessary. Template functions are using the keyword, template. Templates are blueprints of a function that can be applied to different data types.

Syntax of Template

```
Template < class type 1, type 2 ... >
Void function - name ( type 2 parameter 1, type 1 parameter 2 ) {...}
```

Example

```
Template < class X >
X min ( X a , X b )
{
return (a < b ) ? a : b ;
}
```

This list of parameter types is called the formal parameter list of the template, and it cannot be empty. Each formal parameter consists of the keyword, type name, followed by an identifier. The identifier can be built-in or user-defined data type, or the identifier type. When the function is invoked with actual parameters, the identifier type is substituted with the actual type of the parameter. This allows the use of any data type. The template declaration immediately precedes the definition of the function for which the template is being defined. The template declaration, followed by the function definition, constitutes the template definition. The template of the max ( ) function is coded below:

```
# include < iostream.h >
template < class type >
type max ( type x , type y)
{
return x > y ? x : y ;
}
int main ( )
{
cout << " max ( 'A', 'a') : " << max ( 'A', 'a') << endl ;
cout << " max ( 30, 40 ) : " << max ( 30 , 40) << endl;
cout << " max ( 45 . 67F, 12 . 32 F) : " << max (45. 67F, 12 .
32 F) <,
endl;
return 0 ;
}
```

Output

```
max ( 'A' , 'a') : a
max ( 30 , 40 ) : 40
max ( 45.67F, 12 . 32F ) : 45 . 67
```

In the example, the list of parameters is composed of only one parameter. The next line specifies that the function takes two arguments and returns a value, all of the defined in the formal parameter list. See what happens if the following command is issued, keeping in mind the template definition for max ( ) :

```
max ( a , b ) ;
```

in which a and b are integer type variables. When the user invokes max ( ) , using two int values, the identifier, 'type', is substituted with int, wherever it is present. Now max ( ) works just like the function int max (int, int) defined earlier, to compare two int values. Similarly, if max ( ) is invoked using two double values, 'type' is replaced with 'double'. This process of substitution, depending on the parameter type passed to the function, is called template instantiation. The template specifies how individual functions will be constructed, given a set of actual types. The template facility allows the creation of a blueprint for a function like max ( ).

 Example

```
# include < iostream.h >
template < class type >
type square ( type a)
{
type b;
```

```
b= a*a ;
return b.;
}
int main ( )
{
cout << " square (25 ) : " << square ( 25) << endl;
cout << " square 40 : " << square 40 << endl;
return 0 ;
}
```

Output

Square (25 . 45F) : 1125

Square 90 : 1600

Here is another example of the use of template function. Consider the following classes:

```
class IRON
{
private :
float density ;
public :
IRON ( ) {density = 8.9 }
Float Density ( ) { return density ;}
} ;
```

 Lab Exercise

```
#include <iostream>
using namespace std;
template <typename T>
T add(T num1, T num2) {
   return (num1 + num2);
}
int main() {
   int result1;
   double result2;
   result1 = add<int>(50, 100);
   cout << "Sum of integers = " << result1 << endl;
   result2 = add<double>(10.2, 9.25);
   cout << "Sum of double =  " << result2 << endl;
   return 0;
}
```

Output

```
Sum of integers = 150
Sum of double =  19.45

Process returned 0 (0x0)   execution time : 0.015 s
Press any key to continue.
```

Did you know?

Is the template or blueprint can then be instantiated for all data types?

Yes, the template or blueprint can then be instantiated for all data types, eliminating duplication of the source code. The identifier, 'type', can be used within the function body of a function that, otherwise, remains unchanged.

11.4 The Typename Keyword

The keywords typename and class can be freely interchanged. For example

Template < class T >

Int maxindex (T arr [ ] , int size)

{

. . .

}

11.5 Template Specialization

While template functions and classes are usable for any data type, that would hold true only, as long as the body of the functions or the class is identical throughout. So, if you have,

Template < typename T .

Void swap (T + Ihs , T + rhs)

{

T tmp (+ Ihs) ;

+ Ihs = + rhs ;

+ rhs = tmp ;

}

the previous template will be instantiated correctly for any of the pointer types passed to it -except for the data types char + , unsigned char + , and signed char + . Since, the body of the function.

Caution:

Differs for these types, you would use template specialize the template function.

Specialization would consist of instantiating the template definition for a specific data type. So, when the compiler needs to instantiate the template, it finds a predefined version already existing and uses it. For example, just after the previous template declaration, you could create a template specialization for char + like this:

Template e >

Void swap < char > (char + Ihs , char + rhs)

{

char + tmp = new char [strlen (Ihs) + 1] ;

strcpy (tmp , Ihs) .;

strcpy (tmp , Ihs);

Strcpy (Ihs , rhs) ; Notes

Strcpy (rhs , tmp) ;

}

11.6 Function Templates with Multiple Parameters

Like template classes, we can use more than one generic data type in the template statement, using a comma-separated list as shown below:

```
template<class T1, class T2, ...>
returntype functionname(arguments of types T1, T2,...)
{
        .....
        .....   (Body of function)
        .....
}
```

Example

```
#include<iostream>
#include<string>
using namespace std;
template<class T1,class T2>
void display(T1 x,T2 y)
{
cout<<x<< " "<<y<< "\n";
}
int main(){
display(858,"ABD");
display(858.58, 1024);
return 0;
}
```

Output

858 ABD

858.58 1024

11.7 Point of Instantiation

A template function gets instantiated under the following circumstances:

1. Implicitly instantiated because it is referenced from a function call that depends on a template argument.
2. Implicitly instantiated because it is referenced within a default argument in a declaration.
3. The point of instantiation of a function template specialization immediately follows the declaration or definition that refers to the specialization.

11.8 Overloading of Template Function

A template function may be overloaded either by template functions or ordinary functions of its name. In such cases, the overloading resolution is accomplished as follows:

1. Call an ordinary function that has an exact match.
2. Call a template function that could be created with an exact match.
3. Try normal overloading resolution to ordinary functions and call the one that matches.

An error is generated if no match is found. Note that no automatic conversions are applied to arguments on the template functions. Program 12.8 shows how a template function is overloaded with an explicit function.


Notes

- Overloading a function template means having different sets of function templates which differ in their parameter list.
- If the function template is with the ordinary template, the name of the function remains the same but the number of parameters differs.


Example:

```
#include <iostream>
using namespace std;
template <class T>
void display(T t1)
{
   cout << "Display template value: "
        << t1 << "\n";
}
void display(int t1)
{
   cout << "With integer argument: "
        << t1 << "\n";
}
int main()
{
   display(100);
   display(10.2);
```

```
    display('A');
    return 0;
}
```

Output

```
With integer argument: 100
Display template value: 10.2
Display template value: A

Process returned 0 (0x0)   execution time : 0.045 s
Press any key to continue.
```

Task

Write a program to demonstrate working of function template in C++.

Write a program to demonstrate working of overloading function template in C++.

We have seen that a template can have multiple arguments. It is also possible to use non-type arguments. That is, in addition to the type argument T, we can also use other arguments such as strings, function names, constant expressions and built-in types. Consider the following example:

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

This template supplies the size of the array as an argument. This implies that the size of the array is known to the compiler at the compile time itself. The arguments must be specified whenever a template class is created. Example:

```
array<int,10>      al;          // Array of 10 integers
 array<float,5>    a2;          // Array of 5 floats
array<char,20>     a3;          // String of size 20
```

The size is given as an argument to the template class.

