

# **DIPLOMA IN COMPUTER APPLICATION**

## **DCA-15-T**

# **Object Oriented Programming Using C++**



**Centre for Distance and Online Education  
Guru Jambheshwar University of Science &  
Technology, HISAR-125001**



## Chapter 1

### Fundamentals of C++

#### 1.1 Introduction to C++

The C language was developed in 1972 by Dennis Ritchie at Bell Telephone laboratories, primarily as a systems programming language (a language to write operating systems with). Ritchie's primary goals were to produce a minimalistic language that was easy to compile, allowed efficient access to memory, produced efficient code, and was self-contained (not reliant on other programs). For a high-level language, it was designed to give the programmer a lot of control, while still encouraging platform (hardware and operating system) independence (that is, the code didn't have to be rewritten for each platform).

C ended up being so efficient and flexible that in 1973, Ritchie and Ken Thompson rewrote most of the Unix operating system using C. Many previous operating systems had been written in assembly. Unlike assembly, which produces programs that can only run on specific CPUs, C has excellent portability, allowing Unix to be easily recompiled on many different types of computers and speeding its adoption. C and Unix had their fortunes tied together, and C's popularity was in part tied to the success of Unix as an operating system.

In 1978, Brian Kernighan and Dennis Ritchie published a book called "The C Programming Language". This book, which was commonly known as K&R (after the authors' last names), provided an informal specification for the language and became a de facto standard. When maximum portability was needed, programmers would stick to the recommendations in K&R, because most compilers at the time were implemented to K&R standards.

In 1983, the American National Standards Institute (ANSI) formed a committee to establish a formal standard for C. In 1989 (committees take forever to do anything), they finished, and released the C89 standard, more commonly known as ANSI C. In 1990 the International Organization for Standardization (ISO) adopted ANSI C (with a few minor modifications). This version of C became known as C90. Compilers eventually became ANSI C/C90 compliant, and programs desiring maximum portability were coded to this standard.



In 1999, the ANSI committee released a new version of C called C99. C99 adopted many features which had already made their way into compilers as extensions, or had been implemented in C++.

C++ (pronounced see plus plus) was developed by Bjarne Stroustrup at Bell Labs as an extension to C, starting in 1979. C++ adds many new features to the C language, and is perhaps best thought of as a superset of C, though this is not strictly true (as C99 introduced a few features that do not exist in C++). C++'s claim to fame results primarily from the fact that it is an object-oriented language. As for what an object is and how it differs from traditional programming methods, well, we'll cover that in chapter 8 (Basic object-oriented programming).

C++ was standardized in 1998 by the ISO committee (this means the ISO committee ratified a document describing the C++ language, to help ensure all compilers adhered to the same set of standards). A minor update was released in 2003 (called C++03).

Three major updates to the C++ language (C++11, C++14, and C++17, ratified in 2011, 2014, and 2017 accordingly) have been made since then, each adding additional functionality. C++11 in particular added a huge number of new capabilities, and at this point is widely considered the new baseline. As of the time of writing, C++20 is in the works, promising to bring even more new capabilities. Future upgrades to the language are expected every three or so years.

Each new formal release of the language is called a **language standard** (or **language specification**). Standards are named after the year they are released in. For example, there is no C++15, because there was no new standard in 2015.

## 1.2 Character Set

**Character set** is the combination of English language (Alphabets and White spaces) and math's symbols (Digits and Special symbols). Character Set means that the characters and symbols that a C++ Program can understand and accept. These are grouped to form the commands, expressions, words, c-statements and other tokens for C++ Language.

Character Set is the combination of alphabets or characters, digits, special symbols and white spaces same as learning English is to first learn the alphabets, then learn to combine these alphabets to form words, which in turn are combined to form sentences and sentences are combined to form paragraphs. More about a C++ program we can say that it is a sequence of characters. These characters from the



character set plays the different role in different way in the C++ compiler. The special characters are listed in Table

Special Characters				
+	>	/	[	\
!	;	“	]	{
<	*	.	%	}
:	^	,	~	#
-	(	+	_	
?	)	.	&	

In addition to these characters, C++ also uses a combination of characters to represent special conditions. For example, character combinations such as '\n', '\b' and '\t' are used to represent newline, backspace and horizontal tab respectively.

### 1.3 Tokens

As in the English language, in a paragraph all the words, punctuation mark and the blank spaces are called Tokens. Similarly in a C++ program all the C++ statements having *Keywords*, *Identifiers*, *Constants*, *Strings*, *Operators* and the *Special Symbols* are called C++ Tokens. C++ Tokens are the essential part of a C++ compiler and so are very useful in the C++ programming. A Token is an individual entity of a C++ program.

**For example, some C++ Tokens used in a C++ program are:**

**Reserve:** words long, do if, else etc.

**Identifiers:** Pay, salary etc.

**Constant:** 470.6, 16, 49 etc.

**Strings:** “Dinesh”, “2013-01” etc.

**Operator:** +, \*, <, >=, &&, 11, etc

**Special symbols:** 0, {}, #, @, %, etc.



**Variable:** A variable is used for storing a value either a number or a character and a variable also vary its value means it may change his value Variables are used for given names to locations in the Memory of Computer where the different constants are stored. these locations contain Integer ,Real or Character Constants.

### 1.4 Keywords

A Keyword is that which have a special meaning those are already been explained to the c++ language like cout, cin, for, if, else , etc these are the reserve keywords Always Remember that we can t give a name to a variable as the name of a keyword we cant create new Keywords in c Language. All the keywords of C++ are listed in Table.

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



## 1.5 Identifiers

The Identifiers are those which are used for giving a name to a variables, arrays, functions, classes, structures, namespaces and so on, like a b etc these are used for naming a variable. When we declare any variable then we specify a Name So that identifiers are used for naming a variable. While defining identifiers in C++, programmers must follow the rules listed here .

- An identifier must be unique in a program
- An identifier must contain only upper case and lower case letters, underscore character ( \_ ) ordigits 0 to 9.
- An identifier must start with a letter or an underscore.
- An identifier in upper case is different from that in lower case.
- An identifier must be different from a keyword. In addition, identifiers that start with a double underscore ‘ \_ ’ or an underscore followed by an upper case letter must be avoided as these names are reserved by the Standard c++ Library.
- An identifier must not contain other characters such as ‘\*’, ‘;’ and whitespace characters (tabs, space and newline).

Some valid and invalid identifiers in C++ are listed here.

```
Po178_ddm    //valid
_78hhvt4     //valid
902gt1       //invalid as it starts with a digit
Tyy;ui8      //invalid as it contains the ‘;’ character
for          //invalid as it is a c++ keyword
Fg026 neo    //invalid as it contains spaces
```

## 1.6 Constants

**Constants, also known as literals:** The term constants or literals refers to the fixed value means a Constant is that whose value is never changed at the end of the program.



for ex:

$$3x+2y=10$$

Here 3 and 2 and 10 are constants their value never be change but here x and y are the variables and may be they vary their value

### 1.6.1 Numeric Constants

Numeric constants refer to the numbers consisting of a sequence of digits (with or without decimal point) that can be either positive or negative. However, by default, numeric constants are positive. Numeric constants can be further classified as *integer constants* and *floating-point constants*, which are listed in Table

Type of Numeric constants		
Type	Description	Examples
Integer constants	Integer constants refer to integer-valued numbers. Integer constants can be represented by three different number systems namely . decimal (base 10), octal (base 8) and hexadecimal numbers (base 16). The octal constants are preceded by a 0 (Zero ) and hexadecimal constants are preceded by a 0x or ox.	54, - 646, 01612, 0x38A
Floating- Point constants	Floating - point constants refer to real numbers, that is, the number a decimal point Floating -point constants are also written in the floating -point notation in which the constant is divided into a mantissa and an exponent.	64 .23, - 74.32, 537E-9

### 1.6.2 Character Constants

Character constants refer to a single character enclosed in single quotes ( ' '). The examples of character constants are 'f', 'M', '8', '&', '7', etc. All character constants are internally stored as integer values.

Character constants can represent either the printable characters or the non-printable characters. The examples of printable character constants are 'a', '5', '#', ';', etc. However, there are few character constants that cannot be included in a program directly through a keyboard such as backspace, newline and so on. These character constants are known as non-printable constants and are included in a program using the *escape sequences*. An escape sequence refers to a character preceded by the backslash character (\). Some of the escape sequences used in C++ is listed in Table



Escape-Sequences	
Escape Sequences	Character constant
\a	Alert(bell)
\b	Backspace
\f	Form feed
\n	NewLine(Linefeed)
\r	Carriage return
\t	Horizontal Tab
\v	Vertical Tab
\0	Null
\'	Single quote
\"	Double quote
\\	Backslash
\?	Questionmark
\C	Octal Constant(c ia a three digit constant)
\xC	Hexadecimal Constant(c ia a three digit hexadecimal constant)

### 1.6.3 String Constants

String constants refer to a sequence of any number of characters enclosed in the double quotes (” “). The examples of string constants are “hello”, “name”, “colour”, “date”, etc. Note that string constants are always terminated by the Null ( ‘ \0 ‘ ) character.

The presence of a backslash character in a string constant indicates an escape sequence. For example, the string constant “welcome \ “home” is displayed as welcome” home. Note that the double quote next to the backslash is an escape sequence and not a delimiter for the string constant.

## 1.7 C++ Data Types

Data types in C++ is mainly divided into three types:

1. *Primitive Data Types*: These data types are built-in or predefined data types and can be used directly by the user to declare variables. example: int, char , float, bool etc. Primitive data types available in C++ are:

- Integer
- Character





- Boolean
  - Floating Point
  - Double Floating Point
  - Valueless or Void
  - Wide Character
2. *Derived Data Types:* The data-types that are derived from the primitive or built-in datatypes are referred to as Derived Data Types. These can be of four types namely:
- Function
  - Array
  - Pointer
  - Reference
3. *Abstract or User-Defined Data Types:* These data types are defined by user itself. Like, defining a class in C++ or a structure. C++ provides the following user-defined datatypes
- Class
  - Structure
  - Union
  - Enumeration
  - Typedef defined DataType

This article discusses primitive data types available in C++.

- **Integer:** Keyword used for integer data types is **int**. Integers typically requires 4 bytes of memory space and ranges from -2147483648 to 2147483647.
- **Character:** Character data type is used for storing characters. Keyword used for character data type is **char**. Characters typically requires 1 byte of memory space and ranges from -128 to 127 or 0 to 255.
- **Boolean:** Boolean data type is used for storing boolean or logical values. A boolean variable can store either *true* or *false*. Keyword used for boolean data type is **bool**.



- **Floating Point:** Floating Point data type is used for storing single precision floating point values or decimal values. Keyword used for floating point data type is **float**. Float variables typically requires 4 byte of memory space.
- **Double Floating Point:** Double Floating Point data type is used for storing double precision floating point values or decimal values. Keyword used for double floating point data type is **double**. Double variables typically requires 8 byte of memory space.
- **void:** Void means without any value. void datatype represents a valueless entity. Void data type is used for those function which does not returns a value.
- **Wide Character:** Wide character data type is also a character data type but this data type has size greater than the normal 8-bit datatype. Represented by **wchar\_t**. It is generally 2 or 4 bytes long.

### 1.8 Variables and their Declaration

Variables represent named storage locations, whose values can be manipulated during program run. For instance, to store name and marks of a student during a program run, we require two storage locations, having different name so that these can be distinguished easily.

Variables, called as symbolic variables, serve the purpose. The variables are called symbolic variables because these are named locations. For instance, the following statement declares a variable `i` of the data type `int` :

**`int i;`**

There are the following two values associated with a symbolic variable :

- Its data value, stored at some location in memory. This is sometimes referred to as a variable's rvalue (pronounced "are-value").
- Its location value; that is, the address in memory at which its data value is stored. This is sometimes referred to as a variable's lvalue (pronounced as "el-value").

### C++ Variable Declaration

Here is the general form to declare a variable in C++

type name;



Here, type is any valid C++ data type and name is the name of the variable. A variable name is an identifier. Therefore, all the rules of identifier naming apply in declaring the name of a variable. Following declaration creates a variable age of int type :

```
int age;
```

### C++ Variable Initialization

All the example definitions of above examples are simple definitions. A simple definition does not provide a first value or initial value to the variable i.e., variable is uninitialized and the variable's value is said to be undefined. A first value (initial value) may be specified in the definition of a variable. A variable with a declared first value is said to be an initialised variable.

Here is the general form to initialize values to a variable in C++

```
type variable_name = value;
```

Here is a code fragment showing the variable initialization in C++

```
int val = 100;
```

## 1.9 C++ Standard Library

The C++ Standard Library can be categorized into two parts –

- **The Standard Function Library** – This library consists of general-purpose, stand-alone functions that are not part of any class. The function library is inherited from C.
- **The Object Oriented Class Library** – This is a collection of classes and associated functions.

Standard C++ Library incorporates all the Standard C libraries also, with small additions and changes to support type safety.

The Standard Function Library

The standard function library is divided into the following categories –

- I/O,
- String and character handling,
- Mathematical,
- Time, date, and localization,



- Dynamic allocation,
- Miscellaneous,
- Wide-character functions,

### The Object Oriented Class Library

Standard C++ Object Oriented Library defines an extensive set of classes that provide support for a number of common activities, including I/O, strings, and numeric processing. This library includes the following –

- The Standard C++ I/O Classes
- The String Class
- The Numeric Classes
- The STL Container Classes
- The STL Algorithms
- The STL Function Objects
- The STL Iterators
- The STL Allocators
- The Localization library
- Exception Handling Classes
- Miscellaneous Support Library

### 1.10 Basics of Typical C++ Environment

C++ programs can be created using any text editor. For the UNIX implementation of C++, vi or ed text editor can be used for creating and editing the source code. For the DOS implementations of C++, edlin or any other editor can be used. Some systems such as Turbo C++ and Borland C++ provide an Integrated Development Environment (IDE) for developing programs. These IDEs provide facilities for typing, editing, searching, compiling etc integrated in one package.

A C++ program files should have a proper file extension depending upon the implementation of C++, it is developed/compiled into. C++ implementations use several extensions, for example .c, .cc, .cpp, .cxx



etc. TurboC++ and Borland C++ use .cpp for C++ programs. Zortech C++ uses extension .cxx and UNIX AT&T versions use .c and .cc extension.

### 1.11 Compile and Execute C++ Programs using Borland C++

For writing and compiling programs in Borland C++, the Borland C++ software must either be available on hard disk or floppy disk. After booting your system and once the dos prompt appears on the screen, then do the following steps :

1. Place your working copy of Borland C++ program disk in the floppy disk drive, (if you have BC++ on floppy disk), or change to your Borland C++ subdirectory (if BC++ on the hard disk).
2. Type BC at the dos prompt and press Enter. It will take you to the IDE (Integrated Development Environment) of BC++. The topmost row of the IDE displays its menu that contains options  $\equiv$ , File, Edit, Search, Run, Compile, Debug, Project, Options, Window, Help. Each option on the menu can be involved by pressing Alt key and the highlighted letter of the menu option. The option ( $\equiv$ ), the system menu option, can be involved by pressing Alt+spacebar. For details, you can refer to the user manual of Borland C++.
3. To write a program, then, Press Alt+E to start the editor, and then Type your C++ program line by line
4. To save your program, then, After typing the program, press Alt+F to activate File menu, Press S to save the program file, Type the program name you want to give to your program file. You need not give extension .cpp since Borland C++ adds it automatically, and then Press Enter and your file is saved.
5. To compile your program, press Alt+F9 key combination or alternatively you can press Alt+C followed by C or enter key.
6. To run your program, press ctrl+F9 key combination or alternatively Alt+R followed by R or enter key.

When you execute (run) your program, it waits for the user input if required any and carries out all the instructions.



7. To see your program output, press the key combination Alt+F5. Any keypress or click will take you back to Borland C++ IDE.

### 1.12 Compile and Execute C++ Programs using TurboC++

Now let's look at how to save the C++ programming source code in a file, and how to compile and run it using the TurboC++ compiler. Following are the simple steps:

1. Download TurboC++ compiler
2. Install it on your system
3. Now open TurboC++ and type your C++ program inside it.
4. After typing your complete C++ program, press Alt+F key and go to save button and type your program name followed by .cpp extension and press enter. Now your program will be saved inside BIN directory of TurboC++ in C drive.
5. Now to compile C++ program, simply press F9 key, if an error occurs then watch at your code to make correction and recompile it.
6. If there will no error appears, then press Ctrl+F9 key to run the program and watch the result.

#### 1.12.1 Role of Compiler

A part of compiler's job is to analyse the program code for "correctness". If the meaning of a program is correct, then a compiler can not detect errors (for example, if different statements are used etc.), but a compiler can certainly detect errors in the form of a program. Some common forms of program errors are :

1. Syntax Errors - Syntax errors occur when rules of a programming language are misused i.e., when a grammatical rule of C++ is violated.
2. Semantics Errors - Semantic errors occur when statements are not meaningful. Semantic refers to the set of rules which give the meaning of a statement. For instance, the statement.
3. Type Errors - Data in C++ has an associated data type. The value 7, for instance, is an integer. 'a' is a character constant and "hello" is a string. If a function is given wrong type of data, type error is signalled by the compiler.



4. Run-time Errors (Execution Errors) - A run-time error is that occurs during the execution of a program. It is caused of some illegal operation taking place or in-availability of desired or required conditions for the execution of program.
5. Logical Errors - A logical error is that error which causes a program to produce incorrect or undesired output.

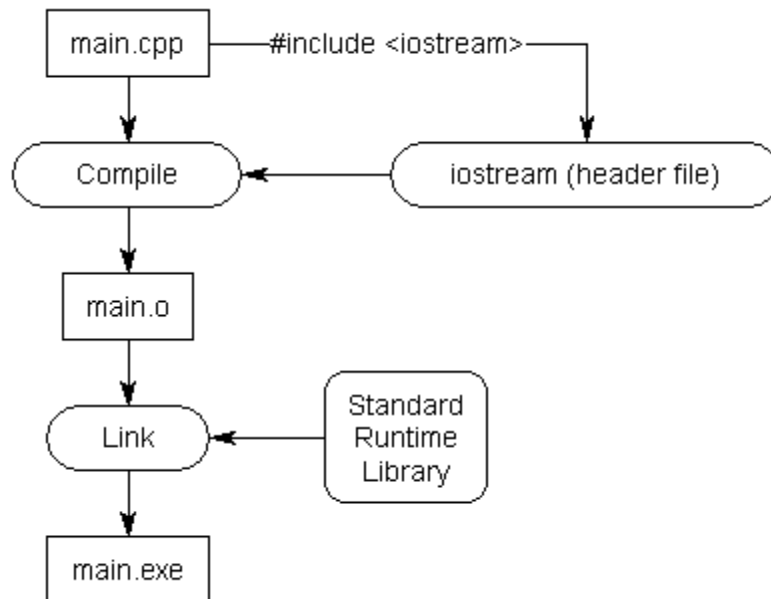
### 1.13 Header Files and Namespaces

As programs grow larger (and make use of more files), it becomes increasingly tedious to have to forward declare every function you want to use that is defined in a different file. Wouldn't it be nice if you could put all your forward declarations in one place and then import them when you need them?

C++ code files (with a .cpp extension) are not the only files commonly seen in C++ programs. The other type of file is called a **header file**. Header files usually have a .h extension, but you will occasionally see them with a .hpp extension or no extension at all. The primary purpose of a header file is to propagate declarations to code files.

Consider what would happen if the *iostream* header did not exist. Wherever you used `std::cout`, you would have to manually type or copy in all of the declarations related to `std::cout` into the top of each file that used `std::cout`! This would require a lot of knowledge about how `std::cout` was implemented, and would be a ton of work. Even worse, if a function prototype changed, we'd have to go manually update all of the forward declarations. It's much easier to just `#include iostream`!

When it comes to functions and variables, it's worth keeping in mind that header files typically only contain function and variable declarations, not function and variable definitions (otherwise a violation of the *one definition rule* could result). `std::cout` is forward declared in the *iostream* header, but defined as part of the C++ standard library, which is automatically linked into your program during the linker phase.



Consider a situation, when we have two persons with the same name, Zara, in the same class. Whenever we need to differentiate them definitely we would have to use some additional information along with their name, like either the area, if they live in different area or their mother's or father's name, etc.

Same situation can arise in your C++ applications. For example, you might be writing some code that has a function called `xyz()` and there is another library available which is also having same function `xyz()`. Now the compiler has no way of knowing which version of `xyz()` function you are referring to within your code.

## Namespaces

A **namespace** is designed to overcome this difficulty and is used as additional information to differentiate similar functions, classes, variables etc. with the same name available in different libraries. Using namespace, you can define the context in which names are defined. In essence, a namespace defines a scope.

### Defining a Namespace

A namespace definition begins with the keyword **namespace** followed by the namespace name as follows –

```
namespace namespace_name {
```





```
// code declarations  
}
```

To call the namespace-enabled version of either function or variable, prepend (::) the namespace name as follows –

name::code; // code could be variable or function.

### 1.14 Library files

#### Structure of C++ Program

Program Structure of a simple C++ program contains a header file, a function main(), then program code. Here is the program structure of a simple C++ program:

header files

```
return_type main()  
{  
    program_code_statements;  
}
```

Here is an example, showing the program structure of a simple C++ program:

```
/* C++ Program Structure - This is a multi-line comment */  
  
#include<iostream.h>           // header file. This is a single-line comment  
  
#include<conio.h> // header file  
  
void main()                    // main function  
{  
    clrscr();                  // to clear the screen  
    cout<<"Hello, C++";      // prints Hello, C++  
    getch();                   // holds output screen until user press a key  
}
```



In the above program, the first line is a multi-line comments. Multi-line comment starts from `/*` and end with `*/`. Then the second line and the third line, both are header files included in the program by using `#include`. The `iostream.h` header file is responsible for `cout` (standard output) and `conio.h` header file is responsible for the functions `clrscr()` (to clear the output screen) and then `getch()` (holds the output screen until user press a key). Now the fourth line contains `main()` function. Here `void` represents that the program will not return any value. And in last, between `{` and `}`, `program_code_statements` is present. In which, `clrscr()` function clears the output screen, `cout` is used to send the string "Hello, C++" on the output screen. In other words, `cout<<"Hello, C++";` simply prints the string "Hello, C++" on the standard output. Now `getch()` is the function, responsible to hold the output screen until and unless, user press any key. You can see from the above program, we have included some short description after `//`, which indicates single-line comments. Anything after `//` becomes a single-line comment. Comments are totally ignored by the compiler. Here is the output of the above C++ program:

## SUMMARY

In C++, we use variables to access memory. Variables have an identifier, a type, and a value (and some other attributes that aren't relevant here). A variable's type is used to determine how the value in memory should be interpreted.

### Multiple Choice Questions

Q 1: Which of the following is not the keyword in C++?

A - volatile

B - friend

C - extends

D - this

Q 2: 'cin' is an \_\_

A. Class

B. Object

C - Package

D - Namespace



Q 3: A user defined header file is included by following statement in general.

A - #include "file.h"

B - #include <file.h>

C - #include <file>

D - #include file.h

Q 4 Which data type can be used to hold a wide character in C++?

A - unsigned char

B - int

C - wchar\_t

D - none of the above.

Q5. In CPP, cin and cout are the predefined stream \_\_\_\_\_ .

A. Operator

B. Functions

C. Objects

D. Data types

**ANS: 1-C    2-B    3-A    4-C    5-C**

Q1. Define the structure of the C++ programs.

Q2. What do you mean by header files in C++? How header files reduces the compilation time and efforts of the programmer?

Q3. Define keywords. What are the keywords available in C++ language?

Q4. What is variable? How they are defines and initialized? Write important rules to define a variable.



## Chapter 2:

# Introduction to Object Oriented Programming

## 2Introduction to Object Oriented Programming

- 2.1 Introduction
- 2.2 Object Oriented Concepts
- 2.3 Objects and Classes
- 2.4 Data Abstraction
- 2.5 Encapsulation
- 2.6 Message Passing
- 2.7 Inheritance
- 2.8 Polymorphism
- 2.9 Generalization and Specialization
- 2.10 Modularity

### 2.1 Introduction

Object-oriented programming (OOP) is a programming paradigm based on the concept of *objects*, which are *data structures* that contain data, in the form of fields (or attributes) and code, in the form of procedures, (or methods). A distinguishing feature of objects is that an object's procedures provide access to and modify its fields.

In object-oriented programming, computer programs are designed by making them out of objects that interact with one another. There is significant diversity in object-oriented programming, but most popular languages are class-based, meaning that objects are instances of classes, which typically also determines their type.

Object orientation is an outgrowth of procedural programming. Procedural programming is a programming paradigm, derived from structured programming, based upon the concept of the procedure call. Procedures, also known as routines, subroutines, or methods define the computational steps to be



carried out.

Any given procedure might be called at any point during a program's execution, including by other procedures or itself. Procedural programming is a list or set of instructions telling a computer what to do step by step and how to perform from the first code to the second code. Procedural programming languages include C, Fortran, Pascal, and BASIC.

The focus of procedural programming is to break down a programming task into a collection of variables, data structures, and subroutines, whereas in object-oriented programming it is to break down a programming task into *objects* that expose behavior (methods) and data (fields) using interfaces. The most important distinction is that while procedural programming uses procedures to operate on data structures, object-oriented programming bundles the two together, so an **object**, which is an **instance of a class**, operates on its "own" data structure.

## 2.2 Object Oriented Concepts

The conceptual framework of object-oriented systems is based upon the object model. In this chapter, we will look into the following basic concepts and terminologies of object-oriented systems:

- ✓ Objects and Classes
- ✓ Data Abstraction
- ✓ Encapsulation
- ✓ Message Passing
- ✓ Inheritance
- ✓ Polymorphism
- ✓ Generalization and Specialization
- ✓ Modularity
- ✓ Persistence
- ✓ Typing
- ✓ Concurrency
- ✓ Hierarchy



## 2.3 Objects and Classes

The concepts of objects and classes are intrinsically linked with each other and form the foundation of object-oriented paradigm.

### *Object*

An object is a real-world element in an object-oriented environment that may have a physical or a conceptual existence. Each object has:

- Identity that distinguishes it from other objects in the system.
- State that determines the characteristic properties of an object as well as the values of the properties that the object holds.
- Behavior that represents externally visible activities performed by an object in terms of changes in its state.

Objects can be modelled according to the needs of the application. An object may have a physical existence, like a customer, a car, etc.; or an intangible conceptual existence, like a project, a process, etc.

### **Classes**

A class represents a collection of objects having same characteristic properties that exhibit common behavior. It gives the blueprint or description of the objects that can be created from it. Creation of an object as a member of a class is called instantiation. Thus, object is an instance of a class.

The constituents of a class are –

- A set of attributes for the objects that are to be instantiated from the class. Generally, different objects of a class have some difference in the values of the attributes. Attributes are often referred as class data.
- A set of operations that portray the behavior of the objects of the class. Operations are also referred as functions or methods.

### *Example*

Let us consider a simple class, Circle, that represents the geometrical figure circle in a two-dimensional space. The attributes of this class can be identified as follows –



- x-coord, to denote x-coordinate of the center
- y-coord, to denote y-coordinate of the center
- a, to denote the radius of the circle

Some of its operations can be defined as follows –

- findArea(), method to calculate area
- findCircumference(), method to calculate circumference
- scale(), method to increase or decrease the radius

During instantiation, values are assigned for at least some of the attributes. If we create an object my\_circle, we can assign values like x-coord : 2, y-coord : 3, and a : 4 to depict its state. Now, if the operation scale() is performed on my\_circle with a scaling factor of 2, the value of the variable a will become 8. This operation brings a change in the state of my\_circle, i.e., the object has exhibited certain behavior.

## 2.4 Data Abstraction

Data abstraction is one of the most essential and important feature of object oriented programming. Abstraction means displaying only essential information and hiding the details. Data abstraction refers to providing only essential information about the data to the outside world, hiding the background details or implementation.

Consider a real-life example of a man driving a car. The man only knows that pressing the accelerators will increase the speed of car or applying brakes will stop the car but he does not know about how on pressing accelerator the speed is actually increasing, he does not know about the inner mechanism of the car or the implementation of accelerator, brakes etc in the car. This is what abstraction is.

## 2.5 Encapsulation

In normal terms Encapsulation is defined as wrapping up of data and information under a single unit. In Object Oriented Programming, Encapsulation is defined as binding together the data and the functions that manipulates them.

Consider a real-life example of encapsulation, in a company there are different sections like the accounts section, finance section, sales section etc. The finance section handles all the financial



transactions and keep records of all the data related to finance. Similarly, the sales section handles all the sales related activities and keep records of all the sales. Now there may arise a situation when for some reason an official from finance section needs all the data about sales in a particular month. In this case, he is not allowed to directly access the data of sales section. He will first have to contact some other officer in the sales section and then request him to give the particular data. This is what encapsulation is. Here the data of sales section and the employees that can manipulate them are wrapped under a single name “sales section”.

Encapsulation also lead to data abstraction or hiding. As using encapsulation also hides the data. In the above example the data of any of the section like sales, finance or accounts is hidden from any other section.

