

Object Oriented Programming

ECAP202

**Edited by
Balraj Kumar**



L OVELY
P ROFESSIONAL
U NIVERSITY



Object Oriented Programming

Edited By:
Balraj Kumar

CONTENT

Unit 1: Principles of OOP	1
<i>Prikshit Kumar Angra, Lovely Professional University</i>	
Unit 2: Basics of C++	17
<i>Prikshit Kumar Angra, Lovely Professional University</i>	
Unit 3: Operators and Type Casting	30
<i>Prikshit Kumar Angra, Lovely Professional University</i>	
Unit 4: Control Structures	50
<i>Prikshit Kumar Angra, Lovely Professional University</i>	
Unit 5: Pointers and Structures	63
<i>Prikshit Kumar Angra, Lovely Professional University</i>	
Unit 6: Classes and Objects	74
<i>Prikshit Kumar Angra, Lovely Professional University</i>	
Unit 7: More on Classes and Objects	90
<i>Prikshit Kumar Angra, Lovely Professional University</i>	
Unit 8: Handling Functions	104
<i>Prikshit Kumar Angra, Lovely Professional University</i>	
Unit 9: More on Functions	113
<i>Prikshit Kumar Angra, Lovely Professional University</i>	
Unit 10: Static Members and Polymorphism	128
<i>Prikshit Kumar Angra, Lovely Professional University</i>	
Unit 11: Constructors and Destructors	141
<i>Prikshit Kumar Angra, Lovely Professional University</i>	
Unit 12: More on Constructors and Destructors	160
<i>Prikshit Kumar Angra, Lovely Professional University</i>	
Unit 13: Inheritance	169
<i>Prikshit Kumar Angra, Lovely Professional University</i>	
Unit 14: File Handling	184
<i>Prikshit Kumar Angra, Lovely Professional University</i>	

Unit 01: Principles of OOP

CONTENTS

Objectives

Introduction

- 1.1 Basic Concept of Object-oriented Programming
- 1.2 Introduction to OOP languages
- 1.3 Procedural programming v/s Object-oriented programming
- 1.4 Procedure oriented Programming Paradigm
- 1.5 Object-oriented Programming Paradigm
- 1.6 Benefits of OOP
- 1.7 Applications of Object Oriented Programming
- 1.8 C++ Class Member Function
- 1.9 Private Member Function

Summary

Keywords

Self Assessment

Answers for Self Assessment

Review Questions

Further Readings

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.

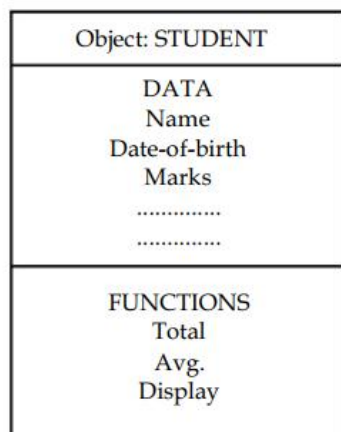
In object-oriented parlance, a problem is viewed in terms of the following concepts:

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

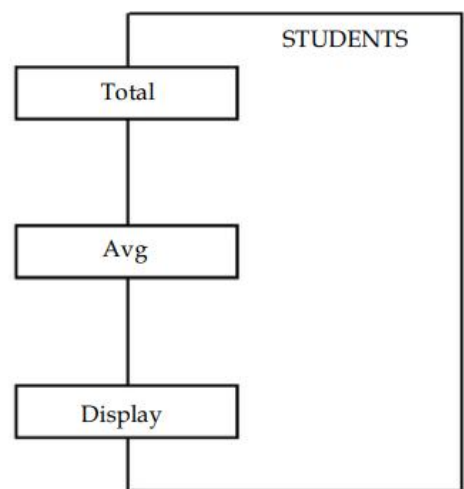
Let us now study the entire concept in detail.

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.



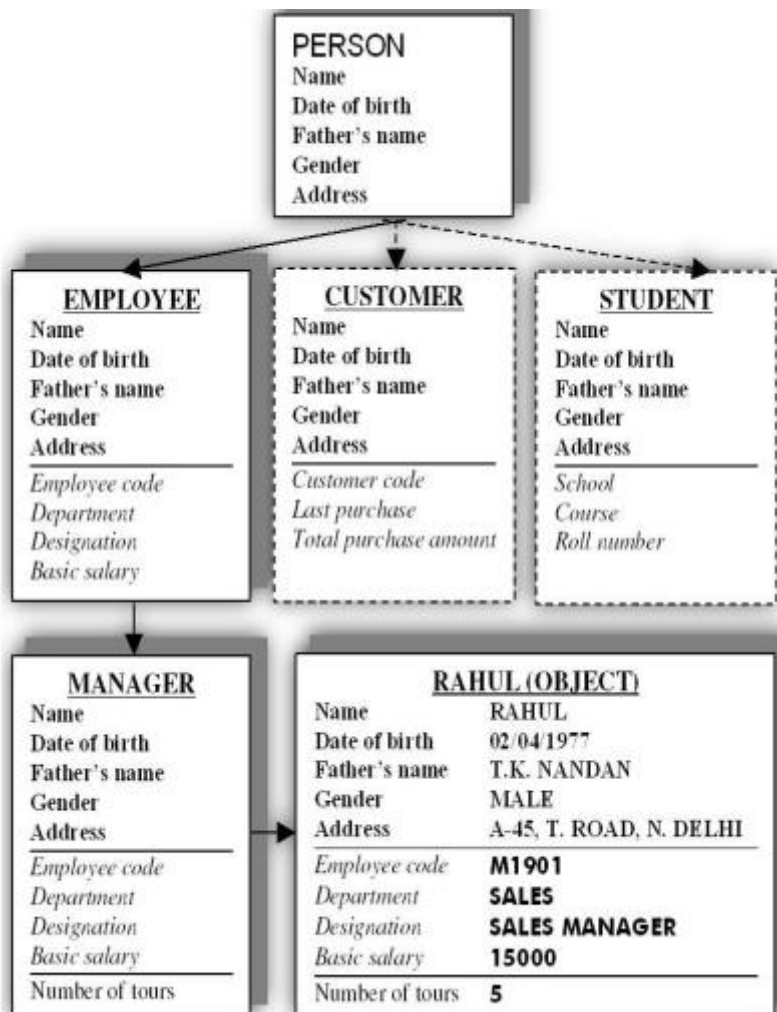
Notes:- A class is a data-type that has its own members i.e. data members and member functions. It is the blueprint for an object in objects oriented programming language. It is the basic building block of object oriented programming in c++.

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 of object oriented programming.

Let's take a real life example to understand the 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 that we should know that's why this is done from the same as a concept of data abstraction.

In C++ programming language, there are two ways using which we can accomplish data abstraction –

1. using class
2. using header file



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

index of text book.

```
class School
{
void sixthclass();
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++ programming language supports the following types of inheritance

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

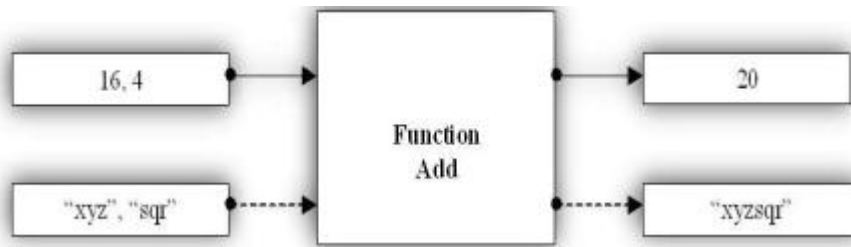


Caution: - Keep in mind that each subclass defines only those features that are unique to it.

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 (+).

$16 + 4 = 20$

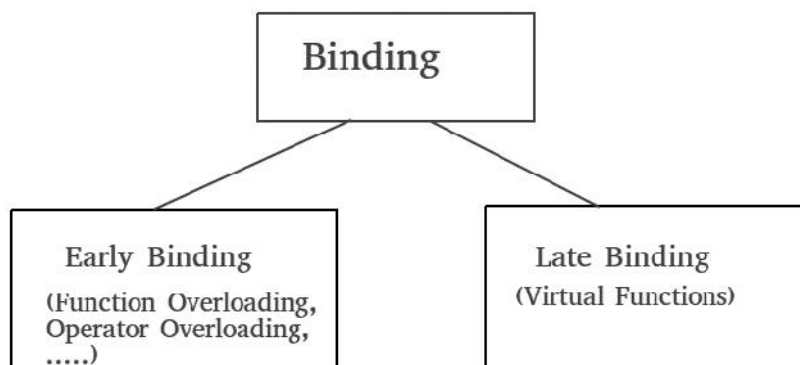
"xyz" + "sqr" = "xyzsqr"



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.



Did you know?

Difference between static and dynamic binding in C++

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.

Object Oriented Programming

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 object-based programming features along with two additional features, 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
-----------------	------------------------	-----------------------------

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

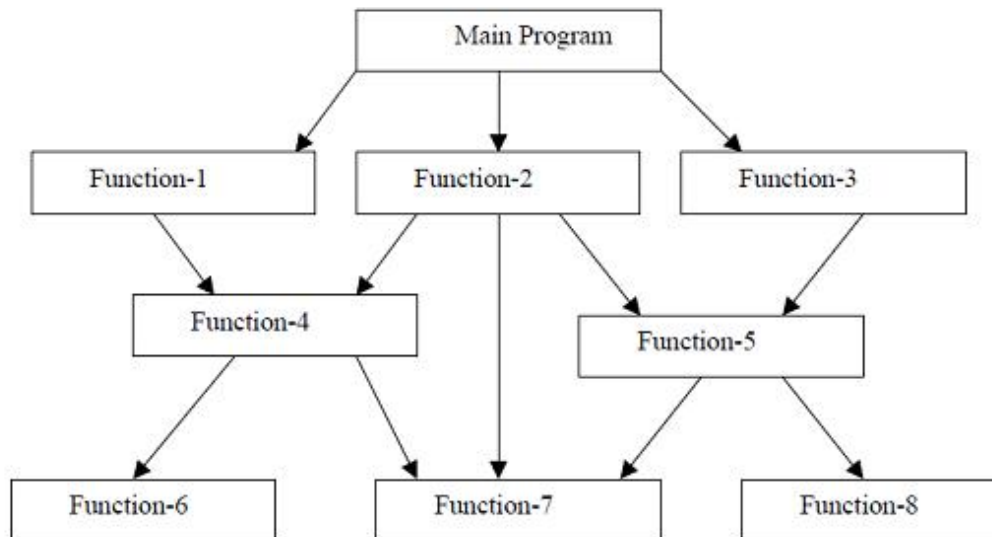
1.4 Procedure oriented Programming Paradigm

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 world.

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

Unit 02: Basics of C++

CONTENTS

Objectives

Introduction

- 2.1 What is C?
- 2.2 What is C++?
- 2.3 Similarities Between C and C++
- 2.4 C Vs. C++
- 2.5 Simple C++ Program
- 2.6 Compiling and linking
- 2.7 Review of Tokens
- 2.8 Keywords
- 2.9 Identifiers
- 2.10 Constants
- 2.11 Strings
- 2.12 Operators
- 2.13 Data Types
- 2.14 Reference Variables

Summary

Keywords

Self Assessment

Answer for Self Assessment

Review Questions

Further Readings

Objectives

After studying this unit, you will be able to:

- Recognize the tokens
- Understand keywords
- Describe the expressions
- Recognize the reference variable
-

Introduction

C++ is a language in essence. It is made up of letters, words, sentences, and constructs just like English language. This unit discusses these elements of the C++ language along with the operators applicable over them.

2.1 What is C?

It is a very powerful and general-purpose language used in programming. We can use C to develop software such as databases, operating systems, compilers, and many more. This programming language is excellent to learn for beginners in programming.

Object Oriented Programming

C is the basic programming language that can be used to develop from the operating systems (like Windows) to complex programs like Oracle database, Git, Python interpreter, and many more. C programming language can be called a god's programming language as it forms the base for other programming languages. If we know the C language, then we can easily learn other programming languages. C language was developed by the great computer scientist Dennis Ritchie at the Bell Laboratories. It contains some additional features that make it unique from other programming languages.

2.2 What is C++?

It is a subset of the C language. C++ is object-oriented, designed as an extension to the C language. Thus, apart from the features of procedural language from C, C++ provides support to object-oriented features as well. For instance, polymorphism, inheritance, encapsulation, abstraction, and more.

C++ is a special-purpose programming language developed by Bjarne Stroustrup at Bell Labs circa 1980. C++ language is very similar to C language, and it is so compatible with C that it can run 99% of C programs without changing any source of code though C++ is an object-oriented programming language, so it is a safer and well-structured programming language than C.

2.3 Similarities Between C and C++

- Both languages have the same code structure.
- They both have a similar syntax.
- They both have a similar compilation.
- Their basic memory model is very close to the hardware.
- Both the languages share the same basic syntax. Also, almost all the operators and keywords of C are present in C++ as well, and they do the same thing.
- Similar notions of heap, stack, static, and file-scope variables are present in both of these languages.
- As compared to C, C++ has more extended grammar. But the basic grammar here is the same.

2.4 C Vs. C++

Parameter	C	C++
Definition	It is a structural programming language that doesn't provide any support for classes and objects.	It is an object-oriented programming language, and it provides support for the concept of classes and objects.
History	Dennis Ritchie developed the C language at the AT&T Bell Laboratories in around 1969.	Bjarne Stroustrup developed the C++ language in 1979-1980 at Bell Labs.
Type of Programming Language	C primarily supports procedural programming for developing codes. Here, it checks the code line by line.	C++ supports both programming paradigms-procedural as well as object-oriented. It is, thus, known as a hybrid language.
Support for OOPs Feature	C has no support for the OOPs concept. Thus, it does not support encapsulation, polymorphism, and inheritance.	The C++ language supports encapsulation, inheritance, and polymorphism because it is an object-oriented programming language.

Unit 02: Basics of C++

Supported Features	C has no support for functions and operator overloading. It also does not have any namespace feature and functionality of reference variables.	C++, on the other hand, supports both of the functions and operator overloading. It also has the namespace feature and the functionality of reference variables.
Driven Type	The C is a function-driven language because it is procedural programming.	The C++ language, on the other hand, is object-driven because it is OOP (object-oriented programming).
Data Security	C is vulnerable to manipulation via outside code. It is because it does not support encapsulation-leading to its data behaving as a free entity.	C++, on the other hand, is a very secure language. It supports encapsulation of data in the form of objects- thus hiding the information and ensuring that one uses the structures and operators as intended.
Type of Subset	It is a subset of the C++ language. It cannot run the codes used in C++.	It is a superset of the C language. It is capable of running 99% of the C language codes.
Segregation of Data and Functions	Since C is a procedural programming language, the data and functions stay separate in it.	In the case of C++, the data and functions stay encapsulated in an object's form.
Hiding Data and Information	C does not support the hiding of data and information.	C++ language hides the data through encapsulation. This process ensures that a user utilizes the structures as operators as intended.
Built-in Data Types	The C language does not support built-in data types.	The C++ language supports built-in data types.
Function Inside Structures	In the case of C, it does not define the functions inside structures.	In the case of C++, it uses functions inside a structure.
Reference Variables	It does not support any reference variables.	It supports reference variables.
Overloading of Functions	Function overloading allows a user to have more than one function with different parameters but the same name. The C language does not support it.	The C++ language supports function overloading.
Overriding of Functions	Function overriding provides the specific implementation to any function that is defined already in the base class. The C language does not support it.	The C++ language supports function overriding.

Object Oriented Programming

Header File	C language uses the <code><stdio.h></code> header file.	C++ language uses the <code><iostream.h></code> header file.
Namespace Features	The namespace feature groups various entities like objects, classes, and functions under a specific name. No namespace features are present in the C language.	The C++ language uses the namespace features to help avoid name collisions.
Virtual and Friend Functions	The C language does not support virtual and friend functions.	The C++ language supports virtual and friend functions.
Primary Focus	C language focuses on the process or method instead of focusing on the data.	C++ language focuses on the data instead of focusing on the procedure or method.
Inheritance	The inheritance feature assists the child class in reusing the parent class's properties. The C language offers no support for inheritance.	The C++ language provides support for inheritance.
Allocation and Deallocation of Memory	The C language provides <code>calloc()</code> and <code>malloc()</code> for dynamic allocation of memory and <code>free()</code> for deallocation of memory.	The C++ language provides a new operator for the allocation of memory and a delete operator for the deallocation of memory.
Exception Handling	It does not provide any direct support for exceptional handling. C language requires the usage of functions that support exception handling.	It provides direct support for exceptional handling. The C++ language uses a try-catch block.
Access Modifiers	The structures in C have no access modifiers.	The structures in C++ do have access modifiers.
Type of Approach	C language follows a top-down approach. It functions to break down the main module into various tasks. Then it breaks these tasks into sub-tasks, and so on.	C++ language follows the bottom-up approach. It means that it first develops the lower-level modules and then moves on to the next-level modules.
Function for Input/Output	The C language uses the <code>scanf()</code> and <code>printf()</code> functions for the input and output operations.	In the C++ language, it uses the <code>cin</code> and <code>cout</code> for the input and output operations.

2.5 Simple C++ Program

A "Hello, World!" is a simple program that outputs Hello, World! on the screen. Since it's a very simple program, it's often used to introduce a new programming language to a newbie.

Let's see simple c++ program example.

**Example**