Summary

- C++ supports a mechanism known as template to implement the concept of generic programming.
- Templates classes may be defined as the layout and operations for an unbounded set of related classes.
- Templates allows us to generate a family of classes or a family of functions to handle different data types.

- Template classes and functions eliminate code duplication for different types and thus make the program development easier and more manageable.
- We can use multiple parameters in both the class templates and function templates. A specific class created from a class template is called a template class and the process of creating a template class is known as instantiation. Similarly, a specific function created from a function template is called a template function.
- Like other functions, template functions can be overloaded.
- Member functions of a class template must be defined as function templates using the parameters of the class template.
- While template functions and classes are usable for any data type, that would hold true only, as long as the body of functions or the class is identical throughout.
- We may also use non-type parameters such basic or derived data types as arguments templates.

Keywords

Template: A template function may be defined as an unbounded functions. All the possible parameters to the function are not known in advance and a copy of the function has to be created as and when necessary.

Template Classes: Template classes may be defined as the layout and operations for an unbounded set of related classes.

Function Template: We write a generic function that can be used for different data types. Function templates are special functions that can operate with generic types.

Generic Programming in C++: Generic programming is a technique where generic types are used as parameters in algorithms so that they can work for a variety of data types.

Self Assessment

1. What is meant by the template parameter?

A. It can be used to pass a type as an argument
B. It can be used to evaluate a type
C. It can of no return type
D. It can be used to delete a type

2. Which keyword can be used in template?

A. class
B. typename
C. both class & typename
D. function

3. What is a template?

A. A template is a formula for creating a generic class
B. A template is used to manipulate the class
C. A template is used for creating the attributes
D. None of the above

4. What is the validity of template parameters?

A. inside that block only
B. inside the class
C. whole program
D. inside the main class

5. Which of the following best defines the syntax for template function?

A. template return_type Function_Name(Parameters)
B. template return_type Function_Name(Parameters)
C. All of above
D. None of the above

6. From where does the template class derived?
A. Regular non-templated C++ class
B. Templated class
C. All of above
D. None of the above

7. Which of the following is used for generic programming?
a. Virtual functions
b. Modules
c. Templates
d. Abstract Classes

8. Which of the following is correct about templates?
A. It is a type of compile time polymorphism
B. It allows the programmer to write one code for all data types
C. Helps in generic programming
D. All of the mentioned

9. What is the difference between normal function and template function?
A. The normal function works with any data types whereas template function works with specific types only
B. Template function works with any data types whereas normal function works with specific types only
C. Unlike a normal function, the template function accepts a single parameter
D. Unlike the template function, the normal function accepts more than one parameters

10. What does this template function indicates?

```
template<class T>

T func(T a)

{

    cout<<a;

}
```

A. A function taking a single generic parameter and returning a generic type
B. A function taking a single generic parameter and returning nothing
C. A function taking single int parameter and returning a generic type
D. A function taking a single generic parameter and returning a specific non-void type

11. Can we have overloading of the function templates?
A. Yes
B. No
C. May Be
D. Can't Say

12. What is a function template?
A. creating a function without having to specify the exact type
B. creating a function with having an exact type
C. creating a function without having blank spaces
D. creating a function without class

13. Which keyword is used for the template?
A. Template
B. template
C. Temp
D. temp

14. Which of the following is correct about templates?
A. It is a type of compile time polymorphism
B. It allows the programmer to write one code for all data types
C. Helps in generic programming
D. All of the mentioned

15. Pick out the correct statement.
A. you only need to write one function, and it will work with many different types
B. it will take a long time to execute
C. Duplicate code is increased
D. it will take a long time to execute & duplicate code is increased

Review Questions

1. What is generic programming? How is it implemented in C++?
2. A template can be considered as a kind of macro. Then, what is the difference between them? Distinguish between overloaded functions and function templates.
3. Distinguish between the terms class template and template class.
4. A class (or function) template is known as a parameterized class (or function). Comment.
5. What are Templates?
6. Define template instantiation.
7. List down the rules for using templates.
8. How can you get a template function instantiated?
9. Define the template class with the explaining program.
10. When can a template function be instantiated?

Answers for Self Assessment

1.	A	2.	C	3.	A	4.	A	5.	C
6.	C	7.	C	8.	D	9.	B	10.	A
11.	A	12.	A	13.	B	14.	C	15.	A

Further Readings

E Balagurusamy; Object-oriented Programming with C++; Tata Mc Graw-Hill.

Herbert Schildt; The Complete Reference C++; Tata Mc Graw Hill.

Robert Lafore; Object-oriented Programming in Turbo C++; Galgotia.

Web Links

https://www.codecademy.com/learn/learn-c-plus-plus

https://isocpp.org/wiki/faq/templates

https://www.startertutorials.com/blog/generic-programming-cpp.html

Unit 12: More on Templates

CONTENTS

Objectives

After studying this unit, you will be able to:

- Describe the class template and inheritance
- Understand recursion with template function
- Understand preprocessors
- Define Macros

Introduction

A template can be considered as a kind of macro. When an object of a specific type is defined for actual use, the template definition for that class is substituted with the required data type. Since a template is defined with a parameter that would be replaced by a specified data type at the time of actual use of the class or function, the templates are sometimes called parameterized classes or functions.

12.1 Class Template and Inheritance

- It is possible to inherit from a template class. All the usual rules for inheritance and polymorphism apply.
- If we want the new, derived class to be generic it should also be a template class; and pass its template parameter along to the base class.

Example

```
#include<iostream>
using namespace std;
template <typename T>
```

```
class base {
protected:
   int x=100,y=10;
};

template <typename T>
class derived : public base<T> {

public:
   int sum() { return this->x+this->y; }
};
int main() {
   derived<int> d;
   cout<<"Sum of x and y is"<<d.sum();
   return 0;
}
```

Output

```
Sum of x and y is110
Process returned 0 (0x0)   execution time : 0.158 s
Press any key to continue.
```

Task: Write a program using C++ that demonstrate working of class template and inheritance.

12.2 Recursion with Template Function

- The basic purpose of templates in the C++ language is to support functions and classes parameterized by type. However, templates can also be used to encode arbitrarily complex computational procedures.
- In order to implement a procedure using templates, the procedure must be formulated recursively.

Example

```
#include <iostream>
 using namespace std;
void tprintf(const char* format)
{
   cout << format;
}
template<typename T, typename... Targs>
void tprintf(const char* format, T value, Targs... Fargs)
{
```

```
   for ( ; *format != '\0'; format++ ) {
      if ( *format == '%' ) {
      cout << value;
        tprintf(format+1, Fargs...);
        return;
      }
 cout << *format;
   }
}
int main()
{
   tprintf("% Programming% %\n","C++",'!',123);
   return 0;
}
```

Output

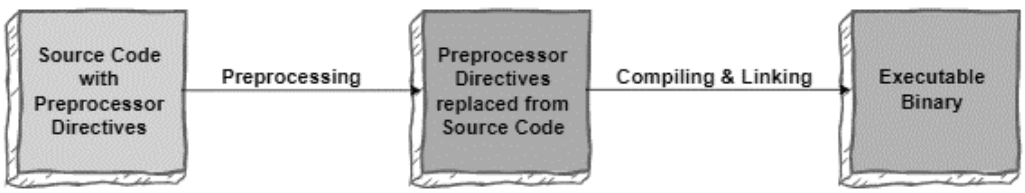
```
C++ Programming! 123

Process returned 0 (0x0)   execution time : 0.017 s
Press any key to continue.
```

Task: Write a program using C++ that demonstrate working of recursion and template function.