***Difference between Abstraction and Encapsulation:***

S.NO	Abstraction	Encapsulation
1.	Abstraction is the process or method of gaining the information.	While encapsulation is the process or method to contain the information.
2.	In abstraction, problems are solved at the design or interface level.	While in encapsulation, problems are solved at the implementation level.
3.	Abstraction is the method of hiding the unwanted information.	Whereas encapsulation is a method to hide the data in a single entity or unit along with a method to protect information from outside.
4.	We can implement abstraction using abstract class and interfaces.	Whereas encapsulation can be implemented using by access modifier i.e. private, protected and public.
5.	In abstraction, implementation complexities are hidden using abstract classes and interfaces.	While in encapsulation, the data is hidden using methods of getters and setters.
6.	The objects that help to perform	Whereas the objects that result in





	abstraction are encapsulated.	encapsulation need not be abstracted.
--	-------------------------------	---------------------------------------

## 2.6 Message Passing

Any application requires a number of objects interacting in a harmonious manner. Objects in a system may communicate with each other using message passing. Suppose a system has two objects: obj1 and obj2. The object obj1 sends a message to object obj2, if obj1 wants obj2 to execute one of its methods.

The features of message passing are –

- Message passing between two objects is generally unidirectional.
- Message passing enables all interactions between objects.
- Message passing essentially involves invoking class methods.
- Objects in different processes can be involved in message passing.

## 2.7 Inheritance

Inheritance is the mechanism that permits new classes to be created out of existing classes by extending and refining its capabilities. The existing classes are called the base classes/parent classes/super-classes, and the new classes are called the derived classes/child classes/subclasses. The subclass can inherit or derive the attributes and methods of the super-class(es) provided that the super-class allows so. Besides, the subclass may add its own attributes and methods and may modify any of the super-class methods. Inheritance defines an “is – a” relationship.

### Example

From a class Mammal, a number of classes can be derived such as Human, Cat, Dog, Cow, etc. Humans, cats, dogs, and cows all have the distinct characteristics of mammals. In addition, each has its own particular characteristics. It can be said that a cow “is – a” mammal.

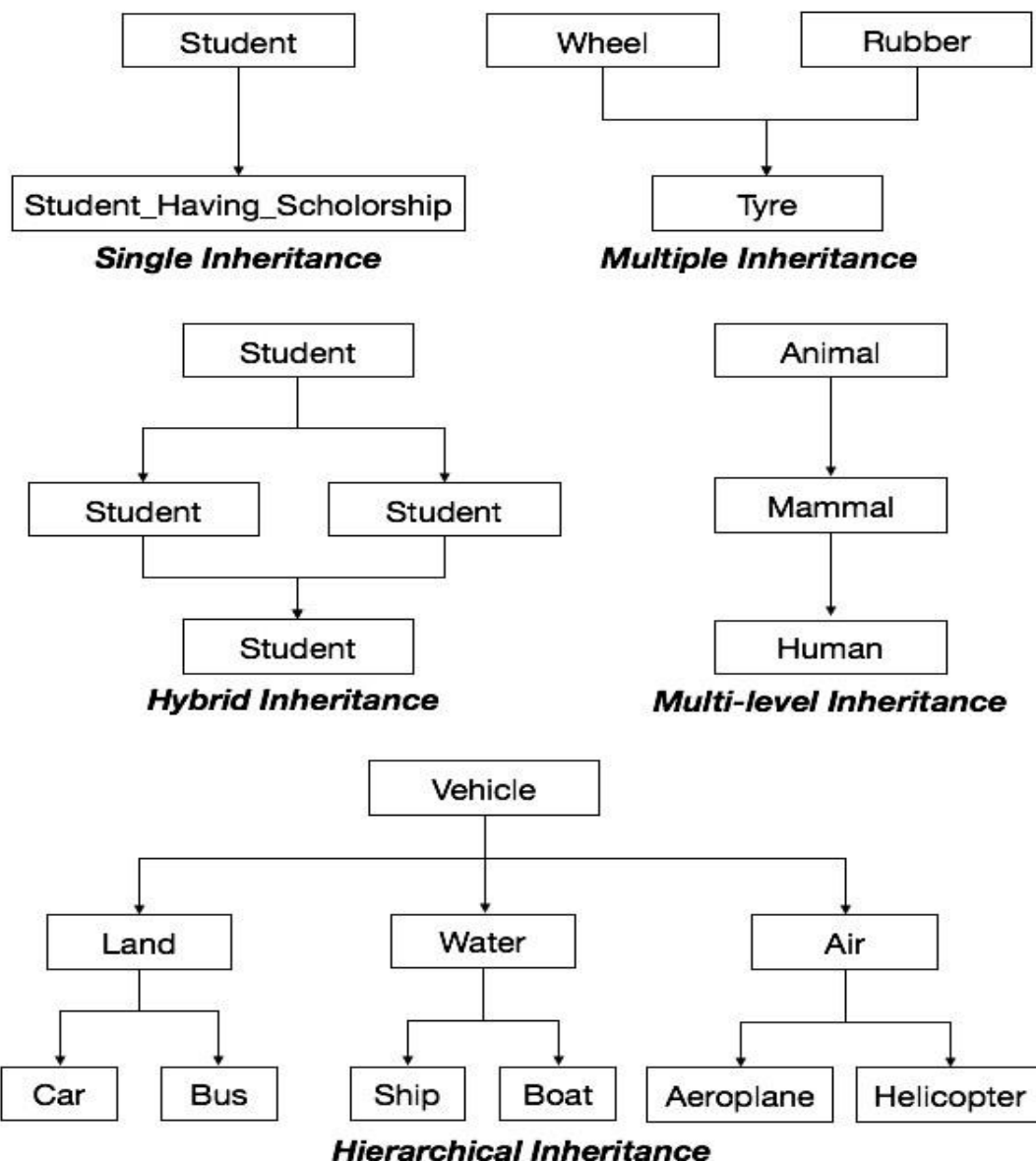
### Types of Inheritance

- **Single Inheritance** – A subclass derives from a single super-class.
- **Multiple Inheritance** – A subclass derives from more than one super-classes.
- **Multilevel Inheritance** – A subclass derives from a super-class which in turn is derived from another class and so on.



- **Hierarchical Inheritance** – A class has a number of subclasses each of which may have subsequent subclasses, continuing for a number of levels, so as to form a tree structure.
- **Hybrid Inheritance** – A combination of multiple and multilevel inheritance so as to form a lattice structure.

The following figure depicts the examples of different types of inheritance.





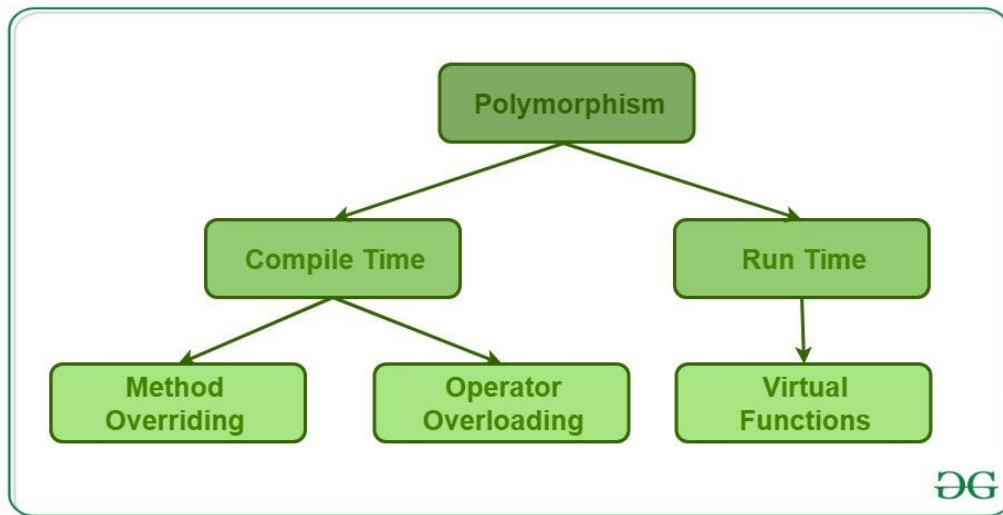
## 2.8 Polymorphism

Polymorphism is the ability of any data to be processed in more than one form. The word itself indicates the meaning as poly means many and morphism means types. Polymorphism is one of the most important concept of object oriented programming language. The most common use of polymorphism in object-oriented programming occurs when a parent class reference is used to refer to a child class object. Here we will see how to represent any function in many types and many forms.

Real life example of polymorphism, a person at the same time can have different roles to play in life. Like a woman at the same time is a mother, a wife, an employee and a daughter. So, the same person has to have many features but has to implement each as per the situation and the condition. Polymorphism is considered as one of the important features of Object-Oriented Programming.

Polymorphism is the key power of object-oriented programming. It is so important that languages that don't support polymorphism cannot advertise themselves as Object-Oriented languages. Languages that possess classes but have no ability of polymorphism are called object-based languages. Thus, it is very vital for an object-oriented programming language.

It is the ability of an object or reference to take many forms in different instances. It implements the concept of function overloading, function overriding and virtual functions.



*“Polymorphism is a property through which any message can be sent to objects of multiple classes, and every object has the tendency to respond in an appropriate way depending on the class properties.”*



This means that polymorphism is the method in an object-oriented programming language that does different things depending on the class of the object which calls it. For example, `$square->area()` will return the area of a square, but `$triangle->area()` might return the area of a triangle. On the other hand, `$object->area()` would have to calculate the area according to which class `$object` was called.

## 2.9 GENERALIZATION AND SPECIALIZATION

Generalization and specialization represent a hierarchy of relationships between classes, where subclasses inherit from super-classes.

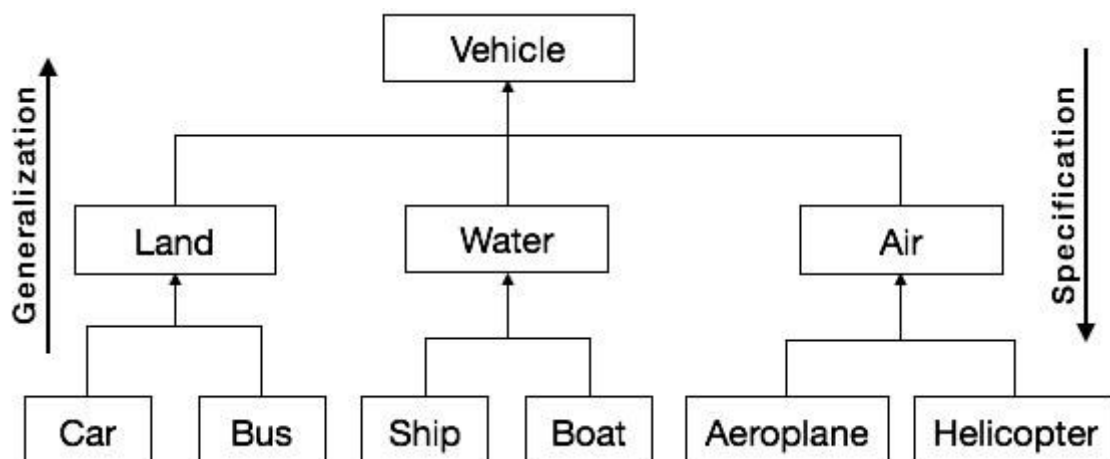
### 2.9.1 Generalization

In the generalization process, the common characteristics of classes are combined to form a class in a higher level of hierarchy, i.e., subclasses are combined to form a generalized super-class. It represents an “is – a – kind – of” relationship. For example, “car is a kind of land vehicle”, or “ship is a kind of water vehicle”.

### 2.9.2 Specialization

Specialization is the reverse process of generalization. Here, the distinguishing features of groups of objects are used to form specialized classes from existing classes. It can be said that the subclasses are the specialized versions of the super-class.

The following figure shows an example of generalization and specialization.



## 2.10 MODULARITY



Modularity is the process of decomposing a problem (program) into a set of modules so as to reduce the overall complexity of the problem. Booch has defined modularity as –

“Modularity is the property of a system that has been decomposed into a set of cohesive and loosely coupled modules.”

Modularity is intrinsically linked with encapsulation. Modularity can be visualized as a way of mapping encapsulated abstractions into real, physical modules having high cohesion within the modules and their inter-module interaction or coupling is low.

### 2.11 TYPING

According to the theories of abstract data type, a type is a characterization of a set of elements. In OOP, a class is visualized as a type having properties distinct from any other types. Typing is the enforcement of the notion that an object is an instance of a single class or type. It also enforces that objects of different types may not be generally interchanged; and can be interchanged only in a very restricted manner if absolutely required to do so.

The two types of typing are –

- **Strong Typing** – Here, the operation on an object is checked at the time of compilation, as in the programming language Eiffel.
- **Weak Typing** – Here, messages may be sent to any class. The operation is checked only at the time of execution, as in the programming language Smalltalk.

### 2.12 CONCURRENCY

Concurrency in operating systems allows performing multiple tasks or processes simultaneously. When a single process exists in a system, it is said that there is a single thread of control. However, most systems have multiple threads, some active, some waiting for CPU, some suspended, and some terminated. Systems with multiple CPUs inherently permit concurrent threads of control; but systems running on a single CPU use appropriate algorithms to give equitable CPU time to the threads so as to enable concurrency.

In an object-oriented environment, there are active and inactive objects. The active objects have independent threads of control that can execute concurrently with threads of other objects. The active objects synchronize with one another as well as with purely sequential objects.



## 2.13 PERSISTENCE

An object occupies a memory space and exists for a particular period of time. In traditional programming, the lifespan of an object was typically the lifespan of the execution of the program that created it. In files or databases, the object lifespan is longer than the duration of the process creating the object. This property by which an object continues to exist even after its creator ceases to exist is known as persistence.

## 2.14 HIERARCHY

In Grady Booch's words, "Hierarchy is the ranking or ordering of abstraction". Through hierarchy, a system can be made up of interrelated subsystems, which can have their own subsystems and so on until the smallest level components are reached. It uses the principle of "divide and conquer". Hierarchy allows code reusability.

The two types of hierarchies in OOA are –

- **“IS–A” hierarchy** – It defines the hierarchical relationship in inheritance, whereby from a super-class, a number of subclasses may be derived which may again have subclasses and so on. For example, if we derive a class Rose from a class Flower, we can say that a rose “is–a” flower.
- **“PART–OF” hierarchy** – It defines the hierarchical relationship in aggregation by which a class may be composed of other classes. For example, a flower is composed of sepals, petals, stamens, and carpel. It can be said that a petal is a “part–of” flower.

## SUMMARY

In an object-oriented system, all data is represented as discrete objects with which the user and other objects may interact. Each object contains data as well as information about the executable file needed to interpret that data. An object-oriented system allows the user to focus completely on tasks rather than tools.

The conceptual framework of object-oriented systems is based upon the object model. There are two categories of elements in an object-oriented system:

Major Elements: by major, it is meant that if a model does not have any one of these elements, it ceases



to be object oriented. The four major elements are:

- Abstraction
- Encapsulation
- Modularity
- Hierarchy

Minor Elements: By minor, it is meant that these elements are useful, but not indispensable part of the object model. The three minor elements are:

- Typing
- Concurrency
- Persistence

### **Multiple Choice Questions**

Q1. Which was the first purely object oriented programming language developed?

- a) Java
- b) C++
- c) SmallTalk
- d) Kotlin

Q2. Which of the following best defines a class?

- a) Parent of an object
- b) Instance of an object
- c) Blueprint of an object
- d) Scope of an object

Q3. Who invented OOP?

- a) Alan Kay
- b) Andrea Ferro
- c) Dennis Ritchie



d) Adele Goldberg

Q4. Which is not feature of OOP in general definitions?

a) Code reusability

b) Modularity

c) Duplicate/Redundant data

d) Efficient Code

Q5. Which Feature of OOP illustrated the code reusability?

a) Polymorphism

b) Abstraction

c) Encapsulation

d) Inheritance

Ans: 1- c 2-c 3-a 4-c 5-d

### **Long Answer Type Questions**

Q1. Define the concept of OOP. Explain all characteristics of OOP.

Q2. What is data encapsulation? How it is achieved in OOP?

Q3. Define polymorphism. What are the different ways of using Polymorphism.

Q4. Define the terms generalization and specialization of object oriented programming.

Q5. What is reusability of code. What are different types of inheritance. How inheritance reduces the efforts of the program developer?

### **Refences**

1. Object-Oriented Modeling and Design with UML, M. Blaha, J.Rumbaugh, Pearson Education-2007
2. Object-Oriented Analysis & Design with the Unified Process, Satzinger, Jackson, Burd, Thomson-2007
3. ObjectOrientedAnalysis&Design, GradyBooch, AddisonWesley-1994.
4. TimothyC.Lethbridge, RobertLaganiere, ObjectOrientedSoftware Engineering, TMH, 2004





Subject Name: <b>Object Oriented Programming Using C++</b>	
Course Code: <b>DCA-15-T</b>	Author:
Lesson No. 3	Vetter:
<b>CONTROL STRUCTURE AND LOOPS</b>	

- 3.1 Introduction
- 3.2 Conditional/Branch statement
  - 3.2.1 If-Statement
  - 3.2.2 Switch-Case Statement
- 3.3 Loop
  - 3.3.1 While
  - 3.3.2 do
  - 3.3.3 For
- 3.4 Breaking Control Statement
  - 3.4.1 Breaking Statement
  - 3.4.2 Continuous Statement
  - 3.4.3 Go-to Statement
- 3.5 Check Your Progress
- 3.6 Summary
- 3.7 Self-Assessment Test
- 3.8 Answers to check your progress
- 3.9 References/ Suggested Readings



## 3.1 INTRODUCTION

### Control Flow Construct Statements

Usually a program is not a linear sequence of instructions. It may repeat code or take decisions for a given path-goal relation. Most programming languages have control flow statements (constructs) which provide some sort of control structures that serve to specify order to what has to be done to perform our program that allow variations in this sequential order:

- statements may only be obeyed under certain conditions (conditionals),
- statements may be obeyed repeatedly under certain conditions (loops),
- a group of remote statements may be obeyed (subroutines).

Logical Expressions as conditions Logical expressions can use logical operators in loops and conditional statements as part of the conditions to be met.

## 3.2 CONDITIONAL/BRANCH STATEMENT

There is likely no meaningful program written in which a computer does not demonstrate basic decision-making skills. It can actually be argued that there is no meaningful human activity in which no decision making, instinctual or otherwise, takes place. For example, when driving a car and approaching a traffic light, one does not think, "I will continue driving through the intersection." Rather, one thinks, "I will stop if the light is red, go if the light is green, and if yellow go only if I am traveling at a certain speed a certain distance from the intersection." These kinds of processes can be simulated using conditionals. A conditional is a statement that instructs the computer to execute a certain block of code or alter certain data only if a specific condition has been met. The most common conditional is the if-else statement, with conditional expressions and switch-case statements typically used as more shorthand methods.

### 3.2.1 If-Statement



if (Fork branching)

The if-statement allows one possible path choice depending on the specified conditions.

### Syntax

if (condition)

```
{  
    statement;  
}
```

### Semantic

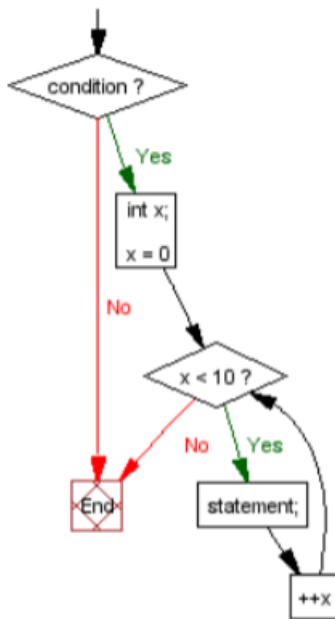
First, the condition is evaluated:

- if condition is true, statement is executed before continuing with the body.
- if condition is false, the program skips statement and continues with the rest of the program.

Note: The statements in an if statement can be any code that's valid; you can declare variables, nest statements, etc. Example:

if(condition)

```
{  
    int x; // Valid code  
    for(x = 0; x < 10; ++x) // Also valid.  
    {  
        statement;  
    }  
}
```



If you wish to avoid typing `std::cout`, `std::cin`, or `std::endl`; all the time, you may include using namespace `std` at the beginning of your program since `cout`, `cin`, and `endl` are members of the `std` namespace. Sometimes the program needs to choose one of two possible paths depending on a condition. For this we can use the `if-else` statement.

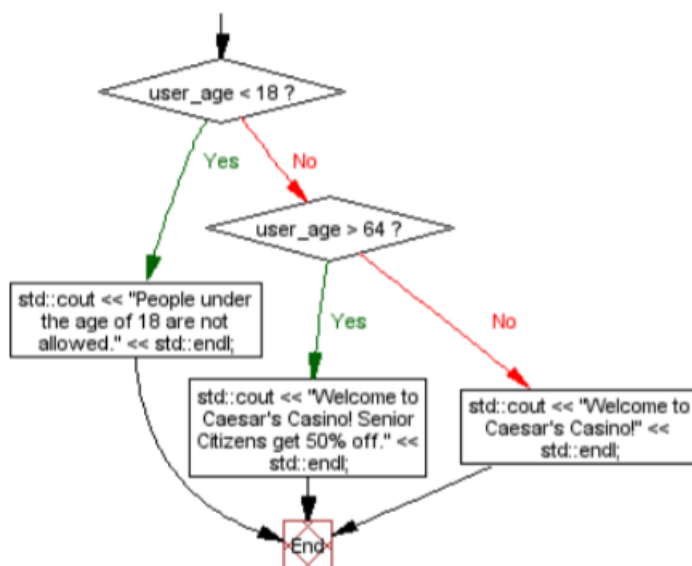
```
if (user_age < 18)
{
    std::cout << "People under the age of 18 are not allowed." << std::endl;
}
else
{
    std::cout << "Welcome to Caesar's Casino!" << std::endl;
}
```

Here we display a message if the user is under 18. Otherwise, we let the user in. The `if` part is executed only if '`user_age`' is less than 18. In other cases (when '`user_age`' is greater than or equal to 18), the `else` part is executed



if conditional statements may be chained together to make for more complex condition branching. In this example we expand the previous example by also checking if the user is above 64 and display another message if so.

```
if (user_age < 18)
{
std::cout << "People under the age of 18 are not allowed." << std::endl;
}
else if (user_age > 64)
{
std::cout << "Welcome to Caesar's Casino! Senior Citizens get 50% off." << std::endl;
}
else
{
std::cout << "Welcome to Caesar's Casino!" << std::endl;
}
}
```



**Example of if statement:**

```
#include<iostream>

using namespace std;

int main(){
    int num=70;
    if( num <100){
        /* This cout statement will only execute,
         * if the above condition is true
         */
        cout<<"number is less than 100";
    }

    if(num >100){
        /* This cout statement will only execute,
         * if the above condition is true
         */
        cout<<"number is greater than 100";
    }
    return 0;
}
```

**Nested if statement in C++**

When there is an if statement inside another if statement then it is called the **nested if statement**. The structure of nested if looks like this:

```
if(condition_1)
```



```
{  
Statement1(s);  
  
if(condition_2)  
{  
Statement2(s);  
}  
}
```

Statement1 would execute if the condition\_1 is true. Statement2 would only execute if both the conditions( condition\_1 and condition\_2) are true.

### Example of Nested if statement

```
#include<iostream>  
usingnamespace std;  
intmain(){  
int num=90;  
  
/* Nested if statement. An if statement  
   * inside another if body  
   */  
  
if( num <100){  
    cout<<"number is less than 100"<<endl;  
    if(num >50){  
        cout<<"number is greater than 50";  
    }  
}  
  
return0;
```



```
}
```

**Output:**

number is less than 100

number is greater than 50

**If else statement**

Sometimes you have a condition and you want to execute a block of code if condition is true and execute another piece of code if the same condition is false. This can be achieved in C++ using if-else statement.

This is how an if-else statement looks:

```
if(condition){  
    Statement(s);  
}  
else{  
    Statement(s);  
}
```

The statements inside “if” would execute if the condition is true, and the statements inside “else” would execute if the condition is false.

**if-else-if Statement**

if-else-if statement is used when we need to check multiple conditions. In this control structure we have only one “if” and one “else”, however we can have multiple “else if” blocks. This is how it looks:

```
if(condition_1){  
    /*if condition_1 is true execute this*/  
    statement(s);  
}
```





```
elseif(condition_2){  
    /* execute this if condition_1 is not met and  
     * condition_2 is met  
     */  
    statement(s);  
}  
elseif(condition_3){  
    /* execute this if condition_1 & condition_2 are  
     * not met and condition_3 is met  
     */  
    statement(s);  
}  
.  
.  
.  
else{  
    /* if none of the condition is true  
     * then these statements gets executed  
     */  
    statement(s);  
}
```

**Note:** The most important point to note here is that in if-else-if, as soon as the condition is met, the corresponding set of statements get executed, rest gets ignored. If none of the condition is met then the statements inside “else” gets executed.

**Example of if-else-if**

```
#include<iostream>

using namespace std;

int main()
{
    int num;

    cout<<"Enter an integer number between 1 & 99999: ";

    cin>>num;

    if(num <100&& num>=1){
        cout<<"Its a two digit number";
    }

    elseif(num <1000&& num>=100){
        cout<<"Its a three digit number";
    }

    elseif(num <10000&& num>=1000){
        cout<<"Its a four digit number";
    }

    elseif(num <100000&& num>=10000){
        cout<<"Its a five digit number";
    }

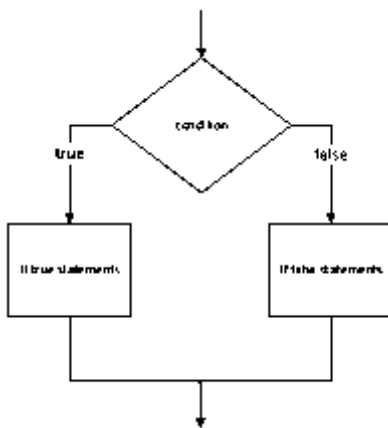
    else{
        cout<<"number is not between 1 & 99999";
    }

    return 0;
}
```

**Output:**

Enter an integer number between 1&999999:8976

Its a four digit number

**Flow diagram of if-else****Example of if-else statement**

```
#include<iostream>

using namespace std;

int main(){
    int num=66;
    if( num <50){
        //This would run if above condition is true
        cout<<"num is less than 50";
    }
    else{
        //This would run if above condition is false
        cout<<"num is greater than or equal 50";
    }
}
```



```
return 0;
```

```
}
```

**Output:**

num is greater than or equal 50

**3.2.2 Switch-Case Statement (Multiple branching)**

The switch statement branches based on specific integer values.

```
switch (integer expression)
```

```
{
```

```
case label1:
```

```
statement(s)
```

```
break;
```

```
case label2:
```

```
statement(s)
```

```
break; ...
```

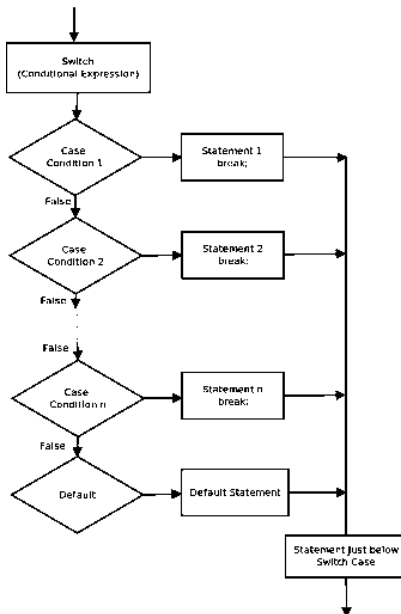
```
default:
```

```
statement(s)
```

```
break;
```

```
}
```

As you can see in the above scheme the case and default have a "break;" statement at the end of block. This expression will cause the program to exit from the switch, if break is not added the program will continue execute the code in other cases even when the integer expression is not equal to that case.

**Example:**

```
#include <iostream>

using namespace std;

int main(){

    int num=5;

    switch(num+2) {

        case 1:

            cout<<"Case1: Value is: "<<num<<endl;

        case 2:

            cout<<"Case2: Value is: "<<num<<endl;

        case 3:

            cout<<"Case3: Value is: "<<num<<endl;

        default:

            cout<<"Default: Value is: "<<num<<endl;

    }
```



```
    return 0;  
}
```

**Output:**

Default:Valueis:5

### 3.3 LOOP(ITERATIONS)

A loop (also referred to as an iteration or repetition) is a sequence of statements which is specified once but which may be carried out several times in succession. The code "inside" the loop (the body of the loop) is obeyed a specified number of times, or once for each of a collection of items, or until some condition is met. Iteration is the repetition of a process, typically within a computer program. Confusingly, it can be used both as a general term, synonymous with repetition, and to describe a specific form of repetition with a mutable state. When used in the first sense, recursion is an example of iteration. However, when used in the second (more restricted) sense, iteration describes the style of programming used in imperative programming languages. This contrasts with recursion, which has a more declarative approach. Due to the nature of C++ there may lead to an even bigger problems when differentiating the use of the word, so to simplify things use "loops" to refer to simple recursions as described in this section and use iteration or iterator (the "one" that performs an iteration) to class iterator (or in relation to objects/classes) as used in the STL.

**Infinite Loops**

Sometimes it is desirable for a program to loop forever, or until an exceptional condition such as an error arises. For instance, an event-driven program may be intended to loop forever handling events as they occur, only stopping when the process is killed by the operator. More often, an infinite loop is due to a programming error in a condition-controlled loop, wherein the loop condition is never changed within the loop.

