

# **Object Oriented Programming with C++**

# Introduction to Object Oriented Programming

# INTRODUCTION

- A programming language is a language specifically designed to express computations that can be performed by the computer.
- Programming languages are used to create programs that control the behaviour of a system, to express algorithms, or can be used as a mode of human communication.
- The term ‘programming language’ usually refers to high-level languages such as **BASIC, C, C++, COBOL, FORTRAN, ADA, and PASCAL**, to name a few.
- While high-level programming languages are easy for us to read and understand, the computer understands the machine language that consists only of numbers.

# INTRODUCTION

- The determination of the language is dependent on the following factors:
- The type of computer (microcontroller, microprocessor, etc.) on which the program has to be executed.
- The type of program (system program, application program, etc.).
- The expertise of the programmer. That is, the proficiency level of a programmer in a particular language.

# INTRODUCTION

- For example, **FORTRAN** is particularly good for processing numerical data but it does not lend itself very well to organizing large programs. Pascal can be used for writing well-structured and readable programs but it is not as flexible as C language. **C++** goes one step ahead of C by incorporating powerful object oriented features but it is complex and difficult to learn

# First Generation: Machine Language

- The first program was programmed using the machine language. This is the lowest level of programming language and is the only language that a computer understands.
- All the commands and data values are expressed using 0s and 1s, corresponding to the off and on electrical states in a computer.
- In the 1950s, each computer had its own native language, and programmers had primitive systems for combining numbers to represent instructions such as *add* and *subtract*. In machine language, all instructions, memory locations, numbers, and characters are represented in strings of ones and zeroes.

- In machine language, all instructions, memory locations, numbers, and characters are represented in strings of 0s and 1s. Although machine language programs are typically displayed with the *binary* numbers represented in *octal (base 8) or hexadecimal (base 16) number systems*, *these programs are* not easy for humans to read, write, or debug.

Advantages	Disadvantages
<ul style="list-style-type: none"><li>• Code can be directly executed by the computer.</li><li>• Execution is fast and efficient.</li><li>• Programs can be written to efficiently utilize memory.</li></ul>	<ul style="list-style-type: none"><li>• Code is difficult to write.</li><li>• Code is difficult to understand by other people.</li><li>• Code is difficult to maintain.</li><li>• There is more possibility for errors to creep in.</li><li>• It is difficult to detect and correct errors.</li><li>• Code is machine dependent and thus non-portable.</li></ul>

# Second Generation—Assembly Language

- 2GLs comprise the assembly languages. Assembly languages are symbolic programming languages that use symbolic notations to represent machine language instructions.
- These languages are closely connected to machine language and the internal architecture of the computer system on which they are used. Since it is close to machine language, assembly language is also a low-level language
- It uses symbolic codes, also known as mnemonic codes, which are easy-to-remember abbreviations, rather than numbers.
- Examples of these codes include ADD for add, CMP for compare, and MUL for multiply.

## Advantages

- It is easy to understand.
- It is easier to write programs in assembly language than in machine language.
- It is easy to detect and correct errors.
- It is easy to modify.
- It is less prone to errors.

## Disadvantages

- Code is machine dependent and thus non-portable.
- Programmers must have a good knowledge of the hardware and internal architecture of the CPU.
- The code cannot be directly executed by the computer.