```
// Simple C++ Program
#include <iostream>
using namespace std;
int main() {
    cout << "Hello world!";
    return 0;
}
```

Output

**Did you know?**

1. // Simple C++ Program

In C++, any line starting with // is a comment. Comments are intended for the person reading the code to better understand the functionality of the program. It is completely ignored by the C++ compiler.

2. #include <iostream>

The #include is a preprocessor directive used to include files in our program. The above code is including the contents of the iostream file.

This allows us to use cout in our program to print output on the screen.

3. int main() {...}

A valid C++ program must have the main() function. The curly braces indicate the start and the end of the function.

The execution of code begins from this function.

4. cout << "Hello World!";

cout prints the content inside the quotation marks. It must be followed by << followed by the format string. In our example, "Hello World!" is the format string.

5. return 0;

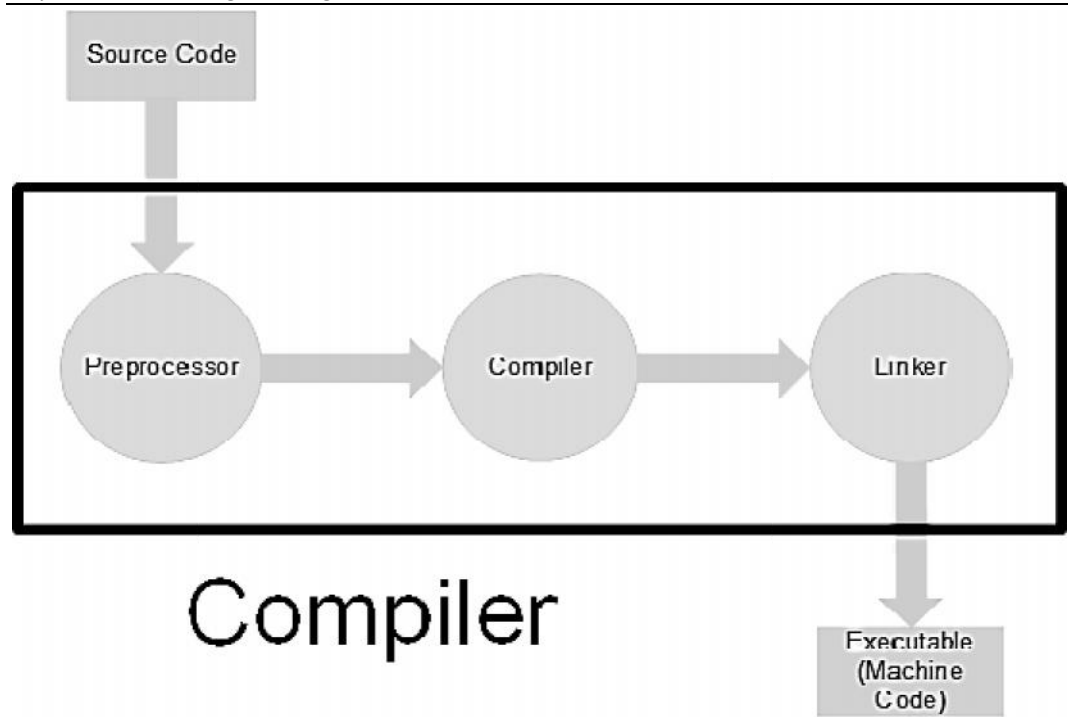
The return 0; statement is the "Exit status" of the program.

**Note**

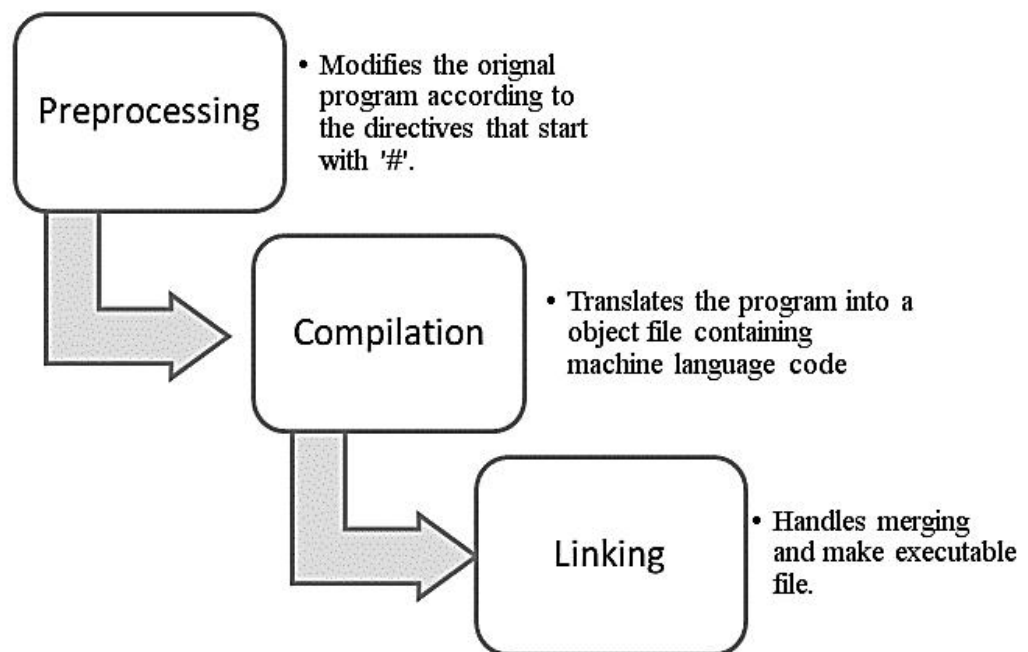
; is used to indicate the end of a statement.

2.6 Compiling and linking

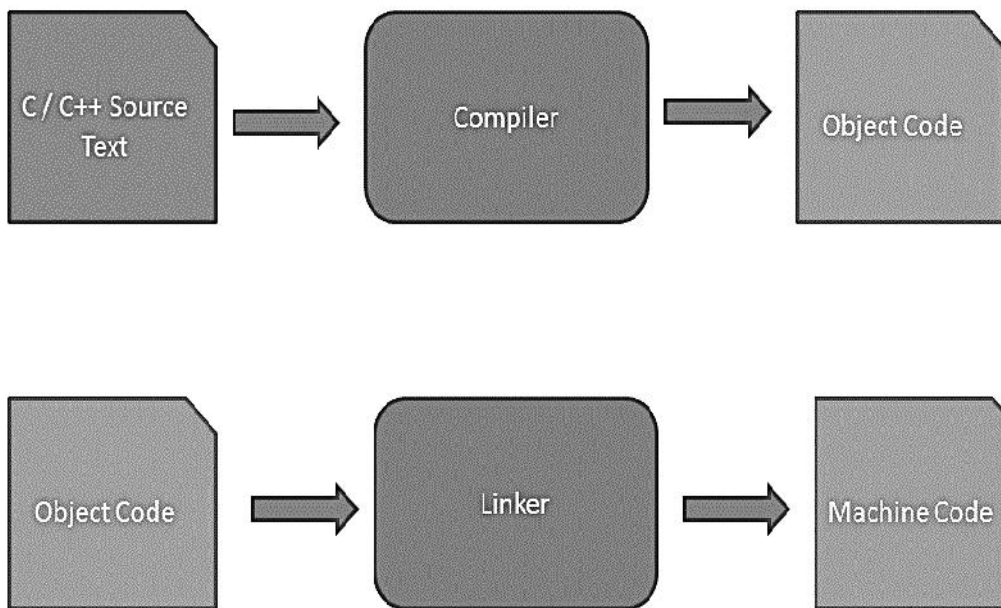
The modified source code is compiled into binary object code. This code is not yet executable.



The object code is combined with the required supporting code to make an executable program.



Compiling and linking



Compiling of C++ Program

1. **Preprocessing**– In simple terms, a C Preprocessor is just a text substitution tool and it instructs the compiler to do required pre-processing before the actual compilation. It handles preprocessing directives like #include, #define, etc.
2. **Compilation** – The compilation takes place on the preprocessed files. The compiler parses the pure C++ source code and converts it into assembly code. The compiler won't give an error unless the source code is not well-formed.
3. **Linking** – The linker produces the final compilation output from the object files the compiler produced. This output can be a shared (or dynamic) library or an executable.

2.7 Review of Tokens

As we know, the smallest individual units in a program are known as tokens. C++ has the following tokens:

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

A C++ program is written using these tokens, white spaces, and the syntax of the language. Most of the C++ tokens are basically similar to the C tokens with the exception of some additions and minor modifications.



Did you Know?

How tokens are being separated?

Tokens are usually separated by “white space.” White space can be one or more:

- a) Blanks
- b) Horizontal or vertical tabs
- c) New lines
- d) Formfeeds
- e) Comments

2.8 Keywords

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

Object Oriented Programming

The following Table gives the complete set of C++ keywords. The keywords not found in ANSI C are shown in boldface. These keywords have been added to the ANSI C keywords in order to enhance its features making it an object-oriented language.

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

2.9 Identifiers

Identifiers refer to the names of variables, functions, arrays, classes, etc., created by the programmer. They are the fundamental requirement of any language. Each language has its own rules for naming these identifiers. The following rules are common to both C and C++:

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

A major difference between C and C++ is the limit on the length of a name. While ANSI C recognizes only the first 32 characters in a name, C++ places no limit on its length and, therefore, all the characters in a name are significant.

Care should be exercised while naming a variable that is being shared by more than one file containing C and C++ programs. Some operating systems impose a restriction on the length of such a variable name.



Caution

The first character in an identifier must be a letter or the _ (underscore) character; however, beginning identifiers with an underscore is considered a poor programming style.

2.10 Constants

In C++, we can create variables whose value cannot be changed. `const` keyword is used to declare constant.



Example

```
const float pi = 3.14;
```

Types of Constants in C++

1. Integer constants
2. Floating constants
3. Character constants
4. String constants
5. Octal constants

6. Hexadecimal constants

2.11 Strings

- A string stores a sequence of characters.
- It terminates with a null character '\0'.
- Unlike characters, strings in C++ are always enclosed within double quotes (" ").



Example

```
char name[30] = "Hello";
```

2.12 Operators

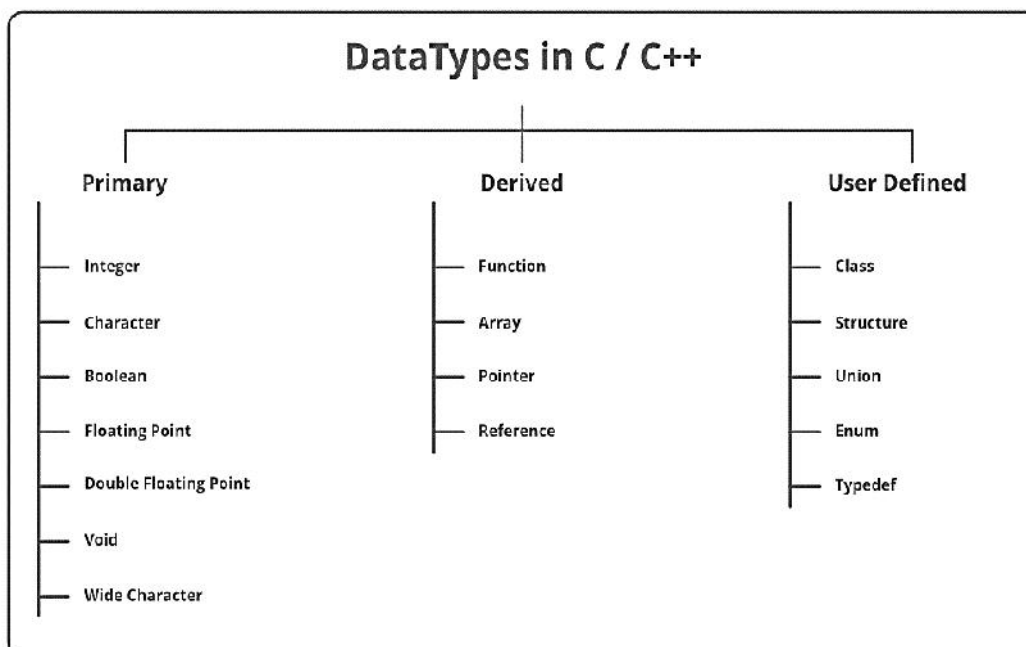
Operators are symbols that operate on operands. These operands can be variables or values. Operators help us to perform mathematical and logical computations.

Common C++ Operators are

1. Arithmetic
2. Assignment
3. Relational
4. Logical
5. Bitwise
6. Other operators

2.13 Data Types

Data types define the type of data a variable can hold, For Example, an integer variable can hold integer data, a character type variable can hold character data, etc.



Primitive Data Types

These data types are built-in or predefined data types and can be used directly by the user to declare variables. For Example: int, char, float, bool etc.

Derived Data Types

Object Oriented Programming

The data-types that are derived from the primitive or built-in datatypes are referred to as Derived Data Types.

2.14 Reference Variables

- Reference variable is an alternate name of already existing variable.
- It cannot be changed to refer another variable and should be initialized at the time of declaration and cannot be NULL.
- The operator '&' is used to declare reference variable.



Lab Exercise

```
#include<iostream>
using namespace std;
int main(){
int x=30;
int &r=x;
cout<<r;
return 0;
}
```

Output

```
30
Process returned 0 (0x0)   execution time : 1.035 s
Press any key to continue.
```

Summary

- C++ language is made up of letters, words, sentences and constructs just like English language.
- A collection of characters, much like a word in English language are called tokens.
- A token can be a keyword, or identifier, constant, string or an operator.
- Keywords are the reserve words that cannot be used as names of variable or other userdefined program elements.
- Identifiers refer to the names of variables, functions, arrays, classes etc. created by the programmer, Basic types.
- An array represents named list of finite number of similar data elements. A function is a named part of a program that can be invoked from the other parts of the program.
- A pointer is a variable that holds a memory address of another variable. A reference is an alternative name for an object. A constant is a data item whose data value can never change during the program run.

Keywords

Character Constant: One or more characters enclosed in single quotes.

Expression: A combination of variables, constants and operators written according to some rules.

Identifiers: The names of variables, functions, arrays, classes, etc. created by the programmer.

SelfAssessment

1. What is C++?
 - A. C++ is an object oriented programming language
 - B. C++ is a procedural programming language
 - C. C++ supports both procedural and object oriented programming language
 - D. C++ is a functional programming language

2. Which of the following approach is used by C++?
 - A. Left-right
 - B. Right-left
 - C. Top-down
 - D. Bottom-up

3. Which of the following is an extension of C program?
 - A. .doc
 - B. .c
 - C. .cc
 - D. .cprog

4. Which of the following type is provided by C++ but not C?
 - A. double
 - B. float
 - C. int
 - D. bool

5. What would be the output of following code snippet?
`#include <stdio.h>`

`int main()`
`{`
`printf("1024");`

`return 0;`
`}`
 - A. 1024
 - B. 10
 - C. Error
 - D. 0

6. C++ provides various types of that include keywords, identifiers, constants, strings, and operators.
 - A. expressions
 - B. structures
 - C. tokens
 - D. None of the above

Object Oriented Programming

7. refer to fixed values that do not change during the execution of a program.
- A. Identifiers
 - B. Constants
 - C. Strings
 - D. Operators
8. _____ used for function calls and parameters.
- A. { }
 - B. ()
 - C. @
 - D. #
9. _____ used to access a structure member.
- A. @
 - B. .
 - C. //
 - D. \$
10. What are the parts of the literal constants?
- A. integer numerals
 - B. floating-point numerals
 - C. strings and boolean values
 - D. all of the above
11. ` Which of the following is not a derived data type?
- A. Function
 - B. Int
 - C. Array
 - D. Pointer
12. Pick the odd one out.
- A. boolean type
 - B. integer type
 - C. array type
 - D. character type
13. What is the size of an int data type?
- A. 4 Bytes
 - B. 2 Bytes
 - C. 8 Bytes
 - D. Depends on the system/compiler
14. _____ data type holds whole numbers? (4 bytes)
- A. Bool
 - B. String
 - C. Double
 - D. Int
15. What would be the output of following code snippet?
- ```
#include <iostream>

using namespace std;

int main() {
 double a;
```

```
cout << sizeof(a);

return 0;

}
```

- A. 1
- B. 0
- C. 8
- D. 4

### **Answer for Self Assessment**

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

### **Review Questions**

1. Explain various types of data types in c++.
2. What is a constant?
3. Why keywords required in the programming? Explain.
4. Write a program that performs reference operation.
5. Explain different types of operators.



### **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://msdn.microsoft.com/en-us/library/wcz57btd(v=vs.80).aspx)
- <https://www.codecademy.com/learn/learn-c-plus-plus>

## **Unit 03: Operators and Type Casting**

### **CONTENTS**

Objectives

Introduction

3.1 Operators in C++

3.2 Scope Resolution Operator

3.3 Member de-referencing operators

3.4 Type Conversion

3.5 Basic Type to Class Type

3.6 Class Type to Basic Type

3.7 Class Type to another Class Type

Summary

Keywords

Self Assessment

Answers for Self Assessment

Review Questions

Further Readings

### **Objectives**

After studying this unit, you will be able to:

- Recognize the type conversions
- Understand operators
- 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.

### 3.1 Operators in C++

An operator is a symbol that operates on a value to perform specific mathematical or logical computations. They form the foundation of any programming language. In C++, we have built-in operators to provide the required functionality.

An operator operates the operands. For example,



#### Example

```
int c = a + b;
```

Here, '+' is the addition operator. 'a' and 'b' are the operands that are being 'added'.

**Operators in C++ can be classified into 6 types:**

1. Arithmetic Operators
2. Assignment Operators
3. Relational Operators
4. Logical Operators
5. Bitwise Operators
6. Other Operators

#### Arithmetic Operators

Arithmetic operators are used to perform arithmetic operations on variables and data. For example,



E

#### Example

```
a+b;
```

| Operator | Operation                                   |
|----------|---------------------------------------------|
| +        | Addition                                    |
| -        | Subtraction                                 |
| *        | Multiplication                              |
| /        | Division                                    |
| %        | Modulo Operation (Remainder after division) |

#### Assignment Operators

Assignment operators are used to assign values to variables. For example,



#### Example

```
X=10;
```

Here, we have assigned x value of 5 to the variable x..

| Operator | Example |
|----------|---------|
| =        | x = y;  |
| +=       | x += y; |

**Unit 03: Operators and Type Casting**

|                 |                      |
|-----------------|----------------------|
| <code>-=</code> | <code>x -= y;</code> |
| <code>*=</code> | <code>x *= y;</code> |
| <code>/=</code> | <code>x /= y;</code> |
| <code>%=</code> | <code>x %= y;</code> |

**Relational Operators**

A relational operator is used to check the relationship between two operands. For example,

**Example**

`a > b;`

Here, `>` is a relational operator. It checks if `a` is greater than `b` or not.

If the relation is true, it returns 1 whereas if the relation is false, it returns 0.

| Operator           | Meaning                  | Example                              |
|--------------------|--------------------------|--------------------------------------|
| <code>==</code>    | Is Equal To              | <code>3 == 5</code> gives us false   |
| <code>!=</code>    | Not Equal To             | <code>3 != 5</code> gives us true    |
| <code>&gt;</code>  | Greater Than             | <code>3 &gt; 5</code> gives us false |
| <code>&lt;</code>  | Less Than                | <code>3 &lt; 5</code> gives us true  |
| <code>&gt;=</code> | Greater Than or Equal To | <code>3 &gt;= 5</code> give us false |
| <code>&lt;=</code> | Less Than or Equal To    | <code>3 &lt;= 5</code> gives us true |

**Logical Operators**

Logical operators are used to check whether an expression is true or false. If the expression is true, it returns 1 whereas if the expression is false, it returns 0.

| Operator                | Example                                         | Meaning                                                      |
|-------------------------|-------------------------------------------------|--------------------------------------------------------------|
| <code>&amp;&amp;</code> | <code>expression1 &amp;&amp; expression2</code> | Logical AND.<br>True only if all the operands are true.      |
| <code>  </code>         | <code>expression1    expression2</code>         | Logical OR.<br>True if at least one of the operands is true. |
| <code>!</code>          | <code>!expression</code>                        | Logical NOT.<br>True only if the operand is false.           |

**Bitwise Operators**

In C++, bitwise operators are used to perform operations on individual bits. They can only be used alongside `char` and `int` data types.

Object Oriented Programming

| Operator | Description             |
|----------|-------------------------|
| &        | Binary AND              |
|          | Binary OR               |
| ^        | Binary XOR              |
| ~        | Binary One's Complement |
| <<       | Binary Shift Left       |
| >>       | Binary Shift Right      |

**Other C++ Operators**

Here's a list of some other common operators available in C++. We will learn about them in later tutorials.

| Operator | Description                                                | Example                                             |
|----------|------------------------------------------------------------|-----------------------------------------------------|
| sizeof   | returns the size of data type                              | sizeof(int); // 4                                   |
| ?:       | returns value based on the condition                       | string result = (5 > 0) ? "even" : "odd"; // "even" |
| &        | represents memory address of the operand                   | &num; // address of num                             |
| .        | accesses members of struct variables or class objects      | s1.marks = 92;                                      |
| ->       | used with pointers to access the class or struct variables | ptr->marks = 92;                                    |
| <<       | prints the output value                                    | cout << 5;                                          |
| >>       | gets the input value                                       | cin >> num;                                         |

**3.2 Scope Resolution Operator**

The scope resolution operator is used to reference the global variable or member function that is out of scope. Therefore, we use the scope resolution operator to access the hidden variable or function of a program. The operator is represented as the double colon (::) symbol.

***Uses of the scope resolution Operator***

1. It is used to access the hidden variables or member functions of a program.
2. It defines the member function outside of the class using the scope resolution.
3. It is used to access the static variable and static function of a class.
4. The scope resolution operator is used to override function in the Inheritance.

**3.3 Member de-referencing operators**

The pointer-related operator & and \* are called referencing and dereferencing operators.

The referencing operator (&) is a unary operator and it returns the address of its operand variable.



### Unit 03: Operators and Type Casting

The dereferencing operator (\*) is a unary operator that returns the value present at the specified address.



#### Example

```
#include<iostream>
using namespace std;
int main()
{
 int n= 50;
 int *ptr;
 ptr=&n;
 cout<<"\nAddress of n ="<<&n;
 cout<<"\nValue in variable ptr ="<< ptr;
 cout<<"\nValue of n ="<<n;
 cout<<"\nValue using dereferencing operator="<<*ptr;
 return 0;
}
```

Output

```
Address of n =0x61fe14
Value in variable ptr =0x61fe14
Value of n =50
Value using dereferencing operator=50
Process returned 0 (0x0) execution time : 1.409 s
Press any key to continue.
```

## 3.4 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*

**Object Oriented Programming**

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 Exercise**

```
//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

### 3.5 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;
```

Object Oriented Programming

---

```

public:
time (int t) // 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 and 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";

```

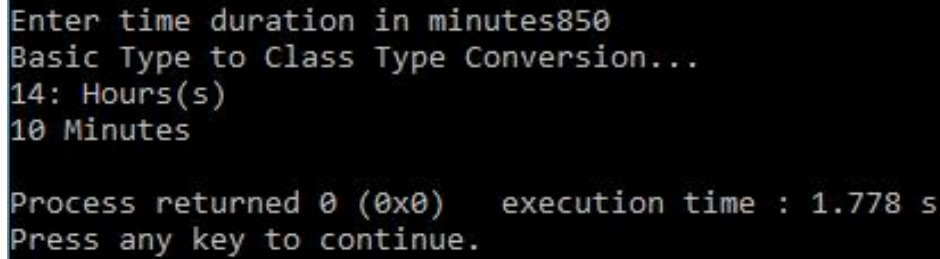
Unit 03: Operators and Type Casting

```

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";

```

Object Oriented Programming

```

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.

**3.6 Class Type to Basic Type**

The constructor functions do not support conversion from a class to basic type. C++ allows us to define an 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*()
{
return(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;
}
```

Object Oriented Programming

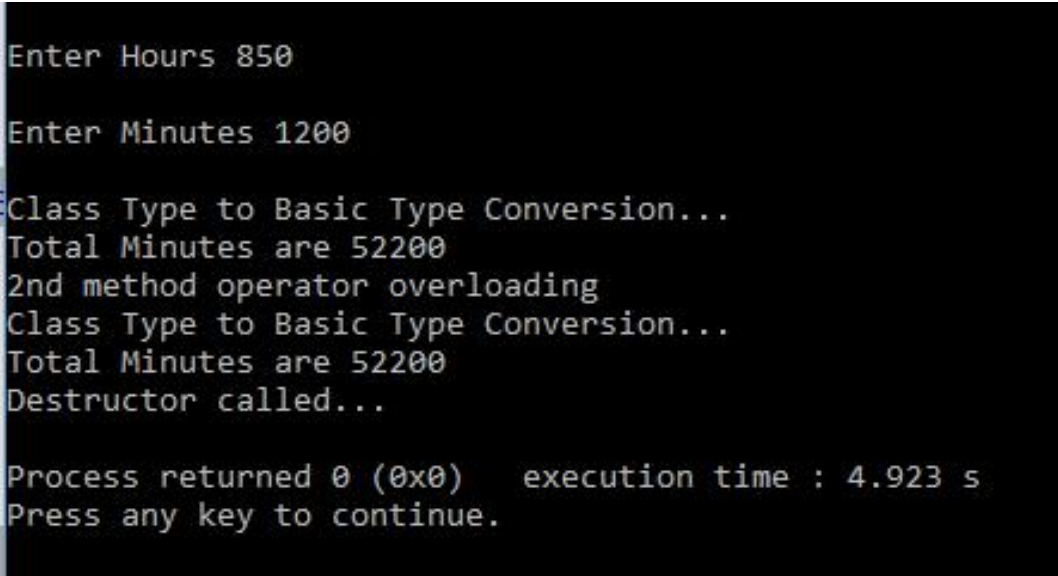
```

};

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.