12.3 Preprocessors in C++

The preprocessor is a directive given to the compiler before the compilation phase of the program. The compiler executes these directives before compiling the code into an executable file.

Take a look at the image below:



The image above illustrates that, after the Preprocessing phase, all preprocessor directives are replaced with C++ code in the source code file. Then, the compilation begins.

Did you know?

- Preprocessors are programs that process our source code before compilation.

- Preprocessor programs provide preprocessors directives which tell the compiler to preprocess the source code before compiling.
- All of these preprocessor directives begin with a '#' (hash) symbol.

Structure of a Preprocessor directive

A preprocessor directive begins with the # character – only white-space characters can come before a preprocessor directive on a line. The directives may have additional tokens after them on the line. For example:

#example_directive token1 token2

Common Preprocessor directives

The following are some preprocessor directives that beginner programmers encounter.

#include Preprocessor directive

The structure of the #include directive is:

#include filename

The filename is the name of a standard library (if it is enclosed in ankle brackets <>) or a custom C++ file (if it is enclosed in double quotes "").

When the compiler encounters the #include directive, it replaces the line with the contents of the file.

#define Preprocessor directive

The #define directive is used to define Macros and Symbolic Constants. The structure of the #define directive is:

#define identifier replacement-text

In the preprocessing phase, the compiler replaces any occurrences of identifier that don't occur as part of a string literal or a comment with the replacement-text.

#error Preprocessor directive

The structure of the #error directive is:

#error tokens

Here, the tokens are characters separated by white-spaces.

In the preprocessing phase, if the compiler encounters the #error preprocessor directive, it terminates the compilation and prints the tokens as error on the standard output.

Conditional inclusion Preprocessor directives

Conditional inclusion Preprocessor directives consist of the following:

#if

#ifdef

#ifndef

#elif

#else

#endif

These directives instruct the compiler to conditionally include parts of the source code for compilation. The following code provides an example:

Example

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

#define PI 3.142

int main() {
  #ifdef PI
    cout << "The value of PI is: " << PI << endl;
  #else
    cout << "PI is not defined!" << endl;
  #endif
  return 0;
}
```

12.4 Macros in C++

A macro is a fragment of code which has been given a name. Whenever the name is used, it is replaced by the contents of the macro. There are two kinds of macros. They differ mostly in what they look like when they are used. Object-like macros resemble data objects when used, function-like macros resemble function calls.

Did you know?

- Macros are a piece of code in a program which is given some name. Whenever this name is encountered by the compiler the compiler replaces the name with the actual piece of code.
- The '#define' directive is used to define a macro.

Importance of macros

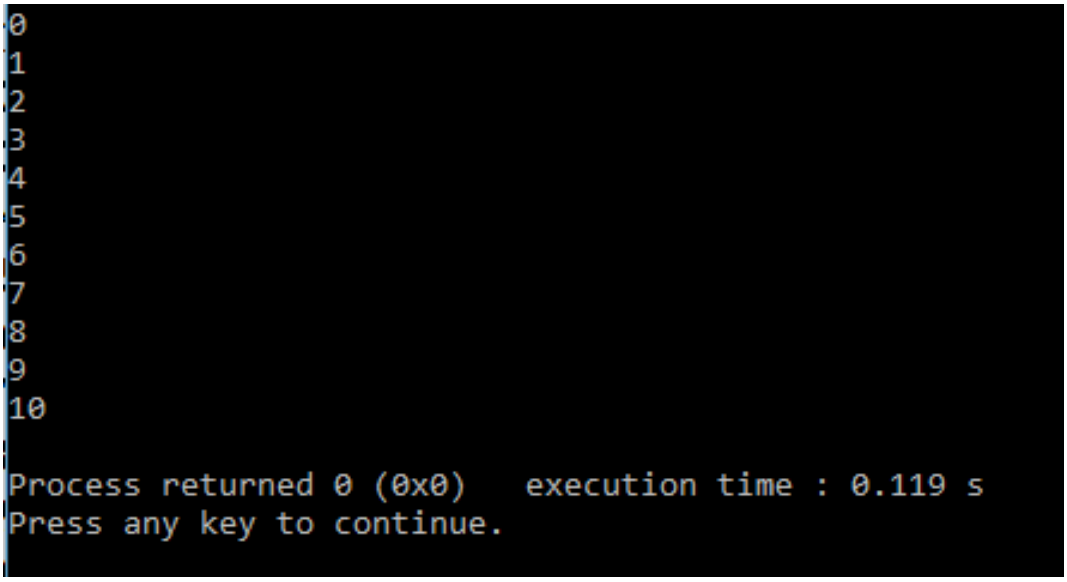
- Unless making a basic file you have to write **#include**, which is a macro that pastes the text contained in a file.
- With a name that gives the statement a meaning, its easier to understand the macro than the lengthy code behind it.

Example

```
#include<iostream>
using namespace std;
#define END 10
int main(){
int i;
for(i=0;i<=END;i++){
```

```
    cout<<i<<endl;
}
return 0;
}
```

Output



Difference between Macro and Function

Macro	Function
It is not compiled, it is pre-processed.	It is not pre-processed but it is compiled.
Macros do not check for compilation error which leads to unexpected output.	Function checks for compilation error and there is a less chance of unexpected output.
Code length is increased.	Code length remains same.
Macros are faster in execution than function.	Functions are bit slower in execution.
Before compilation process the macro name is replaced by the macro value.	In a function call, transfer of control takes place.
Macros are useful when a small piece of code used multiple times in a program.	Functions are helpful when a large piece of code repeated number of times.

Macros vs. Templates

- Macro are used in C and C++ for replacement of numbers, small inline functions etc.
- Template is only available in C++ language. It is used to write small macro like functions etc.

Difference between Macros and Templates

Macro	Template
There is no type checking	There is full type checking
They are preprocessed	They are compiled
They can cause an unexpected result	No such unexpected results are obtained.


Example

```
//Program Macro as Function
#include <iostream>
using namespace std;
#define MIN(a,b) (((a)<(b)) ? a : b)
int main () {
  int i, j;
 cout<<"Enter Two numbers";
 cin>>i>>j;
  cout <<"The minimum is " << MIN(i, j) << endl;
  return 0;
}
```

Output