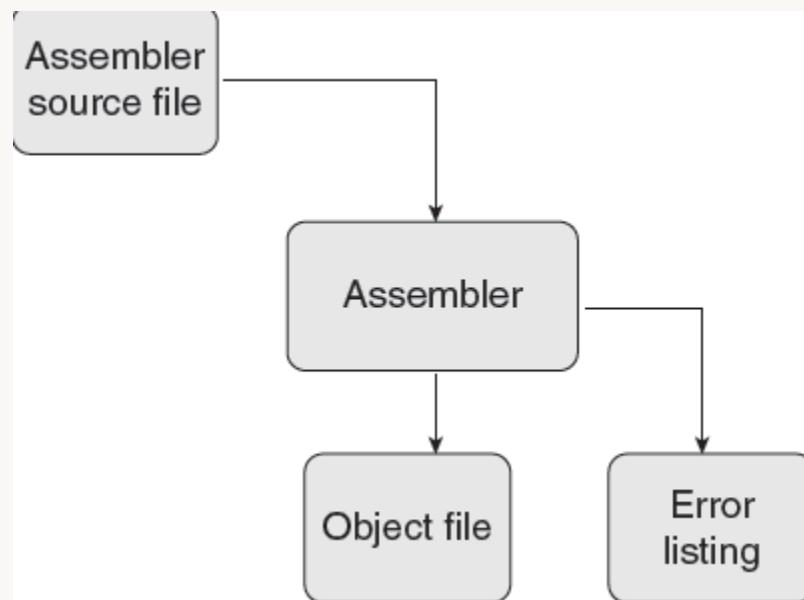
# Second Generation—Assembly Language

- Assembly language programs consist of a series of individual statements or instructions to instruct the computer what to do. Basically, an assembly language statement consists of a label, an operation code, and one or more *operands* .
- Labels are used to identify and refer instructions in the program. The operation code (opcode) is a mnemonic that specifies the operation to be performed, such as *move* , *add* , *subtract* , or *compare* .

# Assembler

- Since computers can execute only codes written in machine language, a special program, called the assembler, is required to convert the code written in assembly language into an equivalent code in machine language, which contains only 0s and 1s.
- There is a one-to-one correspondence between the assembly language code and the machine language code. However, if there is an error, the assembler gives a list of errors. The object file is created only when the assembly language code is free from errors. The object file can be executed as and when required.

# Assembler



# Third Generation: High-level Language

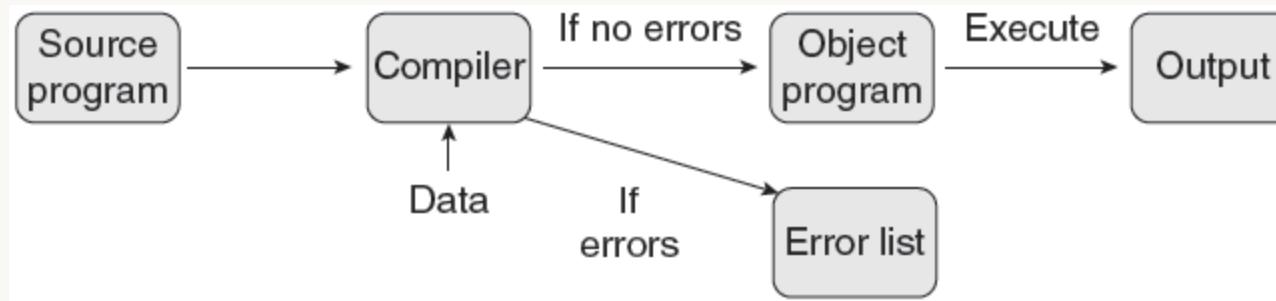
- The third generation was introduced to make the languages more programmer friendly .
- A statement written in a high-level programming language will expand into several machine language instructions.
- A translator is needed to translate the instructions written in a high-level language into the computer-executable machine language. Such translators are commonly known as interpreters and compilers. Each high-level language has many compilers, and there is one for each type of computer.

# Third Generation: High-level Language contd.

Advantages	Disadvantages
<ul style="list-style-type: none"><li>• The code is machine independent.</li><li>• It is easy to learn and use the language.</li><li>• There are few errors.</li><li>• It is easy to document and understand the code.</li><li>• It is easy to maintain the code.</li><li>• It is easy to detect and correct errors.</li></ul>	<ul style="list-style-type: none"><li>• Code may not be optimized.</li><li>• The code is less efficient.</li><li>• It is difficult to write a code that controls the CPU, memory, and registers.</li></ul>

# Compiler

- A compiler is a special type of program that transforms the source code written in a programming language (*the source language* ) *into machine language, which uses only two digits—0 and 1 (the target language)* .
- *The resultant code in 0s and 1s is known as the object code.*
- *The object code is used to create an executable program.*



# Compile Time Errors

- If the source code contains errors, then the compiler will not be able to do its intended task. Errors that limit the compiler in understanding a program are called syntax errors.
- Examples of syntax errors are spelling mistakes, typing mistakes, illegal characters, and use of undefined variables. The other type of error is the logical error, which occurs when the program does not function accurately.
- Logical errors are much harder to locate and correct than syntax errors.
- Whenever errors are detected in the source code, the compiler generates a list of error messages indicating the type of error and the line in which the error has occurred.