**3.7 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**



Unit 03: Operators and Type Casting

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

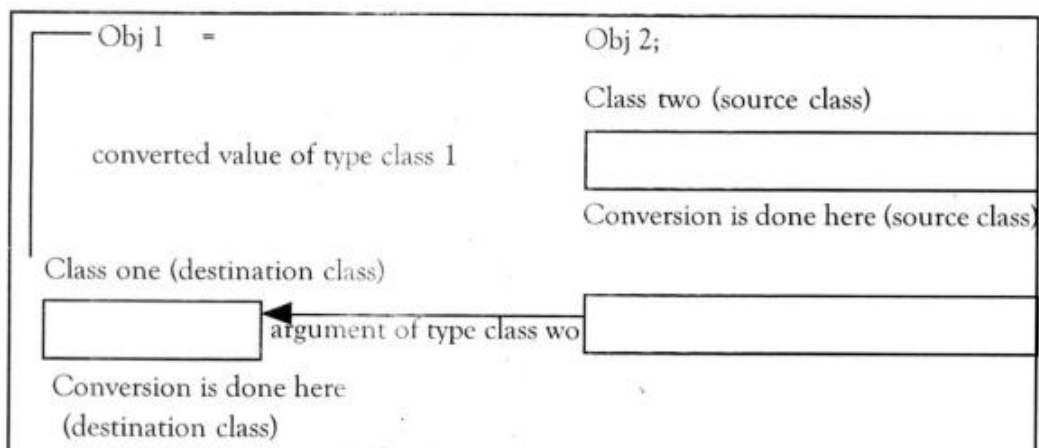
Obj1 is an object of class one and Obj2 is an object of class two. The class two type data is converted to class one type data and the converted value is assigned to the Obj1. Since the conversion takes place from class two to class one, two is known as the source and one is known as the destination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

Object Oriented Programming

---

```

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,intq,float r)
 {
 ino=n;
 qty=q;
 rate=r;
 }
 int getino()

```

Object Oriented Programming

```

{
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?  
 $(\text{float})a * (\text{int})b / (\text{long})c * (\text{double})d$ 
  - 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?

---

**Unit 03: Operators and Type Casting**

---

- 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://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 04: Control Structures**

### **CONTENTS**

Objectives

Introduction

4.1 Decision-Making Controls

4.2 Iterative Controls

4.3 Jumping Controls

Summary

Keywords

Self Assessment

Answers for Self Assessment

Review Questions

Further Readings

### **Objectives**

After studying this unit, you will be able to:

- Understand control structure
- Recognize decision-making control
- Analyze iterative controls
- Recognize the jumping controls

### **Introduction**

A C++ control statement redirects the flow of a program in order to execute additional code. These statements come in the form of conditionals (if-else, switch) and loops (for, while, do-while). Each of them relies on a logical condition that evaluates to a Boolean value in order to run one piece of code over another.

#### **4.1 Decision-Making Controls**

The decision control statements are the decision-making statements that decide the order of execution of statements based on the conditions. In the decision-making statements, the programmer specifies which conditions are to be executed or tested with the statements to be executed if the condition is true or false.

**Decision-making statements are:**

- if statement
- if-else statement
- nested if statements
- switch statement

#### ***If statement***

The if statement consists a condition which is followed by one or some of the statements, if the condition is true then the statements will be executed or else not. This statement is the simple and easy decision control statement.

***Syntax***

```

if (condition)
{
 //statements to be executed if condition is true
}

```

**Lab Exercise**

```

#include<iostream>
Using namespacestd;
int main()
{ int a = 10;
 if (a > 20)
 {
 cout<<"10 is less than 20";
 }
 cout<<"Out of if block";
}

```

***if-else statement***

In the if-else statement the if statement is followed by the else statement which will execute when the expression is false.

***Syntax***

```

if (condition)
{ //statement will execute if the condition is true
}
else
{ //statement will execute if the condition is false
}

```

**Lab Exercise**

```

#include<iostream>
using namespace std;
int main(){
 int a=10,b=20;
 if(a>b)
 {
 cout<<"a is greater than b";
 }
 else
 {
 cout<<"b is greater than a";
 }
}

```

***nested if statements***

Nested if statements means “One if statement within another if statement”. There may be situations where you may need to evaluate a condition based on the result of another condition. In these cases, nested if statements will come in handy. The syntax of nested if statement will be:

```
if (condition 1)
{
 if (condition 2)
 {
 if (condition 3)
 {
 // execute statement1;
 }
 if (condition 4)
 {
 // execute statement2;
 }
 }
}
```

**Lab Exercise**

```
#include <iostream>
using namespace std;
int main()
{
 int a;
 cout<< "Enter a number" << '\n';
 cin>>a;
 if(a>=10)
 {
 if(a>=20)
 {
 if(a>=100)
 {
 cout<<"You entered a value greater than 100." << '\n';
 }
 if(a<=75)
 {
 cout<<"You entered a value less than 75" << '\n';
 }
 }
 if(a<=30)
 {
```

**Object Oriented Programming**

---

```

cout<<"You entered a value less than 30" << '\n';
}
}
if(a<10)
{
cout<<"You entered a value less than 10" << '\n';
}
}

```

**switch statement**

You can use the switch statement when you want to evaluate a condition against different cases (expressions). Depending on the input, the appropriate case will be executed, and the result will be printed. The syntax of switch case statement will be:

```

switch (condition)
{
case expression1:
//statement to execute;
break;
case expression2:
//statement to execute;
break;
case expression3:
//statement to execute;
break;
...
default:
// statement to execute;
}

```

**Lab Exercise**

```

#include <iostream>
using namespace std;
int main()
{
int a;
cout<< "Enter a number between 1 to 7 : " << '\n';
cin>>a;
switch (a)
{
case 1:
{
cout<< "The first colour of the rainbow is VIOLET: " << '\n';
break;
}
}
}

```

```
}
case 2:
{
cout<< "The second colour of the rainbow is INDIGO: " << '\n';
break;
}
case 3:
{
cout<< "The third colour of the rainbow is BLUE: " << '\n';
break;
}
case 4:
{
cout<< "The fourth colour of the rainbow is GREEN: " << '\n';
break;
}
case 5:
{
cout<< "The fifth colour of the rainbow is YELLOW: " << '\n';
break;
}
case 6:
{
cout<< "The sixth colour of the rainbow is ORANGE: " << '\n';
break;
}
case 7:
{
cout<< "The seventh colour of the rainbow is RED: " << '\n';
break;
}
default:
cout<< "You have entered a number other than 1 - 7. Please try again!" << '\n';
}
}
```

## 4.2 Iterative Controls

The iteration (for, while, and do-while loop) statements allows a set of instruction to be performed repeatedly until a certain condition is fulfilled. The iteration statements are also called loops or looping statements. C++ provides three kinds of loops:

- for loop
- while loop

- do-while loop



### Notes

A loop has four elements that have different purposes. These elements are:

Initialization Expression(s)

Test Expression

Update Expression(s)

Loop's Body

### *for loop*

The for loop is one of the most widely used loops in C++. The for loop is a deterministic loop in nature, that is, the number of times the body of the loop is executed is known in advance.

#### *Syntax*

For(initialization expression(s); test-expression; update expression(s))

```
{
 body-of-the-loop;
}
```



### Lab Exercise

```
#include<iostream>
using namespace std;
int main (){
 int n=0;
 for (n=1; n<=10; n++)
 cout<<n<<" ";
 return 0;
}
```

Output:

```
1 2 3 4 5 6 7 8 9 10
Process returned 0 (0x0) execution time : 0.031 s
Press any key to continue.
```

### *while loop*

The second loop available in C++ is the while loop. The while loop is an entry-controlled loop.

#### *Syntax*

While(expression)

```
{
 loop-body;
}
```



### Lab Exercise

```
#include<iostream>
```

```
using namespace std;
int main(){
 int n=0;
 while(n<=10){
 n++;
 cout<<n<<" ";
 }
 return 0;
}
```

Output:

```
1 2 3 4 5 6 7 8 9 10 11
Process returned 0 (0x0) execution time : 0.023 s
Press any key to continue.
```

### *do-while loop*

Unlike the for and while loops, the do-while is an exit-controlled loop i.e., it evaluates its test-expression at the bottom of the loop after executing its loop-body statements. This means that a do-while loop always executes at least once.

#### *Syntax*

```
do
{
 statement ;
}while(test-expression);
```



#### **Lab Exercise**

```
#include<iostream>
using namespace std;
int main(){
 int n=0;
 do{
 n++;
 cout<<n<<" ";
 }
 while(n<10);
 return 0;
}
```

Output

```
1 2 3 4 5 6 7 8 9 10
Process returned 0 (0x0) execution time : 0.024 s
Press any key to continue.
```

### 4.3 Jumping Controls

Jump statements are used to interrupt the normal flow of program. Types of Jump Statements are

- goto statement
- break statement
- continue statement

#### *goto statement*

The goto statement is a control statement which is used to transfer the control from one place to another place without any condition in a program.

| Syntax1            | Syntax2            |
|--------------------|--------------------|
| <b>goto label;</b> | <b>label:</b>      |
| -----              | -----              |
| -----              | -----              |
| -----              | -----              |
| <b>label:</b>      | <b>goto label;</b> |



#### Lab Exercise

```
#include<iostream>
using namespacestd;
int main(){
 int n=0;
 while(n<10){
 if(n==4)
 gotox;
 n++;
 cout<<n<<" ";
 }
 x:
 cout<<"Hello Label";
 return 0;
}
```

Output

```
1 2 3 4 Hello Label
Process returned 0 (0x0) execution time : 0.024 s
Press any key to continue.
```

#### *break statement*



**Lab Exercise**

```
#include<iostream>
using namespace std;
int main(){
 int n=0;
 while(n<10){
 if(n==4)
 break;
 n++;
 cout<<n<<" ";
 }
 return 0;
}
```

Output:

```
1 2 3 4
Process returned 0 (0x0) execution time : 0.031 s
Press any key to continue.
```

***continue statement***

The continue statement works somewhat like the break statement. Instead of forcing termination, however, continue forces the next iteration of the loop to take place, skipping any code in between.

***Syntax***

```
continue;

#include <iostream>
using namespace std;
int main () {
 int a = 5;
 do {
 if(a == 10) {
 a =a+1;
 continue;
 }
 cout<< "value of a: " << a <<endl;
 a = a + 1;
 }
 while(a< 20);
 return 0;
}
```

Output:

```
value of a: 5
value of a: 6
value of a: 7
value of a: 8
value of a: 9
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 15
value of a: 16
value of a: 17
value of a: 18
value of a: 19

Process returned 0 (0x0) execution time : 0.023 s
Press any key to continue.
```

### Summary

- Selection structures are implemented using If , If Else and Switch statements.
- Looping structures are implemented using While, Do While and For statements.
- The statements given as "initialization statements" are executed only once, at the beginning of a for loop.
- There are 3 statements given to a for loop as shown. One for initialization purpose, other for condition testing and last one for iterating the loop. Each of these 3 statements are separated by semicolons.
- The C++ language provides a set of control statements that allows you to conditionally control data input and output. These controls are referred to as loops.

### Keywords

**If statement** - The single if statement in C++ language is used to execute the code if a condition is true.

**If-else statement** - The if-else statement in C++ language is used to execute the code if the condition is true or false. It is also called a two-way selection statement.

**Switch Statement** - Switch statement acts as a substitute for a long if-else-if ladder that is used to test a list of cases.

### SelfAssessment

1. Decision Control statements in C++ can be implemented using
  - A. if
  - B. if-else
  - C. Conditional Operator
  - D. All of the above
2. \_\_\_\_\_ is looping statement in C++.
  - A. If
  - B. Switch
  - C. For

- 
- D. None of above
3. To make decision based on multiple choices, \_\_\_\_\_ is best suited.
- A. If
  - B. If-else
  - C. If-else-if
  - D. None of the above
4. How many case statements are allowed before single break statement in switch statement?
- A. 1
  - B. 2
  - C. Multiple
  - D. 100
5. After case keyword \_\_\_\_\_ symbol is used.
- A. :
  - B. ;
  - C. >
  - D. /
6. Which from the following is not a part of the for statement?
- A. Initialization
  - B. Statement checker
  - C. Update
  - D. Continuation condition
7. How many types of Iterators are there?
- A. 1
  - B. 2
  - C. 3
  - D. 5
8. How many types of loops are there in C++?
- A. 4
  - B. 2
  - C. 3
  - D. 1
9. Which Loop is Faster in C++?
- A. For
  - B. While
  - C. Do While
  - D. All work at same speed

10. Do While Loop is also known as \_\_\_\_\_.  
A. Entry Control  
B. Virtual Control  
C. Exit Control  
D. All of Above
11. What is true about a break?  
A. Break stops the execution of entire program  
B. Break halts the execution and forces the control out of the loop  
C. Break forces the control out of the loop and starts the execution of next iteration  
D. Break halts the execution of the loop for certain time frame
12. Which of the following is used with the switch statement?  
A. Continue  
B. Exit  
C. break  
D. for
13. Which of the following is a decision-making statement?  
A. main()  
B. void  
C. goto  
D. None of above
14. \_\_\_\_\_ loop is guaranteed to execute at least once?  
A. For loop  
B. While loop  
C. Do-while loop  
D. None of above
15. Which of the following is a looping statement?  
A. If  
B. If-else  
C. Switch  
D. None of above

### **Answers for SelfAssessment**

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

## **Review Questions**

1. What do you mean by nested if statement? Explain with suitable example.
2. What is difference between looping statements and jumping statements?
3. Write a program to reverse entered number.
4. What do you mean by switch statement? How it is different from if else statement.
5. Differentiate between while loop and do- while loop with suitable example.



## **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://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 05: Pointers and Structures**

### **CONTENTS**

Objectives

Introduction

5.1 Main Function

5.2 Function Prototyping

5.3 Handling Pointers

5.4 C Structures and Limitations

Summary

Keywords

Self Assessment

Answers for Self Assessment

Review Questions

Further Readings

### **Objectives**

After studying this unit, you will be able to:

- Understand pointers
- Recognize Structures
- Analyze C structures

### **Introduction**

A function is a programming unit with a unique name that may be identified. It can be invoked by a programme once it has been defined. When called, it may accept zero or more inputs. The code placed inside the function specification determines what should be done with the incoming input(s). The function generates a single output after doing the given transformation. The caller of the function receives this output.

Computers use their memory for storing instructions of the programs as well as the values of the variables. Since memory is a sequential collection of storage cells each cell has an address associated with it. Whenever we declare a variable, the system allocates, somewhere in the memory, a memory location and a unique address is assigned to this location. Whenever a value is assigned to this variable the value gets stored in the location having a unique address in the memory associated with that variable. Therefore, the values stored in memory can be manipulated using their addresses. Pointer is an extremely powerful mechanism to write efficient programs. Incidentally, this feature makes C stand out as the most powerful programming language. Pointers are the topic of this unit.

### **5.1 Main Function**

Main () function is the entry point of any C++ program. It is the point at which execution of program is started. When a C++ program is executed, the execution control goes directly to the main() function. Every C++ program have a main() function.

Syntax

```
void main()
{
```

```

.....
.....
}

```

**Example**

```

#include<iostream.h>
Using namespace std;
void main()
{
cout<<"This is main function";
}

```

## 5.2 Function Prototyping

Function Prototyping is the process of declaring a function for the compiler to understand the function name, arguments, and return type.

### *Parts Of Functions*

1. Function Declaration
2. Function Call
3. Function Definition

### **Function Declaration**

```
return_type function_name(param_1,param_2 ... param_n);
```

**Example**

```
int sum(int x,int y)
```

### *return\_type*

A function can return a value using the return statement to the main program. The function does not return value if the return\_type is void.

### *function\_name*

A function\_name is a user-defined name (identifier) for calling/using the function. The function name and the parameter list together constitute the function signature.

### *Parameters (param\_1,param\_2 ... param\_n)*

A parameter is variable which holds data. The Argument transfers the data from the main program to the function definition.

**Actual arguments:** The main program variables(arguments) are known as the actual arguments during the function call.

**Formal arguments:** The Function definition variables(arguments) are known as the formal arguments.

### **Function Call**

When the main program is run, the program reaches the function call statement and invokes the function and passes the control to it.

```

int main()
{
....
}

```

```
add(a,b); // Function Call
```

```
....
```

```
}
```

### Function Definition

The actual function code block or function body is called as the function definition.

When the function call is triggered by the compiler, the program control is passed to the function definition. Then the compiler executes statements in the body of the function and the program control returns when the return statement or closing braces{ } is reached.



#### Example

```
#include<iostream>
using namespace std;
int add(int x,int y); // Function Declaration
int main() {
 int a=100,b=100,c;
 c = add(a,b); // Function Call
 cout<<"Addition : "<<c;
}
int add(int x,int y) // Function Definition
{
 int z;
 z = x+y;
 return z;
}
```

## 5.3 Handling Pointers

A memory variable is merely a symbolic reference given to a memory location. Now let us consider that an expression in a C program is as follows:

```
int a = 10, b = 5, c;
```

```
c = a + b;
```

The above expression implies that a, b and c are the variables which can hold the integer data. Now from the above-mentioned statement let us assume that the variable 'a' occupies the address 3000 in the memory, 'b' occupies 3020 and the variable 'c' occupies 3040 in the memory. Then the compiler will generate the machine instruction to transfer the data from the location 3000 and 3020 into the CPU, add them and transfer the result to the location 3040 referenced as c. Hence

we can conclude that every variable holds two values:

Address of the variable in the memory (l-value)

Value stored at that memory location referenced by the variable. (r-value)

Pointer is nothing but a simple data type in C programming language, which has a special characteristic to hold the address of some other memory location as its r-value. C programming language provides '&' operator to extract the address of any object. These addresses can be stored in the pointer variable and can be manipulated.

The syntax for declaring a pointer variable is,

```
<data type> *<identifier>;
```



## Object Oriented Programming

### Example

```
int n;
int *ptr; /* pointer to an integer*/
```

The following statement assigns the address location of the variable n to ptr, and ptr is a pointer to n.

```
ptr=&n;
```

Since a pointer variable points to a location, the content of that location is obtained by prefixing the pointer variable by the unary operator \* (also called the indirection or dereferencing operator) like, \*<pointer\_variable>.

### Accessing the Address of a Variable

The actual location of a variable in the memory is system dependent and therefore, the address of a variable is not known to us immediately. How can we then determine the address of a variable?

This can be done with the help of the operator & available in C. The operator & immediately preceding a variable return the address of the variable associated with it.

**Example:** The statement

```
P = &quantity;
```

Would assign the address 5000 to the variable p. The & operator can be remembered as 'address of'.

The &operator can be used only with a simple variable or an array element. The following are illegal use of address operator:

&125 (pointing at constant).

```
Intx[10];
```

&x (pointing at array names).

&(x+y) (pointing at expressions).

If x is an array, then expression such as

```
&x[0] and &x[i+3]
```

are valid and represent the addresses of 0th and (i+3)th elements of x.

### Pointer Declaration

Since pointer variables contain address that belongs to a separate data type, they must be declared as pointers before we use them. Pointers can be declared just as any other variables. The declaration of a pointer variable takes the following form:

```
data_type *pt_name;
```

The above statement tells the compiler three things about the variable pt\_name.

1. The asterisk (\*) tells that the variable pt\_name is a pointer variable.
2. pt\_name needs a memory location.
3. pt\_name points to a variable of type data type.



: The statement

```
int *p;
```

declares the variable p as a pointer variable that points to an integer data type (int). The type int refers to the data type of the variable being pointed to by p and not the type of the value of the pointer.

Given below are some more examples of pointer declaration

| Pointer declaration | Interpretation |
|---------------------|----------------|
|---------------------|----------------|

|                  |                                                                                                                                                    |
|------------------|----------------------------------------------------------------------------------------------------------------------------------------------------|
| int *rollnumber; | Create a pointer variable rollnumber capable of pointing to an integer type variable or capable of holding the address of an integer type variable |
| char *name;      | Create a pointer variable name capable of pointing to a character type variable or capable of holding the address of a character type variable     |
| float *salary;   | Create a pointer variable salary capable of pointing to a float type variable or capable of holding the address of a float type variable           |

### Accessing a Variable through its Pointer

Consider the following statements:

```
int q, *i, n;
q = 35;
i = &q;
n = *i;
```

i is a pointer to an integer containing the address of q. In the fourth statement we have assigned the value at address contained in i to another variable n. Thus, indirectly we have accessed the variable q through n. using pointer variable i.

### Pointer Expression

Like other variables, pointer variables can be used in expressions. Arithmetic and comparison operations can be performed on the pointers. For example, if p1 and p2 are properly declared and initialized pointers, then following statements are valid.