```
Enter Two numbers50
12
The minimum is 12

Process returned 0 (0x0)   execution time : 3.868 s
Press any key to continue.
```

Summary

- C++ supports a mechanism known as template to implement the concept of generic programming.
- Templates classes may be defined as the layout and operations for an unbounded set of related classes.
- Templates allows us to generate a family of classes or a family of functions to handle different data types.
- Macros are a piece of code in a program which is given some name. Whenever this name is encountered by the compiler the compiler replaces the name with the actual piece of code. The '#define' directive is used to define a macro.
- Template classes and functions eliminate code duplication for different types and thus make the program development easier and more manageable.

Keywords

Template: A template function may be defined as an unbounded functions. All the possible parameters to the function are not known in advance and a copy of the function has to be created as and when necessary.

Template Classes: Template classes may be defined as the layout and operations for an unbounded set of related classes.

Function Template: We write a generic function that can be used for different data types. Function templates are special functions that can operate with generic types.

Generic Programming in C++: Generic programming is a technique where generic types are used as parameters in algorithms so that they can work for a variety of data types.

Preprocessor in C++: As the name suggests Preprocessors are programs that process our source code before compilation. There are a number of steps involved between writing a program and executing a program in C / C++.

Self Assessment

1. Template is an example of compile time polymorphism.
A. True
B. False

2. How to declare a template?
A. temp
B. template < >
C. tem
D. all of above

3. The same template is used to generate many different instances, this can be done by
A. Functions
B. Template parameters
C. Operators
D. None of them

4. Recursion is a method in which the solution of a problem depends on _____
A. Larger instances of different problems
B. Larger instances of the same problem
C. Smaller instances of the same problem
D. Smaller instances of different problems

5. A template class can have _____
A. More than one generic data type
B. Only one generic data type
C. At most two data types
D. Only generic type of integers and not characters

6. What is the syntax to use explicit class specialization?
A. template <int> class myClass<>{ }
B. template <int> class myClass<int>{ }
C. template <> class myClass<>{ }
D. template <> class myClass<int>{ }

7. Which is the most significant feature that arises by using template classes?

A. Code readability
B. Ease in coding
C. Code reusability
D. Modularity in code

8. It is possible to inherit from a template class.
A. True
B. False

9. What will the output of following code

```
#include<iostream>
using namespace std;
template <typename T>
class base {

protected:
   int x=10,y=10;
};

template <typename T>
class derived : public base<T> {

public:
   int sum() { return this->x+this->y; }
};

int main() {
   derived<int> d;
   cout<<"Sum of x and y is"<<d.sum();
   return 0;
}
```

A. 10 10
B. 110
C. 20
D. 10

10. Which keyword is used to define the macros in c++?
A. macro
B. define
C. #define
D. none of the mentioned

11. Which symbol is used to declare the preprocessor directives?

A. $

B. %

C. *

D. #

12. What is the output of this program?

```
#include < iostream >

using namespace std;

#define MIN(a,b) (((a)<(b)) ? a : b)

int main ()

{

float i, j;

i = 10.1;

j = 10.01;

cout << "The minimum is "  << MIN(i, j) << endl;

return 0;

}
```

A. 10.01

B. 10.1

C. compile time error

D. none of the mentioned

13. What is the output of this program?

```
#include <iostream>

using namespace std;

#define MIN(a,b) (((a)<(b)) ? a : b)

int main () {

  int i, j;

i=90;

j=10;

  cout <<"The minimum is " << MIN(i, j) << endl;

  return 0;