# Interpreter

- The interpreter executes instructions written in a high-level language.
- A program written in a high-level language can be executed in any of the two ways—by compiling the program or by passing the program through an interpreter.
- Interpreter translates the instructions into an intermediate form, which it then executes. The interpreter takes one statement of high-level code, translates it into the machine level code, executes it, and then takes the next statement and repeats the process until the entire program is translated.



# Compiler vs Interpreter

## Compiler

- It translates the entire program in one go.
- It generates error(s) after translating the entire program.
- Execution of code is faster.
- An object file is generated.
- Code need not be recompiled every time it is executed.
- It merely translates the code.
- It requires more memory space (to save the object file).

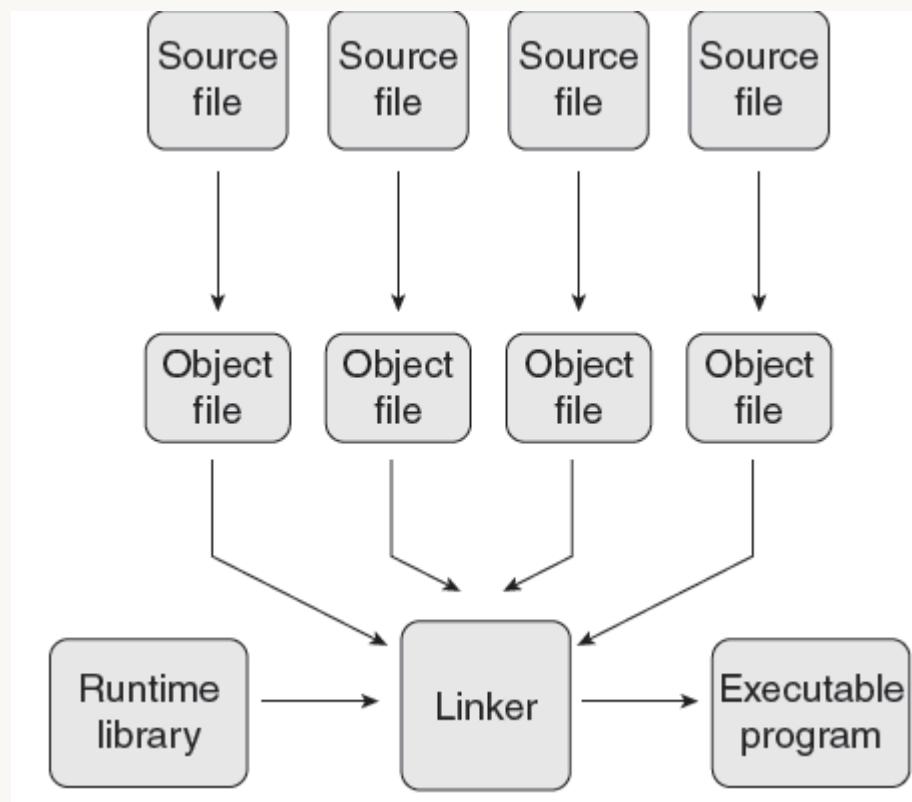
## Interpreter

- It interprets and executes one statement at a time.
- It stops translation after getting the first error.
- Execution of code is slower as every time reinterpretation of statements has to be done.
- No object file is generated.
- Code has to be reinterpreted every time it is executed.
- It translates as well as executes the code.
- It requires less memory space (no object file).

# Linker

- Program is divided into various(smaller) modules as it is easy to code, edit, debug, test, document, and maintain them. Moreover, a module written for one program can also be used for another program. When a module is compiled, an object file of that module is generated.
- Once the modules are coded and tested, the object files of all the modules are combined together to form the final executable file.
- Therefore, a linker, also called a link editor or binder, is a program that combines the object modules to form an executable program..

# Linker contd.



# Loader

- A loader is a special type of program that copies programs from a storage device to the main memory, where they can be executed.
- The functionality and complexity of most loaders are hidden from the users.

# Fourth Generation: Very High-level Languages

- The instructions of the code are written in English-like sentences.
- They are non-procedural, so users concentrate on the ‘what’ instead of the ‘how’ aspect of the task
- The code written in a 4GL is easy to maintain.
- The code written in a 4GL enhances the productivity of programmers, as they have to type fewer lines of code to get something done. A programmer supposedly becomes 10 times more productive when he/she writes the code using a 4GL than using a 3GL.
- A typical example of a 4GL is the query language, which allows a user to request information from a database with precisely worded English-like sentences.