```
y = *p1 * *p2; /multiply values stored in variables pointed to by *p1/and *p2
```

```
sum = sum + *p1; /increment sum by the value stored in the variable/pointed to by p1
```

The pointer may point to any location in the memory therefore you should be careful while using pointers in your programs.

## 5.4 C Structures and Limitations

A structure is a collection of variables referenced under one name providing a convenient means of keeping related information together. The structure definition creates a format that may be used to declare structure variables in a program later on.

The general format of structure definition is as follows:

```
struct tag_name
{
 data_type member1;
 data_type member2;

};
```

A keyword struct declares a structure to hold the details of fields of different datatypes. At this time, no variable has actually been created. Only a format of a new data type has been defined.

Consider the following example:

```
struct addr
{
 char name [30];
 char street [20];
```

## Object Oriented Programming

---

```
char city [15];
```

```
char state [15];
```

```
int pincode;
```

```
};
```

The keyword struct declares a structure to hold the details of fields of address, namely, #name, street, city, state, pin code. The first four members are character array and fifth one is an integer.

### Creating Structure Variables

The structure declaration does not actually create variables. Instead, it defines data type only. For actual use a structure variable needs to be created. This can be done in two ways:

1. Declaration using tagname anywhere in the program.



**Example:** struct book

```
{
 char name [30];
 char author [25];
 float price;
}
struct book book1, book2;
```

2. It is also allowed to combine structure declaration and variable declaration in one statement.

This declaration is given below:

```
struct person
{
 char * name;
 int age;
 char *address;
}
p1, p2, p3;
```

While declaring structure variables along with their definition, the use of tag name is optional.

```
struct
{
 char *name;
 int age;
 char *address;
}
p1, p2, p3;
```

### Giving Values to Members

As the members are not themselves variables they should be linked to the structure variables. The link between a member and a variable is established using member operator '.' which is also known as dot operator.

This can be explained using following example:



**Example:** / \* Program to define a structure and assign value to members \*/

```

struct book
{
 char * name;
 int pages;
 char *author;
};

main()
{
 struct book b1;
 printf ("\n Enter Values:");
 scanf ("%s %d %s", b1.name, &b1.page, b1.author);
 printf ("%s, %d, %s, b1.name, b1.page, b1.author);
}

```

### Structure Initialization

A structure variable can be initialized as any other data type.

```

main()
{
 staticstruct
 {
 int weight;
 float height;
 }

 student = {60, 180.75};

```

This assigns the value 60 to student. Weight and 180.75 student. Height. There is a one-to-one correspondence between the members and their initializing values.

A structure must be declared as static if it is to be initialized inside a function (similar to arrays). The following statements initialize two structure variables. Here, it is essential to use a tag

```

name.
main()
{
 structst_record
 {
 int weight;
 float height;
 }

 staticstructst_recordstudent1 = {60, 180.75};
 staticstructst_recordstudent2 = {53, 170.60};

}

```

Another method is to initialize a structure variable outside the function as shown below:

```

structst_record /* No static word */

```

Object Oriented Programming

```

{
 int weight;
 int height;
}
student1 = {60, 180.75};
}
main()
{
 static struct st_record student 2 = {53, 170.60}

}

```

The initialization of individual structure members within the template is permitted. The initialization must be done only in the declaration of the actual variables.

**Limitations of C Structures**

- The struct data type cannot be treated like built-in data types.
- Operators like +, -, and others cannot be used on structure variables.
- C Structures do not permit data hiding. Structure members can be accessed by any function, anywhere in the scope of the Structure.
- C structures do not permit functions inside Structure
- C Programming language does not support access modifiers. So they cannot be used in C Structures.
- Structures in C cannot have constructor inside Structures.

**Summary**

- In this unit, we learnt about "Functions": definition, declaration, prototypes, types, function calls, datatypes and storage classes, types function invoking and lastly Recursion.
- All these subtopics must have given you a clear idea of how to create and call functions from other functions, how to send values through arguments, and how to return values to the called function.
- Structure is a derived data type used to store the instances of variables of different data types.
- Structure definition creates a format that may be used to declare structure variables in the program later on.
- The structure operators like dot operator "." are used to assign values to structure members.

**Keywords**

**Data types:** It refers to the type of information while storage class refers to the life-time of a variable and its scope within the program.

**Function Call:** A function can be called by specifying its name followed by a list of arguments enclosed in parentheses and separated by commas.

**Return Statement:** Information is returned from the function to the calling portion of the program via return statement.

**Structure:** A structure is a collection of variables referenced under one name providing a convenient means of keeping related information together. Structures within structure: It means nesting of structures.

### Self Assessment

1. Which of the following is the default return value of functions in C++?
  - A. int
  - B. double
  - C. float
  - D. nothing
  
2. There are \_\_\_\_ types of functions in C++.
  - A. Four
  - B. Two
  - C. Three
  - D. None of the above
  
3. Where does the execution of the program starts?
  - A. user-defined function
  - B. main function
  - C. void function
  - D. library function
  
4. What are mandatory parts in the function declaration?
  - A. Return type, function name, parameters
  - B. Return type, function name
  - C. Parameters, function name
  - D. Parameters, variable
  
5. How many minimum numbers of functions are present in the C++ program?
  - A. 1
  - B. 2
  - C. 3
  - D. 0
  
6. Which of the following operator used for dereferencing or indirection?
  - A. \*
  - B. &
  - C. ->
  - D. -->>
  
7. Which one of the following is not a possible state for a pointer.
  - A. hold the address of the specific object
  - B. point one past the end of an object

**Object Oriented Programming**

---

- C. zero
  - D. point to a type
- 
- 8. A pointer can be initialized with
    - A. Null
    - B. Zero
    - C. Address of an object of the same type
    - D. All of the above
- 
- 9. Referencing a value through a pointer is called
    - A. Direct calling
    - B. Indirection
    - C. Pointer referencing
    - D. All of the above
- 
- 10. Choose the right option.  
Int \* x, y;
    - A. x is a pointer to an int, y is an int
    - B. y is a pointer to a string, x is an int
    - C. both x and y are pointers to integer types
    - D. y is a pointer to a string
- 
- 11. A structure is a \_\_
    - A. Collection of variables (different types) represented by single name.
    - B. A structure is a user defined data type in C
    - C. Keyword 'struct' is used to define structure in C
    - D. All of above
- 
- 12. Structure members are accessed using\_\_\_\_
    - A. :
    - B. .
    - C. >
    - D. <
- 
- 13. Which keyword is used to define structure in C\_\_
    - A. structure
    - B. struct
    - C. structC
    - D. None of above
- 
- 14. Passing structure in function using\_\_\_\_
    - A. Function by value
    - B. Function by reference

- C. Both function by reference and function by value  
D. None of above
15. Which of the following operation is illegal in structures?  
A. Pointer to a variable of the same structure  
B. Dynamic allocation of memory for structure  
C. Typecasting of structure  
D. All of the mentioned

### **Answers for Self Assessment**

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

### **Review Questions**

1. Explain the uses of pointers.
2. Discuss limitations of C structures in detail.
3. Write a program that demonstrates working of C structure.
4. What is function? Discuss advantages of using functions in OOP.
5. What do you mean by function prototyping?



### **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://msdn.microsoft.com/en-us/library/wcz57btd(v=vs.80).aspx)
- <https://www.codecademy.com/learn/learn-c-plus-plus>



## Unit 06: Classes and Objects

### CONTENTS

Objectives

Introduction

6.1 Specifying a Class

6.2 Defining Member Functions

6.3 Creating Class Objects

6.4 Access Specifiers

6.5 The Public Access Specifier

6.6 The Private Access Specifier

6.7 The Protected Access Specifier

Summary

Keywords

Self Assessment

Answers for Self Assessment

Review Questions

Further Readings

### Objectives

After studying this unit, you will be able to:

- Recognize how to specify a class
- Define the member functions
- Explain the creation of class objects
- Access the class members
- Discuss the access specifiers

### Introduction

Classes and objects are at the core of object-oriented programming. Writing programs in C++ essentially means writing classes and creating objects from them. In this unit you will learn to work with the same.

It is important to note the subtle differences between a class and an object, here. A class is a template that specifies different aspects of the object it models. It has no physical existence. It occupies no memory. It only defines various members (data and/or methods) that constitute the class.

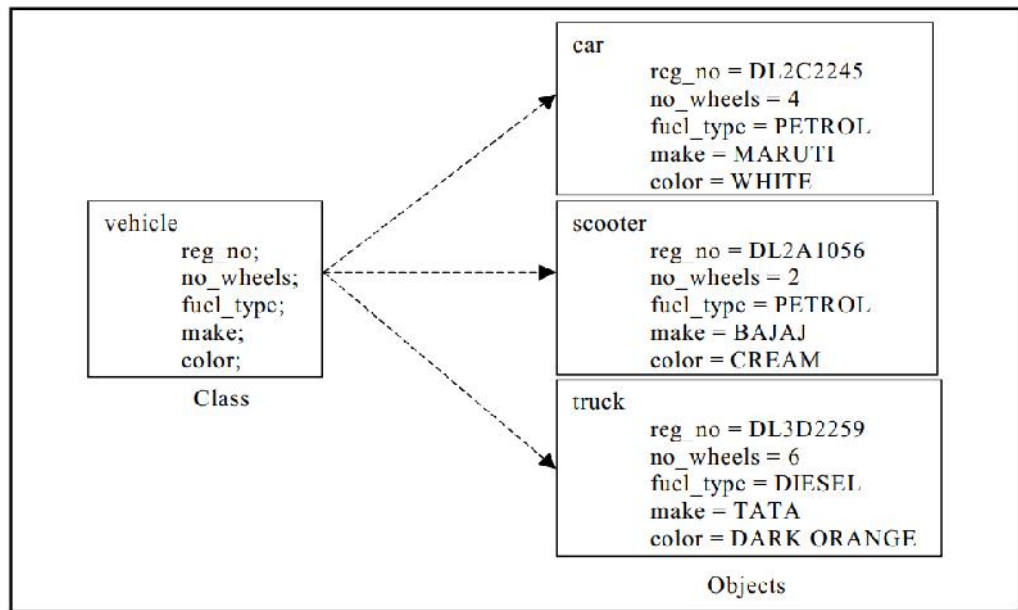
An object, on the other hand, is an instance of a class. It has physical existence and hence occupies memory. You can create as many objects from a class once you have defined a class.

You can think of a class as a data type; and it behaves like one. Just as a data type like int, for example, does not have a physical existence and does not occupy any memory until a variable of that type is declared or created; a class also does not exist physically and occupies no memory until an object of that class is created.



#### Example

To understand the difference clearly, consider a class of vehicle and a few objects of this type as depicted below:



In this example vehicle is a class while car, scooter and truck are instances of the class vehicle and hence are objects of vehicle class. Each instance of the class vehicle – car, scooter and truck – are allocated individual memory spaces for the variables – reg\_no, no\_wheels, fuel\_type, make and, color – so that they all have their own copies of these variables.

## 6.1 Specifying a Class

Like structures a class is just another derived data-type. While structure is a group of elements of different data-type, class is a group of elements of different data-types and functions that operate on them. C++ structure can also have functions defined in it.

There is very little syntactical difference between structures and classes in C++ and, therefore, they can be used interchangeably with minor modifications. Since class is a specially introduced data-type in C++, most of the C++ programmers tend to use the structures for holding only data, and classes to hold both the data and functions.

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

1. Class declaration
2. Class function definitions

The class declaration describes the type and scope of its members. The class function definitions describe how the class functions are implemented. The general form of a class declaration is:

```
class <class_name>
```

```
{
```

```
Private:
```

```
Variables declaration;
```

```
Function declarations;
```

```
Public:
```

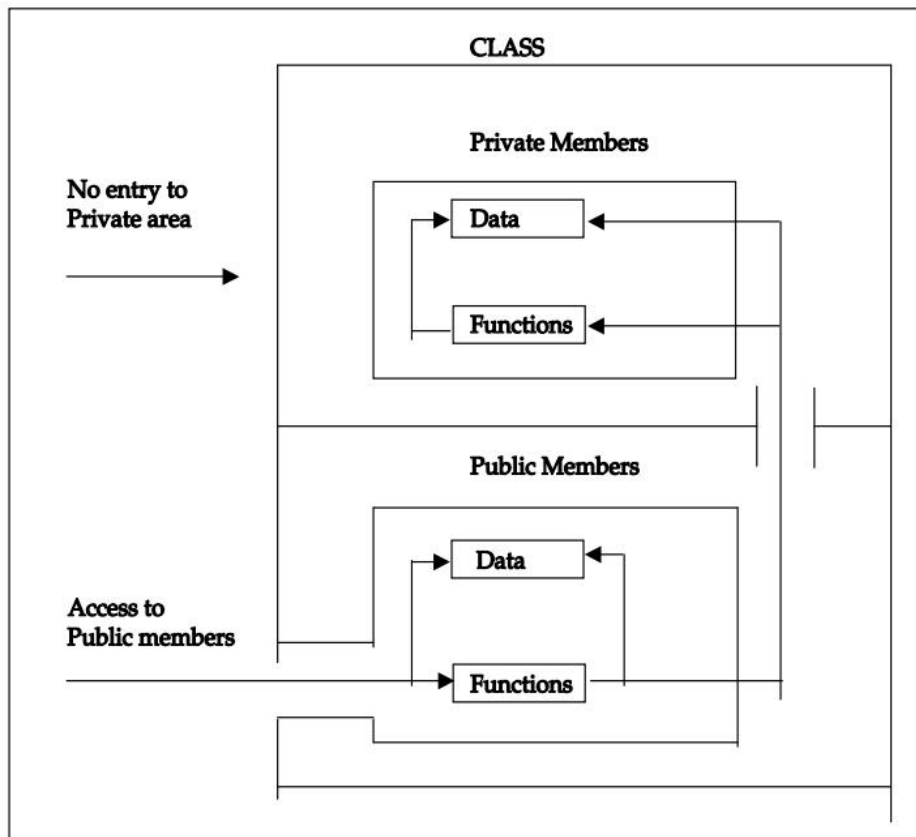
```
Variable declaration;
```

```
function declarations;
```

};

The class declaration is similar to a struct declaration. The keyword `class` specifies that what follows is an abstract data of type `class_name`. The body of a class is enclosed within braces and terminated by a semicolon. The class body contains the declaration of variables and functions.

These functions and variables are collectively called members. They are usually grouped under two sections, namely, private and public to denote which of the members are private and which of them are public. The keywords `private` and `public` are known as visibility labels. The members that have been declared as private can be accessed only from within the class. On the other hand, public members can be accessed from outside the class also. The data hiding (using private declaration) is the key feature of object-oriented programming. The use of the keyword `private` is optional. By default, the members of a class are private. If both the labels are missing, then, by default, all the members are private. Such a class is completely hidden from the outside world and does not serve any purpose.



The variables declared inside a class are known as data members and the functions are known as member functions. Only the member functions can have access to the private data members and private functions. However, the public members (both functions and data) can be accessed from outside the class. The binding of data and functions together into a single class-type variable is referred to as encapsulation. The access to private and public members of a class is well explained diagrammatically in the Figure.

Let us consider the following declaration of a class for student:

```
class student
{
private:
int rollno;
char name [20];
public:
void getdata(void);
```

```
void disp(void);
};
```

The name of the class is student. With the help of this new type identifier, we can declare instances of class student. The data members of this class are int rollno and char name [20]. The two function members are getdata() and disp(). Only these functions can access the data members. Therefore, they provide the only access to the data members from outside the class. The data members are usually declared as private and member functions as public. The member functions are only declared in the class. They shall be defined later.

A class declaration for a machine may be as follows:

```
class machine
{
int totparts, partno;
char partname [20];
public:
void getparts (void);
void disp(part_no);
};
```

Having defined the class, we need to create object of this class. In general, a class is a user defined data type, while an object is an instance of a class. A class provides a template, which defines the member functions and variable that are required for objects of a class type. A class must be defined prior to the class declaration.

The general syntax for defining the object of a class is:

```
class <class_name>
{
private:
//data
// functions
public:
//functions
};
<class_name>object1, object2,...objectN;
```

where object1, object2 and objectN are the instances of the class <class\_name>.

A class definition is very similar to a structure definition except the class definition defines the variables and functions.

## **6.2 Defining Member Functions**

We have learnt to declare member functions. Let us see how member functions of a class can be defined within a class or outside a class.

A member function is defined outside the class using the :: (double colon symbol) scope resolution operator. The general syntax of the member function of a class outside its scope is:

```
<return_type><class_name>::<member_function>(arg1, arg2,...argN)
```

The type of member function arguments must exactly match with the types declared in the class definition of the <class\_name>. The Scope resolution operator (::) is used along with the classname in the header of the function definition. It identifies the function as a member of a particular class. Without this scope operator the function definition would create an ordinary function, subject to the usual function rules of access and scope.

The following program segment shows how a member function is declared outside the class declaration.

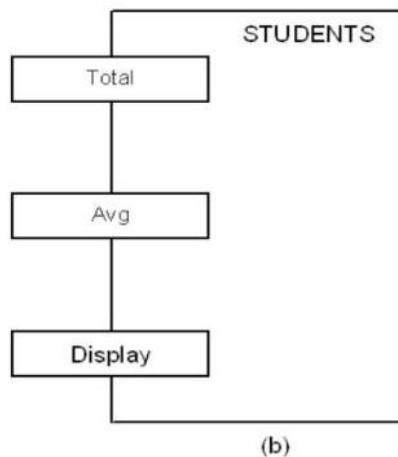
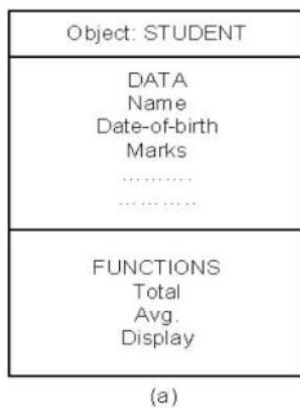
```
class sample
{
private:
int x;
int y;
public:
int sum (); // member function declaration
};
int sample:: sum () //member function definition
{
return (x+y);
}
```



**Notes:** The use of the scope operator double colon (::) is important for defining the member functions outside the class declaration

### 6.3 Creating Class 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. Objects can interact without having to know details of each other data or code.

#### Objects as Function Arguments

Like any other data type argument, objects can also be passed to a function. As you know arguments are passed to a function in two ways:

1. by value
2. by reference

Objects can also be passed as arguments to the function in these two ways. In the first method, a copy of the object is passed to the function. Any modification made to the object in the function does not affect the object used to call the function. The following Program illustrates the

calling of functions by value. The program declares a class integer representing a integer variable x. Only the values of the objects are passed to the function written to swap them.

```
#include<iostream.h>
#include<conio.h>
class integer
{
int x;
public:
void getdata()
{
cout<< "Enter a value for x";
cin>> x;
}
void disp()
{
cout<< x;
}
void swap(integer a1 , integer a2)
{
int temp;
temp = a2.x;
a2.x = a1.x;
a1.x = temp;
}
};
main()
{
integer int1, int2;
int1.getdata();
int2.getdata();
cout<<"\n the value of x belonging to object int1 is ";
int1.disp();
cout<<"\n the value of x belonging to object int2 is ";
int2.disp();
integer int3;
int3.swap(int1, int2); //int3 is just used to invoke the function
cout<<" \nafter swapping ";
cout<<"\n the value of x belonging to object int1 is ";
int1.disp();
cout<<"\n the value of x belonging to object int2 is ";
int2.disp();
}
```

```
cout<< "\n";
getche();
}
```

You should get the following output.

Enter a value for x 15

Enter a value for x 50

the value of x belonging to object int is 15

the value of x belonging to object int2 is 50

after swapping

the value of x belonging to object int is 15

the value of x belonging to object int2 is 50

In the second method, the address of the object is passed to the function instead of a copy of the object. The called function directly makes changes on the actual object used in the call. As against the first method any manipulations made on the object inside the function will occur in the actual object.

### Accessing a Member of Class

Member data items of a class can be static. Static data members are data objects that are common to all objects of a class. They exist only once in all objects of this class. The static members are used when the information is to be shared. They can be public or private data. The main advantage of using a static member is to declare the global data which should be updated while the program lives in memory.

When a static member is declared private, the non-member functions cannot access these members. But a public static member can be accessed by any member of the class. The static data member should be created and initialized before the main function control block begins.

```
class account
{
private:
int acc_no;
static int balance; //static data declaration
public:
void disp(int acc_no);
void getinfo();
};
```

The static variable balance is initialized outside main() as follows:

```
int account::balance = 0; //static data definition
```

Consider the following Program which demonstrates the use of static data member count. The variable count is declared static in the class but initialized to 0 outside the class.