}
```

A. The minimum is 0

B. The minimum is 10

C. The minimum is 90

D. Compile time error

14. Which one is right option for function?

A. It is not pre-processed but it is compiled.

B. Code length remains same.
C. Functions are bit slower in execution.
D. All of above

15. Which one is right option for function?
A. Macros are faster in execution than function.
B. Macros are useful when a small piece of code used multiple times in a program.
C. All of above
D. None of above

Review Questions

1. What is macro? How is it implemented in C++?
2. A template can be considered as a kind of macro. Then, what is the difference between them? Distinguish between overloaded functions and function templates.
3. How to implement recursion using template in C++.
4. A macro can be used as a function. Comment.
5. What are Templates?
6. Define preprocessor in C++.
7. Differentiate between macro and template.
8. Explain recursion with template function with suitable example
9. Write a program in C++ that demonstrate working of preprocessor in C++.
10. Write a program in C++ that demonstrate working of macro.

Answers for Self Assessment

1.	A	2.	B	3.	B	4.	C	5.	A
6.	D	7.	C	8.	A	9.	C	10.	C
11.	D	12.	A	13.	B	14.	D	15.	C

Further Readings

E Balagurusamy; Object-oriented Programming with C++; Tata Mc Graw-Hill.

Herbert Schildt; The Complete Reference C++; Tata Mc Graw Hill.

Robert Lafore; Object-oriented Programming in Turbo C++; Galgotia.

Web Links

https://www.codecademy.com/learn/learn-c-plus-plus

https://isocpp.org/wiki/faq/templates

https://www.startertutorials.com/blog/generic-programming-cpp.html

Unit 13: Exception Handling

CONTENTS

Objectives

After studying this unit, you will be able to:

- Recognize errors and exception
- Understand Exception Handling
- Explain Caching of multiple exceptions
- Analyze multiple cache statements

Introduction

We know that it is very rare that a program works correctly first time. It might have bugs. The two most common types of bugs are logic errors and syntactic errors. The logic errors occur due to poor understanding of the problem and solution procedure. The syntactic errors arise due to poor understanding of the language itself. We can detect these errors by using exhaustive debugging and testing procedures.

We often come across some peculiar exceptions problems other than logic or syntax errors. They are known as exceptions. Exceptions are run time anomalies or unusual conditions that a program may encounter while executing. Anomalies might include conditions such as division by zero, access to an array outside of its bounds, or running out of memory or disk space. When a program encounters an exceptional condition, it is important that it is identified and dealt with effectively. ANSI C++ provides built-in language features to detect and handle exceptions which are basically run time errors.

Exception handling was not part of the original C++. It is a new feature added to ANSI C++. Today, almost all compilers support this feature. C++ exception handling provides a type-safe, integrated approach, for coping with the unusual predictable problems that arise while executing a program.


Did you know?

Error: Error is an illegal operation performed by the user which results in abnormal working of the program.

13.1 Basics of Exception Handling

Exceptions are of two kinds, namely, synchronous exceptions and asynchronous exceptions. Errors such as "out-of-range index" and "over-flow" belong to the synchronous type exceptions The errors that are caused by events beyond the control of the program (such as keyboard interrupts) are called asynchronous exceptions. The proposed exception handling mechanism in C++ is designed to handle only synchronous exceptions.

The purpose of the exception handling mechanism is to provide means to detect and report an "exceptional circumstances" so that appropriate action can be taken. The mechanism suggests a separate error handling code that performs the following tasks:

1. Find the problem I Hit the exception).

2. Inform that an en or has occurred (Throw the exception).

3. Receive the error information (Catch the exception).

4. Take corrective actions (Handle the exception).

The error handling code basically consists of two segments, one to detect errors and to throw exceptions, and the other to catch the exceptions and to take appropriate actions.

 Did you know?

Compile Time Error: Errors caught during compiled time is called Compile time errors. Compile time errors include library reference, syntax error or incorrect class import.

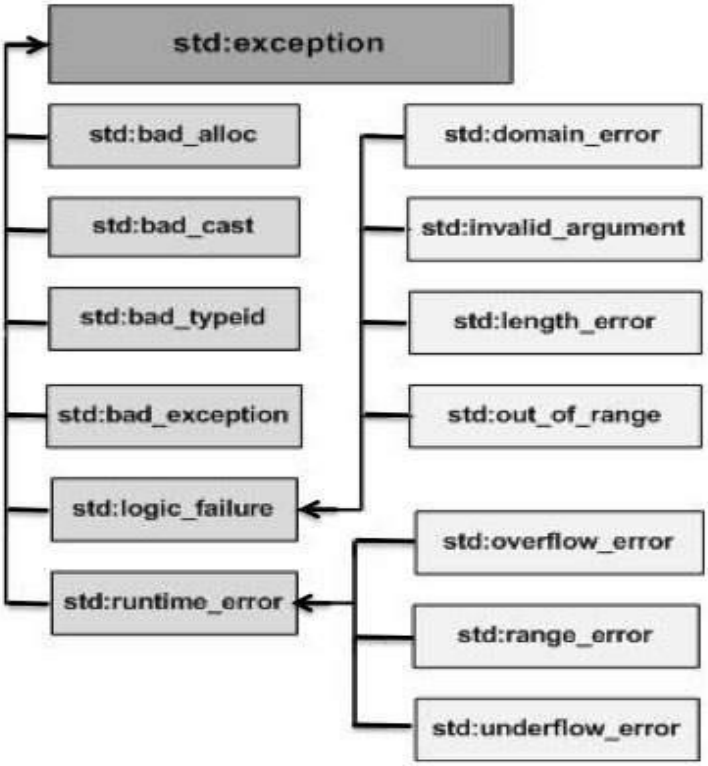
Run Time Error: They are also known as exceptions. An exception caught during run time creates serious issues.

Exceptions are preferred in modern C++ for the following reasons:

- An exception forces calling code to recognize an error condition and handle it. Unhandled exceptions stop program execution.
- An exception jumps to the point in the call stack that can handle the error. Intermediate functions can let the exception propagate. They don't have to coordinate with other layers.
- The exception stack-unwinding mechanism destroys all objects in scope after an exception is thrown, according to well-defined rules.
- An exception enables a clean separation between the code that detects the error and the code that handles the error.

C++ Standard Exceptions

C++ provides a list of standard exceptions defined in <exception> which we can use in our programs. These are arranged in a parent-child class hierarchy shown below –



Description is here

Sr.No	Exception & Description
1	**std::exception** An exception and parent class of all the standard C++ exceptions.
2	**std::bad_alloc** This can be thrown by new.
3	**std::bad_cast** This can be thrown by dynamic_cast.
4	**std::bad_exception** This is useful device to handle unexpected exceptions in a C++ program.
5	std::bad_typeid This can be thrown by typeid.
6	**std::logic_error** An exception that theoretically can be detected by reading the code.
7	**std::domain_error** This is an exception thrown when a mathematically invalid domain is used.
8	**std::invalid_argument** This is thrown due to invalid arguments.

9	**std::length_error** This is thrown when a too big std::string is created.
10	**std::out_of_range** This can be thrown by the 'at' method, for example a std::vector and std::bitset<>::operator[]().
11	**std::runtime_error** An exception that theoretically cannot be detected by reading the code.
12	**std::overflow_error** This is thrown if a mathematical overflow occurs.
13	**std::range_error** This is occurred when you try to store a value which is out of range.
14	**std::underflow_error** This is thrown if a mathematical underflow occurs.

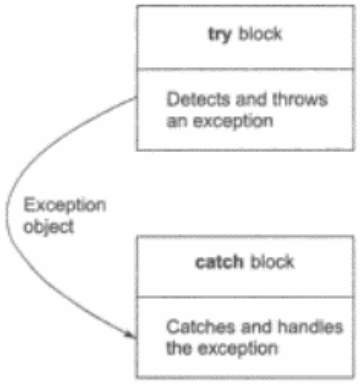
13.2 Exception Handling Mechanism

Exceptions are run time anomalies or unusual conditions that a program may encounter while executing. Anomalies might include conditions such as division by zero, access to an array outside of its bounds, or running out of memory or disk space. When a program encounters an exceptional condition, it is important that it is identified and dealt with effectively.

The mechanism performs following tasks:

- Find the problem (Hit the exception).
- Inform that an error has occurred (Throw the exception).
- Receive the error information (Catch the expression).
- Take corrective actions (Handle the exceptions).
- The error handling code basically consists of two segments one to detect errors and to throw exceptions, and other to catch the exceptions and to take appropriate actions.

C++ exception handling mechanism is basically built upon three keywords, namely, try, throw, and catch. The keyword try is used to preface a block of statements (surrounded by braces) which may generate exceptions. This block of statements is known as try block. When an exception is detected, it is thrown using a throw statement in the try block. A catch block defined by the keyword catch 'catches' the exception 'thrown' by the throw statement in the try block, and handles it appropriately. The relationship is shown in Figure.



The catch block that catches an exception must immediately follow the try block that throws the exception. The general form of these two blocks are as follows:

```
…….
……..
try
{
…….
throw exception;            // Block of statements which
…….                         // detects and throws on exception
……..
}
catch(type org)  {                  // Catches exception
…….                                 // Block of statements that
……..                                       // handles the exception
}
……..
……..
```

When the try block throws an exception, the program control leaves the try block and enters the catch statement of the catch block. Note that exceptions are objects used to transmit information about a problem. If the type of object thrown matches the arg type in the catch statement, then catch block is executed for handling the exception. If they do not 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. That is, the catch block is skipped. This simple try-catch mechanism is illustrated in Program.

```
#include <iostream>
using namespace std;
int main () {
int x;
cout<<"Enter number";
cin>>x;
try {
  if (x >=1) {
    cout << "Input is valid";
  } else {
    throw (x);
  }
}
catch (int num) {
  cout << "Input Invalid\n";
  cout << "Number is: " << num;
}
}
```

Output

```
Enter number12
Input is valid
Process returned 0 (0x0)   execution time : 1.927 s
Press any key to continue.
```

Note

- Exception Handling in C++ is a process to handle runtime errors.
- A C++ exception is a response to an exceptional circumstance that arises while a program is running, such as an attempt to divide by zero.

Did you know?

What is an exception?

Exceptions are run time anomalies or unusual conditions that a program may encounter while executing.

Exception Handling Keywords

- **throw-** when a program encounters a problem, it throws an exception. The throw keyword helps the program perform the throw.
- **catch-** a program uses an exception handler to catch an exception. It is added to the section of a program where you need to handle the problem. It's done using the catch keyword.
- **try-** the try block identifies the code block for which certain exceptions will be activated. It should be followed by one/more catch blocks.

Advantages of Exception Handling

1. **Separating Error-Handling Code from "Regular" Code: -** Exceptions provide the means to separate the details of what to do when something out of the ordinary happens from the main logic of a program. In traditional programming, error detection, reporting, and handling often lead to confusing spaghetti code. Exceptions enable you to write the main flow of your code and to deal with the exceptional cases elsewhere.

2. **Propagating Errors Up the Call Stack: -** A second advantage of exceptions is the ability to propagate error reporting up the call stack of methods.

3. **Grouping and Differentiating Error Types: -** Because all exceptions thrown within a program are objects, the grouping or categorizing of exceptions is a natural outcome of the class hierarchy.

Task: Write a program to demonstrate working of try, throw and catch blocks in C++.

Multiple Cache Statements

It is possible that a program segment has more than one condition to throw an exception. In such cases, we can associate more than one catch statement with a try as shown below:

```
try

{           // try block
……..
}

catch(type1 arg)

{

// catch block1

} catch(type2 arg) {

// catch block2

}
 ……..