# Fifth Generation Programming Language

- Fifth-generation programming languages (5GLs) are centered on solving problems using the constraints given to a program rather than using an algorithm written by a programmer.
- Most constraint-based and logic programming languages and some declarative languages form a part of the 5GLS.
- These languages are widely used in artificial intelligence research. Another aspect of a 5GL is that it contains visual tools to help develop a program.
- Typical examples of 5GLs include Prolog, OPS5, Mercury, and Visual Basic.

# PROGRAMMING PARADIGMS

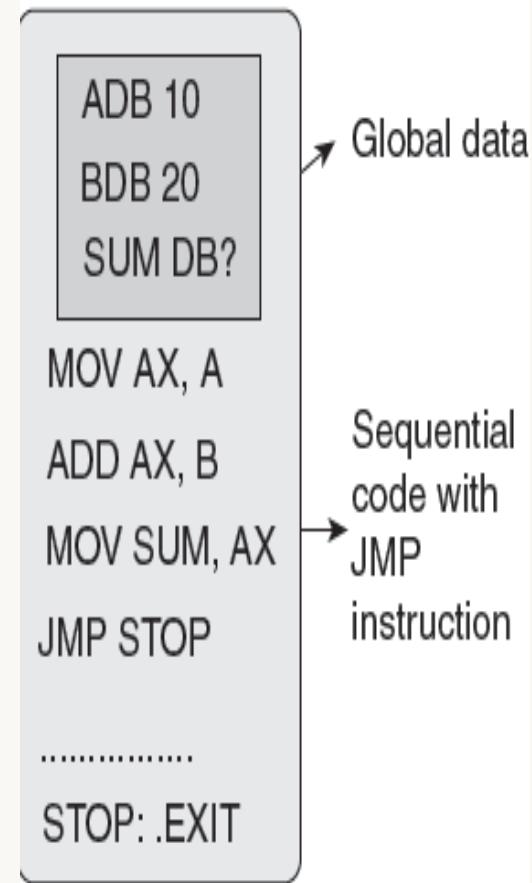
- A programming paradigm is a fundamental style of programming that defines how the structure and basic elements of a computer program will be built. The style of writing programs and the set of capabilities and limitations that a particular programming language has depends on the programming paradigm it supports.
- While some programming languages strictly follow a single paradigm, others may draw concepts from more than one.
- These paradigms, in sequence of their application, can be classified as follows:-
- Monolithic programming—emphasizes on finding a solution

# PROGRAMMING PARADIGMS contd.

- Procedural programming—lays stress on algorithms
- Structured programming—focuses on modules
- Object-oriented programming—emphasizes on classes and objects
- Logic-oriented programming—focuses on goals usually expressed in predicate calculus
- Rule-oriented programming—makes use of ‘if-then-else’ rules for computation
- Constraint-oriented programming—utilizes invariant relationships to solve a problem

# Monolithic Programming

- Programs written using monolithic programming languages such as assembly language and BASIC consist of global data and sequential code. The global data can be accessed and modified (knowingly or mistakenly) from any part of the program, thereby, posing a serious threat to its integrity.
- A sequential code is one in which all instructions are executed in the specified sequence. In order to change the sequence of instructions, jump statements or ‘Go To’ statements are used monolithic programs have just one program module as such programming languages do not support the concept of subroutines.

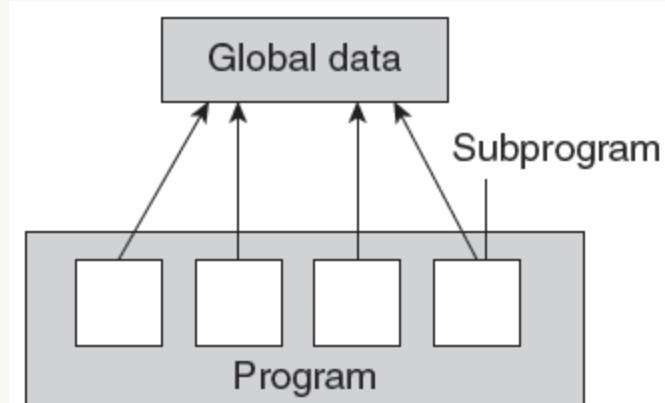


# Procedural Programming

- A program is divided into *n number of subroutines* that access global data
- A subroutine that needs the service provided by another subroutine can call that subroutine.