```
#include <iostream.h>
#include <conio.h>
class counter
{
private:
static int count;
public:
```

```

void disp();
};
int counter::count = 0;
void counter::disp()
{
 count++;
 cout<< "The present value of count is " << count << "\n";
}
main()
{
 counter cnt1 ;
 for(int i=0; i<5; i++)
 cnt1.disp();
 getch();
}

```

You should get the following output from this program.

```

The present value of count is 1
The present value of count is 2
The present value of count is 3
The present value of count is 4
The present value of count is 5

```

## **6.4 Access Specifiers**

Members of a class can access other members (properties and methods) of the same class. Functions, operators and other classes (corresponding objects) outside the class description of a particular class can also access members of that class. An access specifier decides whether or not a function or operator or class, outside the class description can access the members it controls inside its class description. The members an access specifier controls are the members typeDunder it in the class description (until the next specifier). We will use functions and classes in the illustrations of accesses to class members. We will not use operators for the illustrations.

We will be using the phrase, external function. This refers to a function or class method that is not a member of the class description in question. When we say an external function can access class member, we mean the external function can use the name (identifier of property or name of method) of the member as its argument or as an identifier inside its definition.

## **6.5 The Public Access Specifier**

With the public access specifier, an external function can access the public members of the class.

The following code illustrates this (read the explanation below):

```

#include <iostream>
using namespace std;
class Calculator
{
public:
 int num1;

```



```

int num2; Notes
int add()
{
int sum = num1 + num2;
return sum;
}
};
int myFn(int par)
{
return par;
}
int main()
{
Calculator obj;
obj.num1 = 2;
obj.num2 = 3;
int result = obj.add();
cout<< result; cout<< "\n";
int myVar = myFn(obj.num1);
cout<<myVar;
return 0;
}

```

There are two functions in the code: `myFn()` and `main()`. The first line in the `main` function instantiates a class object called, `obj`. In `main`, lines 2 and 3 use the properties of the class as identifiers. Because the class members are public, the `main()` function can access the members of the class. Line 4 of the `main` function also demonstrates this. In line 6 of the `main` function, the function, `myFn()` uses the property `num1` of the class as its argument. It could do so because the member, `num1` is public in the class.

## 6.6 The Private Access Specifier

With the private access specifier an external function cannot access the private members of the class. With the private specifier only a member of a class can access the private member of the class. The following code shows how only a member of a class can access a private member of the class (read the explanation below):

```

#include <iostream>
using namespace std;
class Calculator
{
private:
int num1;
int num2;
public:
int add()
{

```

```

num1 = 2;
num2 = 3;
int sum = num1 + num2;
return sum;
}
};
int main()
{
 Calculator obj;
 int result = obj.add();
 cout<< result;
 return 0;
}

```

The class has two private members (properties) and one public member (method). In the class description, the add() method uses the names of the private members as identifiers. So the add() method, a member of the class has accessed the private members of the class. The main function definition (second line) has been able to access the add() method of the class because the add() method is public (it has a public access specifier).

The following code will not compile because the main function tries to access (use as identifier) a private member of the class:

```

#include <iostream>
using namespace std;
class Calculator
{
private:
 int num1;
 int num2;
public:
 int add()
 {
 num1;
 num2 = 3;
 int sum = num1 + num2;
 return sum;
 }
};
int main()
{
 Calculator obj;
 obj.num1 = 2;
 int result = obj.add();
 cout<< result;
 return 0;
}

```

```
}
```

The second line in the main function is wrong because at that line, main tries to access (use as identifier) the private member, num1.

## 6.7 The Protected Access Specifier

If a member of a class is public, it can be accessed by an external function including a derived class. If a member of a class is private, it cannot be accessed by an external function; even a derived class cannot access it.

The question is, should a derived class not really be able to access a private member of its base class (since the derived class and base class are related)? Well, to solve this problem you have another access specifier called, protected. If a member of a class is protected, it can be accessed by a derived class, but it cannot be accessed by an external function. It can also be accessed by members within the class. The following code illustrates how a derived class can access a protected member of a base class:

```
#include <iostream>
using namespace std;
class Calculator
{
protected:
int num1;
int num2;
};
class ChildCalculator: public Calculator
{
public:
int add()
{
num1 = 2;
num2 = 3;
int sum = num1 + num2;
return sum;
}
};
int main()
{
ChildCalculator myChildObj;
int result = myChildObj.add();
cout << result;
return 0;
}
```

The base class has just two properties and no method; these properties are protected. The derived class has one method and no property. Inside the derived class, the protected properties of the base class are used as identifiers. Generally, when a derived class is using a member of a base class, it is a method of the derived class that is using the member, as in this example. The

above code is OK. The following code will not compile, because line 2 in the main() function tries to access a protected member of the base class:

```
#include <iostream>
using namespace std;
class Calculator
{
protected:
int num1;
int num2;
};
class ChildCalculator: public Calculator
{
public:
int add()
{
num1;
num2 = 3;
int sum = num1 + num2;
return sum;
}
};
int main()
{
Calculator obj;
obj.num1 = 2;
ChildCalculator myChildObj;
int result = myChildObj.add();
cout<<result;
return 0;
}
```

An external function cannot access a protected member of a class (base class); however, a derived class method can access a protected member of the base class.



**Notes:** A member of a class can access any member of the same class independent of the whether the member is public, protected or private.

You should now know the role of the access specifiers: public, protected and private as applied to classes. In one of the following parts of the series, we shall see the role of the access specifiers in the declaratory of a derived class.

A public member of a class is accessible by external functions and a derived class. A private member of a class is accessible only by other members of the class; it is not accessible by external functions and it is not accessible by a derived class. A protected member of a class is accessible by a derived class (and other members of the class); it is not accessible by an external function.

## Summary

- A class represents a group of similar objects. A class in C++ binds data and associated functions together.
- It makes a data type using which objects of this type can be created. Classes can represent the real-world object which have characteristics and associated behavior.
- While declaring a class, four attributes are declared: data members, member functions, program access levels (private, public, and protected,) and class tag name.
- While defining a class its member functions can be either defined within the class definition or outside the class definition. The public member of a class can be accessed outside the class directly by using object of this class type.
- Private and protected members can be accessed within the class by the member function only. The member function of a class is also called a method. The qualified name of a class member is a class name: class member name.
- A global object can be declared only from a global class whereas a local object can be declared from a global as well as a local class.
- The object is created separately to store their data members. They can be passed to as well as returned from functions. The ordinary member's functions can access both static as well as ordinary member of a class.

## Keywords

**Class:** Represents the real-world objects which have characteristics and associated behavior.

**Global Class:** A class whose definition occurs outside the bodies of all functions in a program. Objects of this class type can be declared from anywhere in the program.

**Local Class:** A class whose definition occurs inside a function body. Objects of this class type can be declared only within the function that defines this class type.

**Private Members:** Class members that are hidden from the outside world.

**Public Members:** Class members (data members and member functions) that can be used by any function.

## Self Assessment

- Which of the following is correct about class?
  - Class can have member functions while structure cannot.
  - Class data members are public by default while that of structure are private.
  - Pointer to structure or classes cannot be declared.
  - Class data members are private by default while that of structure are public by default.
- Which of the following is not an access specifier?
  - Public
  - Char
  - Private
  - Protected
- 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
4. Which among the following best describes member functions?
- A. Functions which are defined within the class
  - B. Functions belonging a class
  - C. Functions in public access of a class
  - D. Functions which are private to class
5. How can static member function can be accessed directly in main() function?
- A. Dot operator
  - B. Colon
  - C. Scope resolution operator
  - D. Arrow operator
6. Which keyword is used to make a nonmember function as friend function of a class?
- A. Friendly
  - B. New
  - C. Friend
  - D. Connect
7. Member functions \_\_\_\_\_
- A. Must be defined inside class body
  - B. Can be defined inside class body or outside
  - C. Must be defined outside the class body
  - D. Can be defined in another 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: functionName(parameters)
  - C. private(functionName(parameters))
  - D. private functionName(parameters)

### **Answers for SelfAssessment**

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

## **Review Questions**

1. Define class. What is the difference between structures and classes in C++?
2. How can you pass an object of a C++ class to/from a C function?
3. Can a method directly access the non-public members of another instance of its class?
4. What's the difference between the keywords struct and class?
5. Write a program to add two numbers defining a member `getdata ()` and `display ()` inside a class named `sum` and displaying the result.
6. How does C++ help with the tradeoff of safety vs. usability?
7. "The main advantage of using a static member is to declare the global data which should be updated while the program lives in memory". Justify your statement.
8. Write a program to print the score board of a cricket match in real time. The display should contain the batsman's name, runs scored, indication if out, mode by which out, bowler's score (overs played, maiden overs, runs given, wickets taken). As and when a ball is thrown, the score should be updated.  
(print: Use separate arrays to store batsmen's and bowler's information)
9. If a member of a class is public, it can be accessed by an external function including a derived class. Explain with an example.
10. Write a program in which a class has one private member (properties) and two public member (method).



## **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://msdn.microsoft.com/en-us/library/wcz57btd(v=vs.80).aspx)
- <https://www.codecademy.com/learn/learn-c-plus-plus>



## Unit 07: More on Classes and Objects

### CONTENTS

Objectives

Introduction

7.1 C++ Class Member Function

7.2 Nested Member Function

7.3 Private Member Functions

7.4 Arrays within the Class

7.5 Memory Allocation of Objects

Summary

Keywords

Self Assessment

Answers for Self Assessment

Review Questions

Further Readings

### Objectives

After studying this unit, you will be able to:

- Understand function definition
- Analyze member functions
- Recognize private member functions
- Explain memory allocation of objects
- Analyze arrays with in class

### Introduction

A class in C++ is the building block that leads to Object-Oriented programming. It is a user-defined data type, which holds its own data members and member functions, which can be accessed and used by creating an instance of that class. A C++ class is like a blueprint for an object.

For Example: Consider the Class of Cars. There may be many cars with different names and brand but all of them will share some common properties like all of them will have 4 wheels, Speed Limit, Mileage range etc. So here, Car is the class and wheels, speed limits, mileage are their properties.

### 7.1 C++ Class Member Function

A 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.

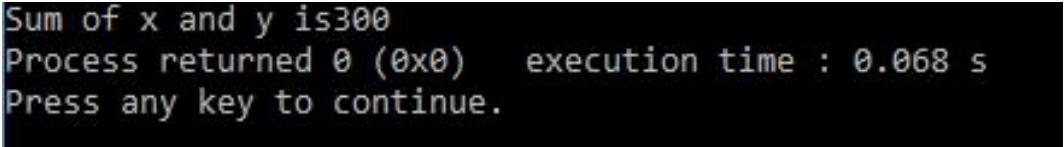
#### *Syntax*

```
return_type class_name :: member_function
```

**Lab Exercise:**

```
#include<iostream>
using namespace std;
class find_sum{
public:
 int x,y;
 int sum();
};
int find_sum ::sum(){
 return x+y;
}
int main(){
 find_sum sum1;
 sum1.x=100;
 sum1.y=200;
 cout<<"Sum of x and y is"<<sum1.sum();
 return 0;
}
```

Output



```
Sum of x and y is300
Process returned 0 (0x0) execution time : 0.068 s
Press any key to continue.
```

## 7.2 Nested Member Function

- A member function may call another member function within itself. This is called nesting of member functions.
- A member function can access not only the public functions but also the private functions of the class it belongs to.
- Calling a member function with in another member function.
- Scope resolution operators ( :: ) used to defining a function outside a class.

**Lab Exercise:**

```
#include<iostream>
using namespace std;
class area{
private:
 int l,w;
public:
 void getdata();
```

```

 void display_area();
};
void area::getdata()
{
 cout<<"Enter length and width";
 cin>>l>>w;
}
void area::display_area()
{
 getdata();
 cout<<"area is"<<l*w;
}
int main(){
 area a1;
 a1.display_area();
}

```

Output

```

Enter length and width15
3
area is45
Process returned 0 (0x0) execution time : 2.984 s
Press any key to continue.

```

### 7.3 Private Member Functions

A function declared inside the class's private section is known as "private member function". A private member function is accessible through the only public 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.



#### Lab Exercise:

```

#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 ;
}
void Box::setWidth(double wid) {
 width = wid;
}
int main() {
 Box box;
 box.length = 5.0;
 cout<< "Length of box : " <<box.length<<endl;
 box.setWidth(5.0);
 cout<< "Width of box : " <<box.getWidth() <<endl;
 return 0;
}

```

Output

```

Length of box : 5
Width of box : 5

Process returned 0 (0x0) execution time : 0.078 s
Press any key to continue.

```

## 7.4 Arrays within the Class

An array is a collection of elements of the same type placed in contiguous memory .

*Syntax*

```
typearrayName [arraySize];
```



### LabExercise

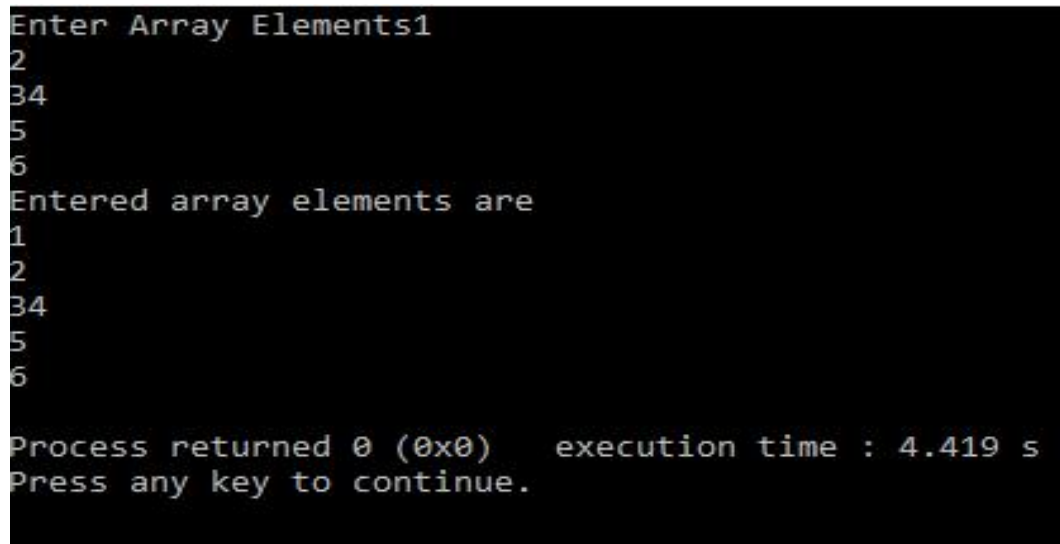
```

#include<iostream>
using namespace std;
int main(){
 int arr[5];
 cout<<"Enter Array Elements";
 for(int i=0;i<=4;i++)
 {
 cin>>arr[i];
 }
 cout<<"Entered array elements are"<<endl;
 for(int i=0;i<=4;i++)
 {

```

```
cout<<arr[i]<<endl;
}
return 0;
}
```

Output



```
Enter Array Elements1
2
34
5
6
Entered array elements are
1
2
34
5
6
Process returned 0 (0x0) execution time : 4.419 s
Press any key to continue.
```

### Array with in class

Arrays can be declared as the members of a class. The arrays can be declared as private, public or protected members of the class.



#### Lab Exercise:

```
#include<iostream>
using namespace std;
class student {
 int marks[5];
public:
 void getdata ();
 void showdata();
};
void student::getdata(){
cout<<"Enter marks";
for(int i=0;i<=4;i++){
cin>>marks[i];
}
}
void student::showdata(){
cout<<"Marks are";
for(int i=0;i<=4;i++){
```

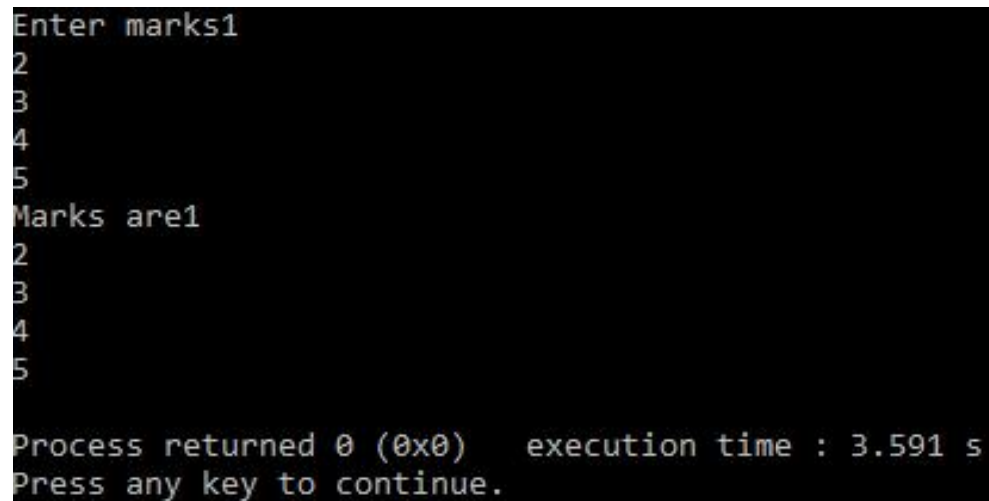
```

cout<<marks[i]<<endl;
}
}

int main(){
 student stu;
 stu.getdata();
 stu.showdata();
}

```

Output



```

Enter marks1
2
3
4
5
Marks are1
2
3
4
5

Process returned 0 (0x0) execution time : 3.591 s
Press any key to continue.

```

## 7.5 Memory Allocation of Objects

When a class is defined, only the specification for the object is defined; no memory or storage is allocated.

An object is a data structure that encapsulates data and functions in a single construct. In an object-oriented framework, they are the basic run-time entities.

### *Format to declare objects*

Class\_name object\_list;

Where

Class\_name – Is name of class defined

Object\_list – comma-seprated list of objects



**Example** – To create an object (obj) for class 'student'

Student obj;

Class\_name object\_list;



**Example** - To create three objects (s1, s2 and s3) for class 'student'

Student s1, s2, s3;

**Significance of Class and Object in C++**

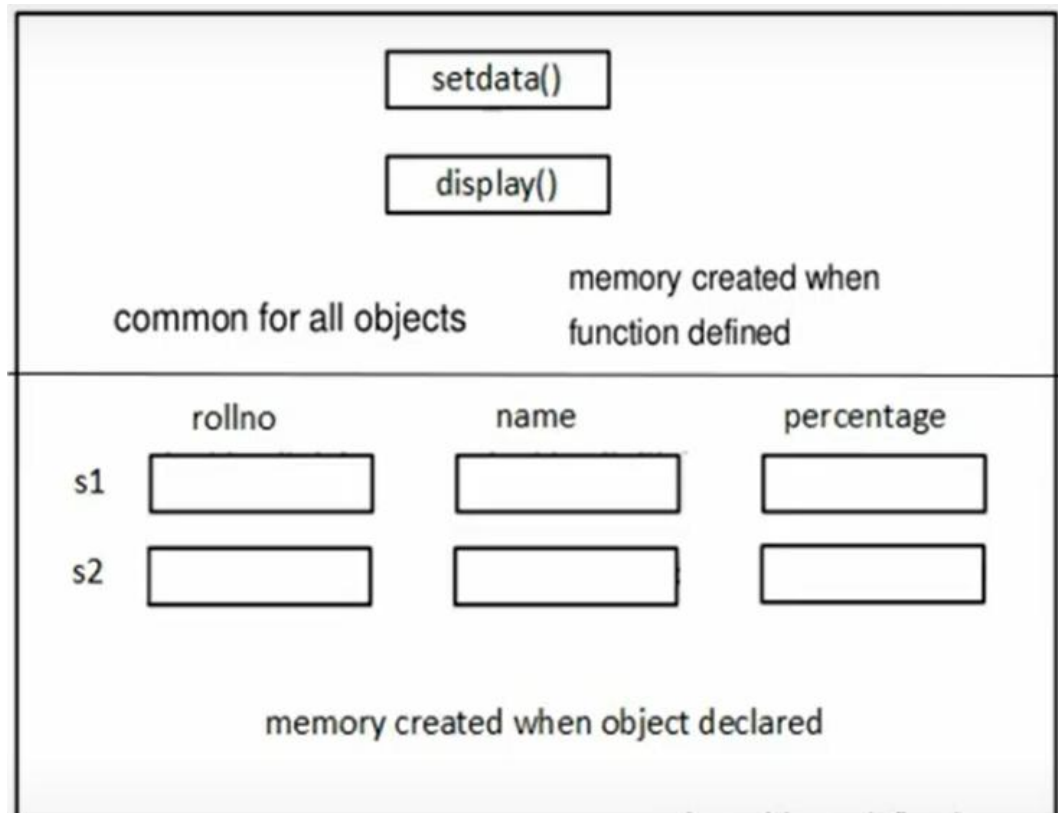
- **Data Hiding:** A class prevents the access of the data from the outside world using access specifiers. It can set permissions to restrict the access of the data.
- **Code Reusability:** You can reduce code redundancy by using reusable code with the help of inheritance. Other classes can inherit similar functionalities and properties, which makes the code clean.
- **Data Binding:** The data elements and their associated functionalities are bound under one hood, providing more security to the data.
- **Flexibility:** You can use a class in many forms using the concept of polymorphism. This makes a program flexible and increases its extensibility.

**Memory allocation of objects**

- Memory allocation of data members is performed each time an object of the class is created, not when data members are declared inside the class.
- Every object of a class has an individual copy of data members.
- Since the member function defined inside the class remains the same for all objects, memory allocation of a member function is performed at the time of defining the class.