……

catch(typeN arg) {

// catch blockN
}
```


Note

- Each catch block catches one type of exceptions
- Need multiple catch blocks
- When the value is not important, the parameter can be omitted.
    - E.g. catch(int) {…}
- catch (…) catches any exception, can serve as the default exception handler
- Although a try-block followed by its catch-block can be nested inside another try-block
- Although a try-block followed by its catch-block can be nested inside another try-block

 Did you know?

Why multiply blocks are useful?

Multiple catch blocks are used when we have to catch a specific type of exception out of many possible type of exceptions i.e. an exception of type char or int or short or long etc.

 Lab Exercise

```
//Program
#include<iostream>
using namespace std;
class compareAges{
public:
   int fathersAge;
   int sonsAge;
   void inputAge(){
      cout<<"Enter Fathers Age";
cin>>fathersAge;
cout<<"Enter Sons Age";
cin>>sonsAge;
   }
   void compare(){
   try{
   if(fathersAge==sonsAge) throw 55;
   else if(sonsAge>fathersAge) throw ('x');
   else throw (2.1f);
   }
   catch(int a){
   cout<<"Invalid..";
   }
   catch(char b){
   cout<<"Age of son is never greater than the age of father";
   }
   catch (float c){
   cout<<"Valid input";
   }
   }
};
main(){
compareAges obj;
obj.inputAge();
obj.compare();
}
```

Output

```
Enter Fathers Age95
Enter Sons Age100
Age of son is never greater than the age of father
Process returned 0 (0x0)   execution time : 16.626 s
Press any key to continue.
```

13.3 Caching Multiple Exceptions

Multiple catch blocks are used when we have to catch a specific type of exception out of many possible type of exceptions i.e. an exception of type char or int or short or long etc. Let's see the use of multiple catch blocks with an example.

Lab Exercise

```
#include<iostream>


using namespace std;


int main()
{
int a=10, b=0, c;
try
{
        //if a is divided by b(which has a value 0);
        if(b==0)
                throw(c);
        else
        c=a/b;


}
catch(char c)                                   //catch block to handle/catch exception
{
        cout<<"Caught exception : char type ";
}
catch(int i)                                     //catch block to handle/catch exception
{
        cout<<"Caught exception : int type ";
}
catch(short s)                          //catch block to handle/catch exception
{
        cout<<"Caught exception : short type ";
}
cout<<"\n Hello World";
```

}

Output

```
Caught exception : int type
 Hello World
Process returned 0 (0x0)   execution time : 0.845 s
Press any key to continue.
```

Summary

- A try block may throw an exception directly or invoke a function that throws an exception. Irrespective of location of the throw point, the catch block is placed immediately after the try block.
- We can place two or more catch blocks together to catch and handle multiple types of exceptions thrown by a try block.
- It is also possible to make a catch statement to catch all types of exceptions using ellipses as its argument.
- We may also restrict a function to throw only a set of specified exceptions by adding a throw specification clause to the function definition.

Keywords

Error - Error is an illegal operation performed by the user which results in abnormal working of the program.

Exception – An exception is a problem that arises during the execution of a program. A C++ exception is a response to an exceptional circumstance that arises while a program is running, such as an attempt to divide by zero.

Run time error - They are also known as exceptions. An exception caught during run time creates serious issues.

Throw- when a program encounters a problem, it throws an exception. The throw keyword helps the program perform the throw.

Catch- a program uses an exception handler to catch an exception. It is added to the section of a program where you need to handle the problem. It's done using the catch keyword.

Try- the try block identifies the code block for which certain exceptions will be activated. It should be followed by one/more catch blocks.

Self Assessment

1. Which is used to handle the exceptions in c++?

A. catch handler
B. handler
C. exception handler
D. throw

2. Which type of program is recommended to include in try block?

A. static memory allocation
B. dynamic memory allocation
C. const reference
D. pointer

3. Which statement is used to catch all types of exceptions?
A. catch()
B. catch(Test t)
C. catch(…)
D. catch(Test)

4. What is an exception in C++ program?
A. A problem that arises during the execution of a program
B. A problem that arises during compilation
C. Also known as the syntax error
D. Also known as semantic error

5. By default, what a program does when it detects an exception?
A. Continue running
B. Results in the termination of the program
C. Calls other functions of the program
D. Removes the exception and tells the programmer about an exception

6. Why do we need to handle exceptions?
A. To avoid unexpected behaviour of a program during run-time
B. To let compiler remove all exceptions by itself
C. To successfully compile the program
D. To get correct output

7. How Exception handling is implemented in the C++ program?
A. Using Exception keyword
B. Using try-catch block
C. Using Exception block
D. Using Error handling schedules

8. Which is used to throw a exception?
A. Try
B. Throw
C. Catch
D. None of the above

9. How do define the user-defined exceptions?
A. Inherting & overriding exception class functionlity
B. Overriding class functionlity
C. Inherting class functionlity
D. None of the above

10. What is an error in C++?
A. Violation of syntactic and semantic rules of a languages
B. Missing of Semicolon
C. Missing of double quotes
D. Violation of program interface

11. We can prevent a function from throwing any exceptions.
A. TRUE
B. FALSE
C. May Be
D. Can't Say

12. An exception can be of only built-In type.
A. True
B. False

13. Generic catch handler must be placed at the end of all the catch handlers.
A. True
B. False

14. A try block can be nested under another try block.
A. True
B. False

15. What is the difference between error and exception?
A. Both are the same
B. Errors can be handled at the run-time but the exceptions cannot
C. Exceptions can be handled at the run-time but the errors cannot
D. Both can be handled during run-time

Answers for Self Assessment

1.	C	2.	B	3.	C	4.	A	5.	B
6.	A	7.	B	8.	B	9.	A	10.	A
11.	A	12.	B	13.	A	14.	A	15.	C

Review Questions

1. What do you mean by an error?
2. What is an exception? How it is different from error.
3. What is an exception?
4. How is an exception handle in C++?
5. List the advantages of exception handling.
6. What are the basic implementations of exception handling?
7. Explain in detail "exception handling mechanism".
8. When do we use multiple cache statements?
9. What are main keywords that responsible for exception handling in C++.
10. What should we place inside the catch block?

Further Readings

E Balagurusamy; Object-oriented Programming with C++; Tata Mc Graw-Hill.

Herbert Schildt; The Complete Reference C++; Tata Mc Graw Hill.

Robert Lafore; Object-oriented Programming in Turbo C++; Galgotia.

Web Links

https://study.com/academy/lesson/practical-application-for-c-plus-plus-programming-

https://www.codecademy.com/learn/learn-c-plus-plus

https://www.guru99.com/cpp-file-read-write-open.html

Unit 14: More on Exception Handling

CONTENTS

Objectives

After studying this unit, you will be able to:

- Recognize errors and exception
- Understand Exception Handling
- Analyze C++ standard exceptions
- Describe re-throwing exception

Introduction

We know that it is very rare that a program works correctly first time. It might have bugs. The two most common types of bugs are logic errors and syntactic errors. The logic errors occur due to poor understanding of the problem and solution procedure. The syntactic errors arise due to poor understanding of the language itself. We can detect these errors by using exhaustive debugging and testing procedures.

We often come across some peculiar exceptions problems other than logic or syntax errors. They are known as exceptions. Exceptions are run time anomalies or unusual conditions that a program may encounter while executing. Anomalies might include conditions such as division by zero, access to an array outside of its bounds, or running out of memory or disk space. When a program encounters an exceptional condition, it is important that it is identified and dealt with effectively. ANSI C++ provides built-in language features to detect and handle exceptions which are basically run time errors.

Exception handling was not part of the original C++. It is a new feature added to ANSI C++. Today, almost all compilers support this feature. C++ exception handling provides a type-safe, integrated approach, for coping with the unusual predictable problems that arise while executing a program.


Did you know?

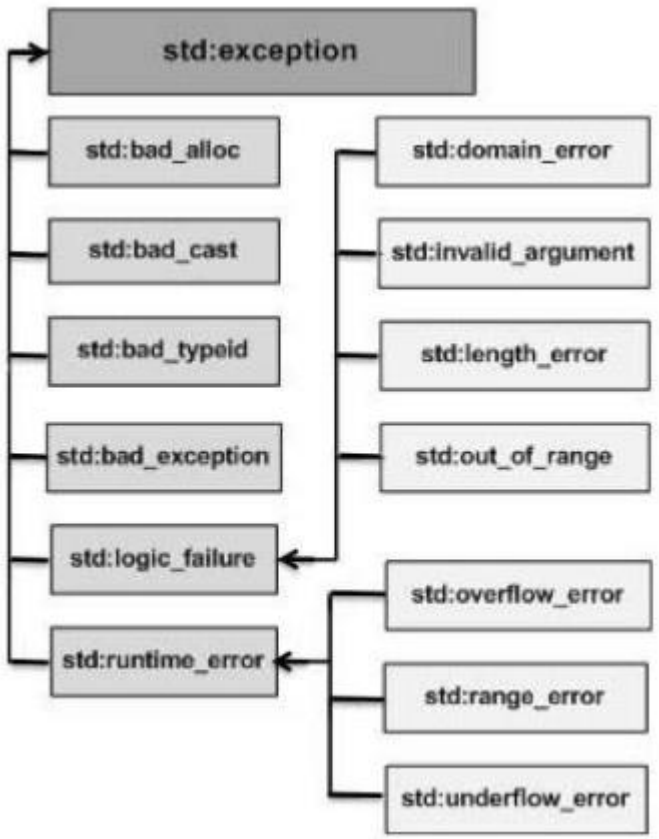
Error: Error is an illegal operation performed by the user which results in abnormal working of the program.

Advantages of Exception Handling

- Exception handling helps programmers to create reliable systems.
- Exception handling separates the exception handling code from the main logic of program.
- Exceptions can be handled outside of the regular code by throwing the exceptions from a function definition or by re-throwing an exception.
- Functions can handle only the exceptions they choose i.e., a function can throw many exceptions, but may choose handle only some of them.

14.1 C++ standard exceptions

C++ provides a list of standard exceptions defined in <exception> which we can use in our programs. These are arranged in a parent-child class hierarchy shown below:



Exception	Description
std::exception	An exception and parent class of all the standard C++ exceptions.
std::bad_alloc	This can be thrown by **new**.
std::bad_cast	This can be thrown by **dynamic_cast**.
std::bad_exception	This is useful device to handle unexpected exceptions in a C++ program
std::bad_typeid	This can be thrown by **typeid**.

std::logic_error	An exception that theoretically can be detected by reading the code.
std::domain_error	This is an exception thrown when a mathematically invalid domain is used
std::invalid_argument	This is thrown due to invalid arguments.
std::length_error	This is thrown when a too big std::string is created
std::out_of_range	This can be thrown by the at method from for example a std::vector and std::bitset<>::operator[]().
std::runtime_error	An exception that theoretically can not be detected by reading the code.
std::overflow_error	This is thrown if a mathematical overflow occurs.
std::range_error	This is occured when you try to store a value which is out of range.
std::underflow_error	This is thrown if a mathematical underflow occurs.

Define New Exceptions:

You can define your own exceptions by inheriting and overriding exception class functionality. Following is the example, which shows how you can use std::exception class to implement your own exception in standard way: -

 Lab exercise