## Advantages

- The only goal is to write correct programs.
- Programs were easier to write as compared to monolithic programming.



# Procedural Programming contd.

## Disadvantages

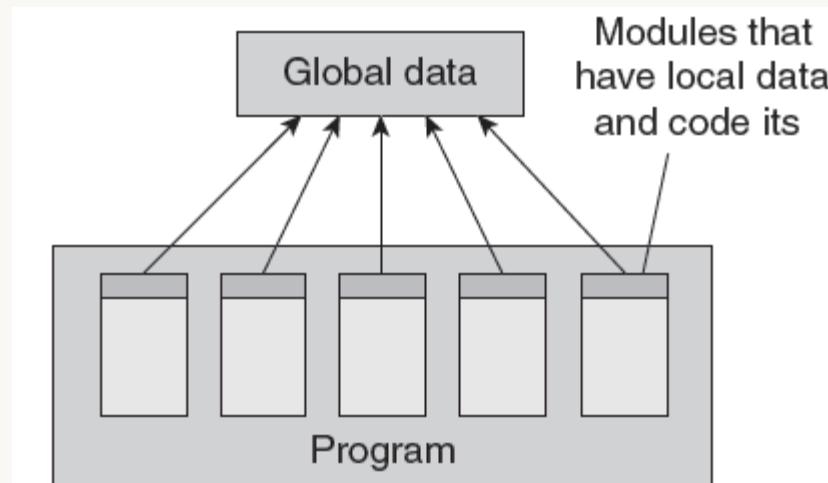
- Writing programs is complex.
- No concept of reusability.
- Requires more time and effort to write programs.
- Programs are difficult to maintain.
- Global data is shared and therefore may get altered (mistakenly).

# Structured Programming

- Structured programming, also referred to as modular programming, employs a top-down approach in which the overall program structure is broken down into separate modules.
- This allows the code to be loaded into memory more efficiently and also be reused in other programs. Modules are coded separately and once a module is written and tested individually, it is then integrated with other modules to form the overall program structure.

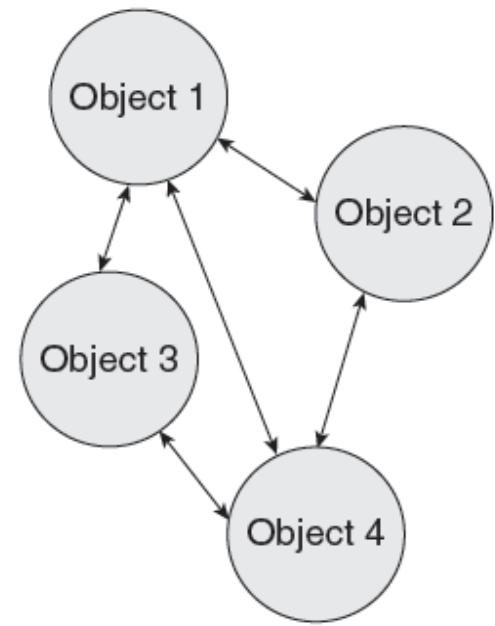
# Structured Programming contd.

- Not data-centered
- Global data is shared and therefore may get inadvertently modified
- Main focus on functions



# Object Oriented Programming

- It treats data as a critical element in the program development and restricts its flow freely around the system.
- The object oriented paradigm is task based (as it considers operations) as well as data-based (as these operations are grouped with the relevant data).
- Every object contains some data and the operations, methods, or functions that operate on that data. While some objects



Objects of a program interact by sending messages to each other

Figure 1.9 Object oriented paradigm

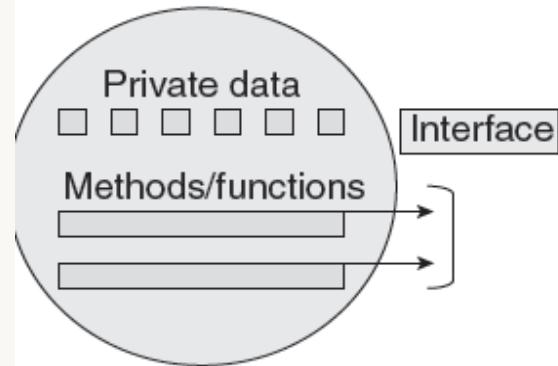


Figure 1.10 Object

# Object Oriented Programming contd.

- May contain only basic data types such as characters, integers, floating types, the other object, the other objects on the other hand may incorporate complex data types such as trees or graphs.
- Programs that need the object will access the object's methods through a specific interface. The interface specifies how to send a message to the object, that is, a request for a certain operation to be performed.
- The programs are data centered.
- Programs are divided in terms of objects and not procedures.
- Functions that operate on data are tied together with the data.