```
#include <iostream>
using namespace std;
class student{
int roll_no;
char name[20];
int percentage;
public:
void setdata(){
cout<<"Enter Roll Number" ;
cin>>roll_no;
cout<<"Enter Name ";
cin>>name;
cout<<"Enter Percentage";
cin>>percentage;
}
void display(){
cout<<"\n Roll Number " <<roll_no<<" \tName " << name <<" \t Percentage " << percentage
<<endl;
}
};
int main(){
student s1,s2;
s1.setdata();
s2.setdata();
s1.display();
```

```
s2.display();
return 0;
}
```



```
Enter Roll Number1
Enter Name ABC
Enter Percentage90
Enter Roll Number2
Enter Name PQR
Enter Percentage89

Roll Number 1 Name ABC Percentage 90

Roll Number 2 Name PQR Percentage 89

Process returned 0 (0x0) execution time : 182.106 s
Press any key to continue.
```

### Memory Allocation

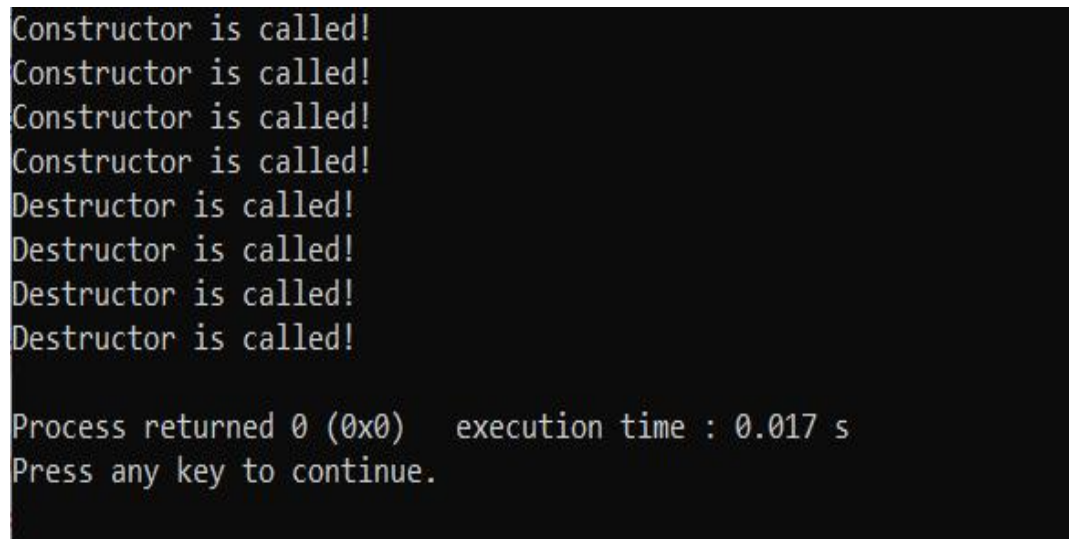
- Many times, you are not aware in advance how much memory you will need to store particular information in a defined variable and the size of required memory can be determined at run time.



- You can allocate memory at run time within the heap for the variable of a given type using a special operator in C++ which returns the address of the space allocated. This operator is called new operator.
- If you are not in need of dynamically allocated memory anymore, you can use delete operator, which de-allocates memory previously allocated by new operator.

```
#include <iostream>
using namespace std;
class sample
{
 public:
 sample() {
 cout<< "Constructor is called!" <<endl;
 }
 ~sample() {
 cout<< "Destructor is called!" <<endl;
 }
};
int main()
{
 sample* obj = new sample[4];
 delete [] obj; // Delete array of objects
 return 0;
}
```

Output



```
Constructor is called!
Constructor is called!
Constructor is called!
Constructor is called!
Destructor is called!
Destructor is called!
Destructor is called!
Destructor is called!

Process returned 0 (0x0) execution time : 0.017 s
Press any key to continue.
```

```
Constructor is called!
Constructor is called!
Constructor is called!
Constructor is called!

Process returned 0 (0x0) execution time : 0.017 s
Press any key to continue.
```

### Summary

- A class represents a group of similar objects. A class in C++ binds data and associated functions together.
- It makes a data type using which objects of this type can be created. Classes can represent the real-world object which have characteristics and associated behavior.
- While declaring a class, four attributes are declared: data members, member functions, program access levels (private, public, and protected,) and class tag name.

### Keywords

**Class:** Represents the real-world objects which have characteristics and associated behavior.

**Global Class:** A class whose definition occurs outside the bodies of all functions in a program. Objects of this class type can be declared from anywhere in the program.

**Local Class:** A class whose definition occurs inside a function body. Objects of this class type can be declared only within the function that defines this class type.

**Private Members:** Class members that are hidden from the outside world.

**Public Members:** Class members (data members and member functions) that can be used by any function.

### SelfAssessment

1. 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
2. How many public members are allowed in a class?
  - A. 1
  - B. Maximum 7
  - C. Exactly 3
  - D. As many as required
3. Which operator is used to define a member function outside the class?

- A. \*
  - B. ( )
  - C. +
  - D. ::
4. 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
5. 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
6. 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
7. The combination of abstraction of the data and code is viewed in\_\_\_\_\_.
- A. Inheritance
  - B. Class
  - C. Object
  - D. Interfaces
8. 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
9. Which syntax among the following shows that a member is private in a class?
- A. private ::Name(parameters)
  - B. private: functionName(parameters)
  - C. private(functionName(parameters))
  - D. private functionName(parameters)
10. Where does the object is created?
- A. Class
  - B. Constructor
  - C. Destructors
  - D. Attributes

11. Which is used to define the member of a class externally?
- A. ;
  - B. ::
  - C. #
  - D. !!\$
12. Choose the correct class declaration from the followings?
- A. Class A { int a; };
  - B. Class B { }
  - C. Public class A { }
  - D. Object A { int y; };
13. Choose the right observation from the following?
- A. Base class pointer object cannot point to a derived class object
  - B. A derived class pointer object cannot point to a base class object
  - C. A derived class cannot have pointer objects
  - D. A base class cannot have pointer objects
14. A static member function can be called using the ..... instead of its objects.
- A. variable name
  - B. function name
  - C. Class name
  - D. object name
15. Constructors are used to \_\_\_\_\_
- A. initialize the objects
  - B. construct the data members
  - C. both initialize the objects & construct the data members
  - D. delete the objects

### **Answers for SelfAssessment**

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

### **Review Questions**

1. What do you mean by classes and objects in object-oriented programming?
2. Write a program that demonstrate working of member functions and classes.
3. Explain memory allocation of objects with the help of programing example.
4. Differentiate between public, private and protected access specifiers.
5. What is the significance of member functions, explain using an example.



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

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



## Unit 08: Handling Functions

### CONTENTS

Objectives

Introduction

8.1 C++ Functions

8.2 Function Activities

8.3 Default Arguments

Summary

Keywords

Self Assessment

Answers for Self Assessment

Review Questions

Further Readings

### Objectives

After studying this unit, you will be able to:

- Understand functions in OOP
- Analyze execution of code using call by reference and call by value mechanism
- Understand default arguments
- Construct C++ programs using function and default arguments

### Introduction

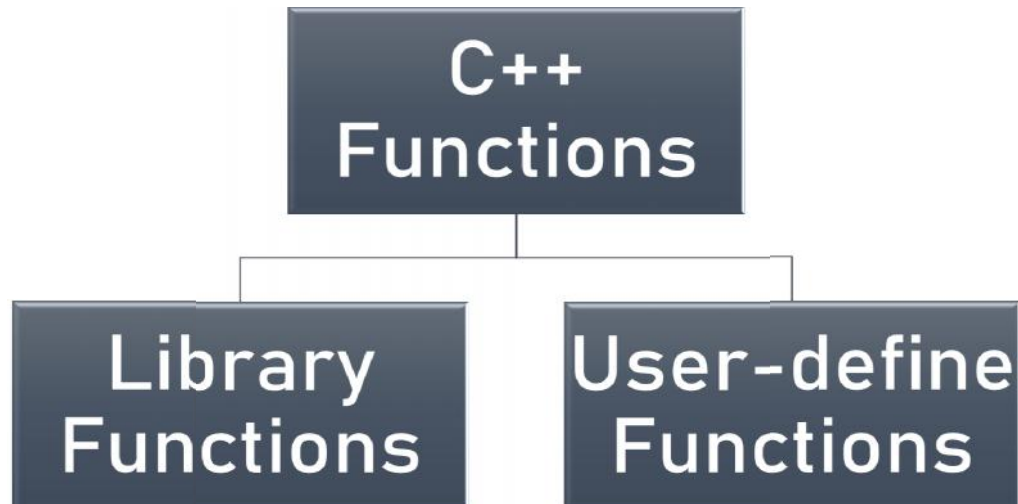
A function is a set of statements that are put together to perform a specific task. It can be statements performing some repeated tasks or statements performing some specialty tasks like printing etc.

Code can be made simpler by using functions to divide it into smaller, more manageable chunks. Using functions also save us from repeatedly writing the same code, which is another another benefit. Instead of repeatedly writing the same set of statements, we only need to write one function and then call it as and when necessary.

### 8.1 C++ Functions

The function in C++ language is also known as procedure or subroutine in other programming languages. 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. Debugging the code would be easier if you use functions, as errors are easy to be traced. The use of functions reduce the size of code. A duplicate set of statements is replaced by function calls. Improve reusability of code, the same function can be used in any program. Use of function improves readability of code.

### **C++ Functions Types**

**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 its require.

**8.2 Function Activities**

There are the following activities a function has.

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

**Function Declaration**

A function declaration is 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.

**Function Declaration**

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

**Function Definition**

- 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.

**Function Definition**

```
int sum(int a,int b)
{
```



```
int c;
c=a+b;
return c;
}
```

### 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 Types

- Call by value
- Call by reference

### 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.



#### Lab Exercise:

```
#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.
```

**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.

**Lab Exercise:**

```
#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 a and b is"<<sum(&a,&b);
 return 0;
}
```

**Output**

```
Enter two numbers15
25
Sum of a and b is40
Process returned 0 (0x0) execution time : 2.884 s
Press any key to continue.
```

**Call by Value Vs. Call by reference**

| Parameters         | Call by value                                                         | Call by reference                                                                                                       |
|--------------------|-----------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------|
| Arguments          | Copy of variable is passed.                                           | Variable passed itself.                                                                                                 |
| Value modification | Original value not modified.                                          | The original value is modified.                                                                                         |
| Safety             | Actual arguments remain safe as they cannot be modified accidentally. | Actual arguments are not Safe. They can be accidentally modified, so you need to handle arguments operations carefully. |

### 8.3 Default Arguments

Like any other data type, an object may be used as a function argument.

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



#### Lab Exercise:

```
#include<iostream>
using namespace std;
class sample{
int x=10;
public:
 void display(sample s){
cout<<"Value of x accessed by passing object is "<<s.x;
 }
};
main(){
sample s1;
s1.display(s1);
}
```

Output

```
Value of x accessed by passing object is 10
Process returned 0 (0x0) execution time : 6.083 s
Press any key to continue.
```

### Summary

- Each function puts related code together. This makes it easier for programmers to understand code.
- Functions make programming easier by eliminating code repetition.
- Functions facilitate code reuse. You can call the same function to perform a task at different sections of the program or even outside the program.
- A function in C++ helps you group related code into one.
- Functions facilitate code reuse.
- Instead of writing similar code, again and again, you simply group it into a function. You can then call the function from anywhere within the code.
- Functions can be library or user-defined.
- Library functions are the functions built-in various C++ functions.
- To use library functions, you simply include its library of definition and call the function. You don't define the function.

- User-defined functions are the functions you define as a C++ programmer.
- A function declaration tells the compiler about the function name, return type, and parameter types.
- A function definition adds the body of the function.
- If a function takes parameters, their values should be passed during the function call.

### **Keywords**

**Function:** A function is defined as a group of statements or a block of code that carries out specific tasks.

**Call by value:** In this parameter passing method, values of actual parameters are copied to the function's formal parameters and the two types of parameters are stored in different memory locations. So, any changes made inside functions are not reflected in the actual parameters of the caller.

**Call by reference:** Both the actual and formal parameters refer to the same locations, so any changes made inside the function are actually reflected in the actual parameters of the caller.

### **Self Assessment**

- Function call type is
  - Call by value
  - Call by reference
  - Both call by value and class by reference
  - Neither call by value nor call by reference
- int x and int y is a\_\_\_\_\_  
add(intx,int y)
 

```
{
 int z;
 z=x+y;
 printf("result is= %d",z);
}
```

  - formal Parameter
  - actual Parameter
  - intermediate
  - both A and B
- Actual parameter is\_\_\_\_\_
  - parameters that appear in function calls.
  - parameters that appear in function definition.
  - local to the function definition
  - above all

4. Call by value and call by reference is part of \_\_\_\_
  - A. pointers
  - B. array
  - C. functions
  - D. loops
5. In program we can modify original value in
  - A. Call by value
  - B. Call by reference
  - C. Header file
  - D. above all
6. OOP C++ programming using \_\_\_\_
  - A. Library functions
  - B. User define functions
  - C. Both Library functions and user define functions
  - D. None of above
7. By default, how the values are passed in C++?
  - A. Call by pointer
  - B. Call by value
  - C. Call by reference
  - D. None of the above
8. What will happen when we use void in argument passing?
  - A. It will not return value to its caller
  - B. It will return value to its caller
  - C. All of above
  - D. none of the above
9. Passing objects to a function
  - A. Can be done in one way
  - B. Can be done in more than one way
  - C. Is not possible
  - D. Not possible in OOP C++
10. The object
  - A. Can be passed by reference
  - B. Can be passed by value
  - C. Can be passed by reference or value
  - D. Can't be passed to function
11. \_\_\_\_ symbol is used to pass the object by reference in OOP C++?

- A. @
  - B. #
  - C. \$
  - D. &
12. Pass by reference of an object to a function \_\_\_\_\_
- A. Affects the object in the called function
  - B. Affects the object and its properties
  - C. Affects the object in the caller function
  - D. None of above
13. The name of a function ends with
- A. double quotes
  - B. single quotes
  - C. parenthesis
  - D. #@
14. What would be the output of following code snippet?
- ```
#include<iostream>
using namespace std;
class sample{
int x=10;
public:
    void display(sample s){
        cout<<s.x;
    }
};
main(){
    sample s1;
    s1.display(s1);
}
```
- A. 10
 - B. Error
 - C. 0
 - D. None of above
15. What would be the output of the following code snippet?
- ```
#include<iostream>
using namespace std;
class sample{
```

```

int x=100;

public:

 void display(sample s){

 cout<<s.x;

 }

};

main(){

 sample s1;

 s1.display();

}

```

- A. Error
- B. 10
- C. 100
- D. None of above

### **Answers for Self Assessment**

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

### **Review Questions**

1. What do you mean by function? Explain using a program example.
2. Write a program that demonstrates the concept of call by value.
3. Write a program that demonstrates the concept of call by reference.
4. Why do we need functions in OOP? Comment.
5. Explain default arguments.



### **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://msdn.microsoft.com/en-us/library/wcz57btd(v=vs.80).aspx)  
<https://www.codecademy.com/learn/learn-c-plus-plus>

## **Unit 09: More on Functions**

### **CONTENTS**

Objectives

Introduction

9.1 Why We Need Functions in C++

9.2 The Main Function

9.3 Function Types

9.4 Function Activities

9.5 Inline function

9.6 C++ Objects as Function Arguments

9.7 C++ Friend Functions

9.8 Characteristics of Friend Function

9.9 C++ Static Data Members & Functions

9.10 Polymorphism in C++

9.11 Function Overloading

Summary

Keywords

Self Assessment

Answers for Self Assessment

Review Questions

Further Readings

### **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.



## 9.1 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.

## 9.2 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[])`

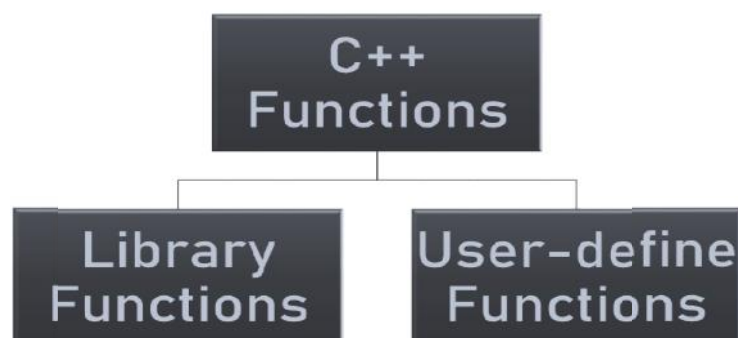
As is evident from the above forms, return type of the main method specifies the value returned to the operating system once the program finishes its execution. If your main method does not return any value to the operating system (caller of this method), then return type should be specified as void.



**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.

## 9.3 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 required.

## 9.4 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 means just writing the body of a function. We are just going to write the body of the function, what kind of work we can do with the function, we are just writing the logic. The body of the function consists of the statements, which are going to be performed a specific task like a sub routine or a sub program, it is also known as the procedure. We are writing 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.

- 3.1 Call by value
- 3.2 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.
```

## 9.5 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

```
Number is 50
Number is 150
Number is 200

Process returned 0 (0x0) execution time : 0.014 s
Press any key to continue.
```

## 9.6 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.

```

## 9.7 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);

}

```

## 9.8 Characteristics of Friend Function

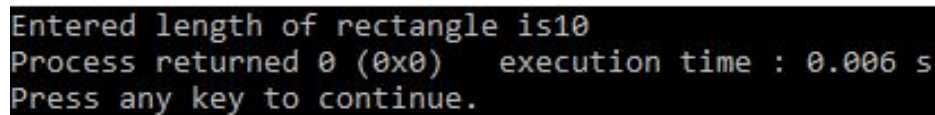
1. Friend function is not in the scope of the class in which it has been declared as friend.
2. It allows to generate more efficient code.
3. It cannot be called using the object of the class as it is not in the scope of the class.
4. It can be called similar to a normal function without the help of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

A screenshot of a terminal window with a black background and green text. The output shows the program's execution: it prints "Entered length of rectangle is10", followed by "Process returned 0 (0x0) execution time : 0.006 s", and finally "Press any key to continue.".

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

## 9.9 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;
}
```

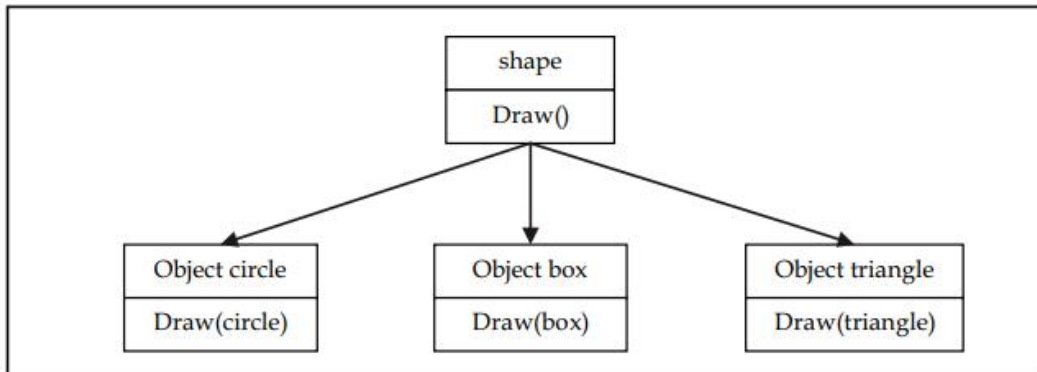
## Output

```
Value of a: 500
Process returned 0 (0x0) execution time : 0.142 s
Press any key to continue.
```

## 9.10 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 two numbers, the operation will generate a sum. If the operands are strings, then the operation will produce a third string by concatenation. 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.

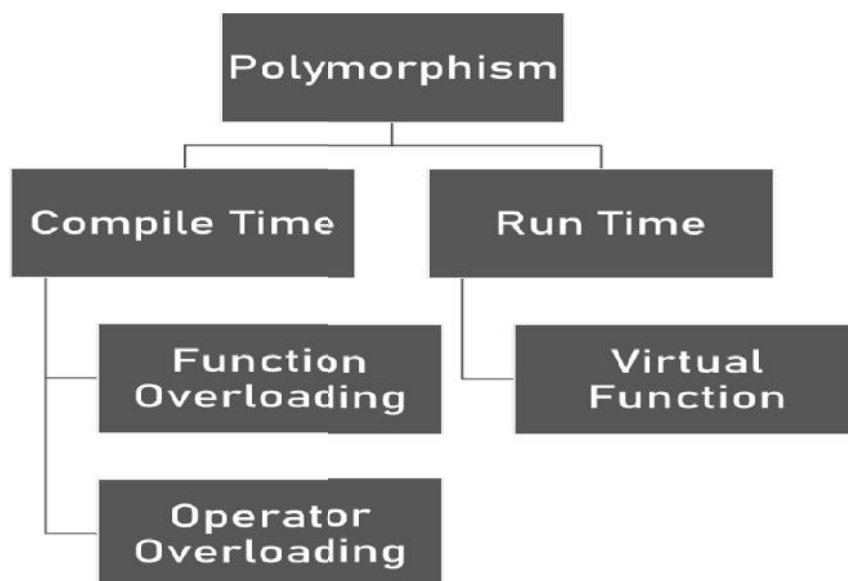
Polymorphism plays an important role in allowing objects having different internal structures to share the same external interface. This means that a general class of operations may be accessed in the same manner even though specific actions associated with each operation may differ. Polymorphism is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

### Types of C++ Polymorphism

1. Compile time Polymorphism
2. Runtime Polymorphism



### 9.11 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);
}

```



*Object Oriented Programming*

---

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 memberName dataType;
  - B. dataType static memberName;

Object Oriented Programming

---

- 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](https://en.wikipedia.org/wiki/Object-oriented_programming)  
[www.web-source.net](http://www.web-source.net)  
[www.webopedia.com](http://www.webopedia.com)

## Unit 10 : Static Members and Polymorphism

### CONTENTS

Objectives

Introduction

10.1 Static Data Member

10.2 Static Member Function

10.3 Polymorphism

10.4 Types of C++ Polymorphism

10.5 Function Overloading

10.6 Difference between Polymorphism and Inheritance

Summary

Keywords

Self Assessment

Answers for Self-Assessment

Review Questions

Further Readings

### Objectives

After studying this unit, you will be able to:

- Understand functions in OOP
- Analyze execution of code using call by reference and call by value mechanism
- Understand default arguments
- Construct C++ programs using function and default arguments

### 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.

### 10.1 Static Data Member

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 data member is similar to the static member function because the static data can only be accessed using the static data member or static member function.