**Condition-controlled loops**

Most programming languages have constructions for repeating a loop until some condition changes. Condition-controlled loops are divided into two categories Preconditional or Entry-Condition that place



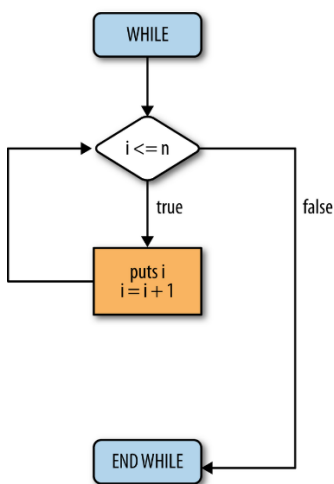
the test at the start of the loop, and Postconditional or Exit-Condition iteration that have the test at the end of the loop. In the former case the body may be skipped completely, while in the latter case the body is always executed at least once

### 3.3.1 while (Preconditional loop)

#### Syntax

```
while ("condition") "statement"; "statement2";
```

#### Flow Chart



#### Semantic

First, the condition is evaluated:

1. if condition is true, statement is executed and condition is evaluated again.
2. if condition is false continues with statement2

Remark: statement can be a block of code { ... } with several instructions. What makes 'while' statements different from the 'if' is the fact that once the body (referred to as statement above) is executed, it will go back to 'while' and check the condition again. If it is true, it is executed again. In fact, it will execute as many times as it has to until the expression is false.

#### Example 1

```
#include <iostream>
```

```
using namespace std;
```



```
int main()
{
    int i=0;
    while (i<< "The value of i is " << i << endl;
    i++;
}
cout << "The final value of i is : " << i<< endl;
return 0;
}
```

#### Execution

The value of i is 0

The value of i is 1

The value of i is 2

The value of i is 3

The value of i is 4

The value of i is 5

The value of i is 6

The value of i is 7

The value of i is 8

The value of i is 9

The final value of i is 10

#### **Example 2 // validation of an input #include using namespace std;**

```
int main()
{
    int a;
```





```
bool ok=false;
while (!ok)
{
cout << "Type an integer from 0 to 20 : ";
cin >> a;
ok = ((a>=0) && (a<=20));
if (!ok) cout << "ERROR - ";
}
return 0;
}
```

#### Execution

Type an integer from 0 to 20 : 30

ERROR - Type an integer from 0 to 20 : 40

ERROR - Type an integer from 0 to 20 : -6

ERROR - Type an integer from 0 to 20 : 14

#### 3.3.2 do-while (Postconditional loop)Syntax

```
do
{
statement(s)
}
while (condition);
statement2;
```

#### Semantic

1. statement(s) are executed.
2. condition is evaluated.



3. if condition is true goes to 1).

4. if condition is false continues with statement2

The do - while loop is similar in syntax and purpose to the while loop. The construct moves the test that continues condition of the loop to the end of the code block so that the code block is executed at least once before any evaluation.

Example

```
#include  
  
using namespace std;  
  
int main()  
{  
    int i=0;  
    do  
    {  
        cout << "The value of i is " << i << endl;  
        i++;  
    }  
    while (i<< "The final value of i is : " << i << endl;  
    return 0;  
}
```

Execution

The value of i is 0

The value of i is 1

The value of i is 2

The value of i is 3

The value of i is 4



The value of i is 5

The value of i is 6

The value of i is 7

The value of i is 8

The value of i is 9

The final value of i is 10

### 3.3.3 for (Preconditional and Counter-controlled loop)

The for loop is designed to iterate a number of times. Its syntax is: for (initialization; condition; increase) statement;

Like the while-loop, this loop repeats statement while condition is true. But, in addition, the for loop provides specific locations to contain an initialization and an increase expression, executed before the loop begins the first time, and after each iteration, respectively. Therefore, it is especially useful to use counter variables as condition.

It works in the following way:

1. initialization is executed. Generally, this declares a counter variable, and sets it to some initial value. This is executed a single time, at the beginning of the loop.
2. condition is checked. If it is true, the loop continues; otherwise, the loop ends, and statement is skipped, going directly to step 5.
3. statement is executed. As usual, it can be either a single statement or a block enclosed in curly braces { }.
4. increase is executed, and the loop gets back to step 2.
5. the loop ends: execution continues by the next statement after it.

Here is the countdown example using a for loop:

```
// countdown using a for loop
```

```
#include <iostream>
```



```
using namespace std;
```

```
int main ()  
{  
    for (int n=10; n>0; n--) {  
        cout << n << ", ";  
    }  
    cout << "Hello\n";  
}
```

Output:

10, 9, 8, 7, 6, 5, 4, 3, 2, 1, Hello!

The three fields in a for-loop are optional. They can be left empty, but in all cases the semicolon signs between them are required. For example, for (;n<10;) is a loop without *initialization* or *increase* (equivalent to a while-loop); and for (;n<10;++n) is a loop with *increase*, but no *initialization* (maybe because the variable was already initialized before the loop). A loop with no *condition* is equivalent to a loop with true as condition (i.e., an infinite loop). Because each of the fields is executed in a particular time in the life cycle of a loop, it may be useful to execute more than a single expression as any of *initialization*, *condition*, or *statement*. Unfortunately, these are not statements, but rather, simple expressions, and thus cannot be replaced by a block. As expressions, they can, however, make use of the comma operator (,): This operator is an expression separator, and can separate multiple expressions where only one is generally expected. For example, using it, it would be possible for a for loop to handle two counter variables, initializing and increasing both:

```
for ( n=0, i=100 ; n!=i ; ++n, --i )  
{  
    // whatever here...
```



```
}
```

This loop will execute 50 times if neither `n` or `i` are modified within the loop:

```
for ( n=0, i=100 ; n!=i ; ++n, --i )
```

`n` starts with a value of 0, and `i` with 100, the condition is `n!=i` (i.e., that `n` is not equal to `i`). Because `n` is increased by one, and `i` decreased by one on each iteration, the loop's condition will become false after the 50th iteration, when both `n` and `i` are equal to 50.

### ***Range-based for loop***

The for-loop has another syntax, which is used exclusively with ranges: `for ( declaration : range ) statement;`

This kind of for loop iterates over all the elements in range, where declaration declares some variable able to take the value of an element in this range. Ranges are sequences of elements, including arrays, containers, and any other type supporting the functions `begin` and `end`; Most of these types have not yet been introduced in this tutorial, but we are already acquainted with at least one kind of range: strings, which are sequences of characters.

An example of range-based for loop using strings:

```
// range-based for loop
#include <iostream>
#include <string>
using namespace std;

int main ()
{
    string str {"Hello!"};
    for (char c : str)
```



```
{  
    cout << "[" << c << " ]";  
}  
  
cout << "\n";  
}
```

Output:

[H][e][l][l][o][!]

Note how what precedes the colon (:) in the for loop is the declaration of a char variable (the elements in a string are of type char). We then use this variable, *c*, in the statement block to represent the value of each of the elements in the range. This loop is automatic and does not require the explicit declaration of any counter variable. Range based loops usually also make use of type deduction for the type of the elements with *auto*. Typically, the range-based loop above can also be written as:

```
1 for (auto c : str)  
2     cout << "[" << c << " ]";
```

Here, the type of *c* is automatically deduced as the type of the elements in *str*.

### 3.4 BREAKING CONTROL STATEMENTS

The *goto* statement is strongly discouraged as it makes it difficult to follow the program logic, this way inducing to errors. In some (mostly rare) cases the *goto* statement allows to write uncluttered code, for example, when handling multiple exit points leading to the cleanup code at a function exit (and exception handling is not a better option). Except in those rare cases, the use of unconditional jumps is a frequent symptom of a complicated design, as the presence of many levels of nested statements. In exceptional cases, like heavy optimization, a programmer may need more control over code behavior; a *goto* allows the programmer to specify that execution flow jumps directly and unconditionally to a desired label.



### 3.4.1 Breaking Statement

The **break** statement has the following two usages in C++ –

- When the **break** statement is encountered inside a loop, the loop is immediately terminated and program control resumes at the next statement following the loop.
- It can be used to terminate a case in the **switch** statement (covered in the next chapter).

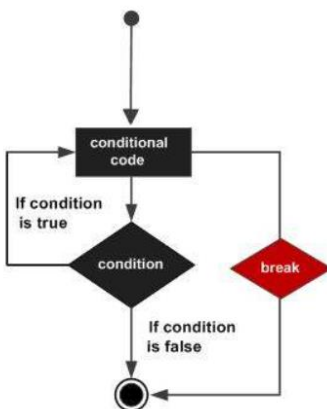
If you are using nested loops (i.e., one loop inside another loop), the break statement will stop the execution of the innermost loop and start executing the next line of code after the block.

#### Syntax

The syntax of a break statement in C++ is –

```
break;
```

#### Flow Diagram



Example:

```
#include<iostream>

usingnamespace std;
```

```
int main (){
```

```
// Local variable declaration:
```



```
int a =10;

// do loop execution
do{
    cout <<"value of a: "<< a << endl;
    a = a +1;
    if( a >15){
        // terminate the loop
        break;
    }
}while( a <20);

return0;
}
```

Output:

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 15
```

### 3.4.2 The 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.

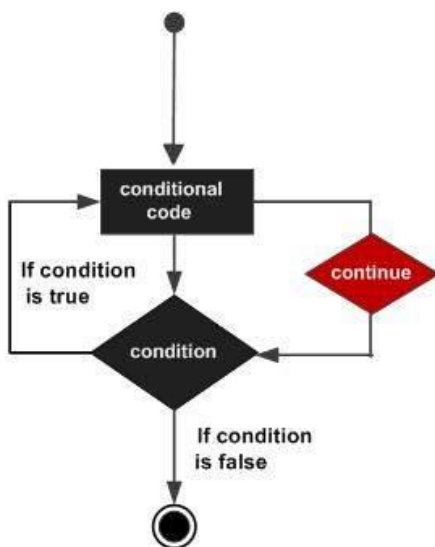
For the **for** loop, continue causes the conditional test and increment portions of the loop to execute. For the **while** and **do...while** loops, program control passes to the conditional tests.

### 2.10 Syntax

The syntax of a continue statement in C++ is –

continue;

### 2.11 Flow Diagram



### 2.12 continue statement inside for loop

As you can see that the output is missing the value 3, however the for loop iterate though the num value 0 to 6. This is because we have set a condition inside loop in such a way, that the continue statement is encountered when the num value is equal to 3. So for this iteration the loop skipped the cout statement and started the next iteration of loop.

```
#include<iostream>

usingnamespace std;

int main(){

for(int num=0; num<=6; num++){
```



```
/* This means that when the value of  
    * num is equal to 3 this continue statement  
    * would be encountered, which would make the  
    * control to jump to the beginning of loop for  
    * next iteration, skipping the current iteration  
    */
```

```
if(num==3){  
    continue;  
}  
  
    cout<<num<<" ";  
}  
return 0;  
}
```

**Output:**

012456

**Use of continue in While loop**

```
#include<iostream>  
  
using namespace std;  
  
int main(){  
    int j=6;  
    while(j >=0){  
        if(j==4){  
            j--;
```



```
continue;
}
    cout<<"Value of j: "<<j<<endl;
    j--;
}
return 0;
}
```

**Output:**

```
Value of j:6
Value of j:5
Value of j:3
Value of j:2
Value of j:1
Value of j:0
```

**Example of continue in do-While loop**

```
#include<iostream>
using namespace std;
int main(){
    int j=4;
    do{
        if(j==7){
            j++;
            continue;
        }
        cout<<"j is: "<<j<<endl;
```



```
j++;  
}while(j<10);  
return0;  
}
```

**Output:**

```
j is:4  
j is:5  
j is:6  
j is:8  
j is:9
```

**3.4.3 Goto Statement**

The goto statement is used for transferring the control of a program to a given label. The syntax of goto statement looks like this:

```
goto label_name;
```

**Program structure:**

```
label1:  
...  
...  
goto label2;  
...  
..  
label2:  
...
```



In a program we have any number of goto and label statements, the goto statement is followed by a label name, whenever goto statement is encountered, the control of the program jumps to the label specified in the goto statement. goto statements are almost never used in any development as they are complex and makes your program much less readable and more error prone. In place of goto, you can use continue and break statement.

**Example of goto statement in C++**

```
#include<iostream>

using namespace std;

int main(){

int num; cout<<"Enter a number: "; cin>>num;

if(num %2==0){

gotoprint;

}

else{

    cout<<"Odd Number";

}

print:

    cout<<"Even Number";

return0;

}
```

**Output:**

Enter a number:42

EvenNumber



### 3.5 CHECK YOUR PROGRESS

1. How are many sequences of statements present in C++?
  - a) 4
  - b) 3
  - c) 5
  - d) 6
2. The if..else statement can be replaced by which operator?
  - a) Bitwise operator
  - b) Conditional operator
  - c) Multiplicative operator
  - d) Addition operator
3. The switch statement is also called as?
  - a) choosing structure
  - b) selective structure
  - c) certain structure
  - d) bitwise structure
4. The destination statement for the goto label is identified by what label?
  - a) \$
  - b) @
  - c) \*
  - d) :
5. Which looping process is best used when the number of iterations is known?
  - a) for
  - b) while
  - c) do-while
  - d) all looping processes require that the iterations be known
6. How many types of loops are there in C++?
  - a) 4



- b) 2
- c) 3
- d) 1

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

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

- a) 0123456789
- b) 10
- c) 012345678910
- d) compile time error

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

```
#include <iostream>
using namespace std;
int main()-----
{
    int n =15;
    for(;;)
        cout<< n;
```



```
return0;  
}
```

- a) error
- b) 15
- c) infinite times of printing n
- d) none of the mentioned

### 3.6 SUMMARY

There are three kinds of loops in C++. The for loop is most often used when you know in advance how many times you want to execute the loop. The while loop and do loops are used when the condition causing the loop to terminate arises within the loop, with the while loop not necessarily executing at all, and the do loop always executing at least once. A loop body can be a single statement or a block of multiple statements delimited by braces. A variable defined within a block is visible only within that block. There are four kinds of decision-making statements. The if statement does something if a test expression is true. The if...else statement does one thing if the test expression is true, and another thing if it isn't. The else if construction is a way of rewriting a ladder of nested if...else statements to make it more readable. The switch statement branches to multiple sections of code, depending on the value of a single variable. The conditional operator simplifies returning one value if a test expression is true, and another if it's false. The logical AND and OR operators combine two Boolean expressions to yield another one, and the logical NOT operator changes a Boolean value from true to false, or from false to true. The break statement sends control to the end of the innermost loop or switch in which it occurs. The continue statement sends control to the top of the loop in which it occurs. The goto statement sends control to a label. Precedence specifies which kinds of operations will be carried out first. The order is unary, arithmetic, relational, logical, conditional, assignment.

### 3.7 SELF ASSESSMENT TEST

Ques:-1 Name and describe the usual purpose of three expressions in a for statement.

Ques:-2 Write a while loop that displays the numbers from 100 to 110.





Ques:-3 Explain the control Structure in detail with example.

Ques:- Explain goto statement in detail.

Ques:- What are various conditional statements available in C++?

### **3.8 ANSWER TO CHECK YOUR PROGRESS**

Answer:1 c

Answer:2 b

Answer:3 b

Answer:4 d

Answer:5 a

Answer:6 a

Answer:7 b

Answer:8 c

### **3.9 REFERENCES/SUGGESTED READING**



Subject Name: <b>Object Oriented Programming Using C++</b>	
Course Code: <b>DCA-15-T</b>	Author:
Lesson No. 4	Vetter:
<b>Introduction to Pointers</b>	

#### **4.1 Introduction to pointers**

#### **4.2 Pointers and address operator**

##### **4.2.1 Pointers and address operator**

#### **4.3 Pointers Expression**

#### **4.4 Pointers arithmetics**

#### **4.5 Pointers and functions**

##### **4.5.1 Address of function**

###### **4.5.1.1 Syntax for Declaration**

###### **4.5.1.2 Calling a function indirectly**

###### **4.5.1.3 Passing a function pointer as a parameter**

#### **4.6 Pointer to function**

#### **4.7 Pointer and array**

#### **4.8 Array of Pointer**

#### **4.9 Pointer to pointer**

#### **4.10 Check your progress**

#### **4.11 Summary**

#### **4.12 Self Assessment Test**

#### **4.13 Answer to check your progress**

#### **4.14 References/suggested reading**



## 4.1 INTRODUCTION TO POINTERS

For a program, the memory of a computer is like a succession of memory cells, each one byte in size, and each with a unique address. These single-byte memory cells are ordered in a way that allows data representations larger than one byte to occupy memory cells that have consecutive addresses. This way, each cell can be easily located in the memory by means of its unique address. For example, the memory cell with the address 1776 always follows immediately after the cell with address 1775 and precedes the one with 1777, and is exactly one thousand cells after 776 and exactly one thousand cells before 2776. When a variable is declared, the memory needed to store its value is assigned a specific location in memory (its memory address). Generally, C++ programs do not actively decide the exact memory addresses where its variables are stored. Fortunately, that task is left to the environment where the program is run - generally, an operating system that decides the particular memory locations on runtime. However, it may be useful for a program to be able to obtain the address of a variable during runtime in order to access data cells that are at a certain position relative to it. Pointer is a variable that holds the address of another variable. They have data type just like variables, for example an integer type pointer can hold the address of an integer variable and a character type pointer can hold the address of a char variable.

Here, are pros/benefits of using Pointers

- Pointers are variables which store the address of other variables in C++.
- More than one variable can be modified and returned by function using pointers.
- Memory can be dynamically allocated and de-allocated using pointers.
- Pointers help in simplifying the complexity of the program.
- The execution speed of a program improves by using pointers.

### Syntax of pointer

```
data_type *pointer_name;
```



Here, data\_**type** is the pointer's base type; it must be a valid C++ type and pointer\_name is the name of the pointer variable. The asterisk you used to declare a pointer is the same asterisk that you use for multiplication. However, in this statement the asterisk is being used to designate a variable as a pointer. Following are the valid pointer declaration –

```
int *ip;    // pointer to an integer
```

```
double *dp; // pointer to a double
```

```
float *fp;  // pointer to a float
```

```
char *ch    // pointer to character
```

```
int*p,var
```

```
/* This pointer p can hold the address of an integer
```

```
* variable, here p is a pointer and var is just a
```

```
* simple integer variable
```

```
*/
```

## 4.2 POINTERS AND ADDRESS OPERATOR

### 4.2.1 Pointers and address operator

#### Address-of operator (&)

The address of a variable can be obtained by preceding the name of a variable with an ampersand sign (&), known as *address-of operator*. For example:

```
foo = &myvar;
```

This would assign the address of variable myvar to foo; by preceding the name of the variable myvar with the *address-of operator* (&), we are no longer assigning the content of the variable itself to foo, but its address.



The actual address of a variable in memory cannot be known before runtime, but let's assume, in order to help clarify some concepts, that `myvar` is placed during runtime in the memory address 1776.

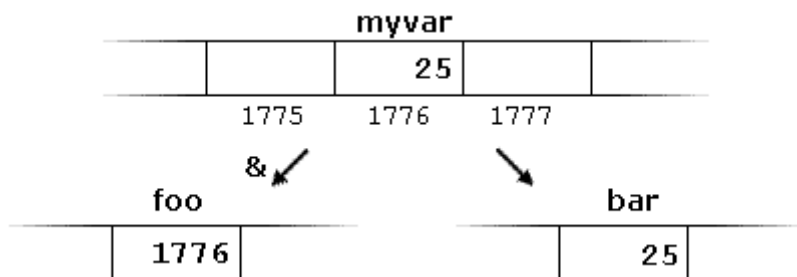
In this case, consider the following code fragment:

```
myvar = 25;
```

```
foo = &myvar;
```

```
bar = myvar;
```

The values contained in each variable after the execution of this are shown in the following diagram:



First, we have assigned the value 25 to `myvar` (a variable whose address in memory we assumed to be 1776).

The second statement assigns `foo` the address of `myvar`, which we have assumed to be 1776.

Finally, the third statement, assigns the value contained in `myvar` to `bar`. This is a standard assignment operation, as already done many times in earlier chapters.

The main difference between the second and third statements is the appearance of the *address-of operator* (`&`).

The variable that stores the address of another variable (like `foo` in the previous example) is what in C++



is called a *pointer*. Pointers are a very powerful feature of the language that has many uses in lower level programming. A bit later, we will see how to declare and use pointers.

### How to use Pointer?

```
// This will print the address of variable var
cout<<&var;

/* This will also print the address of variable
 * var because the pointer p holds the address of var
 */

cout<<p;

/* This will print the value of var, This is
 * important, this is how we access the value of
 * variable through pointer
 */

cout<<*p;
```

### Example:

```
#include<iostream>

usingnamespace std;

int main(){

//Pointer declaration

int*p,var=101;

//Assignment

p =&var;

cout<<"Address of var: "<<&var<<endl;

cout<<"Address of var: "<<p<<endl;

cout<<"Address of p: "<<&p<<endl;
```



```
cout<<"Value of var: "<<*p;
return 0;
}
```

**Output:**

Address of var:0x7fff5dffc0c

Address of var:0x7fff5dffc0c

Address of p:0x7fff5dffc10

Value of var:101

## 4.3 POINTER EXPRESSION

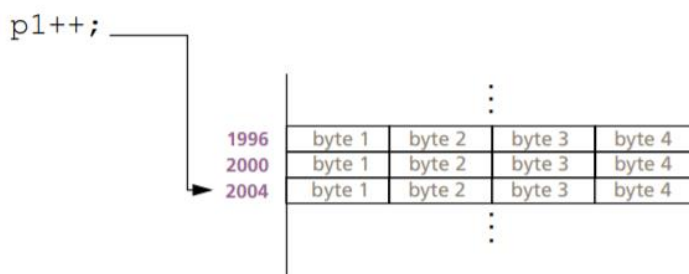
**Pointer Expressions**

Pointers can be used in most C++ expressions. Keep in mind to use parentheses around pointer expressions. Pointer Arithmetic Only four arithmetic operators can be used on pointers:

- ++
- --
- +
- -

Example: (assuming 32-bit integers)

```
int *p1; // assume: p1 == 2000
```





- Integers can be added or subtracted from pointers:
- You can subtract pointers of the same type from one another. You can not add pointers! However, you can add int numbers to pointers:

```
void main()
{
int i[10], *intPtr;
double d[10], *doublePtr;
int x;
intPtr = i; // i_ptr points to first element of i
doublePtr = d; // f_ptr points to first element of f
for(x=0; x < 10; x++)
cout << intPtr + x;
cout << ' ';
cout << doublePtr + x;
cout << endl;
}
```

Output of the example program:

The addresses of the array elements:

4 bytes	int 8 bytes double
0xeffffd9c	0xeffffd48
0xeffffda0	0xeffffd50
0xeffffda4	0xeffffd58





0xeffffda8	0xeffffd60
0xeffffdac	0xeffffd68
0xeffffdb0	0xeffffd70
0xeffffdb4	0xeffffd78
0xeffffdb8	0xeffffd80
0xeffffdbc	0xeffffd88
0xeffffdc0	0xeffffd90

If we want to see the values at these addresses, we have to use the "value at ..." operator (\*):

```
void main()
{
    int i[3]={1,2,3}, *intPtr;
    double d[3]={1.1,2.2,3.3}, *doublePtr; int x;
    intPtr = i; // i_ptr points to first element of i
    doublePtr = d; // f_ptr points to first element of f
    for(x=0; x < 3; x++)
        cout << *(intPtr + x);
    cout << ' ';
    cout << *(doublePtr + x);
    cout << endl;
}
```

## 4.4 POINTER ARITHMETICS

### Pointer Arithmetics

To conduct arithmetical operations on pointers is a little different than to conduct them on regular integer types. To begin with, only addition and subtraction operations are allowed; the others make no



sense in the world of pointers. But both addition and subtraction have a slightly different behavior with pointers, according to the size of the data type to which they point.

When fundamental data types were introduced, we saw that types have different sizes. For example: char always has a size of 1 byte, short is generally larger than that, and int and long are even larger; the exact size of these being dependent on the system. For example, let's imagine that in a given system, char takes 1 byte, short takes 2 bytes, and long takes 4.

Suppose now that we define three pointers in this compiler:

```
int myvar;
```

```
int *foo = &myvar;
```

```
int *bar = foo;
```

and that we know that they point to the memory locations 1000, 2000, and 3000, respectively.

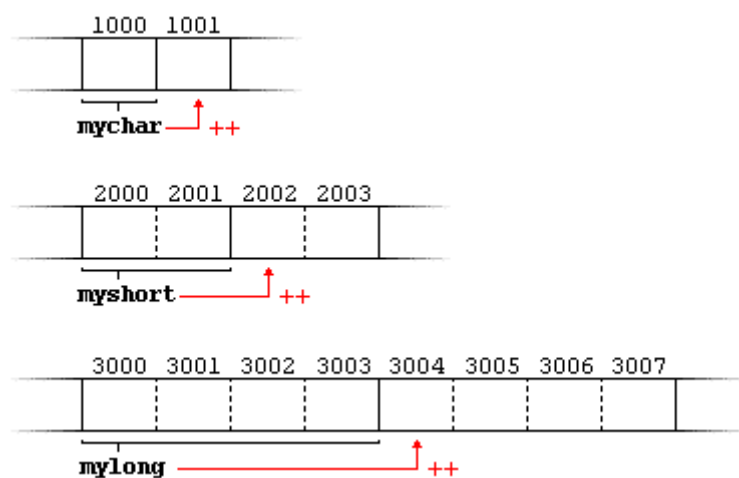
Therefore, if we write:

```
++mychar;
```

```
++myshort;
```

```
++mylong;
```

mychar, as one would expect, would contain the value 1001. But not so obviously, myshort would contain the value 2002, and mylong would contain 3004, even though they have each been incremented only once. The reason is that, when adding one to a pointer, the pointer is made to point to the following element of the same type, and, therefore, the size in bytes of the type it points to is added to the pointer.





This is applicable both when adding and subtracting any number to a pointer. It would happen exactly the same if we wrote:

```
mychar = mychar +1;
```

```
myshort = myshort +1;
```

```
mylong = mylong +1;
```

Regarding the increment (++) and decrement (--) operators, they both can be used as either prefix or suffix of an expression, with a slight difference in behavior: as a prefix, the increment happens before the expression is evaluated, and as a suffix, the increment happens after the expression is evaluated. This also applies to expressions incrementing and decrementing pointers, which can become part of more complicated expressions that also include dereference operators (\*). Remembering operator precedence rules, we can recall that postfix operators, such as increment and decrement, have higher precedence than prefix operators, such as the dereference operator (\*). Therefore, the following expression:

```
*p++
```

is equivalent to `*(p++)`. And what it does is to increase the value of `p` (so it now points to the next element), but because `++` is used as postfix, the whole expression is evaluated as the value pointed originally by the pointer (the address it pointed to before being incremented).

Essentially, these are the four possible combinations of the dereference operator with both the prefix and suffix versions of the increment operator (the same being applicable also to the decrement operator):

```
*p++ // same as *(p++): increment pointer, and dereference      unincremented address
```

```
*++p // same as *(++p): increment pointer, and dereference incremented address
```

```
++*p // same as ++(*p): dereference pointer, and increment the value it points to
```



(*\*p*)++ // dereference pointer, and post-increment the value it points to

A typical -but not so simple- statement involving these operators is:

```
*p++ = *q++;
```

Because ++ has a higher precedence than \*, both *p* and *q* are incremented, but because both increment operators (++) are used as postfix and not prefix, the value assigned to *\*p* is *\*q* before both *p* and *q* are incremented. And then both are incremented. It would be roughly equivalent to:

```
1 *p = *q;
```

```
2 ++p;
```

```
3 ++q;
```

## 4.5 POINTERS AND FUNCTIONS

### Pointers and Functions

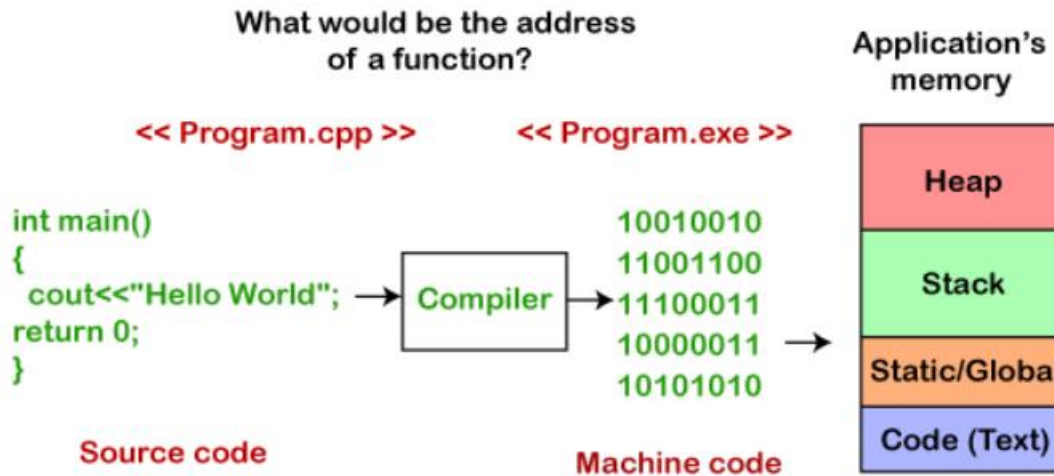
As we know that pointers are used to point some variables; similarly, the function pointer is a pointer used to point functions. It is basically used to store the address of a function. We can call the function by using the function pointer, or we can also pass the pointer to another function as a parameter.

They are mainly useful for event-driven applications, callbacks, and even for storing the functions in arrays.

#### 4.5.1 What is the address of a function?



## Function Pointers



Computer only understands the low-level language, i.e., binary form. The program we write in C++ is always in high-level language, so to convert the program into binary form, we use compiler. Compiler is a program that converts source code into an executable file. This executable file gets stored in RAM. The CPU starts the execution from the `main()` method, and it reads the copy in RAM but not the original file. All the functions and machine code instructions are data. This data is a bunch of bytes, and all these bytes have some address in RAM. The function pointer contains RAM address of the first instruction of a function.

### 4.5.1.1 Syntax for Declaration

The following is the syntax for the declaration of a function pointer:

```
int (*FuncPtr) (int,int);
```

The above syntax is the function declaration. As functions are not simple as variables, but C++ is a type safe, so function pointers have return type and parameter list. In the above syntax, we first supply the return type, and then the name of the pointer, i.e., `FuncPtr` which is surrounded by the brackets and preceded by the pointer symbol, i.e., `(*)`. After this, we have supplied the parameter list `(int,int)`. The above function pointer can point to any function which takes two integer parameters and returns integer type value.



### Address of a function

We can get the address of a function very easily. We just need to mention the name of the function, we do not need to call the function. Example:

```
#include <iostream>

using namespace std;

int main()
{
    std::cout << "Address of a main() function is : " << &main << std::endl;

    return 0;
}
```

In the above program, we are displaying the address of a main() function. To print the address of a main() function, we have just mentioned the name of the function, there is no bracket not parameters. Therefore, the name of the function by itself without any brackets or parameters means the address of a function. We can use the alternate way to print the address of a function, i.e., &main.

#### 4.5.1.2 Calling a function indirectly

We can call the function with the help of a function pointer by simply using the name of the function pointer. The syntax of calling the function through the function pointer would be similar as we do the calling of the function normally. Let's understand this scenario through an example.