# Object Oriented Programming contd.

- Data is hidden and not accessible by external functions.
- New data and functions can be easily added as and when required.
- Follows a bottom-up approach for problem solving.

# Classes

- OOP, being specifically designed to solve real world problems, allows its users to create user defined data types in the form of classes.
- A class provides a template or a blueprint that describes the structure and behaviour of a set of similar objects.
- Once we have the definition for a class, a specific instance of the class can be easily created.

```
class student
{
    private:
        int roll_no;
        char name[20];
        float marks;
    public:
        get_details();
        show_details();
};
```

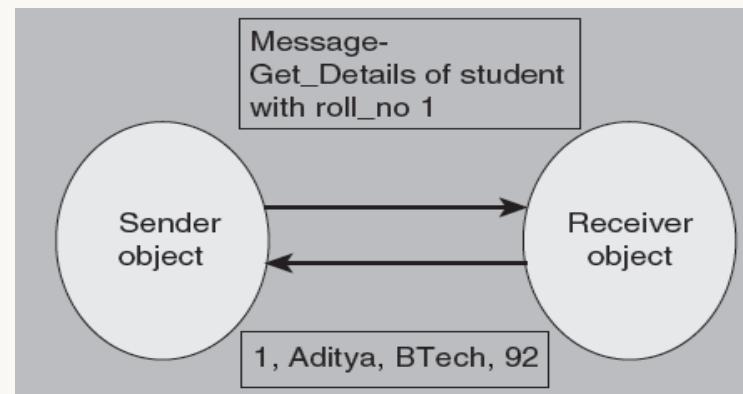
# Objects

- We have taken an example of student class and have mentioned that a class is used to create instances, known as objects. Therefore, if student is a class, then all the 60 students
- In a course (assuming there are maximum 60 students in a particular course) are the objects of the student class. Therefore, all students such as Aditya, Chaitanya, Deepti, and Esha are objects of the class.
- Hence, a class can have multiple instances. Every object contains some data and procedures. They are also called methods.

Object Name
Attribute 1
Attribute 2
.....
Attribute N
Function 1
Function 2
.....
Function N

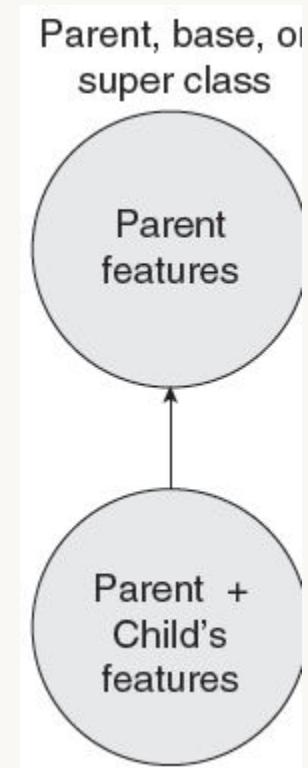
# Method and Message Passing

- A method is a function associated with a class. It defines the operations that the object can execute when it receives a message. In object oriented language, only methods of the class can access and manipulate the data stored in an instance of the class (or object).
- Two objects can communicate with each other through messages. An object asks another object to invoke one of its methods by sending it a message.



# Inheritance

- Inheritance is a concept of OOP in which a new class is created from an existing class. The new class, often known as a sub-class, contains the attributes and methods of the parent class (the existing class from which the new class is created).
- The new class, known as sub-class or derived class, inherits the attributes and behaviour of the pre-existing class, which is referred to as super-class or parent class (refer to Fig.).
- The inheritance relationship of sub- and super classes generates a hierarchy. Therefore, inheritance relation is also called ‘is-a’ relation.
- The main advantage of inheritance is the ability to reuse the code.