A static data member has certain special characteristics.

1. It is initialized to zero when the first object of its class is created.
2. Only one copy of that member is created for the entire class and is shared by all the objects of that class, no matter how many objects are created.
3. It is visible only within the class, but its lifetime is the entire program.

**Syntax**

```

Class class_name{
private:
static data_member;
public:
static return_type function_name()
{
//body
}
};

```

**Example:**

```

#include <iostream>
using namespace std;
class Demo
{
public:
static int n;
};
int Demo :: n =100;
int main()
{
cout<<"\nValue of n is: "<<Demo::n;
return 0;
}

```

**Output**

```

Value of n is: 100
Process returned 0 (0x0) execution time : 2.471 s
Press any key to continue.

```

**10.2 Static Member Function**

Like a static member variable, we can also have static member functions. A member function that is declared static has the following properties:-



1. A static function can have access to only other static members (function or variable) declared in the same class.
2. A static member function can be called using the class name (instead of its object) as follows-

Class\_name::Function\_name();



**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 =50;
int main()
{
 Demo obj;
 obj.fun();
 return 0;
}
```

Output

```
Value of a: 50

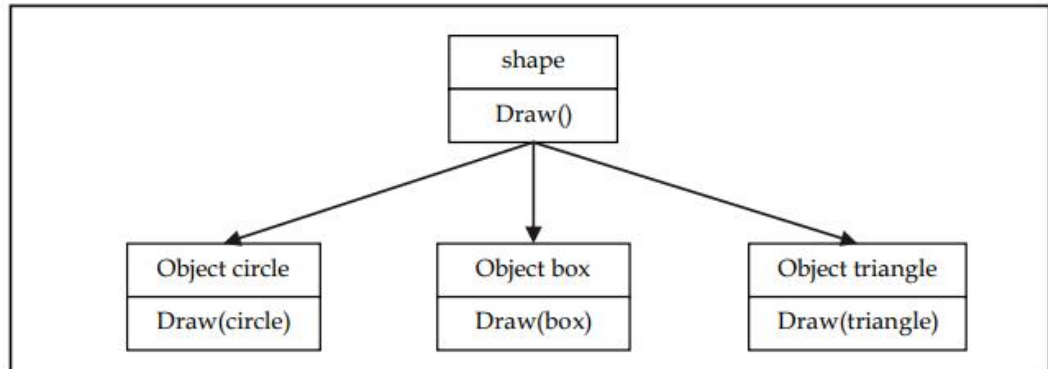
Process returned 0 (0x0) execution time : 1.653 s
Press any key to continue.
```

### 10.3 Polymorphism

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.

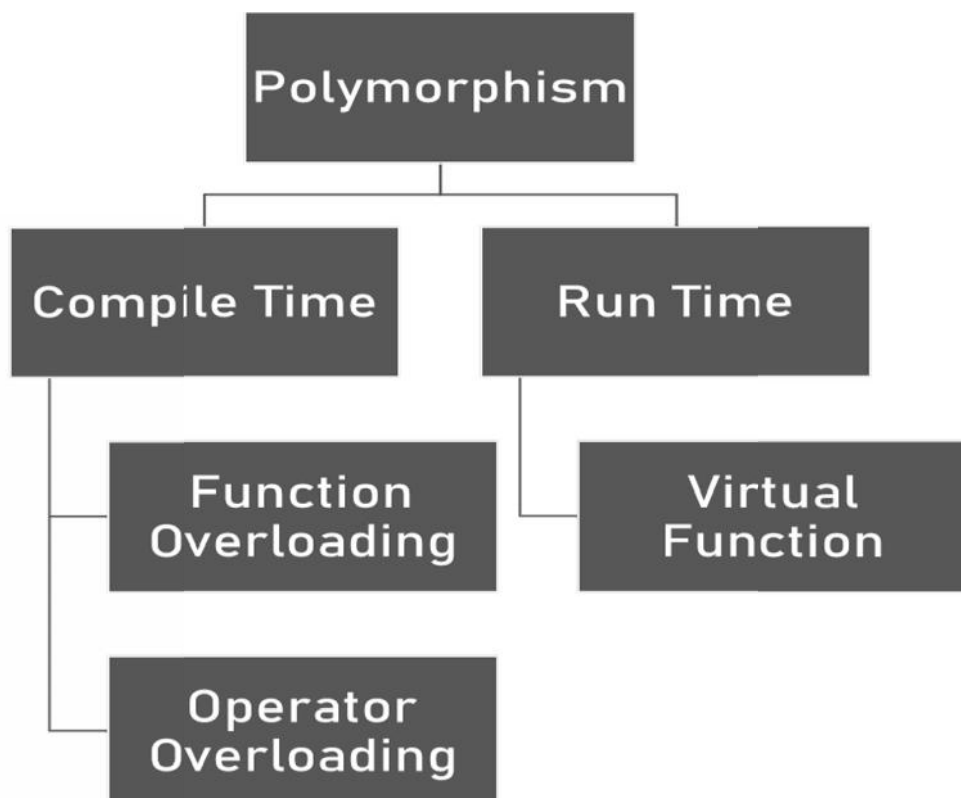
Polymorphism plays an important role in allowing objects having different internal structures to share the same external interface. This means that a general class of operations may be accessed in the same manner even though specific actions associated with each operation may differ. Polymorphism is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

#### **10.4 Types of C++ Polymorphism**

1. Compile time Polymorphism
2. Runtime Polymorphism



## 10.5 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.

```

## 10.6 Difference between Polymorphism and Inheritance

| Inheritance                                                                                                                             | Polymorphism                                                                                                                                     |
|-----------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------|
| Inheritance is one in which a new class (derived class) is formed that inherits the characteristics from the current class(base class). | Whereas polymorphism is what can be characterized in different ways.                                                                             |
| Basically, it is used for lessons.                                                                                                      | Whereas functions or techniques are simply implemented.                                                                                          |
| The principle of reusability is supported by inheritance and decreases code length in object-oriented programming.                      | Polymorphism enables the object to determine which function type is to be enforced at both compile-time (overloading) and run-time (overriding). |
| Inheritance may be an inheritance of pure, mixed, mixed, hierarchical and multilevel.                                                   | Whereas-time polymorphism (overload) as well as run-time polymorphism (overriding) can be compiled.                                              |
| In pattern building, it is used.                                                                                                        | Since it is also used in creating patterns.                                                                                                      |

## Summary

- Polymorphism is the occurrence of two or more distinct variants of a plant in the same ecosystem together in such quantities that the rarest of them cannot be preserved by repeated mutation.
- If there are even a few percent of a population with a genetically regulated type, it may have been preferred by option.
- Polymorphism may either be intermittent, in which a chromosome is in the process of rollout to an unopposed inhabitant, or regulated, in which the equilibrium of selective agencies is retained at a fixed stage.

- In general, due to the repeated creation of the variation, intermittent polymorphism is due to environmental changes that make the results of an earlier disadvantageous chromosome advantageous.

### **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.

### **SelfAssessment**

1. The static member functions \_\_\_\_\_
  - A. Have access to all the members of a class
  - B. Have access to only constant members of a class
  - C. Have access to only the static members of a class
  - D. Have direct access to all other class members also
2. The static member functions \_\_\_\_\_
  - A. Can be called using class name
  - B. Can be called using program name
  - C. Can be called directly
  - D. Can't be called outside the function
3. Choose the right observation from the followings.
  - A. Static member functions can't be virtual
  - B. Static member functions can be virtual
  - C. Static member functions can be declared virtual if it is pure virtual class
  - D. Static member functions can be used as virtual in Java
4. Which keyword should be used to declare the static member functions?
  - A. stat
  - B. const
  - C. common
  - D. static
5. Which is the correct syntax for declaring static data member?
  - A. static mamberNamedataType; .
  - B. dataType static memberName;

- C. memberName static dataType;
- D. static dataTypememberName;

6. What would be the output of the following code?

```
#include <iostream>
```

```
using namespace std;
```

```
class Demo
```

```
{
```

```
 private:
```

```
 static int a;
```

```
 public:
```

```
 static void fun()
```

```
 {
```

```
 cout<< a <<endl;
```

```
 }
```

```
};
```

```
int Demo :: a =20;
```

```
int main()
```

```
{
```

```
 Demo obj;
```

```
 obj.fun();
```

```
 return 0;
```

```
}
```

- A. 0
- B. 20
- C. Error
- D. None of above

7. A static member function can be called using the class as

- A. Class\_name::Function\_name();
- B. Member ::Function\_name();
- C. Class\_name;
- D. Function\_name();

8. 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

9. Function overloading is \_\_\_\_\_ in C++.

A. Class

B. Object

C. Compile Time Polymorphism

D. None of above

10. 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

11. Which of the following in Object-Oriented Programming is supported by Function overloading and default arguments features of C++?

A. Inheritance

B. Encapsulation

C. Polymorphism

D. None of the above

12. When will we use the function overloading?

A. same function name but same number of arguments

B. different function name but different number of arguments

C. same function name but different number of arguments



D. different function name but same number of arguments

13. Function overloading is also similar to which of the following?

- A. function overloading
- B. destructor overloading
- C. operator overloading
- D. constructor overloading

14. In which of the following we cannot overload the function?

- A. Caller
- B. Return function
- C. Called function
- D. All of above

15. Several functions of the same name can be defined, as long as they have different parameters, this is called

- A. Function overloading
- B. Functions reusing
- C. Operators overloading
- D. None of them

### **Answers for Self-Assessment**

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

### **Review Questions**

1. Program to demonstrate working static data members and static member functions in object-oriented programming C++.
2. To calculate the area of circle, rectangle, and triangle using function overloading.
3. Do inline functions improve performance?
4. How can inline functions help with the tradeoff of safety vs. speed?
5. Write a program that demonstrates the working of function overloading.
6. Write a program that demonstrates the 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**

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

<https://teachcomputerscience.com/polymorphism/>

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



## Unit 11: Constructors and Destructors

### CONTENTS

Objectives

Introduction

11.1 Constructor and Destructor

11.2 Difference Between Constructor and Destructor in C++

11.3 Copy Constructor

11.4 Dynamic Constructor

11.5 Parameterized Constructors

11.6 Constructors with Default Arguments

11.7 Constructor overloading

11.8 Destructors

Summary

Keywords

Self Assessment

Answers for Self Assessment

Review Questions

Further Readings

### 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 of 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.

## 11.1 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.

```
class abc
{
}
main()
{
}
int x,y;
abc myabc;
...;
```

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:

```
class abc
{
}
main()
{
 abc myabc(100,200);
 ---;
}
```

Unit 11: Constructors and Destructors

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;

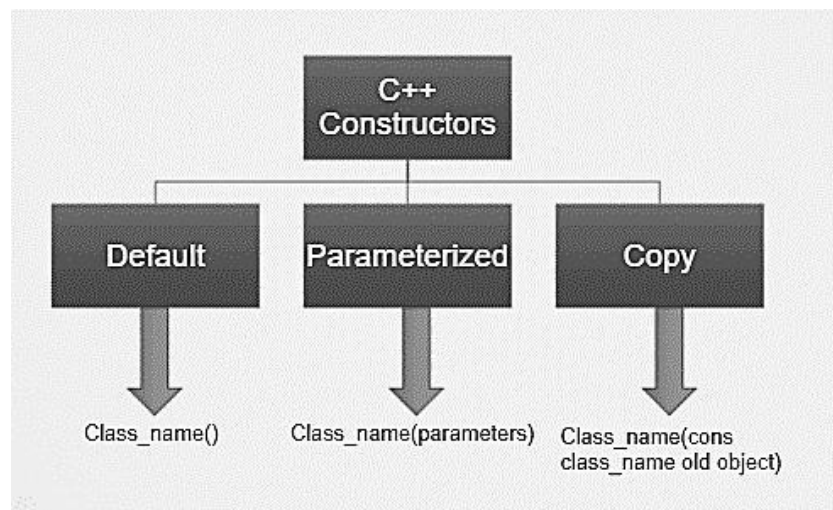
y=a;

}
```

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.

Unit 11: Constructors and Destructors

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
```

## 11.2 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                              | A destructor allows an object to execute                                    |

**Unit 11: Constructors and Destructors**

|                                                                              |                                                                                     |
|------------------------------------------------------------------------------|-------------------------------------------------------------------------------------|
| some of its value before it is used.                                         | 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.                              |

**11.3 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?

- If a new object has to be created before the copying can occur, the copy constructor is used.
- If an object does not have to be created before the copying can occur, the assignment operator is used.

**11.4 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>

class dyncons
{
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

## **11.5 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 ();
}

```

## 11.6 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;
}

```

Object Oriented Programming

---

```
};
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 i2 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=0)

**Lab Exercise**

```
// # Program - Default constructor
#include<iostream>
using namespace std;
class constructor{
private:
int x,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;
```

Object Oriented Programming

---

```

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**

```

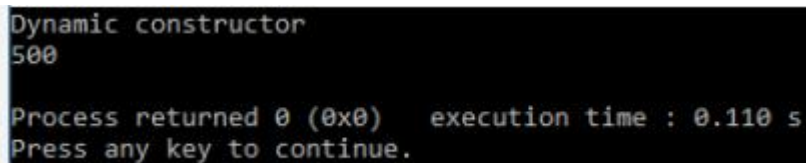
#include <iostream>
using namespace std;
class demo {
 int* p;

```

```
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.
```

### 11.7 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.

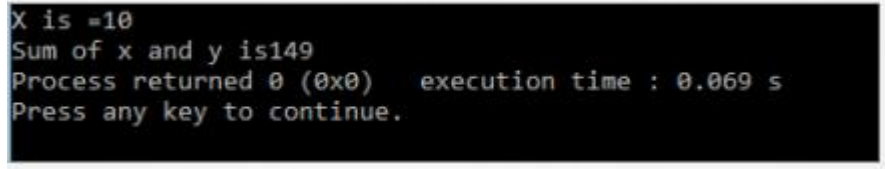


Object Oriented Programming

- 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{
 int x,y;
public:
 demo(){
 x=10;
 cout<<"X is "<<x<<endl;

 }
 demo(int a,int b){
 x=a;
 y=b;
 cout<<"Sum of x and y is"<<x+y;
 }
};
main(){
 demo d1,d2(90,59);
}
```

**Output**


```
X is =10
Sum of x and y is149
Process returned 0 (0x0) execution time : 0.069 s
Press any key to continue.
```

**11.8 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 is 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 creates an object of that class. Constructors are public member functions unless otherwise there is a good reason against.
- A constructor may take argument(s). A constructor that takes no argument(s) is known as a 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 the 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 function that is automatically called when an object is no longer required is known as a 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 deinitializes an object before it goes out of scope.

**SelfAssessment**

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

**Object Oriented Programming**

- 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](https://en.wikipedia.org/wiki/Object-oriented_programming)
- [www.web-source.net](http://www.web-source.net)

- [www.webopedia.com](http://www.webopedia.com)

## Unit 12: More on Constructors and Destructors

### CONTENTS

Objectives

Introduction

12.1 Default Arguments

12.2 Constructors with Default Arguments

12.3 Dynamic initialization of objects in C++

12.4 Destructor

Summary

Keywords

Self Assessment

Answers for Self Assessment

Review Questions

Further Readings

### Objectives

After studying this unit, you will be able to:

- Understand constructors with default arguments in C++.
- Analyze default arguments
- Discuss the destructors
- Understand dynamic initializing of objects

### 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.

### 12.1 Default Arguments

A default argument is a value provided in a function declaration that is automatically assigned by the compiler if the caller of the function doesn't provide a value for the argument with a default value.



#### Example

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

Object Oriented Programming

```
int sum(int a,int b,int c=0,int d=0)
{
 return(a+b+c+d);
}
int main(){
 cout<<sum(10,20)<<endl;
 cout<<sum(35,65,25)<<endl;
 return 0;
}
```

Output

```
30
125

Process returned 0 (0x0) execution time : 7.214 s
Press any key to continue.
```

*Default arguments are different from constant arguments as constant arguments can't be changed whereas default arguments can be overwritten if required.*

## 12.2 Constructors with Default Arguments

Default arguments of the constructor are those which are provided in the constructor declaration.



### Example

```
#include <iostream>
#include <string>
using namespace std;
class demo{
 int a,b,c;
public:
 demo(int x,int y, int z=0){
 a=x;
 b=y;
 c=z;
 }
 void display_sum();
};
void demo::display_sum(){
 cout<<"Sum is "<< a+b+c <<endl;
}
int main() {
 demo d(20,85);
 demo d1(35,85,96);
 d.display_sum();
```



```
d1.display_sum();
 return 0;
}
```

Output

```
Sum is 105
Sum is 216

Process returned 0 (0x0) execution time : 0.768 s
Press any key to continue.
```

### 12.3 Dynamic initialization of objects in C++

- When the initial values are provided during runtime then it is called dynamic initialization.
- It's possible to do so by using constructors and passing parameters to them.
- When there are several constructors of the same class with different inputs, this comes in handy.
- The memory is allocated at runtime using a new operator and similarly, memory is de-allocated at runtime using the delete operator.



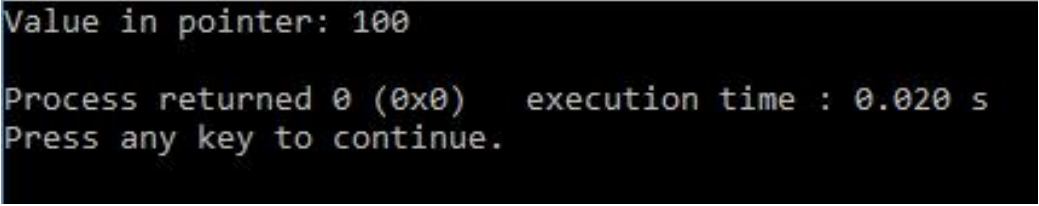
#### Example

```
#include <iostream>
using namespace std;
class demo {
 int* ptr;
public:
 demo()
 {
 ptr = new int;
 *ptr = 100;
 }
 void display()
 {
 cout << "Value in pointer: " << *ptr
 << endl;
 }
};
int main()
{
 demo* obj1 = new demo();
 obj1->display();
 delete obj1;
}
```

```
return 0;
```

```
}
```

Output



```
Value in pointer: 100

Process returned 0 (0x0) execution time : 0.020 s
Press any key to continue.
```

## 12.4 Destructor

- Destructor is a member function which destructs or deletes an object.
- It can be defined only once in a class. Like constructors, it is invoked automatically.

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.



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
```

#### **Difference between Constructor and Destructor in C++**

| Constructors                                     | Destructors                               |
|--------------------------------------------------|-------------------------------------------|
| The constructor initializes the class and allots | If the object is no longer required, then |

Object Oriented Programming

|                                                                                   |                                                                                     |
|-----------------------------------------------------------------------------------|-------------------------------------------------------------------------------------|
| the memory to an object.                                                          | 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.                              |

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 is 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.
- A constructor may take argument (s). A constructor taking 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 less number of arguments than required an error occurs. Every time an object is created its constructor is invoked.