```
#include <iostream>

using namespace std;

int add(int a , int b)
{
    return a+b;
}

int main()
```



```
{  
    int (*funcptr)(int,int); // function pointer declaration  
    funcptr=add; // funcptr is pointing to the add function  
    int sum=funcptr(5,5);  
    std::cout << "value of sum is :" <<sum<< std::endl;  
    return 0;  
}
```

In the above program, we declare the function pointer, i.e., `int (*funcptr)(int,int)` and then we store the address of `add()` function in `funcptr`. This implies that `funcptr` contains the address of `add()` function. Now, we can call the `add()` function by using `funcptr`. The statement `funcptr(5,5)` calls the `add()` function, and the result of `add()` function gets stored in `sum` variable.

**Output:**

```
value of sum is :10  
  
...Program finished with exit code 0  
Press ENTER to exit console.█
```

**look at another example of function pointer.**

```
#include <iostream>  
  
using namespace std;  
  
void printname(char *name)  
{  
    std::cout << "Name is :" <<name<< std::endl;  
}  
  
int main()
```



```
{  
    char s[20]; // array declaration  
    void (*ptr)(char*); // function pointer declaration  
    ptr=printname; // storing the address of printname in ptr.  
    std::cout << "Enter the name of the person: " << std::endl;  
    cin>>s;  
    cout<<s;  
    ptr(s); // calling printname() function  
    return 0;  
}
```

In the above program, we define the function `printname()` which contains the char pointer as a parameter. We declare the function pointer, i.e., `void (*ptr)(char*)`. The statement `ptr=printname` means that we are assigning the address of `printname()` function to `ptr`. Now, we can call the `printname()` function by using the statement `ptr(s)`.

### Output:

```
Enter the name of the person:  
john  
john  
Name is :john  
  
...Program finished with exit code 0  
Press ENTER to exit console.
```

#### 4.5.1.3 Passing a function pointer as a parameter

The function pointer can be passed as a parameter to another function. Let's understand through an example.

```
#include <iostream>
```

```
using namespace std;
```

```
void func1()
```





```
{  
    cout<<"func1 is called";  
}  
  
void func2(void (*funcptr)())  
{  
    funcptr();  
}  
  
int main()  
{  
    func2(func1);  
  
    return 0;  
}
```

In the above code, the func2() function takes the function pointer as a parameter. The main() method calls the func2() function in which the address of func1() is passed. In this way, the func2() function is calling the func1() indirectly.

#### Output:

```
func1 is called  
...Program finished with exit code 0  
Press ENTER to exit console.
```

## 4.6 POINTER TO FUNCTION

### Pointers to function



C++ allows operations with pointers to functions. The typical use of this is for passing a function as an argument to another function. Pointers to functions are declared with the same syntax as a regular function declaration, except that the name of the function is enclosed between parentheses () and an asterisk (\*) is inserted before the name:

```
// pointer to functions
```

```
#include <iostream>
```

```
using namespace std;
```

```
int addition (int a, int b)
```

```
{ return (a+b); }
```

```
int subtraction (int a, int b)
```

```
{ return (a-b); }
```

```
int operation (int x, int y, int (*functocall)(int,int))
```

```
{
```

```
    int g;
```

```
    g = (*functocall)(x,y);
```

```
    return (g);
```

```
}
```

```
int main ()
```

```
{
```

```
    int m,n;
```

```
    int (*minus)(int,int) = subtraction;
```



```
m = operation (7, 5, addition);  
n = operation (20, m, minus);  
cout <<n;  
return 0;  
}
```

In the example above, minus is a pointer to a function that has two parameters of type int. It is directly initialized to point to the function subtraction:

Int (\* minus)(int,int) = subtraction;

## 4.7 POINTER AND ARRAY

### Pointers and Array

The concept of arrays is related to that of pointers. In fact, arrays work very much like pointers to their first elements, and, actually, an array can always be implicitly converted to the pointer of the proper type. For example, consider these two declarations:

```
int myarray [20];  
int * mypointer;
```

The following assignment operation would be valid:

```
mypointer = myarray;
```

Examples that displaying the elements of an array using pointer:

Example 1:

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



```
int main ()
{
    int numbers[5];
    int * p;
    p = numbers; *p = 10;
    p++; *p = 20;
    p = &numbers[2]; *p = 30;
    p = numbers + 3; *p = 40;
    p = numbers; *(p+4) = 50;
    for (int n=0; n<5; n++)
        cout << numbers[n] << ", ";
    return 0;
}
```

Output:

10,20,30,40,50

Example 2:

```
#include<iostream.h>

void main()
{
    int array []={78,45,12,89,56,23,79,46,13,82}; // Array of 10 elements
    int *ptr; // Pointer variable
    ptr = array; // Assigning reference of array in
                // pointer variable
}
```



```
cout << "\nValues : ";  
for(int a=1;a<=10;a++)  
{  
    cout << *ptr;           // Displaying values of array  
                             // using pointer  
    ptr++;                  // Incrementing pointer variable  
}  
}
```

Output :

Values : 78, 45, 12, 89, 56, 23, 79, 46, 13, 82,

In the above example statement 1 creates an array of 10 elements. Statement 2 creates a pointer variable ptr. As said above array name works as pointer variable therefore statement 3 is assigning the address of array in pointer variable ptr. Now ptr have the address of first element in an array. Statement 4 will display the value at address of ptr. After display the first value, statement 5 increase the pointer variable ptr to point to next element in an array and statement 4 will display the next value in an array until the loop ends.

Pointers and arrays support the same set of operations, with the same meaning for both. The main difference being that pointers can be assigned new addresses, while arrays cannot. In the chapter about arrays, brackets ([]) were explained as specifying the index of an element of the array. Well, in fact these brackets are a dereferencing operator known as *offset operator*. They dereference the variable they follow just as \* does, but they also add the number between brackets to the address being dereferenced. For example:

```
a[5] = 0;           // a [offset of 5] = 0  
*(a+5) = 0;        // pointed to by (a+5) = 0
```

These two expressions are equivalent and valid, not only if a is a pointer, but also if a is an array. Remember that if an array, its name can be used just like a pointer to its first element.



## 4.8 ARRAY OF POINTER

### Array of Pointers

Array is a collection of values of similar type. It can also be a collection of references of similar type. Syntax:

**Data\_type \* array [size];**

#### Example:

```
#include<iostream.h>

void main()
{
    int x=10,y=20,z=30;

    int *array[3];           // Declaring array of three pointer
    array[0] = &x;           // Assigning reference of x to array 0th position
    array[1] = &y;           // Assigning reference of y to array 1th position
    array[2] = &z;           // Assigning reference of z to array 2nd position

    cout << "\nValues : ";

    for(int a=0;a<3;a++)
        cout << *arr[a];

}
```

Output :

Values : 10, 20, 30,

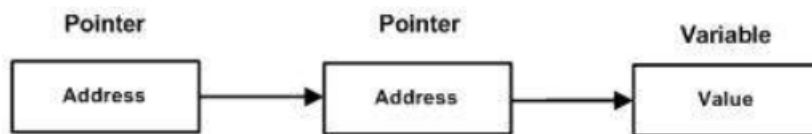
In the above example, we have declared three variable x, y, z and assigning the addresses of these variables into an array of pointer(\*arr[]).

## 4.9 POINTERS TO POINTERS



### Pointer to pointer

A pointer to a pointer is a form of multiple indirection or a chain of pointers. Normally, a pointer contains the address of a variable. When we define a pointer to a pointer, the first pointer contains the address of the second pointer, which points to the location that contains the actual value as shown below.



A variable that is a pointer to a pointer must be declared as such. This is done by placing an additional asterisk in front of its name. For example, following is the declaration to declare a pointer to a pointer of type int –

```
int **var;
```

When a target value is indirectly pointed to by a pointer to a pointer, accessing that value requires that the asterisk operator be applied twice, as is shown below in the example –

```
#include <iostream>
```

```
using namespace std;
```

```
int main () {
```

```
    int var;
```

```
    int *ptr;
```

```
    int **pptr;
```

```
    var = 3000;
```



```
// take the address of var
ptr = &var;

// take the address of ptr using address of operator &
pptr = &ptr;

// take the value using pptr
cout << "Value of var :" << var << endl;
cout << "Value available at *ptr :" << *ptr << endl;
cout << "Value available at **pptr :" << **pptr << endl;

return 0;
}
```

When the above code is compiled and executed, it produces the following result –

Value of var :3000

Value available at \*ptr :3000

Value available at \*\*pptr :3000

## 4.10 CHECK YOUR PROGRESS

Ques:1 What is meaning of following declaration?

```
int(*p[5])();
```

- A. p is pointer to function.
- B. p is array of pointer to function
- C. p is pointer to such function which return type is array.
- D. p is pointer to array of function.

Ques:2 What is size of generic pointer in c?





- A. 0
- B. 1
- C. 2
- D. Null

Ques:3 Void pointer can point to which type of objects?

- A. int
- B. float
- C. double
- D. all of the mentioned

Ques4 When does the void pointer can be dereferenced?

- A. when it doesn't point to any value
- B. when it cast to another type of object
- C. using delete keyword
- D. none of the mentioned

Ques:5 The pointer can point to any variable that is not declared with which of these?

- A. const
- B. volatile
- C. both a & b
- D. static

Ques:6 A void pointer cannot point to which of these?

- A. methods in c++
- B. class member in c++
- C. both a & b
- D. none of the mentioned



Ques:7 What we can't do on a void pointer?

- A. pointer arithmetic
- B. pointer functions
- C. both of the mentioned
- D. none of the mentioned

## 4.11 SUMMARY

## 4.12 SELF ASSESSMENT TEST

Ques1: What is the difference between a pointer and a reference?

Ques:2 When should I use references, and when should I use pointers?

Ques:3 What is the difference between `const char *myPointer` and `char *const myPointer`?

Ques:4 Explain pointer and array in detail.

Ques:5 Define pointer to pointer with example.

## 4.13 ANSWER TO CHECK YOUR PROGRESS

Answer: Option B

Answer: Option C

Answer: Option D

Answer: Option B

Answer: Option C

Answer: Option B

Answer: Option A

## 4.14 REFERENCES/SUGGESTED READING



Subject Name: <b>Object Oriented Programming Using C++</b>	
Course Code: <b>DCA-15-T</b>	Author:
Lesson No. 5	Vetter:
<b>STRUCTURE</b>	

### 5.1 Introduction to array

### 5.2 Structure definition

### 5.3 INITIALIZATION OF STRUCTURE

### 5.4 ACCESSING MEMBERS OF STRUCTURE

### 5.5 NESTED STRUCTURE

#### 5.1 Syntax of Structure with in structure

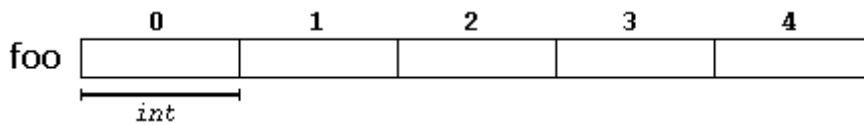
### 5.6 STRUCTURE AND POINTER

### 5.7 CHECK YOUR PROGRESS

## 5.1 INTRODUCTION TO ARRAY

### Introduction to Array

An array is a series of elements of the same type placed in contiguous memory locations that can be individually referenced by adding an index to a unique identifier. That means that, for example, five values of type int can be declared as an array without having to declare 5 different variables (each with its own identifier). Instead, using an array, the five int values are stored in contiguous memory locations, and all five can be accessed using the same identifier, with the proper index. For example, an array containing 5 integer values of type int called foo could be represented as:



where each blank panel represents an element of the array. In this case, these are values of type `int`.

These elements are numbered from 0 to 4, being 0 the first and 4 the last; The first element in an array is always numbered with a zero (not a one), no matter its length. Like a regular variable, an array must be declared before it is used. A typical declaration for an array is:

`type name [elements];`

where `type` is a valid type (such as `int`, `float`...), `name` is a valid identifier and the `elements` field (which is always enclosed in square brackets `[]`), specifies the length of the array in terms of the number of elements. Therefore, the `foo` array, with five elements of type `int`, can be declared as:

`int foo [5];`

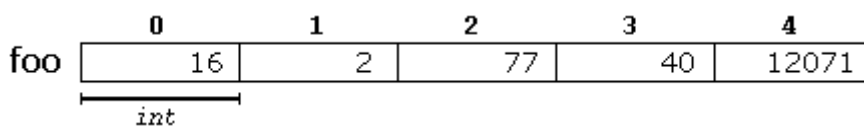
The `elements` field within square brackets `[]`, representing the number of elements in the array, must be a *constant expression*, since arrays are blocks of static memory whose size must be determined at compile time, before the program runs.

### Initializing arrays

By default, regular arrays of *local scope* (for example, those declared within a function) are left uninitialized. This means that none of its elements are set to any particular value; their contents are undetermined at the point the array is declared. But the elements in an array can be explicitly initialized to specific values when it is declared, by enclosing those initial values in braces `{ }`. For example:

`int foo [5] = { 16, 2, 77, 40, 12071 };`

This statement declares an array that can be represented like this:

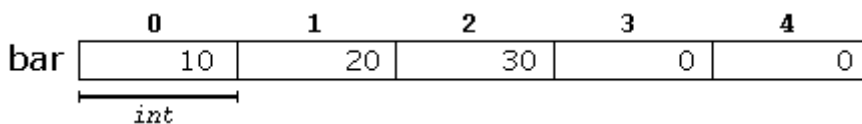




The number of values between braces { } shall not be greater than the number of elements in the array. For example, in the example above, foo was declared having 5 elements (as specified by the number enclosed in square brackets, []), and the braces { } contained exactly 5 values, one for each element. If declared with less, the remaining elements are set to their default values (which for fundamental types, means they are filled with zeroes). For example:

```
int bar [5] = { 10, 20, 30 };
```

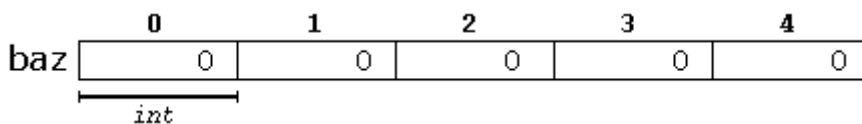
Will create an array like this:



The initializer can even have no values, just the braces:

```
int baz [5] = { };
```

This creates an array of five int values, each initialized with a value of zero:



When an initialization of values is provided for an array, C++ allows the possibility of leaving the square brackets empty []. In this case, the compiler will assume automatically a size for the array that matches the number of values included between the braces { }:

```
int foo [] = { 16, 2, 77, 40, 12071 };
```

After this declaration, array foo would be 5 int long, since we have provided 5 initialization values.

Finally, the evolution of C++ has led to the adoption of *universal initialization* also for arrays.

Therefore, there is no longer need for the equal sign between the declaration and the initializer. Both these statements are equivalent:



```
1 int foo[] = { 10, 20, 30 };
```

```
2 int foo[] { 10, 20, 30 };
```

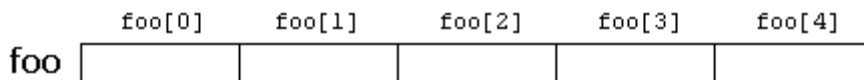
Static arrays, and those declared directly in a namespace (outside any function), are always initialized. If no explicit initializer is specified, all the elements are default-initialized (with zeroes, for fundamental types).

### Accessing the value of an array

The values of any of the elements in an array can be accessed just like the value of a regular variable of the same type. The syntax is:

name[index]

Following the previous examples in which foo had 5 elements and each of those elements was of type int, the name which can be used to refer to each element is the following:



For example, the following statement stores the value 75 in the third element of foo:

```
foo [2] = 75;
```

and, for example, the following copies the value of the third element of foo to a variable called x:

```
x = foo[2];
```

Therefore, the expression foo[2] is itself a variable of type int. Notice that the third element of foo is specified foo[2], since the first one is foo[0], the second one is foo[1], and therefore, the third one is foo[2]. By this same reason, its last element is foo[4]. Therefore, if we write foo[5], we would be accessing the sixth element of foo, and therefore actually exceeding the size of the array. It is syntactically correct to exceed the valid range of indices for an array. This can create problems, since accessing out-of-range elements do not cause errors on compilation, but can cause errors on runtime. At this point, it is important to be able to clearly distinguish between the two uses that brackets [] have related to arrays. They perform two different tasks: one is to specify the size of arrays when they are



declared; and the second one is to specify indices for concrete array elements when they are accessed.

```
1 int foo[5];    // declaration of a new array
2 foo[2] = 75;   // access to an element of the array.
```

The main difference is that the declaration is preceded by the type of the elements, while the access is not. Some other valid operations with arrays:

```
1 foo[0] = a;
2 foo[a] = 75;
3 b = foo [a+2];
4 foo[foo[a]] = foo[2] + 5;
```

For example: *// arrays example*

```
#include <iostream>
```

```
using namespace std;
```

```
int foo [] = { 16, 2, 77, 40, 12071 };
```

```
int n, result=0;
```

```
int main ()
```

```
{
```

```
for ( n=0 ; n<5 ; ++n )
```

```
{
```

```
    result += foo[n];
```

```
}
```

```
cout << result;
```



```
return 0;
```

```
}
```

Output: 12206

### Multidimensional array

Multidimensional arrays can be described as "arrays of arrays". For example, a bidimensional array can be imagined as a two-dimensional table made of elements, all of them of a same uniform data type.

		<b>0</b>	<b>1</b>	<b>2</b>	<b>3</b>	<b>4</b>
<b>jimmy</b> {	<b>0</b>					
	<b>1</b>					
	<b>2</b>					

jimmy represents a bidimensional array of 3 per 5 elements of type int. The syntax is:

```
int jimmy [3][5];
```

and, for example, the way to reference the second element vertically and fourth horizontally in an expression would be:

```
jimmy[1][3]
```

		<b>0</b>	<b>1</b>	<b>2</b>	<b>3</b>	<b>4</b>
<b>jimmy</b> {	<b>0</b>					
	<b>1</b>					
	<b>2</b>					

↓  
**jimmy[1][3]**

(remember that array indices always begin with zero).

Multidimensional arrays are not limited to two indices (i.e., two dimensions). They can contain as many indices as needed. Although be careful: the amount of memory needed for an array increases exponentially with each dimension. For example:

```
char century [100][365][24][60][60];
```





declares an array with an element of type char for each second in a century. This amounts to more than 3 billion char! So this declaration would consume more than 3 gigabytes of memory!

At the end, multidimensional arrays are just an abstraction for programmers, since the same results can be achieved with a simple array, by multiplying its indices:

1 `int jimmy [3][5];` // is equivalent to

2 `int jimmy [15];` // ( $3 * 5 = 15$ )

## 5.2 STRUCTURE DEFINATION

### Introduction

Arrays are used to store similar type of data. But if you want to store **dissimilar** data. The only way is to use **structures** to store different types of data. Structure is a collection of variables of different data types under a single name. It is similar to a class in that, both holds a collection of data of different data types.

**For example:** You want to store some information about a person: his/her name, citizenship number and salary. You can easily create different variables name, citNo, salary to store these information separately. However, in the future, you would want to store information about multiple persons. Now, you'd need to create different variables for each information per person: name1, citNo1, salary1, name2, citNo2, salary2. You can easily visualize how big and messy the code would look. Also, since no relation between the variables (information) would exist, it's going to be a daunting task. A better approach will be to have a collection of all related information under a single name person, and use it for every person. Now, the code looks much cleaner, readable and efficient as well. This collection of all related information under a single name person is a structure.



## 5.3 INITIALIZATION OF STRUCTURE

### Structure Declaration

The struct keyword defines a structure type followed by an identifier (name of the structure). Then inside the curly braces, you can declare one or more members (declare variables inside curly braces) of that structure. For example:

**struct** Person

```
{  
    char name[50];  
    int age;  
    float salary;  
};
```

Here a structure person is defined which has three members: name, age and salary. When a structure is created, no memory is allocated.

The structure definition is only the blueprint for the creating of variables. You can imagine it as a datatype. When you define an integer as below:

```
int var;
```

The `int` specifies that, variable `var` can hold integer element only. Similarly, structure definition only specifies that, what property a structure variable holds when it is defined. Remember to end the declaration with a semicolon (;)

### Define a structure variable

Once you declare a structure `person` as above. You can define a structure variable as:

```
Person bill;
```

Here, a structure variable `bill` is defined which is of type structure `Person`.

When structure variable is defined, only then the required memory is allocated by the



compiler. Considering you have either 32-bit or 64-bit system, the memory of `float` is 4 bytes, memory of `int` is 4 bytes and memory of `char` is 1 byte. Hence, 58 bytes of memory is allocated for structure variable `bill`.

## 5.4 ACCESSING MEMBERS OF STRUCTURE

### Access members of a structure

The members of structure variable is accessed using a **dot (.)** operator.

Suppose, you want to access `age` of structure variable `bill` and assign it 50 to it. You can perform this task by using following code below:

```
bill.age = 50;
```

**Example: Structure** (Program to assign data to members of a structure variable and display it.)

```
#include <iostream>
```

```
using namespace std;
```

```
struct Person
```

```
{
```

```
    char name[50];
```

```
    int age;
```

```
    float salary;
```

```
};
```

```
int main()
```

```
{
```

```
    Person p1;
```



```
cout << "Enter Full name: ";
```

```
cin.get(p1.name, 50);
```

```
cout << "Enter age: ";
```

```
cin >> p1.age;
```

```
cout << "Enter salary: ";
```

```
cin >> p1.salary;
```

```
cout << "\nDisplaying Information." << endl;
```

```
cout << "Name: " << p1.name << endl;
```

```
cout << "Age: " << p1.age << endl;
```

```
cout << "Salary: " << p1.salary;
```

```
return 0;
```

```
}
```

## Output

Enter Full name: kamal kishore

Enter age: 43

Enter salary: 1132.5

Displaying Information.

Name: kamal kishore

Age: 43

Salary: 1132.5

Here a structure `Person` is declared which has three members `name`, `age` and `salary`. Inside `main()` function, a structure variable `p1` is defined. Then, the



user is asked to enter information and data entered by user is displayed.

## Structure and Functions

Structure variables can be passed to a function and returned in a similar way as normal arguments.

### Passing structure to function

A structure variable can be passed to a function in similar way as normal argument. Consider this example:

Example: Structure and Function

```
#include<iostream>
```

```
usingnamespacestd;
```

```
structPerson
```

```
{
```

```
char name[50];
```

```
int age;
```

```
float salary;
```

```
};
```

```
voidshowData(Person); // Function declaration
```

```
intmain()
```

```
{
```

```
Person p;
```

```
cout<<"Enter Full name: ";
```



```
cin.get(p.name, 50);
cout<<"Enter age: ";
cin>> p.age;
cout<<"Enter salary: ";
cin>> p.salary;

// Function call with structure variable as an argument
showData(p);

return 0;
}
```

```
void showData(Person p)
{
    cout<<"\nDisplaying Information."<<endl;
    cout<<"Name: "<< p.name <<endl;
    cout<<"Age: "<< p.age <<endl;
    cout<<"Salary: "<< p.salary;
}
```

### Output

Enter Full name: Aman Gour

Enter age: 53

Enter salary: 45231.6

Displaying Information.



Name: Aman Gour

Age: 53

Salary: 4523.6

In this program, user is asked to enter the `name`, `age` and `salary` of a Person inside `main()` function.

Then, the structure variable `p` is to be passed to a function using.

```
showData(p);
```

The return type of `showData()` is `void` and a single argument of type structure `Person` is passed. Then the members of structure `p` are displayed from this function.

#### Example: Returning structure from function

```
#include<iostream>
```

```
using namespace std;
```

```
struct Person {
```

```
    char name[50];
```

```
    int age;
```

```
    float salary;
```

```
};
```

```
Person getData(Person);
```

```
void showData(Person);
```

```
int main()
```

```
{
```



```
Person p;  
  
p = getData(p);  
showData(p);  
  
return 0;  
}  
  
Person getData(Person p){  
  
cout<<"Enter Full name: ";  
cin.get(p.name, 50);  
  
cout<<"Enter age: ";  
cin>> p.age;  
  
cout<<"Enter salary: ";  
cin>> p.salary;  
  
return p;  
}  
  
void showData(Person p)  
{  
cout<<"\nDisplaying Information."<<endl;
```





```
cout<<"Name: "<< p.name <<endl;
cout<<"Age: "<< p.age <<endl;
cout<<"Salary: "<< p.salary;
}
```

The output of this program is same as program above. In this program, the structure variable `p` of type structure `Person` is defined under `main()` function. The structure variable `p` is passed to `getData()` function which takes input from user which is then returned to main function.

```
p = getData(p);
```

The value of all members of a structure variable can be assigned to another structure using assignment operator `=` if both structure variables are of same type. You don't need to manually assign each members. Then the structure variable `p` is passed to `showData()` function, which displays the information.

### Array of Structures

We can also make an array of structures. In the first example in structures, we stored the data of 3 students. Now suppose we need to store the data of 100 such children. Declaring 100 separate variables of the structure is definitely not a good option. For that, we need to create an array of structures.

Let's see an example for 5 students.

```
#include<iostream>
#include<cstring>
using namespace std;
struct student
{
    int roll_no;
    string name;
    int phone_number;
```



```
};  
  
int main(){  
    struct student stud[5];  
  
    int i;  
  
    for(i=0;i<5;i++){           //taking values from user  
        cout<<"Student "<<i+1<<endl;  
        cout<<"Enter roll no"<<endl;  
        cin>>stud[i].roll_no;  
        cout<<"Enter name"<<endl;  
        cin>>stud[i].name;  
        cout<<"Enter phone number"<<endl;  
        cin>>stud[i].phone_number;  
    }  
  
    for(i=0;i<5;i++){           //printing values  
        cout<<"Student "<<i+1<<endl;  
        cout<<"Roll no : "<<stud[i].roll_no<<endl;  
        cout<<"Name : "<<stud[i].name<<endl;  
        cout<<"Phone no : "<<stud[i].phone_number<<endl;  
    }  
  
    return 0;  
}
```

Output:

Student 1

Enter roll no 1

Enter name



Brow

Enter phone number

82349545

Student 2

Enter roll no

2

Enter name

wany

Enter phone number

85679331

Student 3

Enter roll no

3

Enter name

romi

Enter phone number

94829678

Student 4

Enter roll no

4

Enter name

goly

Enter phone number

98932665

Student 5

Enter roll no

5

Enter name

jacx

Enter phone number



43957670

Student 1

Roll no : 1

Name : Brow

Phone no : 82349545

Student 2

Roll no : 2

Name : wany

Phone no : 85679331

Student 3

Roll no : 3

Name : romi

Phone no : 94829678

Student 4

Roll no : 4

Name : goly

Phone no : 98932665

Student 5

Roll no : 5

Name : jacx

Phone no : 43957670

Here we created an array named **stud** having 5 elements of structure student. Each of the element stores the information of a student. For example, stud[0] stores the information of the first student, stud[1] for the second and so on. We can also copy two structures at one go.