# Polymorphism: Static Binding and Dynamic Binding

- Polymorphism, refers to having several different forms.
- It is a concept that enables the programmers to assign a different meaning or usage to a variable, function, or an object in different contexts.
- Dynamic binding or late binding, also known as run time polymorphism, is a feature that enables programmers to associate a function call with a code at the execution (or run) time.
- For example, if we have a function `print_result()` in class `student`, then both the inherited classes, `undergraduate` and `postgraduate` students will also have the same function

# Polymorphism: Static Binding and Dynamic Binding contd.

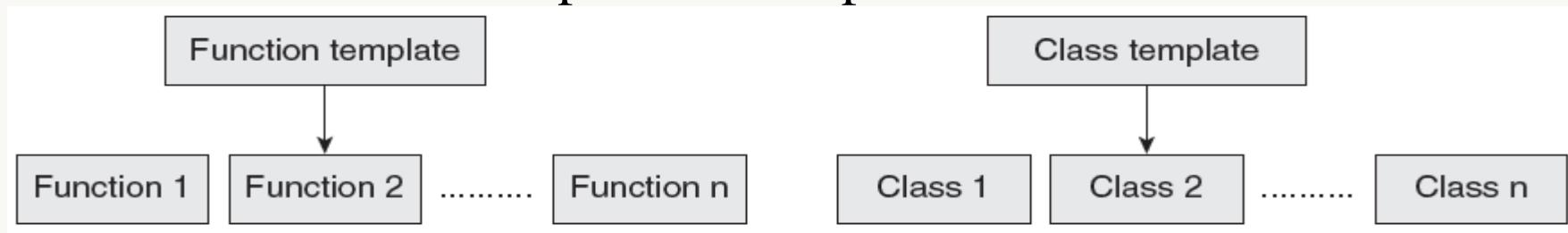
- implemented in their respective classes. Now, when there is a call to print\_result(), ptr\_to\_student-> print\_result(); .
- Then, the decision regarding which version to call—the one in undergraduate class or the one in postgraduate class—will be taken at the execution time.

# Containership

- The ability of a class to contain object(s) of one or more classes as member data. For example,
- Class One can have an object of class Two as its data member. This would allow the object of class One to call the public functions of class Two .
- Here, class One becomes the container, whereas class Two becomes the contained class.
- Containership is also called composition because as in our example, class One is composed of class Two . In OOP, containership represents a ‘has-a’ relationship

# Genericity

- To reduce code duplication and generate short, simpler code, C++ supports the use of generic codes (or templates) to define the same code for multiple data types.
- This means that a C++ function or a class can perform similar operations on different data types like integers, float, and double. This means that a generic function can be invoked with arguments of any compatible type.
- Generic programs, therefore, act as a model of function or class that can be used to generate functions or classes. During program compilation, C++ compiler generates one or more functions or classes based on the specified template.



# Delegation

- In delegation, more than one object is involved in handling a request. The object that receives the request for a service, delegates it to another object called its delegate. The property of delegation emphasizes on the ideology that a complex object is made of several simpler objects.
- For example, our body is made up of brain, heart, hands, eyes, ears, etc., the functioning of the whole body as a system rests on correct functioning of the parts it is composed of. Similarly, a car has a wheel, brake, gears, etc. to control it.
- Delegation differs from inheritance in that two classes that participate in inheritance share an ‘is-a’ relationship; however, in delegation, they have a ‘has-a’ relationship

# Data Abstraction and Encapsulation

- Data abstraction refers to the process by which data and functions are defined in such a way that only essential details are revealed and the implementation details are hidden. The main focus of data abstraction is to separate the interface and the implementation of a program.
- In OOP languages, classes provide public methods to the outside world to provide the functionality of the object or to manipulate the object's data. Any entity outside the world does not know about the implementation details of the class or that method.

# Data Abstraction and Encapsulation contd.

- Data encapsulation, also called data hiding, is the technique of packing data and functions into a single component (class) to hide implementation details of a class from the users.
- Users are allowed to execute only a restricted set of operations (class methods) on the data members of the class.
- Therefore, encapsulation organizes the data and methods into a structure that prevents data access by any function (or method) that is not specified in the class. This ensures the integrity of the data contained in the object

# MERITS OF OOP LANGUAGE

- Elimination of redundant code through inheritance (by extending existing classes).
- Higher productivity and reduced development time due to reusability of the existing modules.
- Secure programs as data cannot be modified or accessed by any code outside the class.
- Real world objects in the problem domain can be easily mapped objects in the program.
- A program can be easily divided into parts based on objects.
- The data centered design approach captures more details of a model in a form that can be easily implemented.