Keywords

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

**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 reinitializes an object before it goes out of scope.

SelfAssessment

1. What are default arguments?
  - A. Arguments which are not mandatory to be passed
  - B. Arguments with default value that aren't mandatory to be passed
  - C. Arguments which are not passed to functions
  - D. Arguments which always take same data value

- 
2. The Constructors with all the default arguments are similar as default constructors. State true or false.
- A. True
  - B. False
  - C. May be
  - D. Can't say
3. Choose the right observation from the following?
- A. The constructors overloading can be done by using different names
  - B. The constructors overloading can be done by using different return types
  - C. The constructors can be overloaded by using only one argument
  - D. The constructors must have same name as that of class.
4. How destructor overloading is done?
- A. By changing the number of parameters
  - B. By changing the type of parameters
  - C. No chance for destructor overloading
  - D. None of above
5. A constructor that accepts \_\_\_\_\_ parameters is called the default constructor.
- A. 1
  - B. 2
  - C. 3
  - D. 0
6. What would be the output of following code
- ```
#include <iostream>
using namespace std;
class C{
private:
    C(){

        cout<<"This is a constructor";
    }

};

int main() {

    C obj;
    return 0;
}
```
- A. This is a constructor

- B. 0
 - C. 1
 - D. Compile time error
7. Which constructor function is designed to copy objects of the same class type?
- A. Create constructor
 - B. Object constructor
 - C. Dynamic constructor
 - D. Copy constructor
8. Which is executed automatically when the control reaches the end of the class scope?
- A. Constructor
 - B. Destructor
 - C. Overloading
 - D. Copy constructor
9. The special character related to destructor is ____
- A. +
 - B. !
 - C. ?
 - D. ~
10. A destructor is used to destroy the objects that have been created by a
- A. Class
 - B. Object
 - C. Constructor
 - D. Destructor
11. 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
12. Destructors _____ for automatic objects if the program terminates with a call to function exit or function abort
- A. Are inherited
 - B. Are created
 - C. Are called
 - D. Are not called

13. Choose the right observation from following for the destructors concept?
- Destructors can be overloaded
 - Destructors can have only one parameter at maximum
 - Destructors are always called after object goes out of scope
 - There can be only one destructor in a class
14. What is actual syntax of destructor in c++?
- !Classname()
 - @Classname()
 - \$Classname()
 - ~Classname()
15. Can a class have virtual destructor?
- Yes
 - No
 - Sometimes
 - Can't say

Answers for Self Assessment

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

Review Questions

- What do you mean by constructor?
- What are used of destructor in program?
- Differentiate between constructor and destructors.
- Write a program that demonstrates working of default arguments.
- What is destructor? Explain using program.



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 13: Inheritance

CONTENTS

Objectives

Introduction

13.1 Defining Derived Class

13.2 Modes of Inheritance

13.3 Types of Inheritance in C++

13.4 Making a Private Member Inheritable

Summary

Keywords

Self Assessment

Answers for Self Assessment

Review Questions

Further Readings

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

13.1 Defining Derived Class

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

```
class derivedclassname :access_specifierbaseclassname
```



```
{
.....
..... // 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,

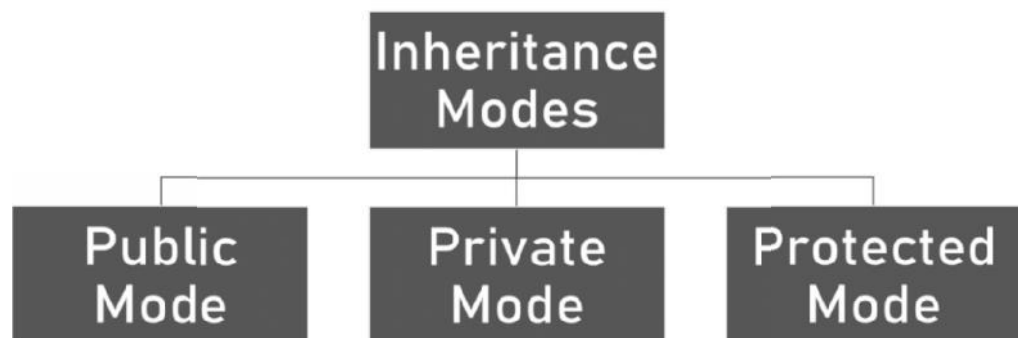
```
class xyz //base class
{
//members of xyz
};
class ABC: public xyz //public derivation
{
//members of ABC
};
class ABC : XYZ //private derivation (by default)
{
//members of ABC
};
```

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.

13.2 Modes of Inheritance

Inheritance Modes :-

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



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.

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.

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:

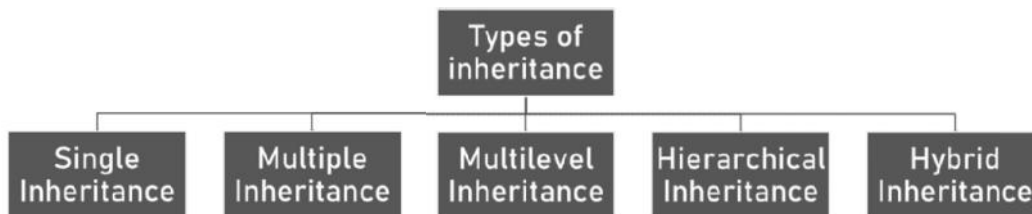
Development model closer to real life object model with hierarchical relationships

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

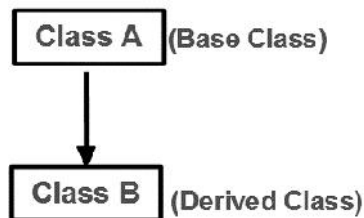
Data hiding –base class can decide to keep some data private so that it cannot be altered by the derived class.

13.3 Types of Inheritance in C++



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;
    void getdata()
    {
        cout<< "Enter the value of x = "; cin>> x;
    }
}
  
```

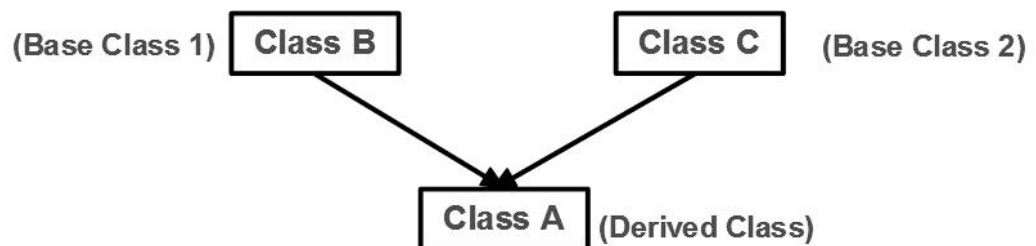
```
};  
class derive : public base  
{  
    private:  
        int y;  
    public:  
        void readdata()  
        {  
            cout<< "Enter the value of y = "; cin>> y;  
        }  
        void product()  
        {  
            cout<< "Product = " << x * y;  
        }  
};  
int main()  
{  
    derive a;  
    a.getdata();  
    a.readdata();  
    a.product();  
    return 0;  
}
```

Output

```
Enter the value of x = 12  
Enter the value of y = 12  
Product = 144  
Process returned 0 (0x0)   execution time : 2.462 s  
Press any key to continue.
```

Multiple Inheritance

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



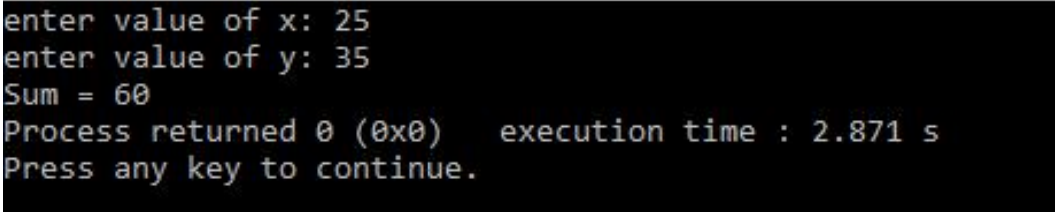
Following program shows working of multiple inheritance.



Example

```
#include<iostream>
using namespace std;
class A
{
    public:
    int x;
    void get_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
```

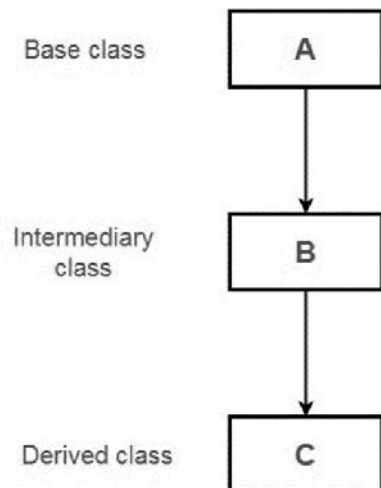
Output

A screenshot of a terminal window showing the output of the C++ program. The text is as follows:

```
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.
```

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;
    void get_data(){
cout<<"Enter Marks";
cin>>marks;
    }
};
class B:public A
{
public:
```

```

    int show_data(){
cout<<"Entered Marks: " <<marks;
    }
};
class C:public B{
    };
    main(){
    C obj;
obj.get_data();
obj.show_data();
    }

```

Output

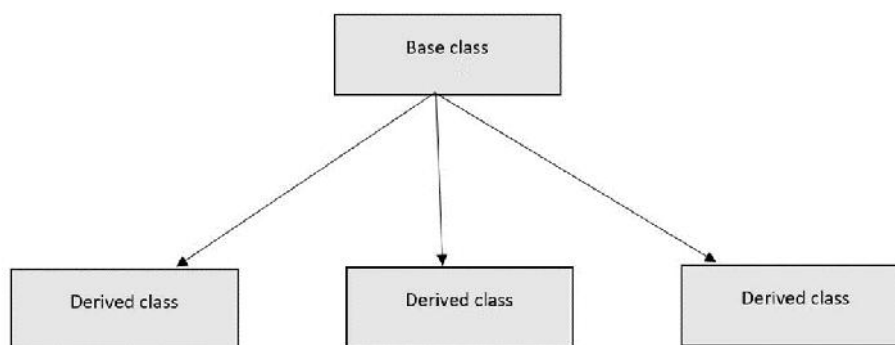
```

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

```

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:
    int x,y;

```

```
        void get_data()
    {
        cout<< "Enter value of x: ";
        cin>> x;
        cout<<"Enter value of y: ";
        cin>>y;
    }
};
class B:public A
{
    public:
    void show_sum()
    {
        cout<< "Sum of x and y is : "<<x+y<<endl;
    }
};
class C : public A
{
    public:
    void show_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

```

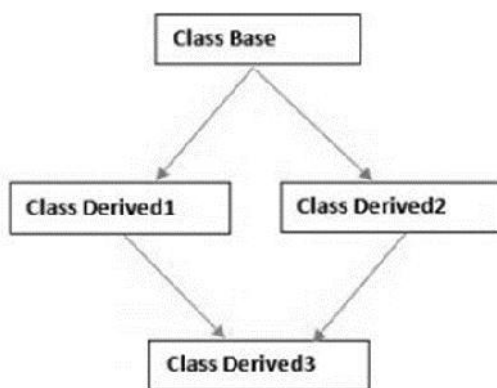
Enter value of x: 15
Enter value of y: 2
Sum of x and y is : 17
Enter value of x: 35
Enter value of y: 2
Product of x and y is : 70
Process returned 0 (0x0)   execution time : 5.966 s
Press any key to continue.

```

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

```



```

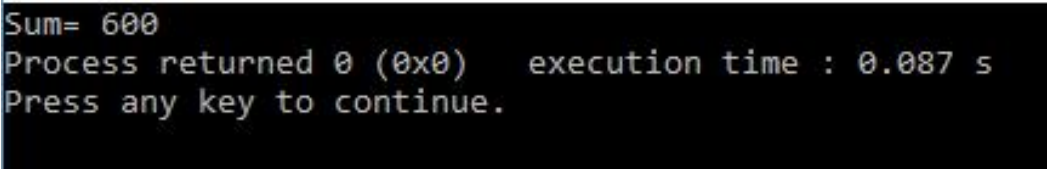
{
    public:
    int y;
    C()
    {
        y = 100;
    }
};

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.

```

13.4 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 public members of the base class can be accessed by its own objects using the dot operator. The result is that we have no member of base class that is accessible to the objects of the derived class.
- 2. Public:** On the other hand, when the base class is publicly inherited, 'public members' of the base class become 'public members' of derived class and therefore they are accessible to the objects of the derived class.
- 3. Protected:** C++ provides a third visibility modifier, protected, which serve a little purpose in the inheritance. A member declared as protected is accessible by the member functions within its class and any class immediately derived from it. It cannot be accessed by functions outside 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

```
class base_class_name
{
----
----
}
class derived_class_name : private base_class_name
{
----
----
}
```

```
//      Program
#include<iostream>
using namespace std;
class emp{
private:
    int id;
    char name[10];
    int salary;
    void get_data(){
cout<<"Enter Id,Name and Salary of  employee"<<endl;
cin>>id>>name>>salary;

    }
public:
    void disp(){
get_data();
cout<<"Details are"<<endl;
cout<<"Emp ID  "<<id<<endl<<"Emp Name  "<<name <<endl<<"Emp Salary "<<salary;
    }
};
main(){
emp obj;
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 life objects. Inheritance has many forms: Single inheritance, multiple inheritance, hierarchical inheritance, multilevel inheritance and hybrid inheritance.

A subclass can derive itself publicly, privately or protected. The derived class constructor is responsible for invoking the base class constructor, the derived class can directly access only 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 are constructed first before constructing the objects of the enclosing class.

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

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

Multiple Inheritance: A class inherits 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 as 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 allowed 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 specifier
 - C. Class name : access specifier 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

Answers for SelfAssessment

- | | | | | |
|-------|-------|-------|-------|-------|
| 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 |

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.



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://msdn.microsoft.com/en-us/library/wcz57btd(v=vs.80).aspx)
- <http://www.learncpp.com/cpp-tutorial/117-multiple-inheritance/>

Unit 14: File Handling

CONTENTS

Objectives

Introduction

14.1 Classes for File Stream Operations

14.2 Creating A File

14.3 Opening a File

14.4 Opening and Closing File

14.5 Detection end-of-file

Summary

Keywords

Self Assessment

Answers for Self Assessment

Review Questions

Further Readings

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

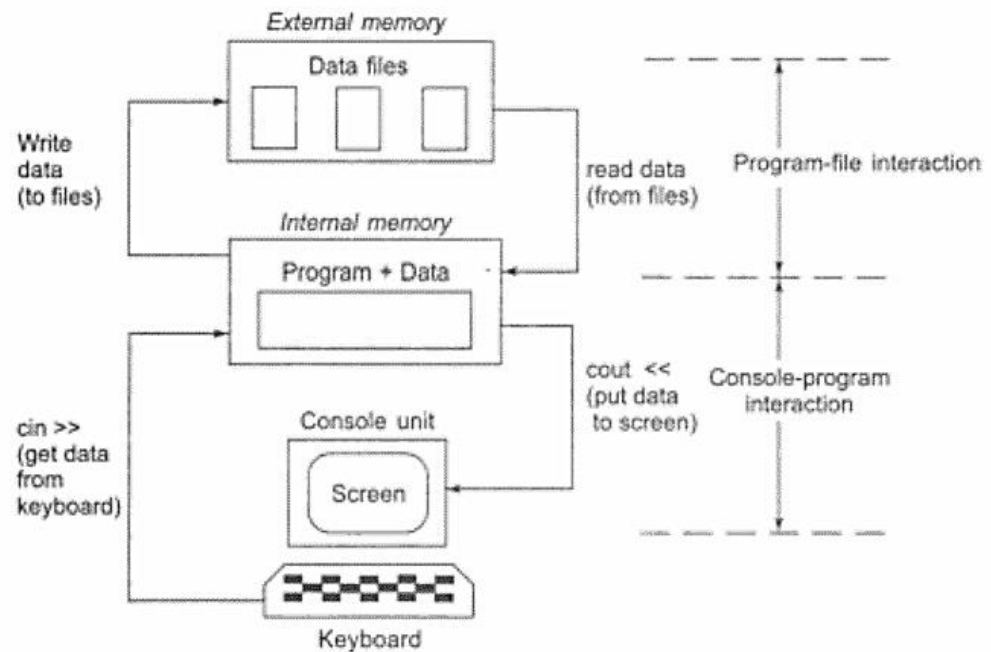
Introduction

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:

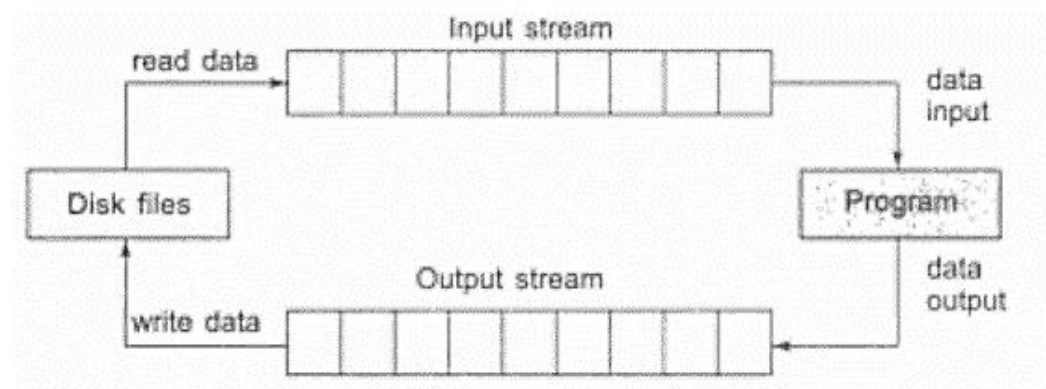
- Data transfer between control unit and the program.
- Data transfer between the program and a disk file.

This is illustrated in figure.



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.

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.

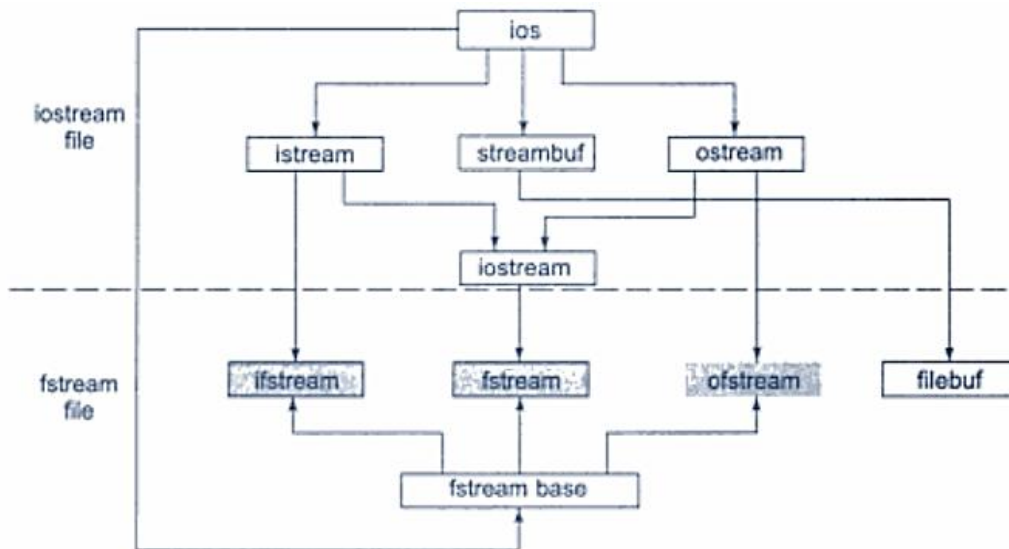


Notes

- 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.

14.1 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 is 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 ostream .
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 .

14.2 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()
{
    fstreamst;
    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.
```

14.3 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.

14.4 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 `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:

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

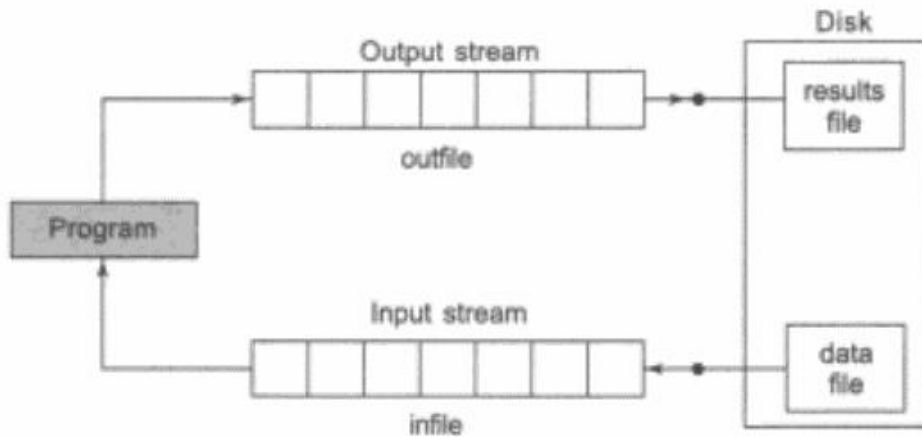
We know that a constructor is used to initialize an object while it is being created. Here, a filename is used to initialize the file stream object. This involves the following steps:

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;
}
```

14.5 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(fin.eof() != 0) (exit(1);)
```

eof() is a member function of ifstream 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 fstream 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.

SelfAssessment

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. istream 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. istream

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 SelfAssessment

- | | | | | |
|----------|----------|-------|----------|-------|
| 1. C | 2. C | 3. D | 4. D | 5. C |
| 6. A | 7. B | 8. C | 9. C | 10. A |
| 11. True | 12. True | 13. B | 14. True | 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>

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