```
#include<iostream>
```

```
#include<cstring>
```

```
using namespace std;
```



```
struct student
{
    int roll_no;
    string name;
    int phone_number;
};

int main() {

    struct student p1 = { 1, "Brown", 123443 };
    struct student p2;
    p2 = p1;

    cout << "roll no : " << p2.roll_no << endl;
    cout << "name : " << p2.name << endl;
    cout << "phone number : " << p2.phone_number << endl;
    return 0;
}
```

Output:

```
roll no : 1
```

```
name : Brown
```

```
phone number : 123443
```

We just have to write `p1 = p2` and that's it. By writing this, all the elements of `p1` will get copied to `p2`.



## 5.5 NESTED STRUCTURE

When a structure contains another structure, it is called nested structure. For example, we have two structures named Address and Employee. To make Address nested to Employee, we have to define Address structure before and outside Employee structure and create an object of Address structure inside Employee structure.

### 5.5.1 Syntax for structure within structure or nested structure

```
struct structure1
{
    -----
    -----
};

struct structure2
{
    -----
    -----
    struct structure1 obj;
};
```

### 5.5.2 Example for structure within structure or nested structure

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



```
struct Address
{
    char HouseNo[25];
    char City[25];
    char PinCode[25];
};
```

```
struct Employee
{
    int Id;
    char Name[25];
    float Salary;
```

```
struct Address Add;
```

```
};

void main()
{
    int i;
    Employee E;

    cout << "\n\tEnter Employee Id : ";
    cin >> E.Id;

    cout << "\n\tEnter Employee Name : ";
    cin >> E.Name;
```



```
cout << "\n\tEnter Employee Salary : ";
cin >> E.Salary;

cout << "\n\tEnter Employee House No : ";
cin >> E.Add.HouseNo;

cout << "\n\tEnter Employee City : ";
cin >> E.Add.City;

cout << "\n\tEnter Employee House No : ";
cin >> E.Add.PinCode;

cout << "\nDetails of Employees";
cout << "\n\tEmployee Id : " << E.Id;
cout << "\n\tEmployee Name : " << E.Name;
cout << "\n\tEmployee Salary : " << E.Salary;
cout << "\n\tEmployee House No : " << E.Add.HouseNo;
cout << "\n\tEmployee City : " << E.Add.City;
cout << "\n\tEmployee House No : " << E.Add.PinCode;

}
```

Output :

Enter Employee Id : 101





Enter Employee Name : Suresh

Enter Employee Salary : 45000

Enter Employee House No : 4598/D

Enter Employee City : Delhi

Enter Employee Pin Code : 110056

#### Details of Employees

Employee Id : 101

Employee Name : Suresh

Employee Salary : 45000

Employee House No : 4598/D

Employee City : Delhi

Employee Pin Code : 110056

## 5.6 STRUCTURE AND POINTER

C++ allows pointers to structures just as it allows pointers to int or char or float or any other data type. The pointers to structures are known as structure pointers.

### 5.6.1 Declaration and Use of Structure Pointers

Just like other pointers, the structure pointers are declared by placing asterisk (\*) in front of a structure pointer's name.

It takes the following general form :

```
struct-name *struct-pointer ;
```



where struct-name is the name of an already defined structure and struct-pointer is the pointer to this structure. For example, to declare dt\_ptr as a pointer to already defined structure date, we shall write

```
date * dt_ptr ;
```

The declaration of structure type and the structure pointer can be combined in one statement. For example,

```
struct date
{
    short int dd, mm, yy ;
};
```

```
date * dt_ptr ;
```

is same as

```
struct date
{
    short int dd, mm, yy ;
} * dt_ptr ;
```

Using structure pointers, the members of structures are accessed using arrow operator ->. To refer to the structure members using -> operator, it is written as

```
struct-pointer -> structure-member
```

That is, to access dd and yy with dt\_ptr, we shall write

```
dt_ptr -> yy
```

and

```
dt_ptr -> dd
```

### Structure Pointers Example

Let's look at the following program which demonstrates the usage of these pointers :

```
/* program
```



\* demonstrates about declaration and the use of

\* Structure Pointers \*/

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

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

```
void main()
```

```
{
```

```
    struct date
```

```
    {
```

```
        short int dd, mm, yy;
```

```
    }join_date = { 19, 12, 2006};
```

```
    date *date_ptr;
```

```
    date_ptr = &join_date;
```

```
    cout<<"Printing the structure elements using the structure variable\n";
```

```
    cout<<"dd = "<<join_date.dd<<", mm = "<<join_date.mm<<, yy = "<<join_date.yy<<"\n";
```

```
    cout<<"\nPrinting the structure elements using the structure pointer\n";
```

```
    cout<<"dd = "<<date_ptr->dd<<, mm = "<<date_ptr->mm<<, yy = "<<date_ptr->yy<<"\n";
```

```
    getch();
```

```
}
```

output:



```
C:\TURBOC~1\Disk\TurboC3\SOURCE\1.EXE
Printing the structure elements using the structure variable
dd = 19, mm = 12, yy = 2006
Printing the structure elements using the structure pointer
dd = 19, mm = 12, yy = 2006
```

The above program is self-explanatory. Remember that the dot operator ('.') requires a structure variable on its left and the arrow operator ('->') requires a structure pointer on its left. The arrow operator consists of the minus sign followed by greater than sign.

There are the following two primary uses of for structure pointers :

- to generate a call by reference call to a function.
- to create dynamic data structures like linked lists, stacks, queues, trees etc. using C++'s dynamic allocation system.

The structure pointer are very much useful in generating call by reference call to function because when a pointer to a structure is passed to a function, only the address of the structure is passed. This makes way for very fast function calls. A second advantage of it is when a function needs to reference the actual structure argument instead of a copy of them, by passing a pointer, it can modify the contents of the actual elements of the structure used in the call.

Here is another example program given, that you can take a look, for the complete understanding on the structure pointers in C++

```
/* This C++ program
```

```
* also demonstrates the structure pointer in C++ */
```

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

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

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

```
#include<stdio.h>
```



```
struct emp
{
    int empno;
    char empname[20];
    float empbasic;
    float empexperience;
};

void display(emp *e);
void increase(emp *e);
void main()
{
    clrscr();
    emp mgr, *eptr;
    cout<<"Enter employee number: ";
    cin>>mgr.empno;
    cout<<"Enter name: ";
    gets(mgr.empname);
    cout<<"Enter basic pay: ";
    cin>>mgr.empbasic;
    cout<<"Enter experience (in years): ";
    cin>>mgr.empexperience;
    eptr = &mgr;
    cout<<"\nEmployee details before increase()\n";
    display(eptr);
    increase(eptr);
}
```



```
        cout<<"\nEmployee details after increase()\n";
        display(eptr);
        getch();
    }
    void display(emp *e)
    {
        int len=strlen(e->empname);
        cout<<"Employee number: "<<e->empno;
        cout<<"\nName: ";
        cout.write(e->empname, len);
        cout<<"\tBasic: "<<e->empbasic;
        cout<<"\tExperience: "<<e->empexperience<<" years\n";
    }
    void increase(emp *e)
    {
        if(e->empexperience >= 5)
        {
            e->empbasic = e->empbasic + 15000;
        }
    }
```

Here are the two sample runs of the above C++ program. One for the employee having experience less than 5 years and the other having experience more than 5 years.



```
C:\TURBOC~1\Disk\TurboC3\SOURCE\2.EXE
Enter employee number: 3358
Enter name: Hritik
Enter basic pay: 56000
Enter experience (in years): 4

Employee details before increase()
Employee number: 3358
Name: Hritik    Basic: 56000    Experience: 4 years

Employee details after increase()
Employee number: 3358
Name: Hritik    Basic: 56000    Experience: 4 years
```

```
C:\TURBOC~1\Disk\TurboC3\SOURCE\2.EXE
Enter employee number: 3379
Enter name: Shivam
Enter basic pay: 78000
Enter experience (in years): 7

Employee details before increase()
Employee number: 3379
Name: Shivam    Basic: 78000    Experience: 7 years

Employee details after increase()
Employee number: 3379
Name: Shivam    Basic: 93000    Experience: 7 years
```

## 5.7 CHECK YOUR PROGRESS

Ques:1 Can a Structure contain pointer to itself?

- (A) Yes
- (B) No
- (C) Compilation Error
- (D) Runtime Error

Error

Ques:2 What is Self Referencial Structure?

- (A) Structure which contains pointers
- (B) Structure which has pointer to itself
- (C) Structure which contains another structure



(D) None of these

Ques: 3 What should be output of below program? program is compiled on g++ compiler.

```
#include<iostream>
```

```
using namespace std;
```

```
struct student{
```

```
    char a; char b; int c;
```

```
};
```

```
int main()
```

```
{
```

```
    cout<<sizeof(student);
```

```
    return 0;
```

```
}
```

(A) 4

(B) 6

(C) 8

(D) 12

Ques:4 #include<iostream>

```
using namespace std;
```

```
struct student{
```

```
    char a; int c;  char b;
```

```
};
```

```
int main()
```

```
{
```

```
    cout<<sizeof(student);
```





```
    return 0;  
}
```

- (A) 4
- (B) 6
- (C) 8
- (D) 12

Ques:5 The data elements in structure are also known as ?

- (A) data
- (B) members
- (C) objects
- (D) none of these

Ans: A

Ans: B

Ans: C

Ans: D

Ans: B

## 5.8 REFERENCES/SUGGESTED READING



## Chapter 6

### Classes and Objects

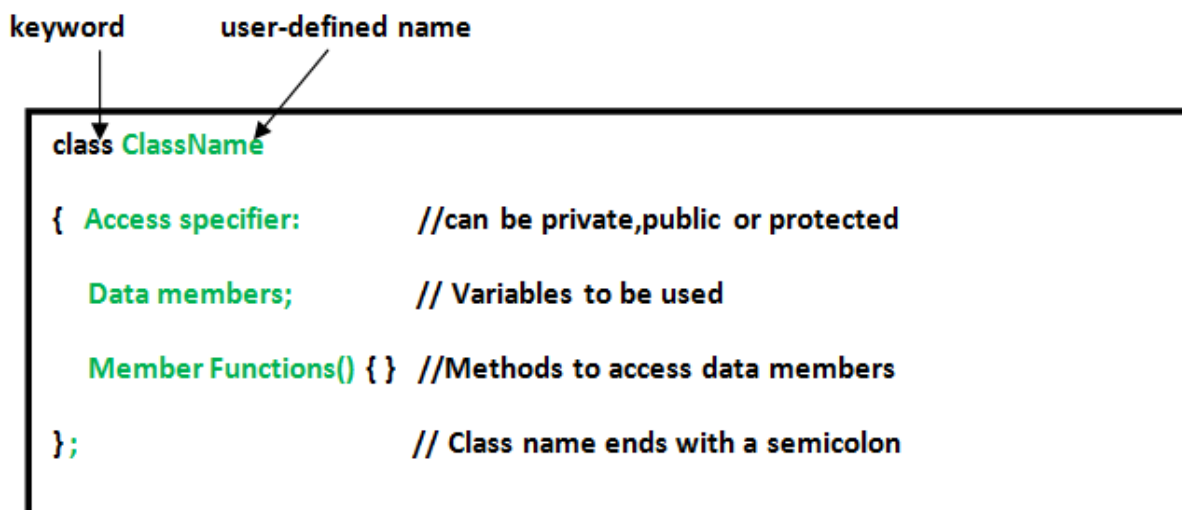
#### Introduction of Classes and Objects

The main purpose of C++ programming is to add object orientation to the C programming language and classes are the central feature of C++ that supports object-oriented programming and are often called user-defined types.

A class is used to specify the form of an object and it combines data representation and methods for manipulating that data into one neat package. The data and functions within a class are called members of the class.

#### 5.7 Class Definition

When you define a class, you define a blueprint for a data type. This doesn't actually define any data, but it does define what the class name means, that is, what an object of the class will consist of and what operations can be performed on such an object.



A class definition starts with the keyword class followed by the class name; and the class body, enclosed by a pair of curly braces. A class definition must be followed either by a semicolon or a list of declarations. For example, we defined the Box data type using the keyword class as follows –



```
class Box {  
    public:  
        double length; // Length of a box  
        double breadth; // Breadth of a box  
        double height; // Height of a box  
};
```

The keyword `public` determines the access attributes of the members of the class that follows it. A public member can be accessed from outside the class anywhere within the scope of the class object. You can also specify the members of a class as private or protected which we will discuss in this chapter.

### 5.8 Class variable/ Object Declaration

A class provides the blueprints for objects, so basically an object is created from a class. We declare objects of a class with exactly the same sort of declaration that we declare variables of basic types. Following statements declare two objects of class `Box` –

```
Box Box1;    // Declare Box1 of type Box  
Box Box2;    // Declare Box2 of type Box
```

Both of the objects `Box1` and `Box2` will have their own copy of data members.

### 5.9 Defining and Accessing Member Functions

Data hiding is one of the important features of Object Oriented Programming which allows preventing the functions of a program to access directly the internal representation of a class type. The access restriction to the class members is specified by the labeled `public`, `private`, and `protected` sections within the class body. The keywords `public`, `private`, and `protected` are called access specifiers.

A class can have multiple `public`, `protected`, or `private` labeled sections. Each section remains in effect until either another section label or the closing right brace of the class body is seen. The default access for members and classes is `private`.

```
class Base {
```



public:

// public members go here

protected:

// protected members go here

private:

// private members go here

};

#### The public Members

A public member is accessible from anywhere outside the class but within a program. You can set and get the value of public variables without any member function as shown in the following example –

```
#include<iostream>

usingnamespace std;

classLine{
public:
double length;
void setLength(double len );
double getLength(void);
};

// Member functions definitions
```



```
doubleLine::getLength(void){
return length ;
}

voidLine::setLength(double len){
    length = len;
}

// Main function for the program
int main(){
Line line;

// set line length
line.setLength(6.0);
cout <<"Length of line : "<< line.getLength()<<endl;

// set line length without member function
line.length =10.0;// OK: because length is public
cout <<"Length of line : "<< line.length <<endl;

return0;
}
```

When the above code is compiled and executed, it produces the following result –

Length of line : 6

Length of line : 10



### The private Members

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.

By default all the members of a class would be private, for example in the following class width is a private member, which means until you label a member, it will be assumed a private member –

```
class Box {  
    double width;  
  
    public:  
        double length;  
        void setWidth( double wid );  
        double getWidth( void );  
};
```

Practically, we define data in private section and related functions in public section so that they can be called from outside of the class as shown in the following program.

```
#include<iostream>  
  
usingnamespace std;  
  
classBox{  
public:  
    double length;  
    void setWidth(double wid );  
    double getWidth(void);  
};
```



```
private:
double width;
};

// Member functions definitions
doubleBox::getWidth(void){
return width ;
}

voidBox::setWidth(double wid ){
    width = wid;
}

// Main function for the program
int main(){
Box box;

// set box length without member function
box.length =10.0;// OK: because length is public
cout <<"Length of box : "<< box.length <<endl;

// set box width without member function
// box.width = 10.0; // Error: because width is private
box.setWidth(10.0);// Use member function to set it.
cout <<"Width of box : "<< box.getWidth()<<endl;
```



```
return0;  
}
```

When the above code is compiled and executed, it produces the following result –

Length of box : 10

Width of box : 10

The protected Members

A protected member variable or function is very similar to a private member but it provided one additional benefit that they can be accessed in child classes which are called derived classes.

You will learn derived classes and inheritance chapter 9. For now you can check following example where I have derived one child class SmallBox from a parent class Box.

Following example is similar to above example and here width member will be accessible by any member function of its derived class SmallBox.

```
#include<iostream>  
usingnamespace std;  
  
classBox{  
protected:  
double width;  
};  
  
classSmallBox:Box{// SmallBox is the derived class.  
public:  
void setSmallWidth(double wid );  
double getSmallWidth(void);
```





```
};

// Member functions of child class
doubleSmallBox::getSmallWidth(void){
return width ;
}

voidSmallBox::setSmallWidth(double wid ){
    width = wid;
}

// Main function for the program
int main(){
    SmallBox box;

    // set box width using member function
    box.setSmallWidth(5.0);
    cout <<"Width of box : "<< box.getSmallWidth()<< endl;

    return0;
}
```

When the above code is compiled and executed, it produces the following result –

Width of box : 5



### 5.10 Const(Constant) Object

Class objects can also be made const by using the const keyword. Initialization is done via class constructors:

```
1 const Date date1; // initialize using default constructor
```

```
2 const Date date2(2020, 10, 16); // initialize using parameterized constructor
```

```
3 const Date date3 { 2020, 10, 16 }; // initialize using parameterized constructor (C++11)
```

Once a const class object has been initialized via constructor, any attempt to modify the member variables of the object is disallowed, as it would violate the const-ness of the object. This includes both changing member variables directly (if they are public), or calling member functions that set the value of member variables. Consider the following class:

```
class Something
{
public:
    int m_value;

    Something(): m_value{0} { }

    void setValue(int value) { m_value = value; }
    int getValue() { return m_value ; }
};

int main()
{
    const Something something{ }; // calls default constructor
```



```
something.m_value = 5; // compiler error: violates const

something.setValue(5); // compiler error: violates const

return 0;
}
```

Both of the above lines involving variable something are illegal because they violate the constness of something by either attempting to change a member variable directly, or by calling a member function that attempts to change a member variable.

Just like with normal variables, you'll generally want to make your class objects const when you need to ensure they aren't modified after creation.

### 5.11 Const Member Function

Now, consider the following line of code:

```
1 std::cout << something.getValue();
```

Perhaps surprisingly, this will also cause a compile error, even though getValue() doesn't do anything to change a member variable! It turns out that const class objects can only explicitly call *const* member functions, and getValue() has not been marked as a const member function.

A const member function is a member function that guarantees it will not modify the object or call any non-const member functions (as they may modify the object).

To make getValue() a const member function, we simply append the const keyword to the function prototype, after the parameter list, but before the function body:

```
class Something
{
public:
    int m_value;
```



```
Something(): m_value{0} { }

void resetValue() { m_value = 0; }

void setValue(int value) { m_value = value; }

int getValue() const { return m_value; } // note addition of const keyword after parameter
list, but before function body
};
```

Now `getValue()` has been made a `const` member function, which means we can call it on any `const` objects.

For member functions defined outside of the class definition, the `const` keyword must be used on both the function prototype in the class definition and on the function definition:

```
class Something
{
public:
    int m_value;

    Something(): m_value{0} { }

    void resetValue() { m_value = 0; }

    void setValue(int value) { m_value = value; }

    int getValue() const; // note addition of const keyword here
};
```



```
int Something::getValue() const // and here
{
    return m_value;
}
```

Futhermore, any const member function that attempts to change a member variable or call a non-const member function will cause a compiler error to occur. For example:

```
class Something
{
public:
    int m_value ;

    void resetValue() const { m_value = 0; } // compile error, const functions can't change
member variables.
};
```

In this example, resetValue() has been marked as a const member function, but it attempts to change m\_value. This will cause a compiler error.

Note that constructors cannot be marked as const. This is because constructors need to be able to initialize their member variables, and a const constructor would not be able to do so. Consequently, the language disallows const constructors.

### Const references

Although instantiating const class objects is one way to create const objects, a more common way is by passing an object to a function by const reference.

In the lesson on passing arguments by reference, we covered the merits of passing class arguments by const reference instead of by value. To recap, passing a class argument by value causes a copy of the class to be made (which is slow) -- most of the time, we don't need a copy, a reference to the original argument works just fine, and is more performant because it avoids the needless copy. We typically



make the reference const in order to ensure the function does not inadvertently change the argument, and to allow the function to work with R-values (e.g. literals), which can be passed as const references, but not non-const references.

Can you figure out what's wrong with the following code?

```
#include <iostream>

class Date
{
private:
    int m_year;
    int m_month;
    int m_day;

public:
    Date(int year, int month, int day)
    {
        setDate(year, month, day);
    }

    void setDate(int year, int month, int day)
    {
        m_year = year;
        m_month = month;
        m_day = day;
    }
}
```



```
int getYear() { return m_year; }

int getMonth() { return m_month; }

int getDay() { return m_day; }

};

// note: We're passing date by const reference here to avoid making a copy of date
void printDate(const Date &date)
{
    std::cout << date.getYear() << '/' << date.getMonth() << '/' << date.getDay() << '\n';
}

int main()
{
    Date date{2016, 10, 16};

    printDate(date);

    return 0;
}
```

The answer is that inside of the `printDate` function, `date` is treated as a `const` object. And with that `const` `date`, we're calling functions `getYear()`, `getMonth()`, and `getDay()`, which are all non-`const`. Since we can't call non-`const` member functions on `const` objects, this will cause a compile error.

The fix is simple: make `getYear()`, `getMonth()`, and `getDay()` `const`:

```
class Date
{
```



```
private:

    int m_year;

    int m_month;

    int m_day;

public:

    Date(int year, int month, int day)
    {
        setDate(year, month, day);
    }

    // setDate() cannot be const, modifies member variables
    void setDate(int year, int month, int day)
    {
        m_year = year;
        m_month = month;
        m_day = day;
    }

    // The following getters can all be made const
    int getYear() const { return m_year; }
    int getMonth() const { return m_month; }
    int getDay() const { return m_day; }
};
```

Now in function printDate(), const date will be able to successfully call getYear(), getMonth(), and





getDay().

### 5.12 Object as Member of Classes

An object of a class can also be a data member for another class.

Here, in the given program, we are doing the same. We have a class named Marks that contains two data members rno - to store roll number and perc - to store the percentage of the student. We have another class named Students that contains two members name - to store the name of the student and objM - which is an object of Marks class.

In Students class there are two member functions:

1. readStudent() - To read the name of the student, and Here, we called readMarks() member function of Marks class by using objM - it will read roll number and percentage.
2. printStudent() - To print the name of the student, and Here, we called printMarks() member function of Marks class by using objM - it will print roll number and percentage.

Program:

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

Using namespace std;

Class marks
{
    Private:
        int  rno;
        float perc;
    Public:
        //constructor
        Marks()
        {rno=0;perc=0.0f;}
}
```



```
//input roll numbers and percentage

Void read marks(void)
{
    Cout<<"Roll No.:"<<rno<<endl;
    Cin>>rno;
    Cout<<"Enter percentage:";
    Cin>>perc;
}

//print roll number and percentage

Void printMarks(void)
{
    Cout<<"Roll No.:"<<rno<<endl;
    Cout<<"Percentage:"<<perc<<"%"<<endl;
}

};

Class student
{
    Private:

    //object to Marks class

    Marks objM;
    Char name[30];
```



Public:

```
//input student details

Void resdStudent(void)
{
    //Input name
    Cout<<"Enter name:"
    Cin.getline(name,30);
    //input Marks
    objM.readMarks();
}

//print student details

Void printStudent(void)
{
    //print name
    Cout<<"Name:"<<name<<endl;

    //print Marks
    objM.printMarks();
}

};

// main code
int main()
{
    //creat object to student class
```



```
Student ss;  
ss.readStudent();  
ss.printStudent();  
  
return 0;  
}
```

Output:

Enter name: Krishan Kumar

Enter roll number 111

Enter percentage 76.32

Name: Krishan Kumar

Roll No.: 111

Percentage: 76.32

## 6.8 SUMMARY

- A class is used to specify the form of an object and it combines data representation and methods for manipulating that data into one neat package
- A class definition starts with the keyword `class` followed by the class name; and the class body, enclosed by a pair of curly braces. A class definition must be followed either by a semicolon or a list of declarations
- A public member is accessible from anywhere outside the class but within a program.
- A private member variable or function cannot be accessed, or even viewed from outside the class.
- A protected member variable or function is very similar to a private member but it provided one additional benefit that they can be accessed in child classes which are called derived classes.

## Multiple Choice Questions

Q1. Which of the following means "The use of an object of one class in definition of another class"?



- a. Encapsulation
- b. Inheritance
- c. Composition
- d. Abstraction

Q2 Which of the following statements is correct?

- a. Data items in a class must be private.
- b. Both data and functions can be either private or public.
- c. Member functions of a class must be private.
- d. Constructor of a class cannot be private.

Q3. How many objects can be created from an abstract class

- a. Zero
- b. One
- c. Two
- d. As many as we want

Q4. Class is \_\_\_\_\_ of an object.

- a) Basic function definition
- b) Detailed description with values
- c) Blueprint
- d) Set of constant values

Q5. Which of the following also known as an instance of a class?

- a. Friend Functions
- b. Object
- c. Member Functions
- d. Member Variables

Ans: 1-c      2-d      3-d      4-c      5-b

**Long Answer Type Questions**

- Q1. Define class in C++. Write down the syntax to declare a class. What is access specifier of member of a class? Write a program to create a class information of students of class.
- Q2. How can we define member functions of a class explicitly?
- Q3. What do you mean by constant member function and constant objects?
- Q4. Define Public, private and protected access specifiers. How can we differentiate them?

**References**

The C++ Programming Language' (5<sup>th</sup> Edition) by Bjarne Stroustrup.

C++ Primer' (5<sup>th</sup> Edition) by Stanley B. Lippman, Josée Lajoie, and Barbara E. Moo



## Chapter 7

### Constructors and Destructors

#### Constructor

The previous chapter class `Box` shows two ways that member functions can be used to give values to the data items in an object. Sometimes, however, it's convenient if an object can initialize itself when it's first created, without requiring a separate call to a member function. Automatic initialization is carried out using a special member function called a *constructor*. A constructor is a member function that is executed automatically whenever an object is created.

As an example, we'll create a class of objects that might be useful as a general-purpose programming element. A *counter* is a variable that counts things. Maybe it counts file accesses,

or the number of times the user presses the Enter key, or the number of customers entering a bank. Each time such an event takes place, the counter is incremented (1 is added to it). The counter can also be accessed to find the current count. Let's assume that this counter is important in the program and must be accessed by many different functions. In procedural languages such as C, a counter would probably be implemented as a global variable. However, global variables complicate the program's design and may be modified accidentally. This example, `COUNTER`, provides a counter variable that can be modified only through its member functions.

```
// counter.cpp
// object represents a counter variable

#include <iostream>
using namespace std;

class Counter
{
    private:
```



```
    unsigned int count; //count

public:

    Counter() : count(0) //constructor
    { /*empty body*/ }

    void inc_count() //increment count
    { count++; }

    int get_count() //return count
    { return count; }

};

////////////////////////////////////

int main()
{
    Counter c1, c2; //define and initialize
    cout << "c1=" << c1.get_count() << endl; //display
    cout << "c2=" << c2.get_count() << endl;
    c1.inc_count(); //increment c1
    c2.inc_count(); //increment c2
    c2.inc_count(); //increment c2
    cout << "c1=" << c1.get_count() << endl; //display again
    cout << "c2=" << c2.get_count() << endl;
    cout << endl;
    return 0;
}
```





The Counter class has one data member: count, of type unsigned int (since the count is always positive). It has three member functions: the constructor Counter(), which we'll look at in a moment; inc\_count(), which adds 1 to count; and get\_count(), which returns the current value of count.

- **Automatic Initialization**

When an object of type Counter is first created, we want its count to be initialized to 0. After all, most counts start at 0. We could provide a set\_count() function to do this and call it with an argument of 0, or we could provide a zero\_count() function, which would always set count to 0. However, such functions would need to be executed every time we created a Counter object.

```
Counter c1; //every time we do this,  
c1.zero_count(); //we must do this too
```

This is mistake prone, because the programmer may forget to initialize the object after creating it. It's more reliable and convenient, especially when there are a great many objects of a given class, to cause each object to initialize itself when it's created. In the Counter class, the constructor Counter() does this. This function is called automatically whenever a new object of type Counter is created. Thus in main() the statement

```
Counter c1, c2;
```

creates two objects of type Counter. As each is created, its constructor, Counter(), is executed. This function sets the count variable to 0. So the effect of this single statement is to not only create two objects, but also to initialize their count variables to 0.

- **Same Name as the Class**

There are some unusual aspects of constructor functions. First, it is no accident that they have exactly the same name (Counter in this example) as the class of which they are members. This is one way the compiler knows they are constructors.

Second, no return type is used for constructors. Why not? Since the constructor is called automatically by the system, there's no program for it to return anything to; a return value wouldn't make sense. This is the second way the compiler knows they are constructors.

- **Initializer List**

One of the most common tasks a constructor carries out is initializing data members. In the Counter



class the constructor must initialize the count member to 0. You might think that this would be done in the constructor's function body, like this:

```
count()
{ count = 0; }
```

However, this is not the preferred approach (although it does work). Here's how you should initialize a data member:

```
count() : count(0)
{ }
```

The initialization takes place following the member function declarator but before the function body. It's preceded by a colon. The value is placed in parentheses following the member data. If multiple members must be initialized, they're separated by commas. The result is the *initializer list* (sometimes called by other names, such as the *member-initialization list*).

```
someClass() : m1(7), m2(33), m2(4) ← initializer list
{ }
```

Why not initialize members in the body of the constructor? The reasons are complex, but have to do with the fact that members initialized in the initializer list are given a value before the constructor even starts to execute. This is important in some situations. For example, the initializer list is the only way to initialize const member data and references. Actions more complicated than simple initialization must be carried out in the constructor body, as with ordinary functions.

### Counter Output

The main() part of this program exercises the Counter class by creating two counters, c1 and c2. It causes the counters to display their initial values, which—as arranged by the constructor—are 0. It then increments c1 once and c2 twice, and again causes the counters to display themselves (non-criminal behavior in this context). Here's the output:

```
c1=0
```

```
c2=0
```



```
c1=1
```

```
c2=2
```

If this isn't enough proof that the constructor is operating as advertised, we can rewrite the constructor to print a message when it executes.

```
Counter() : count(0)
```

```
{ cout << "I'm the constructor\n"; }
```

Now the program's output looks like this:

```
I'm the constructor
```

```
I'm the constructor
```

### 5.13 Parameterize Constructors

Parameterized constructor in C++ are some special types of method which gets instantiated as soon as an object is created. Therefore, there are two types of constructors defined in C++ namely default constructor, Parametrized constructor. There is a minute difference between default constructor and Parametrized constructor. The default constructor is a type of constructor which has no arguments but yes object instantiation is performed there also. On the other hand, as the name suggests Parameterized constructor is a special type of constructor where an object is created, and further parameters are passed to distinct objects.

Whenever a parameterized constructor is defined simultaneously an object gets instantiated which holds details or the values and parameters the object will contain or possess. It becomes a possible situation to pass arguments to that object. To create a parameterized constructor, it is needed to just add parameters as a value to the object as the way we pass a value to a function.