```
#include <iostream>
#include <exception>
using namespace std;
struct  MyException  : public exception
{
const char * what () const throw ()
{
return "C++ Exception";
}
};

int main()
```

```
{
try
{
throw MyException();
}
catch(MyException& e)
{
std::cout << "MyException caught" << std::endl; std::cout << e.what() << std::endl;
}
catch(std::exception& e)
{
//Other errors
}
}
```

This would produce the following result:

```
MyException caught
C++ Exception
```

Here, what() is a public method provided by exception class and it has been overridden by all the child exception classes. This returns the cause of an exception.

 Lab exercise

```
//Program - exception of type bad_typeid
#include <iostream>
#include <exception>
#include <typeinfo>
using namespace std;
class A {virtual fun() {}; };
int main () {
 try {
   A * a = NULL;
   typeid (*a);
 }
 catch (exception& e)
 {
   cout << "Exception: " << e.what();
 }
 return 0;
}
```

Output

```
Exception: std::bad_typeid
Process returned 0 (0x0)   execution time : 0.037 s
Press any key to continue.
```

14.2 Re-throwing Exceptions

If you want to rethrow an exception from within an exception handler, you can do so by calling throw by itself, with no exception. This causes the current exception to be passed on to an outer try/catch sequence. An exception can be rethrown only from within a catch block or from any function called from within that block. When you rethrow an exception, it will not be recaught by the same catch statement. It will propagate to the immediately enclosing try/catch sequence.

- If a catch block cannot handle the particular exception it has caught, we can rethrow the exception.
- The rethrow expression causes the originally thrown object to be rethrown.

 Lab exercise

```
//Program – rethrowing exception


#include <iostream>
using namespace std;
int main()
{
try
{
int a, b;
cout<<"Enter two integer values: ";
cin>>a>>b;
try
{
if(b == 0)
{
throw b;
}
else
{
cout<<"Division of a and b is "<<(a/b);
}
}
catch(...)
{
throw;
}
```

```
}
catch(int)
{
cout<<"Second value cannot be zero";
}
return 0;
}
```

Output

Run 1

```
Enter two integer values: 12 0
Second value cannot be zero
Process returned 0 (0x0)   execution time : 4.307 s
Press any key to continue.
```

Run 2

```
Enter two integer values: 12 2
Division of a and b is 6
Process returned 0 (0x0)   execution time : 2.522 s
Press any key to continue.
```

14.3 Exceptions in Constructors and Destructors

It is possible that exceptions might raise in a constructor or destructors. If an exception is raised in a constructor, memory might be allocated to some data members and might not be allocated for others. This might lead to memory leakage problem as the program stops and the memory for data members stays alive in the RAM.

Similarly, when an exception is raised in a destructor, memory might not be deallocated which may again lead to memory leakage problem. So, it is better to provide exception handling within the constructor and destructor to avoid such problems. Following program demonstrates handling exceptions in a constructor and destructor:

 Lab exercise

```
//Program
#include<iostream>
using namespace std;
class Divide
{
        private:
                int *x;
                int *y;
```

```
        public:
                Divide()
                {
                        x = new int();
                        y = new int();
                        cout<<"Enter two numbers: ";
                        cin>>*x>>*y;
                        try
                        {
                                if(*y == 0)
                                {
                                        throw *x;
                                }
                        }
                        catch(int)
                        {
                                delete x;
                                delete y;
                                cout<<"Second number cannot be zero!"<<endl;
                                throw;
                        }
                }
                ~Divide()
                {
                        try
                        {
                                delete x;
                                delete y;
                        }
                        catch(...)
                        {
                                cout<<"Error while deallocating memory"<<endl;
                        }
                }
                float division()
                {
                        return (float)*x / *y;
                }
};
int main()
{
```

```
        try
        {
                Divide d;
                float res = d.division();
                cout<<"Result of division is: "<<res;


        }
        catch(...)
        {
                cout<<"Unkown exception!"<<endl;
        }
        return 0;
}
```

Output

Run 1

```
Enter two numbers: 25
4
Result of division is: 6.25
Process returned 0 (0x0)   execution time : 5.604 s
Press any key to continue.
```

Run 2

```
Enter two numbers: 121
0
Second number cannot be zero!
Unkown exception!