Somewhat similar scenario we do by passing the parameterized values to the object created with the class. Parameters are used to initialize the objects which are defined in the constructor's body. Whenever a parameterized constructor is declared the values should be passed as arguments to the function of constructor i.e. constructor functions otherwise the conventional way of object declaration will not work. These constructors can be called both implicitly or explicitly.



There are some uses or benefits of using parameterized constructors:

- When constructors are created or instantiated, they are used to initialize and hold the various data elements of different objects having different values.
- One more interesting scenario is that they are used to overload constructors.

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

Class ParamA
{
    Private:
        int b,c;
    Public:
        ParamA(int b1, int c1)
        {
            b = b1;
            c = c1;
        }
        int getX()
        {
            return b;
        }
        int getY()
        {
            return c;
        }
};
```



```
int main()

{
    ParamA p1(50,80)
    cout << "p1.b = " << p1. getX() <<endl<< " p1.c = " << p1.getY();
    return 0;
}
```

Output:

P1.b = 50

p1.c = 80

In this class, ParamA contains two access specifiers one as a private access specifier and one as a public access specifier. Private access specifier involves a declaration of two variables which will be called and references later at some point of time. Followed by public access specifier where the constructor implementation gets started ParamA (int b1, int c1) refers to constructor initialization with int b1 and int c1 as parameters to be passed as values to the object which will call these values later. Output comes out as 50 and 80(values being passed).

```
#include <iostream>
using namespace std;
class faculty {
public:
    int ide;
    string name;
    float pay;
    faculty (int i, string n, float s)
    {
        ide = i;
        name = n;
```



```
pay = s;
}
void show_data ()
{
cout<<ide<<" "<<name<<" "<<pay<<endl;
}
};
int main(void) {
faculty fac1=faculty(501, "Girish", 89700);
faculty fac2=faculty(151, "Anu", 75900);
fac1.show_data();
fac2. show_data ();
return 0;
}
```

Output:

501 Girish 89700

151 Kalpana 75900

**Explanation:** In this example, a class faculty is declared which includes an access specifier as public type and then it is followed with data members as int ide and string name the output which includes the implementation of the constructor will display the name of the faculty, Ide of the faculty and pay of the faculty.

## 5.14 Default Constructors

A constructor that takes no parameters (or has parameters that all have default values) is called a **default constructor**. The default constructor is called if no user-provided initialization values are provided.

Here is an example of a class that has a default constructor:



```
#include <iostream>

class Fraction
{
private:
    int m_numerator;
    int m_denominator;

public:
    Fraction() // default constructor
    {
        m_numerator = 0;
        m_denominator = 1;
    }

    int getNumerator() { return m_numerator; }
    int getDenominator() { return m_denominator; }
    double getValue() { return static_cast<double>(m_numerator) / m_denominator; }
};

int main()
{
    Fraction frac; // Since no arguments, calls Fraction() default constructor
    std::cout << "The fractional number is:" << endl << frac.getNumerator() << "/" <<
    frac.getDenominator() << "\n";
}
```



```
    return 0;  
}
```

Output:

The fractional number is:

0/1

This class was designed to hold a fractional value as an integer numerator and denominator. We have defined a default constructor named Fraction (the same as the class).

Because we're instantiating an object of type Fraction with no arguments, the default constructor will be called immediately after memory is allocated for the object, and our object will be initialized.

Note that our numerator and denominator were initialized with the values we set in our default constructor! Without a default constructor, the numerator and denominator would have garbage values until we explicitly assigned them reasonable values, or initialize them by other means

## 5.15 Copy Constructors

### The Default Copy Constructor

We've seen two ways to initialize objects. A no-argument constructor can initialize data members to constant values, and a multi-argument constructor can initialize data members to values passed as arguments. Let's mention another way to initialize an object: you can initialize it with *another object of the same type*. Surprisingly, you don't need to create a special constructor for this; one is already built into all classes. It's called the *default copy constructor*. It's a one-argument constructor whose argument is an object of the same class as the constructor. The ECOPYCON program shows how this constructor is used.





```
// ecopycon.cpp

// initialize objects using default copy constructor

#include <iostream>

using namespace std;

////////////////////////////////////

class Distance //English Distance class
{
private:
int feet;
float inches;
public:
//constructor (no args)
Distance() : feet(0), inches(0.0)
{ }
//Note: no one-arg constructor
//constructor (two args)
Distance(int ft, float in) : feet(ft), inches(in)
{ }
void getdist() //get length from user
{
cout << "\nEnter feet: "; cin >> feet;
cout << "Enter inches: "; cin >> inches;
}
void showdist() //display distance
{ cout << feet << "\'-" << inches << \'"; }
```



```
};  
  
/////////////////////////////////////  
  
int main()  
{  
    Distance dist1(11, 6.25); //two-arg constructor  
    Distance dist2(dist1); //one-arg constructor  
    Distance dist3 = dist1; //also one-arg constructor  
    //display all lengths  
    cout << "\ndist1 = "; dist1.showdist();  
    cout << "\ndist2 = "; dist2.showdist();  
    cout << "\ndist3 = "; dist3.showdist();  
    cout << endl;  
    return 0;  
}
```

We initialize dist1 to the value of 11'-6.25" using the two-argument constructor. Then we define two more objects of type Distance, dist2 and dist3, initializing both to the value of dist1. You might think this would require us to define a one-argument constructor, but initializing an object with another object of the same type is a special case. These definitions both use the default copy constructor. The object dist2 is initialized in the statement `Distance dist2(dist1);`

## 5.16 Destructor

### Destructors

We've seen that a special member function—the constructor—is called automatically when an object is first created. You might guess that another function is called automatically when an object is destroyed. This is indeed the case. Such a function is called a *destructor*. A destructor has the same name as the constructor (which is the same as the class name) but is preceded by a tilde:



```
class Foo
{
private:
int data;

public:
Foo() : data(0) //constructor (same name as class)
{ }
~Foo() //destructor (same name with tilde)
{ }
};
```

Like constructors, destructors do not have a return value. They also take no arguments (the assumption being that there's only one way to destroy an object). The most common use of destructors is to deallocate memory that was allocated for the object by the constructor.

```
#include <iostream>
using namespace std;
class ParamCode {
public:
int x;
ParamCode (int i);
~ParamCode ();
};
ParamCode::ParamCode (int i) {
x = i;
}
ParamCode::~~ParamCode() {
```



```
cout<< "Destructing those objects whose x value is " << x <<" \n";
}
int main () {
ParamCode t1(20);
ParamCode t2(15);
cout<< t1.x << " " << t2.x <<"\n";
return 0;
}
```

Output:

20 15

Destructing those objects whose x value is 15

Destructing those objects whose x value is 20

### 5.17 Friend Function

A **friend function** is a function that can access the private members of a class as though it were a member of that class. In all other regards, the friend function is just like a normal function. A friend function may be either a normal function, or a member function of another class.

#### 2.1 Declaration of Friend Function

To declare a friend function, simply use the *friend* keyword in front of the prototype of the function you wish to be a friend of the class. It does not matter whether you declare the friend function in the private or public section of the class.

Here's an example of using a friend function:

```
class Accumulator
{
private:
```



```
int m_value;

public:

    Accumulator() { m_value = 0; }

    void add(int value) { m_value += value; }

    // Make the reset() function a friend of this class
    friend void reset(Accumulator &accumulator);
};

// reset() is now a friend of the Accumulator class
void reset(Accumulator &accumulator)
{
    // And can access the private data of Accumulator objects
    accumulator.m_value = 0;
}

int main()
{
    Accumulator acc;

    acc.add(5); // add 5 to the accumulator

    reset(acc); // reset the accumulator to 0

    return 0;
}
```

In this example, we've declared a function named `reset()` that takes an object of class `Accumulator`, and



sets the value of `m_value` to 0. Because `reset()` is not a member of the `Accumulator` class, normally `reset()` would not be able to access the private members of `Accumulator`. However, because `Accumulator` has specifically declared this `reset()` function to be a friend of the class, the `reset()` function is given access to the private members of `Accumulator`.

Note that we have to pass an `Accumulator` object to `reset()`. This is because `reset()` is not a member function. It does not have a `*this` pointer, nor does it have an `Accumulator` object to work with, unless given one.

Here's another example:

```
class Value
{
private:
    int m_value;
public:
    Value(int value) { m_value = value; }
    friend bool isEqual(const Value &value1, const Value &value2);
};

bool isEqual(const Value &value1, const Value &value2)
{
    return (value1.m_value == value2.m_value);
}
```

In this example, we declare the `isEqual()` function to be a friend of the `Value` class. `isEqual()` takes two `Value` objects as parameters. Because `isEqual()` is a friend of the `Value` class, it can access the private members of all `Value` objects. In this case, it uses that access to do a comparison on the two objects, and returns true if they are equal.

### Multiple friends



A function can be a friend of more than one class at the same time. For example, consider the following example:

```
#include <iostream>

class Humidity;

class Temperature
{
private:
    int m_temp;
public:
    Temperature(int temp=0) { m_temp = temp; }

    friend void printWeather(const Temperature &temperature, const Humidity &humidity);
};

class Humidity
{
private:
    int m_humidity;
public:
    Humidity(int humidity=0) { m_humidity = humidity; }

    friend void printWeather(const Temperature &temperature, const Humidity &humidity);
};
```



```
void printWeather(const Temperature &temperature, const Humidity &humidity)
{
    std::cout << "The temperature is " << temperature.m_temp <<
        " and the humidity is " << humidity.m_humidity << "\n";
}

int main()
{
    Humidity hum(10);
    Temperature temp(12);

    printWeather(temp, hum);

    return 0;
}
```

There are two things worth noting about this example. First, because PrintWeather is a friend of both classes, it can access the private data from objects of both classes. Second, note the following line at the top of the example:

```
1 class Humidity;
```

This is a class prototype that tells the compiler that we are going to define a class called Humidity in the future. Without this line, the compiler would tell us it doesn't know what a Humidity is when parsing the prototype for PrintWeather() inside the Temperature class. Class prototypes serve the same role as function prototypes -- they tell the compiler what something looks like so it can be used now and defined later. However, unlike functions, classes have no return types or parameters, so class prototypes are always simply `class ClassName`, where `ClassName` is the name of the class.





### 5.18 Inline Friend Function

Inline function is one of the important features of C++. So, let's first understand why inline functions are used and what is the purpose of inline function?

When the program executes the function call instruction the CPU stores the memory address of the instruction following the function call, copies the arguments of the function on the stack and finally transfers control to the specified function. The CPU then executes the function code, stores the function return value in a predefined memory location/register and returns control to the calling function. This can become overhead if the execution time of function is less than the switching time from the caller function to called function (callee). For functions that are large and/or perform complex tasks, the overhead of the function call is usually insignificant compared to the amount of time the function takes to run. However, for small, commonly-used functions, the time needed to make the function call is often a lot more than the time needed to actually execute the function's code. This overhead occurs for small functions because execution time of small function is less than the switching time.

C++ provides an inline functions to reduce the function call overhead. Inline function is a function that is expanded in line when it is called. When the inline function is called whole code of the inline function gets inserted or substituted at the point of inline function call. This substitution is performed by the C++ compiler at compile time. Inline function may increase efficiency if it is small.

The syntax for defining the function inline is:

```
inline return-type function-name(parameters)
{
    // function code
}
```

Remember, inlining is only a request to the compiler, not a command. Compiler can ignore the request for inlining. Compiler may not perform inlining in such circumstances like:

- 1) If a function contains a loop. (for, while, do-while)
- 2) If a function contains static variables.
- 3) If a function is recursive.



- 4) If a function return type is other than void, and the return statement doesn't exist in function body.
- 5) If a function contains switch or goto statement.

**Inline functions provide following advantages:**

- 1) Function call overhead doesn't occur.
- 2) It also saves the overhead of push/pop variables on the stack when function is called.
- 3) It also saves overhead of a return call from a function.
- 4) When you inline a function, you may enable compiler to perform context specific optimization on the body of function. Such optimizations are not possible for normal function calls. Other optimizations can be obtained by considering the flows of calling context and the called context.
- 5) 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.

**Inline function disadvantages:**

- 1) The added variables from the inlined function consumes additional registers, After in-lining function if variables number which are going to use register increases than they may create overhead on register variable resource utilization. This means that when inline function body is substituted at the point of function call, total number of variables used by the function also gets inserted. So the number of register going to be used for the variables will also get increased. So if after function inlining variable numbers increase drastically then it would surely cause an overhead on register utilization.
- 2) If you use too many inline functions then the size of the binary executable file will be large, because of the duplication of same code.
- 3) Too much inlining can also reduce your instruction cache hit rate, thus reducing the speed of instruction fetch from that of cache memory to that of primary memory.
- 4) Inline function may increase compile time overhead if someone changes the code inside the inline function then all the calling location has to be recompiled because compiler would require to replace all the code once again to reflect the changes, otherwise it will continue with old functionality.
- 5) Inline functions may not be useful for many embedded systems. Because in embedded systems code size is more important than speed.
- 6) Inline functions might cause thrashing because inlining might increase size of the binary executable file. Thrashing in memory causes performance of computer to degrade.

The following program demonstrates the use of use of inline function.



```
#include <iostream>
using namespace std;
inline int cube(int s)
{
    return s*s*s;
}
int main()
{
    cout << "The cube of 3 is: " << cube(3) << "\n";
    return 0;
} //Output: The cube of 3 is: 27
```

### 5.19 Friend Class

#### Friend classes

It is also possible to make an entire class a friend of another class. This gives all of the members of the friend class access to the private members of the other class. Here is an example:

```
#include <iostream>

class Storage
{
private:
    int m_nValue;
    double m_dValue;
public:
```



```
Storage(int nValue, double dValue)
{
    m_nValue = nValue;
    m_dValue = dValue;
}

// Make the Display class a friend of Storage
friend class Display;
};

class Display
{
private:
    bool m_displayIntFirst;

public:
    Display(bool displayIntFirst) { m_displayIntFirst = displayIntFirst; }

    void displayItem(const Storage &storage)
    {
        if (m_displayIntFirst)
            std::cout << storage.m_nValue << ' ' << storage.m_dValue << '\n';
        else // display double first
            std::cout << storage.m_dValue << ' ' << storage.m_nValue << '\n';
    }
}
```



```
};

int main()
{
    Storage storage(5, 6.7);
    Display display(false);

    display.displayItem(storage);

    return 0;
}
```

Because the Display class is a friend of Storage, any of Display's members that use a Storage class object can access the private members of Storage directly. This program produces the following result:

6.7 5

A few additional notes on friend classes. First, even though Display is a friend of Storage, Display has no direct access to the `*this` pointer of Storage objects. Second, just because Display is a friend of Storage, that does not mean Storage is also a friend of Display. If you want two classes to be friends of each other, both must declare the other as a friend. Finally, if class A is a friend of B, and B is a friend of C, that does not mean A is a friend of C.

Be careful when using friend functions and classes, because it allows the friend function or class to violate encapsulation. If the details of the class change, the details of the friend will also be forced to change. Consequently, limit your use of friend functions and classes to a minimum.

### Friend member functions

Instead of making an entire class a friend, you can make a single member function a friend. This is done similarly to making a normal function a friend, except using the name of the member function with the `className::` prefix included (e.g. `Display::displayItem`).



However, in actuality, this can be a little trickier than expected. Let's convert the previous example to make `Display::displayItem` a friend member function. You might try something like this:

```
class Display; // forward declaration for class Display

class Storage
{
private:
    int m_nValue;
    double m_dValue;
public:
    Storage(int nValue, double dValue)
    {
        m_nValue = nValue;
        m_dValue = dValue;
    }

    // Make the Display::displayItem member function a friend of the Storage class
    friend void Display::displayItem(const Storage& storage); // error: Storage hasn't seen
the full definition of class Display
};

class Display
{
private:
    bool m_displayIntFirst;
```



```
public:
    Display(bool displayIntFirst) { m_displayIntFirst = displayIntFirst; }

    void displayItem(const Storage &storage)
    {
        if (m_displayIntFirst)
            std::cout << storage.m_nValue << ' ' << storage.m_dValue << '\n';
        else // display double first
            std::cout << storage.m_dValue << ' ' << storage.m_nValue << '\n';
    }
};
```

However, it turns out this won't work. In order to make a member function a friend, the compiler has to have seen the full definition for the class of the friend member function (not just a forward declaration). Since class `Storage` hasn't seen the full definition for class `Display` yet, the compiler will error at the point where we try to make the member function a friend.

Fortunately, this is easily resolved simply by moving the definition of class `Display` before the definition of class `Storage`.

```
class Display
{
private:
    bool m_displayIntFirst;

public:
    Display(bool displayIntFirst) { m_displayIntFirst = displayIntFirst; }
```



```
void displayItem(const Storage &storage) // error: compiler doesn't know what a
Storage is
{
    if (m_displayIntFirst)
        std::cout << storage.m_nValue << ' ' << storage.m_dValue << '\n';
    else // display double first
        std::cout << storage.m_dValue << ' ' << storage.m_nValue << '\n';
}

};

class Storage
{
private:
    int m_nValue;
    double m_dValue;
public:
    Storage(int nValue, double dValue)
    {
        m_nValue = nValue;
        m_dValue = dValue;
    }

    // Make the Display::displayItem member function a friend of the Storage class
    friend void Display::displayItem(const Storage& storage); // okay now
```





```
};
```

However, we now have another problem. Because member function `Display::displayItem()` uses `Storage` as a reference parameter, and we just moved the definition of `Storage` below the definition of `Display`, the compiler will complain it doesn't know what a `Storage` is. We can't fix this one by rearranging the definition order, because then we'll undo our previous fix.

Fortunately, this is also fixable in a couple of simple steps. First, we can add class `Storage` as a forward declaration. Second, we can move the definition of `Display::displayItem()` out of the class, after the full definition of `Storage` class.

```
#include <iostream>

class Storage; // forward declaration for class Storage

class Display
{
private:
    bool m_displayIntFirst;

public:
    Display(bool displayIntFirst) { m_displayIntFirst = displayIntFirst; }

    void displayItem(const Storage &storage); // forward declaration above needed for
this declaration line
};

class Storage // full definition of Storage class
```



```
{
private:
    int m_nValue;
    double m_dValue;
public:
    Storage(int nValue, double dValue)
    {
        m_nValue = nValue;
        m_dValue = dValue;
    }

    // Make the Display::displayItem member function a friend of the Storage class
    (requires seeing the full declaration of class Display, as above)
    friend void Display::displayItem(const Storage& storage);
};

// Now we can define Display::displayItem, which needs to have seen the full definition of
class Storage
void Display::displayItem(const Storage &storage)
{
    if (m_displayIntFirst)
        std::cout << storage.m_nValue << ' ' << storage.m_dValue << '\n';
    else // display double first
        std::cout << storage.m_dValue << ' ' << storage.m_nValue << '\n';
}
```



```
int main()
{
    Storage storage(5, 6.7);
    Display display(false);

    display.displayItem(storage);

    return 0;
}
```

Now everything will compile properly: the forward declaration of class `Storage` is enough to satisfy the declaration of `Display::displayItem()` inside the `Display` class, the full definition of `Display` satisfies declaring `Display::displayItem()` as a friend of `Storage`, and the full definition of class `Storage` is enough to satisfy the definition of member function `Display::displayItem()`. If that's a bit confusing, see the comments in the program above.

If this seems like a pain -- it is. Fortunately, this dance is only necessary because we're trying to do everything in a single file. A better solution is to put each class definition in a separate header file, with the member function definitions in corresponding `.cpp` files. That way, all of the class definitions would have been visible immediately in the `.cpp` files, and no rearranging of classes or functions is necessary!

## SUMMARY

Constructors are the special member function of C++ which is initialized at the time of object creation. There is no need of special function call for the constructors. A constructor have same name of class to which it belongs. There are two types of constructors parameterized constructor and copy constructors. The reverse of constructor is destructor that destroys object of the class.

## Multiple Choice Questions

Q 1. If class `C` inherits class `B`. And `B` has inherited class `A`. Then while creating the object of class `C`, what



will be the sequence of constructors getting called?

- a) Constructor of C then B, finally of A
- b) Constructor of A then C, finally of B
- c) Constructor of C then A, finally B
- d) Constructor of A then B, finally C

Q 2. Which among the following is not a necessary condition for constructors?

- a) Its name must be same as that of class
- b) It must not have any return type
- c) It must contain a definition body
- d) It can contains arguments

Q3. In which access should a constructor be defined, so that object of the class can be created in any function?

- a) Public
- b) Protected
- c) Private
- d) Any access specifier will work

Q4. Which among the following is true for copy constructor?

- a) The argument object is passed by reference
- b) It can be defined with zero arguments
- c) Used when an object is passed by value to a function
- d) Used when a function returns an object

Q5. How many types of constructors are available for use in general (with respect to parameters)?

- a) 2
- b) 3
- c) 4
- d) 5



Ans: 1-D      2-C      3-A      4-B      5-A

**Long Answer type question**

Q1. What do you mean by constructor? Define different types of constructor in C++.

Q2. Define Friend Function with suitable example.

Q3. Define inline friend function.

Q4. Define default constructor in detail

Q5. What is the use of destructor?

**References**

**‘C++ Primer’, Fifth Edition by Stanley B. Lippman, Barbara E. Moo**

**‘The C Programming Language’ (2<sup>nd</sup> Edition) by Brian W. Kernighan and Dennis M. Ritchie**



## Chapter 8

# Function and Operator Overloading

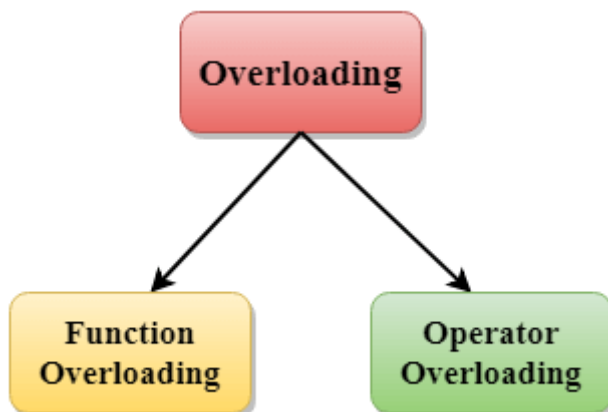
### Introduction

C++ allows you to specify more than one definition for a **function** name or an **operator** in the same scope, which is called **function overloading** and **operator overloading** respectively. An overloaded declaration is a declaration that is declared with the same name as a previously declared declaration in the same scope, except that both declarations have different arguments and obviously different definition (implementation).

When you call an overloaded **function** or **operator**, the compiler determines the most appropriate definition to use, by comparing the argument types you have used to call the function or operator with the parameter types specified in the definitions. The process of selecting the most appropriate overloaded function or operator is called **overload resolution**.

Types of overloading in C++ are:

- Function overloading
- Operator overloading



An overloaded declaration is a declaration that is declared with the same name as a previously declared declaration in the same scope, except that both declarations have different arguments and obviously different definition (implementation).



When you call an overloaded **function** or **operator**, the compiler determines the most appropriate definition to use, by comparing the argument types you have used to call the function or operator with the parameter types specified in the definitions. The process of selecting the most appropriate overloaded function or operator is called **overload resolution**.

### 5.20 Function Overloading

Function Overloading is defined as the process of having two or more function with the same name, but different in parameters is known as function overloading. In function overloading, the function is redefined by using either different types of arguments or a different number of arguments. It is only through these differences compiler can differentiate between the functions.

The **advantage** of Function overloading is that it increases the readability of the program because you don't need to use different names for the same action.

Let's see the simple example of function overloading where we are changing number of arguments of add() method.

```
#include <iostream>

using namespace std;

class Cal {
    public:
    static int add(int a,int b){
        return a + b;
    }

    static int add(int a, int b, int c)
    {
        return a + b + c;
    }
};

int main(void) {
```



```
Cal C; // class object declaration.

cout<<C.add(10, 20)<<endl;

cout<<C.add(12, 20, 23);

return 0;

}
```

**Output:****30****55****5.21 Constructor Overloading**

Like functions constructors of a class can be overloaded.

Following is the example where constructors of a class **print** is being used to print different data types

```
#include<iostream.h>

Class print
{
    public:
    void print(int i)
    {
        cout<<"printing int"<<i<<endl;
    }
    void print(double f)
    {
        cout<<"Printing float"<<f<<endl;
    }
    void print (char* c)
    {
        cout<<"Print character:"<<c<< endl;
    }
}
```





```
}  
};  
void main()  
{  
    Print pr;  
    //call print to print integer  
    pr.print(12);  
    //call print to print float  
    pr.print(123.67);  
    //call print to print character  
    pr.print("I am computer expert");  
}
```

When the above code is compiled and executed, it produces the following result –

Printing int: 12

Printing float: 123.67

Printing character: I am computer expert

## 5.22 Operator Overloading

You can redefine or overload most of the built-in operators available in C++. Thus, a programmer can use operators with user-defined types as well.

Overloaded operators are functions with special names: the keyword "operator" followed by the symbol for the operator being defined. Like any other function, an overloaded operator has a return type and a parameter list.

```
Box operator+(const Box&);
```

declares the addition operator that can be used to **add** two Box objects and returns final Box object. Most overloaded operators may be defined as ordinary non-member functions or as class member functions. In case we define above function as non-member function of a class then we would have to pass two arguments for each operand as follows –



Box operator+(const Box&, const Box&);

Following is the example to show the concept of operator over loading using a member function. Here an object is passed as an argument whose properties will be accessed using this object, the object which will call this operator can be accessed using **this** operator as explained below

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

class Box{
public:
double getVolume(void){
return length * breadth * height;
}
void setLength(double len ){
    length = len;
}
void setBreadth(double bre ){
    breadth = bre;
}
void setHeight(double hei ){
    height = hei;
}

// Overload + operator to add two Box objects.
Box operator+(const Box& b){
Box box;
```



```
        box.length =this->length + b.length;
        box.breadth =this->breadth + b.breadth;
        box.height =this->height + b.height;
return box;
}

private:
double length;// Length of a box
double breadth;// Breadth of a box
double height;// Height of a box
};

// Main function for the program
int main(){
BoxBox1;// Declare Box1 of type Box
BoxBox2;// Declare Box2 of type Box
BoxBox3;// Declare Box3 of type Box
double volume =0.0;// Store the volume of a box here

// box 1 specification
Box1.setLength(6.0);
Box1.setBreadth(7.0);
Box1.setHeight(5.0);

// box 2 specification
```



```
Box2.setLength(12.0);
Box2.setBreadth(13.0);
Box2.setHeight(10.0);

// volume of box 1
volume =Box1.getVolume();
cout <<"Volume of Box1 : "<< volume <<endl;

// volume of box 2
volume =Box2.getVolume();
cout <<"Volume of Box2 : "<< volume <<endl;

// Add two object as follows:
Box3=Box1+Box2;

// volume of box 3
volume =Box3.getVolume();
cout <<"Volume of Box3 : "<< volume <<endl;

return0;
}
```

When the above code is compiled and executed, it produces the following result –

Volume of Box1 : 210

Volume of Box2 : 1560

Volume of Box3 : 5400



### Overloadable/Non-overloadable Operators

Following is the list of operators which can be overloaded

+	-	*	/	%	^
&		~	!	,	=
<	>	<=	>=	++	--
<<	>>	==	!=	&&	
+=	-=	/=	%=	^=	&=
=	*=	<<=	>>=	[]	()
->	->*	New	new []	delete	delete []

Following is the list of operators, which cannot be overloaded –

::	.*	.	?
----	----	---	---

#### 5.22.1 Unary Operator Overloading

The unary operators operate on a single operand and following are the examples of Unary operators –

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

The unary operators operate on the object for which they were called and normally, this operator appears on the left side of the object, as in !obj, -obj, and ++obj but sometime they can be used as postfix as well like obj++ or obj--.

Following example explain how minus (-) operator can be overloaded for prefix as well as postfix



usage.

```
#include <iostream.h>

class Distance {

private:

    int feet;        // 0 to infinite

    int inches;      // 0 to 12


public:

    // required constructors

    Distance() {

        feet = 0;

        inches = 0;

    }

    Distance(int f, int i) {

        feet = f;

        inches = i;

    }


    // method to display distance

    void displayDistance() {

        cout << "F: " << feet << " I:" << inches << endl;

    }


    // overloaded minus (-) operator

    Distance operator- () {
```



```
        feet = -feet;
        inches = -inches;
        return Distance(feet, inches);
    }
};

int main() {
    Distance D1(11, 10), D2(-5, 11);

    -D1;           // apply negation
    D1.displayDistance(); // display D1

    -D2;           // apply negation
    D2.displayDistance(); // display D2

    return 0;
}
```

When the above code is compiled and executed, it produces the following result –

F: -11 I:-10

F: 5 I:-11

## 5.22.2 Binary Operator Overloading

### 5.22.2.1 Arithmetic Operator Overloading

The binary operators take two arguments and following are the examples of Binary operators. You use binary operators very frequently like addition (+) operator, subtraction (-) operator and division (/)



operator.

Following example explains how addition (+) operator can be overloaded. Similar way, you can overload subtraction (-) and division (/) operators.

```
#include <iostream>

using namespace std;

class Box {
    double length;    // Length of a box
    double breadth;    // Breadth of a box
    double height;    // Height of a box

public:

    double getVolume(void) {
        return length * breadth * height;
    }

    void setLength( double len ) {
        length = len;
    }

    void setBreadth( double bre ) {
        breadth = bre;
    }
}
```





```
void setHeight( double hei ) {
    height = hei;
}

// Overload + operator to add two Box objects.
Box operator+(const Box& b) {
    Box box;
    box.length = this->length + b.length;
    box.breadth = this->breadth + b.breadth;
    box.height = this->height + b.height;
    return box;
}
};

// Main function for the program
int main() {
    Box Box1;          // Declare Box1 of type Box
    Box Box2;          // Declare Box2 of type Box
    Box Box3;          // Declare Box3 of type Box
    double volume = 0.0; // Store the volume of a box here

    // box 1 specification
    Box1.setLength(6.0);
    Box1.setBreadth(7.0);
    Box1.setHeight(5.0);
```



```
// box 2 specification
Box2.setLength(12.0);
Box2.setBreadth(13.0);
Box2.setHeight(10.0);

// volume of box 1
volume = Box1.getVolume();
cout << "Volume of Box1 : " << volume <<endl;

// volume of box 2
volume = Box2.getVolume();
cout << "Volume of Box2 : " << volume <<endl;

// Add two object as follows:
Box3 = Box1 + Box2;

// volume of box 3
volume = Box3.getVolume();
cout << "Volume of Box3 : " << volume <<endl;

return 0;
}
```

When the above code is compiled and executed, it produces the following result –

Volume of Box1 : 210



Volume of Box2 : 1560

Volume of Box3 : 5400

#### 5.22.2.2 *Comparison Operator Overloading*

There are various relational operators supported by C++ language like (<, >, <=, >=, ==, etc.) which can be used to compare C++ built-in data types.

You can overload any of these operators, which can be used to compare the objects of a class.

Following example explains how a < operator can be overloaded and similar way you can overload other relational operators.

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

class Distance {
private:
    int feet;        // 0 to infinite
    int inches;      // 0 to 12

public:
    // required constructors
    Distance() {
        feet = 0;
        inches = 0;
    }
    Distance(int f, int i) {
```



```
    feet = f;
    inches = i;
}

// method to display distance
void displayDistance() {
    cout << "F: " << feet << " I:" << inches << endl;
}

// overloaded minus (-) operator
Distance operator- () {
    feet = -feet;
    inches = -inches;
    return Distance(feet, inches);
}

// overloaded < operator
bool operator <(const Distance& d) {
    if(feet < d.feet) {
        return true;
    }
    if(feet == d.feet && inches < d.inches) {
        return true;
    }
}
```



```
        return false;
    }
};

int main() {
    Distance D1(11, 10), D2(5, 11);

    if( D1 < D2 ) {
        cout << "D1 is less than D2 " << endl;
    } else {
        cout << "D2 is less than D1 " << endl;
    }

    return 0;
}
```

When the above code is compiled and executed, it produces the following result –

D2 is less than D1

### 5.22.3 Assignment Operator Overloading

#### 5.22.3.1 Assignment (=) Operator Overloading

You can overload the assignment operator (=) just as you can other operators and it can be used to create an object just like the copy constructor.

Following example explains how an assignment operator can be overloaded.

```
#include <iostream>
```



```
using namespace std;

class Distance {
private:
    int feet;        // 0 to infinite
    int inches;      // 0 to 12

public:
    // required constructors
    Distance() {
        feet = 0;
        inches = 0;
    }
    Distance(int f, int i) {
        feet = f;
        inches = i;
    }
    void operator = (const Distance &D ) {
        feet = D.feet;
        inches = D.inches;
    }

    // method to display distance
    void displayDistance() {
        cout << "F: " << feet << " I:" << inches << endl;
    }
}
```



```
    }  
};  
  
int main() {  
    Distance D1(11, 10), D2(5, 11);  
  
    cout << "First Distance : ";  
    D1.displayDistance();  
    cout << "Second Distance :";  
    D2.displayDistance();  
  
    // use assignment operator  
    D1 = D2;  
    cout << "First Distance :";  
    D1.displayDistance();  
  
    return 0;  
}
```

When the above code is compiled and executed, it produces the following result –

First Distance : F: 11 I:10

Second Distance :F: 5 I:11

First Distance :F: 5 I:11



### 5.22.3.2 Arithmetic Assignment (+=) Operator Overloading

### 5.23 Restrictions on Operator Overloading

In C++, following are the general rules for operator overloading.

- 1) Only built-in operators can be overloaded. New operators cannot be created.
- 2) Arity of the operators cannot be changed.
- 3) Precedence and associativity of the operators cannot be changed.
- 4) Overloaded operators cannot have default arguments except the function call operator () which can have default arguments.
- 5) Operators cannot be overloaded for built in types only. At least one operand must be used defined type.
- 6) Assignment (=), subscript ([]), function call ("()"), and member selection (->) operators must be defined as member functions.
- 7) Except the operators specified in point 6, all other operators can be either member functions or a non-member functions.
- 8) Some operators like (assignment)=, (address)& and comma (,) are by default overloaded.

### SUMMERY

- Function Overloading is defined as the process of having two or more function with the same name, but different in parameters is known as function overloading. In function overloading, the function is redefined by using either different types of arguments or a different number of arguments.
- The **advantage** of Function overloading is that it increases the readability of the program

### Multiple Choice Questions

Q 1 - Choose the operator which cannot be overloaded.

- a. /
- b. \





- c. ::
- d. %

Q 2 - Pick up the valid declaration for overloading ++ in postfix form where T is the class name.

- a. T operator++();
- b. T operator++(int);
- c. T& operator++();
- d. T& operator++(int);

Q3. Function overloading is also similar to which of the following?

a) operator overloading

- a. constructor overloading
- b. destructor overloading
- c. function overloading

**Q4. Overloaded functions are \_\_\_\_\_**

- a. Very long functions that can hardly run**
- b. One function containing another one or more functions inside it**
- c. Two or more functions with the same name but different number of parameters or type**
- d. Very long functions**

Q5. In which of the following we cannot overload the function?

return function

- a. caller
- b. called function
- c. main function

Ans: 1-c      2-b      3-b      4-c      5-a

Q1. What do you mean by function overloading? Explain with the help of suitable example.

Q2. How can we overload a binary operator? Write a program to overload + operator.

Q3. How can we overload constructor of a class? Explain in detail.



Q4. Write a program to overload member function of a class.

Q5. Define function overloading Write a program to overload assignment operator.

**References**

The C++ Programming Language' (5<sup>th</sup> Edition) by Bjarne Stroustrup.

C++ Primer' (5<sup>th</sup> Edition) by Stanley B. Lippman, Josée Lajoie, and Barbara E. Moo

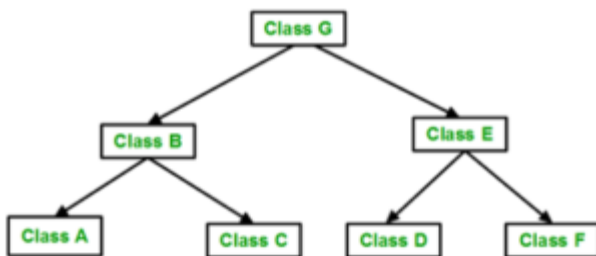


## Chapter 9

### Inheritance

#### 9.1 Introduction to Inheritance

Inheritance is one of the widely used and most powerful programming tools in C++ or any other Object-oriented language. Understanding this concept requires a little knowledge of classes and objects. Using Inheritance, we can create a class that consists of general methods and attributes. This class can be inherited by other classes containing more specific methods.



By doing this we don't have to write the same functionalities again and again which saves time and increases the readability of code.

#### Syntax

1. `Class driven_class: access_type base_class`
2. `{`  
  
`// class body`
3. `}`

#### 9.2 Introduction of Base Classes and Derived Classes

Inheritance allows new classes to be constructed on the basis of existing classes. The new *derived class* “inherits” the data and methods of the so-called *base class*. But you can add more characteristics and functionality to the new class.

#### Defining a derived class



```
class C : public B
{
private:
// Declaration of additional private
// data members and member functions
public:
// Declaration of additional public
// data members and member functions
};
```

### A Person class

Here's a simple class to represent a generic person:

```
#include <stdio.h>
#include <string>
class Person
{
    // In this example, we're making our members public for simplicity
    public:
        string m_name{ };
        int m_age{ };

        Person(const string& name = "", int age = 0)
            : m_name{ name }, m_age{ age }
        {
        }
}
```



```
const std:: string& getName() const { return m_name; }

int getAge() const { return m_age; }

};
```

Because this Person class is designed to represent a generic person, we've only defined members that would be common to any type of person. Every person (regardless of gender, profession, etc...) has a name and age, so those are represented here.

### A BaseballPlayer class

Let's say we wanted to write a program that keeps track of information about some baseball players. Baseball players need to contain information that is specific to baseball players -- for example, we might want to store a player's batting average, and the number of home runs they've hit.

Here's our incomplete Baseball player class:

```
class BaseballPlayer
{
// In this example, we're making our members public for simplicity
public:
    double m_battingAverage{ };
    int m_homeRuns{ };
    BaseballPlayer(double battingAverage = 0.0, int homeRuns = 0)
        : m_battingAverage{ battingAverage }, m_homeRuns{ homeRuns }
    {
    }
};
```

Now, we also want to keep track of a baseball player's name and age, and we already have that information as part of our Person class.



We have three choices for how to add name and age to BaseballPlayer:

- 1) Add name and age to the BaseballPlayer class directly as members. This is probably the worst choice, as we're duplicating code that already exists in our Person class. Any updates to Person will have to be made in BaseballPlayer too.
- 2) Add Person as a member of BaseballPlayer using composition. But we have to ask ourselves, "does a BaseballPlayer have a Person"? No, it doesn't. So this isn't the right paradigm.
- 3) Have BaseballPlayer inherit those attributes from Person. Remember that inheritance represents an is-a relationship. Is a BaseballPlayer a Person? Yes, it is. So inheritance is a good choice here.

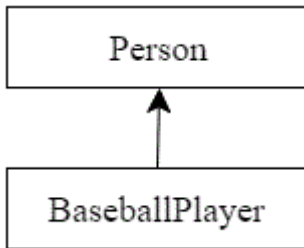
### Making BaseballPlayer a derived class

To have BaseballPlayer inherit from our Person class, the syntax is fairly simple. After the class BaseballPlayer declaration, we use a colon, the word "public", and the name of the class we wish to inherit. This is called *public inheritance*. We'll talk more about what public inheritance means in a future lesson.

```
// BaseballPlayer publicly inheriting Person
class BaseballPlayer : public Person
{
public:
    double m_battingAverage{ };
    int m_homeRuns{ };

    BaseballPlayer(double battingAverage = 0.0, int homeRuns = 0)
        : m_battingAverage{ battingAverage }, m_homeRuns{ homeRuns }
    {
    }
};
```

Using a derivation diagram, our inheritance looks like this:



When `BaseballPlayer` inherits from `Person`, `BaseballPlayer` acquires the member functions and variables from `Person`. Additionally, `BaseballPlayer` defines two members of its own: `m_battingAverage` and `m_homeRuns`. This makes sense, since these properties are specific to a `BaseballPlayer`, not to any `Person`.

Thus, `BaseballPlayer` objects will have 4 member variables: `m_battingAverage` and `m_homeRuns` from `BaseballPlayer`, and `m_name` and `m_age` from `Person`.

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

class Person
{
public:
    string m_name{ };
    int m_age{ };

    Person(const string& name = "", int age = 0)
        : m_name{ name }, m_age{ age }
    {
    }

    const string& getName() const { return m_name; }
```



```
int getAge() const { return m_age; }

};

// BaseballPlayer publicly inheriting Person
class BaseballPlayer : public Person
{
public:
    double m_battingAverage{ };
    int m_homeRuns{ };

    BaseballPlayer(double battingAverage = 0.0, int homeRuns = 0)
        : m_battingAverage{ battingAverage }, m_homeRuns{ homeRuns }
    {
    }
};

int main()
{
    // Create a new BaseballPlayer object
    BaseballPlayer joe{ };

    // Assign it a name (we can do this directly because m_name is public)
    joe.m_name = "Joe";

    // Print out the name
    cout << joe.getName() << '\n'; // use the getName() function we've acquired from the Person
```





```
base class
```

```
    return 0;
```

```
}
```

Which prints the value:

Joe

This compiles and runs because joe is a BaseballPlayer, and all BaseballPlayer objects have a m\_name member variable and a getName() member function inherited from the Person class.

### An Employee derived class

Now let's write another class that also inherits from Person. This time, we'll write an Employee class.

An employee "is a" person, so using inheritance is appropriate:

```
// Employee publicly inherits from Person
class Employee: public Person
{
public:
    double m_hourlySalary{ };
    long m_employeeID{ };

    Employee(double hourlySalary = 0.0, long employeeID = 0)
        : m_hourlySalary{hourlySalary}, m_employeeID{employeeID}
    {
    }

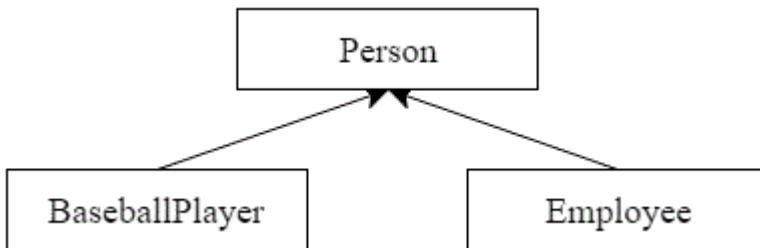
    void printNameAndSalary() const
```



```
{  
    std::cout << m_name << ": " << m_hourlySalary << "\n";  
}  
};
```

Employee inherits `m_name` and `m_age` from `Person` (as well as the two access functions), and adds two more member variables and a member function of its own. Note that `printNameAndSalary()` uses variables both from the class it belongs to (`Employee::m_hourlySalary`) and the parent class (`Person::m_name`).

This gives us a derivation chart that looks like this:



Note that `Employee` and `BaseballPlayer` don't have any direct relationship, even though they both inherit from `Person`.

Here's a full example using `Employee`:

```
#include <iostream>  
  
#include <string>  
  
class Person  
{  
public:  
    std::string m_name{};  
    int m_age{};
```



```
const std::string& getName() const { return m_name; }

int getAge() const { return m_age; }

Person(const std::string& name = "", int age = 0)
    : m_name{name}, m_age{age}
{
}

};

// Employee publicly inherits from Person
class Employee: public Person
{
public:
    double m_hourlySalary{};
    long m_employeeID{};

    Employee(double hourlySalary = 0.0, long employeeID = 0)
        : m_hourlySalary{hourlySalary}, m_employeeID{employeeID}
    {
    }

    void printNameAndSalary() const
    {
        std::cout << m_name << ": " << m_hourlySalary << '\n';
    }
}
```



```
};

int main()
{
    Employee frank{20.25, 12345};

    frank.m_name = "Frank"; // we can do this because m_name is public

    frank.printNameAndSalary();

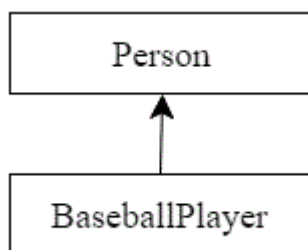
    return 0;
}
```

This prints:

Frank: 20.25

### 9.3 Single Inheritance

Single Inheritance can be considered as the plain vanilla form of inheritance. In single inheritance, a single class inherits from a base class. As shown in previous example BaseballPlayer and class was inherited from class person. Figure shows the pictorial representation of single inheritance.



### 9.4 Type of derivation/ Access Specifies

To this point, you've seen the private and public access specifies, which determine who can access the



members of a class. As a quick refresher, public members can be accessed by anybody. Private members can only be accessed by member functions of the same class or friends. This means derived classes cannot access private members of the base class directly!

```
class Base
{
private:
    int m_private; // can only be accessed by Base members and friends (not derived classes)
public:
    int m_public; // can be accessed by anybody
};
```

#### 9.4.1 Public Derivation

- all the public members in base class are publicly available in the derived class.

Access specifier in base class	Access specifier when inherited privately
Public	Public
Protected	Protected
Private	Inaccessible

#### 5.23.1 Private Derivation

With private inheritance, all members from the base class are inherited as private. This means private members stay private, and protected and public members become private members of derived class.

```
class Base
{
public:
    int m_public;
protected:
```



```
int m_protected;

private:

    int m_private;
};

class Pri: private Base // note: private inheritance
{
    // Private inheritance means:
    // Public inherited members become private (so m_public is treated as private)
    // Protected inherited members become private (so m_protected is treated as private)
    // Private inherited members stay inaccessible (so m_private is inaccessible)

public:
    Pri()
    {
        m_public = 1; // okay: m_public is now private in Pri
        m_protected = 2; // okay: m_protected is now private in Pri
        m_private = 3; // not okay: derived classes can't access private members in the base
    }
};

int main()
{
    // Outside access uses the access specifiers of the class being accessed.
    // In this case, the access specifiers of base.
```



```

Base base;

base.m_public = 1; // okay: m_public is public in Base

base.m_protected = 2; // not okay: m_protected is protected in Base

base.m_private = 3; // not okay: m_private is private in Base


Pri pri;

pri.m_public = 1; // not okay: m_public is now private in Pri

pri.m_protected = 2; // not okay: m_protected is now private in Pri

pri.m_private = 3; // not okay: m_private is inaccessible in Pri


return 0;
}

```

To summarize in table form:

Access specifier in base class	Access specifier when inherited privately
Public	Private
Protected	Private
Private	Inaccessible

Private inheritance can be useful when the derived class has no obvious relationship to the base class, but uses the base class for implementation internally. In such a case, we probably don't want the public interface of the base class to be exposed through objects of the derived class (as it would be if we inherited publicly).

### 5.23.2 Protected Derivation

Protected inheritance is the least common method of inheritance. It is almost never used, except in very particular cases. With protected inheritance, the public and protected members become protected, and



private members stay inaccessible.

```
class Base
{
public:
    int m_public; // can be accessed by anybody
protected:
    int m_protected; // can be accessed by Base members, friends, and derived classes
private:
    int m_private; // can only be accessed by Base members and friends (but not derived
classes)
};

class Derived: public Base
{
public:
    Derived()
    {
        m_public = 1; // allowed: can access public base members from derived class
        m_protected = 2; // allowed: can access protected base members from derived class
        m_private = 3; // not allowed: can not access private base members from derived class
    }
};

int main()
{
```





```

Base base;

base.m_public = 1; // allowed: can access public members from outside class

base.m_protected = 2; // not allowed: can not access protected members from outside class

base.m_private = 3; // not allowed: can not access private members from outside class
}

```

In the above example, you can see that the protected base member `m_protected` is directly accessible by the derived class, but not by the public.

Access specifier in base class	Access specifier when inherited privately
Public	Protected
Protected	Protected
Private	Inaccessible

## 5.24 Types of Base Classes

A base class can be defined classified into two types, direct base and indirect base.

- **Direct base class**

A base class is called a direct base if it is mentioned in the base list. For example:

1. *Class base A*

```

{
    _____
    _____
};

```

Class derived B: public base A

```

{
    _____
    _____
};

```



## 2. Class base B

```
{  
_____  
_____  
};
```

Class base B

```
{  
_____  
_____  
};
```

Class derived C: public base A, public base B

```
{  
_____  
_____  
};
```

Where both classes base A and B are the direct base.

- **Indirect base classes**

A derived class can itself serve as a base of another class. When a derived class is declared as a base of another class, the newly derived class inherits the properties of its base classes as well as its data and function member also.

A class is called indirect base class if it is not a direct base but is a base class of one of the classes mentioned in the above list. For example:

Class base A

```
{  
_____  
_____  
};
```



Class derived B: public base A

```
{  
_____  
_____  
};
```

Class derived: public derived B

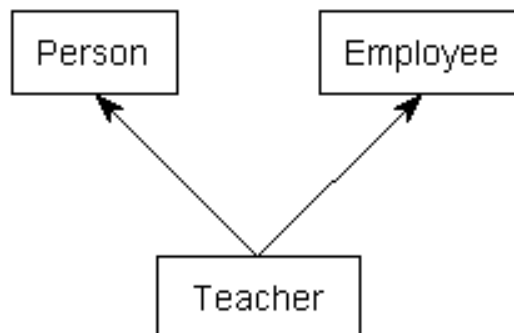
```
{  
_____  
_____  
};
```

Where, base A is an indirect base for derived C.

### 5.25 Multiple Inheritance

So far, all of the examples of inheritance we've presented have been single inheritance -- that is, each inherited class has one and only one parent. However, C++ provides the ability to do multiple inheritance. **Multiple inheritance** enables a derived class to inherit members from more than one parent.

Let's say we wanted to write a program to keep track of a bunch of teachers. A teacher is a person. However, a teacher is also an employee (they are their own employer if working for themselves). Multiple inheritance can be used to create a Teacher class that inherits properties from both Person and Employee. To use multiple inheritance, simply specify each base class (just like in single inheritance), separated by a comma.





```
#include <string>

class Person
{
private:
    std::string m_name;
    int m_age;

public:
    Person(std::string name, int age)
        : m_name(name), m_age(age)
    {
    }

    std::string getName() { return m_name; }
    int getAge() { return m_age; }
};

class Employee
{
private:
    std::string m_employer;
    double m_wage;

public:
```



```
Employee(std::string employer, double wage)
    : m_employer(employer), m_wage(wage)
{
}

std::string getEmployer() { return m_employer; }
double getWage() { return m_wage; }
};

// Teacher publicly inherits Person and Employee
class Teacher : public Person, public Employee
{
private:
    int m_teachesGrade;

public:
    Teacher(std::string name, int age, std::string employer, double wage, int teachesGrade)
        : Person(name, age), Employee(employer, wage), m_teachesGrade(teachesGrade)
    {
    }
};
```

### 5.26 Multilevel Inheritance

It's possible to inherit from a class that is itself derived from another class. There is nothing noteworthy or special when doing so -- everything proceeds as in the examples above.

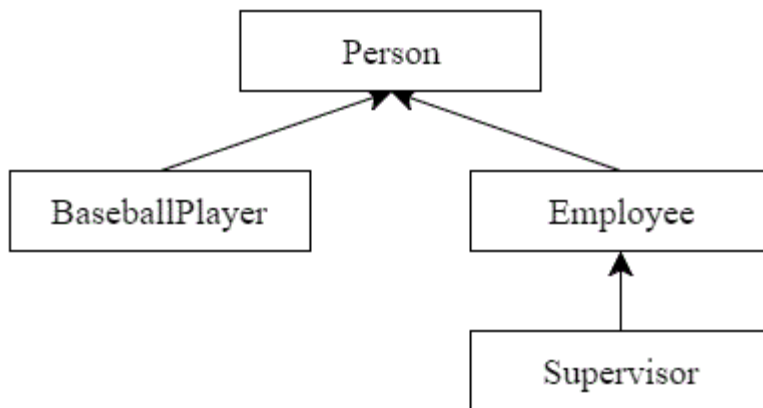
For example, let's write a Supervisor class. A Supervisor is an Employee, which is a Person. We've



already written an Employee class, so let's use that as the base class from which to derive Supervisor:

```
class Supervisor: public Employee
{
public:
    // This Supervisor can oversee a max of 5 employees
    long m_overseesIDs[5]{};
};
```

Now our derivation chart looks like this:



All Supervisor objects inherit the functions and variables from both Employee and Person, and add their own m\_overseesIDs member variable.

By constructing such inheritance chains, we can create a set of reusable classes that are very general (at the top) and become progressively more specific at each level of inheritance.

### 5.27 Casting Base-Class Pointers to Derived-Class Pointer,

### 5.28 Overriding Base-Class Members in a Derived Class

Suppose, the same function is defined in both the derived class and the based class. Now if we call this function using the object of the derived class, the function of the derived class is executed. This is known as function overriding. The function in derived class overrides the function in base class.

When a member function is called with a derived class object, the compiler first looks to see if that



member exists in the derived class. If not, it begins walking up the inheritance chain and checking whether the member has been defined in any of the parent classes. It uses the first one it finds.

Consequently, take a look at the following example:

```
class Base
{
protected:
    int m_value;

public:
    Base(int value)
        : m_value(value)
    {
    }

    void identify() { std::cout << "I am a Base\n"; }
};

class Derived: public Base
{
public:
    Derived(int value)
        : Base(value)
    {
    }
};
```



```
int main()
{
    Base base(5);
    base.identify();

    Derived derived(7);
    derived.identify();

    return 0;
}
```

This prints

I am a Base

I am a Base

When `derived.identify()` is called, the compiler looks to see if function `identify()` has been defined in the Derived class. It hasn't. Then it starts looking in the inherited classes (which in this case is Base). Base has defined an `identify()` function, so it uses that one. In other words, `Base::identify()` was used because `Derived::identify()` doesn't exist.

This means that if the behavior provided by a base class is sufficient, we can simply use the base class behavior.

### Redefining behaviors

However, if we had defined `Derived::identify()` in the Derived class, it would have been used instead.

This means that we can make functions work differently with our derived classes by redefining them in the derived class this is called overriding.

In our above example, it would be more accurate if `derived.identify()` printed "I am a Derived". Let's modify function `identify()` in the Derived class so it returns the correct response when we call function `identify()` with a Derived object.





To modify the way a function defined in a base class works in the derived class, simply redefine the function in the derived class.

```
class Derived: public Base
{
public:
    Derived(int value)
        : Base(value)
    {
    }

    int getValue() { return m_value; }

    // Here's our modified function
    void identify() { std::cout << "I am a Derived\n"; }
};
```

Here's the same example as above, using the new `Derived::Identify()` function:

```
int main()
{
    Base base(5);
    base.identify();

    Derived derived(7);
    derived.identify();

    return 0;
}
```



I am a Base

I am a Derived

Note that when you redefine a function in the derived class, the derived function does not inherit the access specifier of the function with the same name in the base class. It uses whatever access specifier it is defined under in the derived class. Therefore, a function that is defined as `private` in the base class can be redefined as `public` in the derived class, or vice-versa!

```
class Base
{
private:
    void print()
    {
        std::cout << "Base";
    }
};

class Derived : public Base
{
public:
    void print()
    {
        std::cout << "Derived ";
    }
};
```



```
int main()
{
    Derived derived;
    derived.print(); // calls derived::print(), which is public
    return 0;
}
```

### 5.29 Using Constructors and Destructors in Derived Classes,

In section, we'll take a closer look at the role of constructors in the initialization of derived classes. To do so, we will continue to use the simple Base and Derived classes we developed in the previously:

```
class Base
{
public:
    int m_id;

    Base(int id=0)
        : m_id{ id }
    {
    }

    int getId() const { return m_id; }
};

class Derived: public Base
```



```
{  
public:  
    double m_cost;  
  
    Derived(double cost=0.0)  
        : m_cost{ cost }  
    {  
    }  
  
    double getCost() const { return m_cost; }  
};
```

With non-derived classes, constructors only have to worry about their own members. For example, consider Base. We can create a Base object like this:

```
int main()  
{  
    Base base{ 5 }; // use Base(int) constructor  
  
    return 0;  
}
```

Here's what actually happens when base is instantiated:

1. Memory for base is set aside
2. The appropriate Base constructor is called
3. The initialization list initializes variables
4. The body of the constructor executes
5. Control is returned to the caller



This is pretty straightforward. With derived classes, things are slightly more complex:

```
int main()
{
    Derived derived{ 1.3 }; // use Derived(double) constructor

    return 0;
}
```

Here's what actually happens when derived is instantiated:

1. Memory for derived is set aside (enough for both the Base and Derived portions)
2. The appropriate Derived constructor is called
3. **The Base object is constructed first using the appropriate Base constructor.** If no base constructor is specified, the default constructor will be used.
4. The initialization list initializes variables
5. The body of the constructor executes
6. Control is returned to the caller

The only real difference between this case and the non-inherited case is that before the Derived constructor can do anything substantial, the Base constructor is called first. The Base constructor sets up the Base portion of the object, control is returned to the Derived constructor, and the Derived constructor is allowed to finish up its job.

### Initializing base class members

One of the current shortcomings of our Derived class as written is that there is no way to initialize `m_id` when we create a Derived object. What if we want to set both `m_cost` (from the Derived portion of the object) and `m_id` (from the Base portion of the object) when we create a Derived object?

```
class Derived: public Base
{
```



```
public:
    double m_cost;

    Derived(double cost=0.0, int id=0)
        // does not work
        : m_cost{ cost }, m_id{ id }
    {
    }

    double getCost() const { return m_cost; }
};
```

This is a good attempt, and is almost the right idea. We definitely need to add another parameter to our constructor, otherwise C++ will have no way of knowing what value we want to initialize `m_id` to. However, C++ prevents classes from initializing inherited member variables in the initialization list of a constructor. In other words, the value of a member variable can only be set in an initialization list of a constructor belonging to the same class as the variable.

Why does C++ do this? The answer has to do with `const` and reference variables. Consider what would happen if `m_id` were `const`. Because `const` variables must be initialized with a value at the time of creation, the base class constructor must set its value when the variable is created. However, when the base class constructor finishes, the derived class constructors initialization lists are then executed. Each derived class would then have the opportunity to initialize that variable, potentially changing its value! By restricting the initialization of variables to the constructor of the class those variables belong to, C++ ensures that all variables are initialized only once.

The end result is that the above example does not work because `m_id` was inherited from `Base`, and only non-inherited variables can be initialized in the initialization list.