Process returned 0 (0x0)   execution time : 5.213 s
Press any key to continue.
```

14.4 Controlling Uncaught Exceptions

- Whenever an exception arises in C++, it is handled as per the behaviour defined using the try-catch block.
- However, there is often the case when an exception is thrown but isn't caught because the exception handling subsystem fails to find a matching catch block for that particular exception.

In Such cases, following actions take place

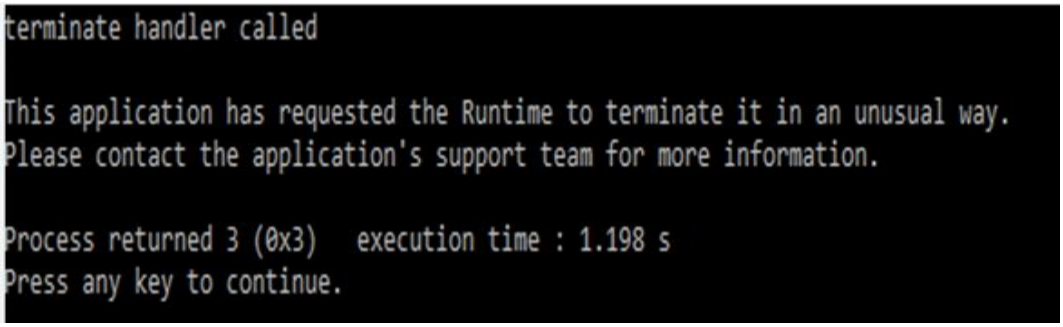
- The exception handling subsystem calls the function: unexpected°. This function, provided by the default C++ library, defines the behavior when an uncaught exception arises. By default, unexpected calls terminate°.
- The terminate function defines the actions that are to be performed during process termination. This, by default, calls abort0.

- The process is aborted.

Lab exercise

```
//          Program
#include <iostream>
#include <exception>
#include <cstdlib>
 using namespace std;
 void myterminate O {
cerr << "terminate handler called\n";
abort();
}
int main (void) {
 set_terminate(myterminate);
throw 0;
 return 0;
}
```

Output



Summary

- A try block may throw an exception directly or invoke a function that throws an exception. Irrespective of location of the throw point, the catch block is placed immediately after the try block.
- We can place two or more catch blocks together to catch and handle multiple types of exceptions thrown by a try block.
- It is also possible to make a catch statement to catch all types of exceptions using ellipses as its argument.
- We may also restrict a function to throw only a set of specified exceptions by adding a throw specification clause to the function definition.

Keywords

Error - Error is an illegal operation performed by the user which results in abnormal working of the program.

Exception – An exception is a problem that arises during the execution of a program. A C++ exception is a response to an exceptional circumstance that arises while a program is running, such as an attempt to divide by zero.

Run time error - They are also known as exceptions. An exception caught during run time creates serious issues.

Throw- when a program encounters a problem, it throws an exception. The throw keyword helps the program perform the throw.

Catch- a program uses an exception handler to catch an exception. It is added to the section of a program where you need to handle the problem. It's done using the catch keyword.

Try- the try block identifies the code block for which certain exceptions will be activated. It should be followed by one/more catch blocks.

Self Assessment

1. If inner catch handler is not able to handle the exception then_____ .

A. Compiler will look for outer try handler

B. Program terminates abnormally

C. Compiler will check for appropriate catch handler of outer try block

D. None of the above

2. What is Re-throwing an exception means in C++?

A. An exception that is thrown again as it is not handled by that catching block

B. An exception that is caught twice

C. An exception that is not handled in one caught hence thrown again

D. All of the mentioned

3. Which header file is used to declare the standard exception?

A. #include<exception>

B. #include<except>

C. #include<error>

D. #include<exce>

4. Where are standard exception classes grouped?

A. namespace std

B. error

C. catch

D. final

5. Which of the following is not a standard exception?

A. bad_cast

B. bad_exception

C. #define

D. bad_function_call

6. Which of the following is a standard exception?

A. #include<iostream>

B. bad_cast

C. All of above

D. None of above

7. If a catch block cannot handle the particular exception it has caught, we can rethrow the exception.

A. True

B. False

8. How to handle error in the destructor?
A. Throwing
B. Terminate
C. Both throwing & terminate
D. None of the above

9. Terminate function will called when we have a uncaught exception?
A. True
B. False

10. How to handle exception in constructor, in c++?
A. We have to return an exception
B. We have to throw an exception
C. All of above
D. None of the above

11. What is the role of a constructor in classes?
A. To modify the data whenever required
B. To destroy an object
C. To initialize the data members of an object when it is created
D. To call private functions from the outer world

12. Destructor has the same name as the constructor and it is preceded by _____ .
A. !
B. ?
C. ~
D. $

13. What will not be called when the terminate() is arised in constructor?
A. main()
B. class
C. destructor
D. none of the mentioned

14. What function will be called when we have a uncaught exception?
A. Catch
B. throw
C. terminate
D. none of the mentioned

15. Catch function will called when we have a uncaught exception?
A. True
B. False

Answers for Self Assessment

1.	C	2.	D	3.	A	4.	A	5.	C
6.	B	7.	A	8.	B	9.	A	10.	B
11.	C	12.	D	13.	C	14.	C	15.	B

Review Questions

1. What do you mean by an uncaught exception?
2. What is an exception? How it is different from error.
3. What is an exception?
4. Explain standard exceptions in C++ with suitable example.
5. List the advantages of exception handling.
6. What are the basic implementations handling exceptions using constructor and destructor?
7. Explain in detail "controlling uncaught exception".
8. Write a program using C++ that demonstrate working of exception handling.

Further Readings

E Balagurusamy; Object-oriented Programming with C++; Tata Mc Graw-Hill.

Herbert Schildt; The Complete Reference C++; Tata Mc Graw Hill.

Robert Lafore; Object-oriented Programming in Turbo C++; Galgotia.

Web Links

https://study.com/academy/lesson/practical-application-for-c-plus-plus-programming-

https://www.codecademy.com/learn/learn-c-plus-plus

https://www.geeksforgeeks.org/exception-handling-c/

LOVELY PROFESSIONAL UNIVERSITY
Jalandhar-Delhi G.T. Road (NH-1)
Phagwara, Punjab (India)-144411
For Enquiry: +91-1824-521360
Fax.: +91-1824-506111
Email: odl@lpu.co.in