However, inherited variables can still have their values changed in the body of the constructor using an assignment. Consequently, new programmers often also try this:



```
class Derived: public Base
{
public:
    double m_cost;

    Derived(double cost=0.0, int id=0)
        : m_cost{ cost }
    {
        m_id = id;
    }

    double getCost() const { return m_cost; }
};
```

While this actually works in this case, it wouldn't work if `m_id` were a `const` or a reference (because `const` values and references have to be initialized in the initialization list of the constructor). It's also inefficient because `m_id` gets assigned a value twice: once in the initialization list of the Base class constructor, and then again in the body of the Derived class constructor. And finally, what if the Base class needed access to this value during construction? It has no way to access it, since it's not set until the Derived constructor is executed (which pretty much happens last).

So how do we properly initialize `m_id` when creating a Derived class object?

In all of the examples so far, when we instantiate a Derived class object, the Base class portion has been created using the default Base constructor. Why does it always use the default Base constructor? Because we never told it to do otherwise!

Fortunately, C++ gives us the ability to explicitly choose which Base class constructor will be called! To do this, simply add a call to the base class Constructor in the initialization list of the derived class:

```
class Derived: public Base
```



```
{
public:
    double m_cost;

    Derived(double cost=0.0, int id=0)
        : Base{ id }, // Call Base(int) constructor with value id!
        m_cost{ cost }
    {
    }

    double getCost() const { return m_cost; }
};
```

Now, when we execute this code:

```
int main()
{
    Derived derived{ 1.3, 5 }; // use Derived(double, int) constructor
    std::cout << "Id: " << derived.getId() << '\n';
    std::cout << "Cost: " << derived.getCost() << '\n';

    return 0;
}
```

The base class constructor Base(int) will be used to initialize m\_id to 5, and the derived class constructor will be used to initialize m\_cost to 1.3!

Thus, the program will print:

Id: 5





Cost: 1.3

In more detail, here's what happens:

1. Memory for derived is allocated.
2. The Derived(double, int) constructor is called, where cost = 1.3, and id = 5
3. The compiler looks to see if we've asked for a particular Base class constructor. We have! So it calls Base(int) with id = 5.
4. The base class constructor initialization list sets m\_id to 5
5. The base class constructor body executes, which does nothing
6. The base class constructor returns
7. The derived class constructor initialization list sets m\_cost to 1.3
8. The derived class constructor body executes, which does nothing
9. The derived class constructor returns

This may seem somewhat complex, but it's actually very simple. All that's happening is that the Derived constructor is calling a specific Base constructor to initialize the Base portion of the object. Because m\_id lives in the Base portion of the object, the Base constructor is the only constructor that can initialize that value.

Note that it doesn't matter where in the Derived constructor initialization list the Base constructor is called -- it will always execute first.

### **Now we can make our members private**

Now that you know how to initialize base class members, there's no need to keep our member variables public. We make our member variables private again, as they should be.

As a quick refresher, public members can be accessed by anybody. Private members can only be accessed by member functions of the same class. Note that this means derived classes can not access private members of the base class directly! Derived classes will need to use access functions to access private members of the base class.

Consider:



```
#include <iostream>

class Base
{
private: // our member is now private
    int m_id;

public:
    Base(int id=0)
        : m_id{ id }
    {
    }

    int getId() const { return m_id; }
};

class Derived: public Base
{
private: // our member is now private
    double m_cost;

public:
    Derived(double cost=0.0, int id=0)
        : Base{ id }, // Call Base(int) constructor with value id!
```



```
        m_cost{ cost }

    {
    }

    double getCost() const { return m_cost; }
};

int main()
{
    Derived derived{ 1.3, 5 }; // use Derived(double, int) constructor
    std::cout << "Id: " << derived.getId() << '\n';
    std::cout << "Cost: " << derived.getCost() << '\n';

    return 0;
}
```

In the above code, we've made `m_id` and `m_cost` private. This is fine, since we use the relevant constructors to initialize them, and use a public accessor to get the values.

This prints, as expected:

Id: 5

Cost: 1.3

We'll talk more about access specifiers in the next lesson.

### Another example

Let's take a look at another pair of classes we've previously worked with:

```
#include <string>
```



```
class Person
{
public:
    std::string m_name;
    int m_age;

    Person(const std::string& name = "", int age = 0)
        : m_name{ name }, m_age{ age }
    {
    }

    const std::string& getName() const { return m_name; }
    int getAge() const { return m_age; }
};

// BaseballPlayer publicly inheriting Person
class BaseballPlayer : public Person
{
public:
    double m_battingAverage;
    int m_homeRuns;

    BaseballPlayer(double battingAverage = 0.0, int homeRuns = 0)
        : m_battingAverage{ battingAverage },
```



```
m_homeRuns{ homeRuns }  
  
{  
  
}  
  
};
```

As we'd previously written it, BaseballPlayer only initializes its own members and does not specify a Person constructor to use. This means every BaseballPlayer we create is going to use the default Person constructor, which will initialize the name to blank and age to 0. Because it makes sense to give our BaseballPlayer a name and age when we create them, we should modify this constructor to add those parameters.

Here's our updated classes that use private members, with the BaseballPlayer class calling the appropriate Person constructor to initialize the inherited Person member variables:

```
#include <iostream>  
  
#include <string>  
  
class Person  
{  
private:  
    std::string m_name;  
    int m_age;  
  
public:  
    Person(const std::string& name = "", int age = 0)  
        : m_name{ name }, m_age{ age }  
    {  
  
    }
```



```
const std::string& getName() const { return m_name; }

int getAge() const { return m_age; }

};

// BaseballPlayer publicly inheriting Person
class BaseballPlayer : public Person
{
private:
    double m_battingAverage;
    int m_homeRuns;

public:
    BaseballPlayer(const std::string& name = "", int age = 0,
        double battingAverage = 0.0, int homeRuns = 0)
        : Person{ name, age }, // call Person(const std::string&, int) to initialize these fields
        m_battingAverage{ battingAverage }, m_homeRuns{ homeRuns }
    {
    }

    double getBattingAverage() const { return m_battingAverage; }
    int getHomeRuns() const { return m_homeRuns; }
};
```

Now we can create baseball players like this:



```
int main()
{
    BaseballPlayer pedro{ "Pedro Cerrano", 32, 0.342, 42 };

    std::cout << pedro.getName() << '\n';
    std::cout << pedro.getAge() << '\n';
    std::cout << pedro.getHomeRuns() << '\n';

    return 0;
}
```

This outputs:

Pedro Cerrano

32

42

As you can see, the name and age from the base class were properly initialized, as was the number of home runs and batting average from the derived class.

### Inheritance chains

Classes in an inheritance chain work in exactly the same way.

```
#include <iostream>

class A
{
public:
    A(int a)
```



```
{  
    std::cout << "A: " << a << '\n';  
}  
};  
  
class B: public A  
{  
public:  
    B(int a, double b)  
    : A{ a }  
    {  
        std::cout << "B: " << b << '\n';  
    }  
};  
  
class C: public B  
{  
public:  
    C(int a , double b , char c)  
    : B{ a, b }  
    {  
        std::cout << "C: " << c << '\n';  
    }  
};
```





```
int main()
{
    C c{ 5, 4.3, 'R' };

    return 0;
}
```

In this example, class C is derived from class B, which is derived from class A. So what happens when we instantiate an object of class C?

First, main() calls C(int, double, char). The C constructor calls B(int, double). The B constructor calls A(int). Because A does not inherit from anybody, this is the first class we'll construct. A is constructed, prints the value 5, and returns control to B. B is constructed, prints the value 4.3, and returns control to C. C is constructed, prints the value 'R', and returns control to main(). And we're done!

Thus, this program prints:

A: 5

B: 4.3

C: R

It is worth mentioning that constructors can only call constructors from their immediate parent/base class. Consequently, the C constructor could not call or pass parameters to the A constructor directly. The C constructor can only call the B constructor (which has the responsibility of calling the A constructor).

## Destructors

When a derived class is destroyed, each destructor is called in the *reverse* order of construction. In the above example, when c is destroyed, the C destructor is called first, then the B destructor, then the A destructor.

### 5.30 Implicit Derived-Class Object to Base-Class Object Conversion.

and the order that derived classes are initialized. In this lesson, we'll take a closer look at the role of



constructors in the initialization of derived classes. To do so, we will continue to use the simple Base and Derived classes we developed in the previously:

```
class Base
{
public:
    int m_id;

    Base(int id=0)
        : m_id{ id }
    {
    }

    int getId() const { return m_id; }
};

class Derived: public Base
{
public:
    double m_cost;

    Derived(double cost=0.0)
        : m_cost{ cost }
    {
    }

    double getCost() const { return m_cost; }
};
```



With non-derived classes, constructors only have to worry about their own members. For example, consider Base. We can create a Base object like this:

```
int main()
{
    Base base{ 5 }; // use Base(int) constructor

    return 0;
}
```

### SUMMARY

- Inheritance allows new classes to be constructed on the basis of existing classes
- All the public members in base class are publicly available in the derived class
- With private inheritance, all members from the base class are inherited as private. This means private members stay private, and protected and public members become private members of derived class
- Protected inheritance is the least common method of inheritance. It is almost never used, except in very particular cases. With protected inheritance, the public and protected members become protected, and private members stay inaccessible

### Multiple Choice Questions

Q 1 - C++ does not support the following

- Multilevel inheritance
- Hierarchical inheritance
- Hybrid inheritance
- None of the above.

Q2. When the inheritance is private, the private methods in base class are \_\_\_\_\_ in the derived class (in C++).

- Inaccessible



- b. Accessible
- c. Protected
- d. Public

**Q3. 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 derived class from more than one base class
- d. None of the mentioned

**Q4. . Inheritance allow in C++ Program?**

- a. Class Re-usability
- b. Creating a hierarchy of classes
- c. Extendibility
- d. All of the above

**Q5. What are the things are inherited from the base class?**

- a. Constructor and its destructor
- b. Operator=() members
- c. Friends
- d. All of the above

Ans: 1-c      2-a      3-c      4-d      5-d

### **Long Answer Type Questions**

**Q1. Explain Types of Inheritance with example.**

**Q2. Explain protected member access specifies for class member.**

**Q3. what is overriding in c++? Explain with example.**

**Q4. What is multilevel inheritance? Explain with example.**

**Q5. Can we use constructor of base class using object of derived class? Explain.**



Subject Name: <b>Object Oriented Programming Using C++</b>	
Course Code: <b>DCA-15-T</b>	Author:
Lesson No. 10	Vetter:
<b>VIRTUAL FUNCTION</b>	

## 10.1 INTRODUCTION TO VIRTUAL FUNCTION

A virtual function is a member function in the base class that we expect to redefine in derived classes. Basically, a virtual function is used in the base class in order to ensure that the function is **overridden**. This especially applies to cases where a pointer of base class points to an object of a derived class. For example, consider the code below:

```
class Base {  
    public:  
    void print() {  
        // code  
    }  
};  
  
class Derived : public Base {  
    public:  
    void print() {  
        // code  
    }  
};
```

Later, if we create a pointer of `Base` type to point to an object of `Derived` class and call



the `print()` function, it calls the `print()` function of the `Base` class.

In other words, the member function of `Base` is not overridden.

**5.30.1** `intmain() {`

**5.30.2** `Derived derived1;`

**5.30.3** `Base* base1 = &derived1;`

**5.30.4**

**5.30.5** `// calls function of Base class`

**5.30.6** `base1->print();`

**5.30.7**

**5.30.8** `return0;`

**5.30.9** `}`

In order to avoid this, we declare the `print()` function of the `Base` class as virtual by using the `virtual` keyword.

```
class Base {  
    public:  
        virtual void print() {  
            // code  
        }  
};
```

Virtual functions are an integral part of polymorphism in.

### Example virtual Function

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



```
class Base {  
    public:  
        virtual void print() {  
            cout << "Base Function" << endl;  
        }  
};  
  
class Derived : public Base {  
    public:  
        void print() {  
            cout << "Derived Function" << endl;  
        }  
};  
  
int main() {  
    Derived derived1;  
  
    // pointer of Base type that points to derived1  
    Base* base1 = &derived1;  
  
    // calls member function of Derived class  
    base1->print();  
  
    return 0;  
}
```



## Output

### Derived Function

Here, we have declared the `print()` function of `Base` as `virtual`.

So, this function is overridden even when we use a pointer of `Base` type that points to the `Derived` object `derived1`.

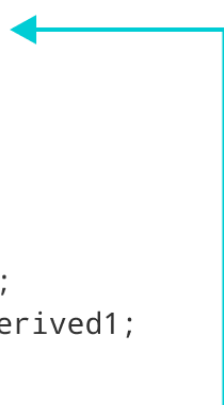
```
class Base {
public:
    virtual void print() {
        // code
    }
};

class Derived : public Base {
public:
    void print() {
        // code
    }
};

int main() {
    Derived derived1;
    Base* base1 = &derived1;

    base1->print();

    return 0;
}
```



`print()` of `Derived` class is called because `print()` of `Base` class is `virtual`

### Use of Virtual Functions

Suppose we have a base class `Animal` and derived classes `Dog` and `Cat`.





Suppose each class has a data member named `type`. Suppose these variables are initialized through their respective constructors.

```
classAnimal {  
private:  
string type;  
... ..  
public:  
    Animal(): type("Animal") {}  
... ..  
};
```

```
classDog :public Animal {  
private:  
string type;  
... ..  
public:  
    Animal(): type("Dog") {}  
... ..  
};
```

```
classCat :public Animal {  
private:  
string type;  
... ..  
public:
```



```
Animal(): type("Cat") {}
```

```
... ..
```

```
};
```

Now, let us suppose that our program requires us to create two `public` functions for each class:

1. `getType()` to return the value of `type`
2. `print()` to print the value of `type`

We could create both these functions in each class separately and override them, which will be long and tedious.

Or we could make `getType()` **virtual** in the `Animal` class, then create a single, separate `print()` function that accepts a pointer of `Animal` type as its argument. We can then use this single function to override the virtual function.

```
classAnimal {
```

```
... ..
```

```
public:
```

```
... ..
```

```
virtualstring getType {...}
```

```
};
```

```
... ..
```

```
... ..
```

```
voidprint(Animal* ani){
```

```
cout<<"Animal: "<< ani->getType() <<endl;
```

```
}
```

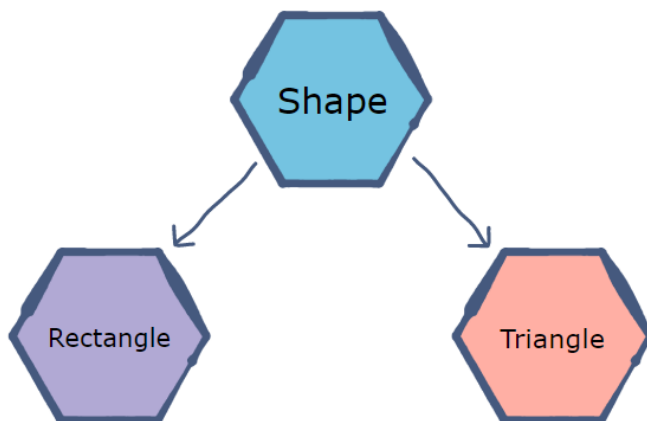
This will make the code **shorter**, **cleaner**, and **less repetitive**.



## 10.2 ABSTRACT BASE CLASSES AND CONCRETE CLASS

By definition, an **abstract class in C++** is a class that has at least *one* pure virtual function (i.e., a function that has no definition). The classes inheriting the abstract class *must* provide a definition for the pure virtual function; otherwise, the subclass would become an abstract class itself. Abstract classes are essential to providing an abstraction to the code to make it reusable and extendable. For example, a *Vehicle* parent class with *Truck* and *Motorbike* inheriting from it is an abstraction that easily allows more vehicles to be added. However, even though all vehicles have wheels, not all vehicles have the same number of wheels – this is where a pure virtual function is needed.

Consider an example of a *shape* base class with sub-classes (*Triangle* and *Rectangle*) that inherit the *Shape* class.



Now, suppose we need a function to return the area of a shape. The function will be declared in the *Shape* class; however, it cannot be defined there as the formula for the area is different for each shape. A non-specific shape does not have an area, but rectangles and triangles do. Therefore, the pure virtual function for calculating area will be implemented differently by each sub-class. The following code snippet implements the abstract *Shape* class along with its sub-classes:

```
1
```

```
#include <iostream>
```



```
using namespace std;

class Shape {
public:
    virtual int Area() = 0; // Pure virtual function is declared as follows.
    // Function to set width.
    void setWidth(int w) {
        width = w;
    }
    // Function to set height.
    void setHeight(int h) {
        height = h;
    }
protected:
    int width;
    int height;
};

// A rectangle is a shape; it inherits shape.
class Rectangle: public Shape {
public:
    // The implementation for Area is specific to a rectangle.
    int Area() {
        return (width * height);
    }
};

// A triangle is a shape too; it inherits shape.
```



```
class Triangle: public Shape {
public:
    // Triangle uses the same Area function but implements it to
    // return the area of a triangle.
    int Area() {
        return (width * height)/2;
    }
};
```

```
int main() {
    Rectangle R;
    Triangle T;

    R.setWidth(5);
    R.setHeight(10);

    T.setWidth(20);
    T.setHeight(8);

    cout << "The area of the rectangle is: " << R.Area() << endl;
    cout << "The area of the triangle is: " << T.Area() << endl;
}
```

#### Output

The area of the rectangle is: 50

The area of the triangle is: 80



## Concrete Class

A concrete class is an ordinary class which has no purely virtual functions and hence can be instantiated. Here is the source code of the C++ program which differentiates between the concrete and abstract class. The C++ program is successfully compiled and run on a Linux system. The program output is also shown below.

```
*  
  
* C++ Program to differentiate between concrete class and abstract class  
  
*/  
  
#include <iostream>  
  
#include <string>  
  
using namespace std;  
  
class Abstract {  
private:  
    string info;  
public:  
    virtual void printContent()=0;  
};  
  
class Concrete {  
private:  
    string info;  
public:  
    Concrete(string s): info(s){ }  
    void printContent(){
```



```
cout<<"Concrete Object Information\n"<< info << endl;
}
};

int main()
{
/*
 * Abstract a;
 * Error : Abstract Instance Creation Failed
 */
string s;

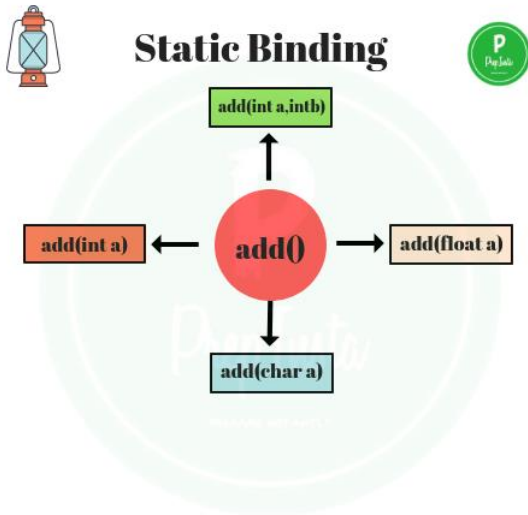
s ="Object Creation Date : 23:26 PM 15 Dec 2013";
Concrete c(s);
c. printContent();
}
```

### 10.3 STATIC BINDING

Static Binding or Early Binding is nothing but most popular compile time polymorphic technique method overloading

**Definition of Static Binding:** Defining multiple methods with the same name but difference in the number of arguments or the data type of arguments or ordering of arguments and resolving this method calls among multiple methods at **compilation** itself is called Static Binding or Early binding or Method overloading

It is called as Static Binding or early binding because the compiler gets clarity among multiple methods, which is the exact method to be executed at *compilation itself*



Example of static binding

```
#include <iostream>
```

```
using namespace std;
```

```
class ComputeSum
```

```
{
```

```
public:
```

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

```
{
```

```
    return x + y;
```

```
}
```

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





```
{  
    return x + y + z;  
}  
};
```

```
int main()  
{  
    ComputeSum obj;  
    cout << "Sum is " << obj.sum(10, 20) << "\n";  
    cout << "Sum is " << obj.sum(10, 20, 30) << "\n";  
  
    return 0;  
}
```

## 10.4 DYNAMIC BINDING

Dynamic binding refers to linking a procedure call to code that will execute only once. The code associated with the procedure is not known until the program is executed, which is also known as late binding.

```
//Dynamic Binding program  
#include <iostream >  
  
using namespace std;  
  
int Square(int x) //Square is =x*x;  
{  
    return x * x;  
}
```



```
}  
  
int Cube(int x) // Cube is =x*x*x;  
{  
    return x * x * x;  
}  
  
main() {  
    int x = 100;  
    int choice;  
    do {  
        cout << "Enter 0 for Square value, 1 for Cube value :\n";  
        cin >> choice;  
    }  
    while (choice < 0 || choice > 1);  
    int( * ptr)(int);  
    switch (choice) {  
        case 0:  
            ptr = Square;  
            break;  
        case 1:  
            ptr = Cube;  
            break;  
    }  
    cout << "The result is :" << ptr(x);  
    return 0;  
}
```



```
C:\Users\KRISHNA RASPUTI\Documents\Dev C Program\Dynamic binding.cpp.exe
Enter 0 for Square value, 1 for Cube value : 0
The result is :10000
Process exited after 22.21 seconds with return value 0
Press any key to continue . . .
```

## 10.5 POLYMORPHISM

The word polymorphism means having many forms. Typically, polymorphism occurs when there is a hierarchy of classes and they are related by inheritance. C++ polymorphism means that a call to a member function will cause a different function to be executed depending on the type of object that invokes the function. Consider the following example where a base class has been derived by other two classes:

```
#include <iostream>

using namespace std;

class Shape {
protected:
int width, height;
public:
Shape( int a=0, int b=0)
{
width = a;
height = b;
}
int area()
{
```



```
cout << "Parent class area :" <<endl;

return 0;

}

};

class Rectangle: public Shape{

public:

Rectangle( int a=0, int b=0):Shape(a, b) { }

int area ()

{

Cout<<"Rectangle class area :" <<endl;

Return(width * height);

}

};

Class Triangle: Public Shape{

Public:

Triangle ( int a=0, int b=0):" <<endl;

Cout<< "Triangle class area :" <<endl;

Return (width * height /2);

}

};

// Main function for the program

int main( )

{

Shape *shape;

Rectangle rec(10,7);
```



```
Triangle tri(10,5);  
// store the address of Rectangle  
shape = &rec;  
// call rectangle area.  
shape->area();  
// store the address of Triangle  
shape = &tri;  
// call triangle area.  
shape->area();  
return 0;  
}
```

When the above code is compiled and executed, it produces the following result:

Parent class area

Parent class area

The reason for the incorrect output is that the call of the function area is being set once by the compiler as the version defined in the base class. This is called static resolution of the function call, or static linkage - the function call is fixed before the program is executed. This is also sometimes called early binding because the area function is set during the compilation of the program. But now, let's make a slight modification in our program and precede the declaration of area in the Shape class with the keyword virtual so that it looks like this:

```
class Shape {  
protected:  
int width, height;  
public:  
Shape( int a=0, int b=0)  
{
```



```
width = a; height = b;
}
virtual int area()
{
cout << "Parent class area :" <<endl;
return 0;
}
};
```

After this slight modification, when the previous example code is compiled and executed, it produces the following result:

Rectangle class area

Triangle class area

This time, the compiler looks at the contents of the pointer instead of its type. Hence, since addresses of objects of tri and rec classes are stored in \*shape the respective area function is called. As you can see, each of the child classes has a separate implementation for the function area. This is how polymorphism is generally used. You have different classes with a function of the same name, and even the same parameters, but with different implementations

## 10.6 PURE VIRTUAL FUNCTION

When a virtual function is not redefined by a derived class, the version defined in the base class will be used. However, in many situations there can be no meaningful definition of a virtual function within a base class. For example, a base class may not be able to define an object sufficiently to allow a base-class virtual function to be created. Further, in some situations you will want to ensure that all derived classes override a virtual function. To handle these two cases, C++ supports the pure virtual function. A pure virtual function is a virtual function that has no definition within the base class. To declare a pure



virtual function, use this general form: virtual type func-name(parameter-list) = 0; When a virtual function is made pure, any derived class must provide its own definition. If the derived class fails to override the pure virtual function, a compile-time error will result. The following program contains a simple example of a pure virtual function. The base class, number, contains an integer called val, the function setval( ), and the pure virtual function show( ). The derived classes hextype, dectype, and octtype inherit number and redefine show( ) so that it outputs the value of val in each respective number base (that is, hexadecimal, decimal, or octal).

```
#include <iostream>

using namespace std;

class number {
protected:
int val;
public:
void setval(int i) { val = i; }
// show() is a pure virtual function
virtual void show() = 0;
};

class hextype : public number {
public:
void show() {
cout << hex << val << "\n"; } };

class dectype : public number {
public:
void show() {
cout << val << "\n";
}
```



```
};  
class octtype : public number {  
public:  
    void show() {  
        cout << oct << val << "\n";  
    }  
};  
int main()  
{  
    dectype d;  
    hextype h;  
    octtype o;  
    d.setval(20);  
    d.show(); // displays 20 - decimal  
    h.setval(20); h.show(); // displays 14 – hexadecimal  
    o.setval(20); o.show(); // displays 24 - octal  
    return 0;  
}
```

Although this example is quite simple, it illustrates how a base class may not be able to meaningfully define a virtual function. In this case, number simply provides the common interface for the derived types to use. There is no reason to define show( ) inside number since the base of the number is undefined. Of course, you can always create a placeholder definition of a virtual function. However, making show( ) pure also ensures that all derived classes will indeed redefine it to meet their own needs. Keep in mind that when a virtual function is declared as pure, all derived classes must override it. If a derived class fails to do this, a compile-time error will result





## 10.7 VIRTUAL DESTRUCTOR

Deleting a derived class object using a pointer of base class type that has a non-virtual destructor results in undefined behavior. To correct this situation, the base class should be defined with a virtual destructor. For example, following program results in undefined behavior.

```
// CPP program without virtual destructor
// causing undefined behavior
#include<iostream>
using namespace std;
class base {
public:
    base()
    { cout<<"Constructing base \n"; }
    ~base()
    { cout<<"Destructing base \n"; }
};
class derived: public base {
public:
    derived()
    { cout<<"Constructing derived \n"; }
    ~derived()
    { cout<<"Destructing derived \n"; }
};
```



```
int main(void)
{
    derived *d = new derived();
    base *b = d;
    delete b;
    getchar();
    return 0;
}
```

Output:-

Constructing base

Constructing derived

Destructing base

// A program with virtual destructor

```
#include<iostream>

using namespace std;

class base {
public:
    base()
    { cout<<"Constructing base \n"; }
    virtual ~base()
    { cout<<"Destructing base \n"; }
};

class derived: public base {
```



```
public:
    derived()
    { cout<<"Constructing derived \n"; }
    ~derived()
    { cout<<"Destructing derived \n"; }
};

int main(void)
{
    derived *d = new derived();
    base *b = d;
    delete b;
    getchar();
    return 0;
}
```

Output:

Constructing base  
Constructing derived  
Destructing derived  
Destructing base

## 10.8 CHECK YOUR PROGRESS

Ques:1. .... binding means that an object is bound to its function call at compile time.

- A) late
- B) static



C) dynamic

D) fixed

Ques:2 C++ supports run time polymorphism with the help of virtual functions, which is called ..... binding.

A) dynamic

B) run time

C) early binding

D) static

Ques:3 In compile-time polymorphism, a compiler is able to select the appropriate function for a particular call at the compile time itself, which is known as .....

A) early binding

B) static binding

C) static linking

D) All of the above

Ques:4. .... are also known as generic pointers, which refer to variables of any type.

A) void pointers

B) null pointers

C) this pointer

D) base pointer

Ques:5 The pointers which are not initialized in a program are called .....

A) void pointers

B) null pointers

C) this pointer

D) base pointer

Ques:6. .... is useful in creating objects at run time.

A) void pointer

B) null pointer

C) this pointer

D) object pointer



Ques:7 A ..... refers to an object that that currently invokes a member function.

- A) void pointers
- B) null pointers
- C) this pointer
- D) base pointer

Ques:8 The ..... cannot be directly used to access all the members of the derived class.

- A) void pointers
- B) null pointers
- C) this pointer
- D) base pointer

Ques:9 Run time polymorphism is achieved only when a ..... is accessed through a pointer to the base class.

- A) member function
- B) virtual function
- C) static function
- D) real function

## **10.9SUMMARY**

Virtual functions, one of advanced features of OOP is one that does not really exist but it« appears real in some parts of a program. This part deals with the polymorphic features which are incorporated using the virtual functions. To make a member function virtual, the keyword virtual is used in the methods while it is declared in the class definition but not in the member function definition. The keyword virtual precedes the return type of the function name. The compiler gets information from the keyword virtual that it is a virtual function and not a conventional function declaration. A virtual function cannot be a static member since a virtual member is always a member of a particular object in a class rather than a member of the class as a whole. When an object is created from its class, the member variables and member functions are allocated memory spaces. The memory spaces have unique addresses. Pointer is a mechanism to access these memory locations using their address rather than the name assigned to



them.

## 10.10 SELF ASSESSMENT TEST

Ques:1 Explain virtual Destructor in detail.

Ques:2 What is the use of Virtual Function?

Ques:3 Define Pure Virtual Function.

Ques:4 Explain Static binding and Dynamic binding.

## 10.11 ANSWER TO CHECK YOUR PROGRESS

Answer:1 B) static

Answer:2 A) dynamic

Answer:3 D) All of the above

Answer:4 A) void pointers

Answer:5 B) null pointers

Answer:6 D) object pointer

Answer:7 C) this pointer

Answer:8 D) base pointer

Answer:9 B) virtual function

## 10.12 REFERENCES/SUGGESTED READING



## NOTES



## NOTES