# MERITS OF OOP LANGUAGE contd.

- Programs designed using OOP are expandable as they can be easily upgraded from small to large systems.
- Message passing between objects simplifies the interface descriptions with external systems. Software complexity becomes easily manageable.
- With polymorphism, behavior of functions, operators, or objects may vary depending upon the circumstances.
- Data abstraction and encapsulation hides implementation details from the external world and provides it a clearly defined interface.
- OOP enables programmers to write easily extendable and maintainable programs.
- OOP supports code reusability to a great extent.

# DEMERITS OF OOP LANGUAGE

- Programs written using object oriented languages have greater processing overhead as they demand more resources.
- Requires more skills to learn and implement the concepts.
- Beneficial only for large and complicated programs.
- Even an easy to use software when developed using OOP is hard to be build.
- OOP cannot work with existing systems.
- Programmers must have a good command in software engineering and programming methodology.

# APPLICATIONS OF OOP LANGUAGE

- Designing user interfaces such as work screens, menus, windows, and so on
- Real-time systems
- Simulation and modelling
- Compiler design
- Client server system
- Object oriented databases
- Object oriented distributed database
- Artificial intelligence—expert systems and neural networks

# APPLICATIONS OF OOP LANGUAGE

- Parallel programming
- Decision control systems
- Office automation systems
- Networks for programming routers, firewalls, and other devices
- Computer-aided design (CAD) systems
- Computer-aided manufacturing (CAM) systems
- Computer animation
- Developing computer games
- Hypertext and hypermedia

# Differences between C and C++

C	C++
<ul style="list-style-type: none"><li>• Procedural language</li><li>• Does not support virtual functions</li><li>• Does not support polymorphism</li><li>• Does not support operator overloading</li><li>• Uses top-down approach to build complex programs</li><li>• Can have multiple declarations for global variables</li><li>• Printf() and scanf() functions in stdio.h file are used for I/O</li><li>• Difficult to determine which function can and cannot modify data</li><li>• Allows main() to be called through other functions</li><li>• All variables must be defined at the beginning of the function (or scope)</li><li>• Does not support inheritance</li><li>• Does not support exception handing</li><li>• Uses malloc(), calloc(), and free() for dynamically allocating or de-allocating memory</li><li>• A character constant is automatically elevated to an integer</li><li>• Identifiers cannot start with two or more consecutive underscores, but may contain them in other positions.</li></ul>	<ul style="list-style-type: none"><li>• Object oriented language</li><li>• Support virtual functions</li><li>• Supports polymorphism</li><li>• Supports operator overloading</li><li>• Uses bottom-up approach to build complex programs</li><li>• Cannot have multiple declarations for global variables</li><li>• Objects cin and cout of iostream.h are used for input or output</li><li>• Easy mapping between data and functions</li><li>• Does not allow main() to be called through other functions</li><li>• Variables can be defined at any location but before their first use.</li><li>• Supports inheritance</li><li>• Supports exception handing</li><li>• Uses new and delete operators for dynamically allocating or de-allocating memory</li><li>• A character constant is not elevated to integer</li><li>• Identifiers are not allowed to contain two or more consecutive underscores in any position</li></ul>

# Comparison between commonly used object oriented languages

Attributes	EIFFEL	C++	JAVA	SMALLTALK
Static typing	Statically typed	Statically typed but supports C-style ‘casts’ which may lead to violation of type rules	Statically typed but needs dynamic typing for generic container structures	Dynamically typed
Proprietary status	Open and standardized	Open, ANSI standard	Licensed from Sun	Not standardized
Compilation technology	Combination of interpretation and compilation	Compiled	Interpreted and on the fly compilation	Initially interpreted but currently mix of interpretation and compilation
Efficiency of code	Fast executable	Fast executable	Performance problems	Executables require a ‘Smalltalk image’
Multiple inheritance	Supported and widely used	Supported but not widely used because of performance problems	Single inheritance	Single inheritance
Understandability	Clear and simple	Complex syntax	Complex syntax	In between simple and complex
Garbage collection	Supported automatically	Not supported	Supported automatically	Supported automatically
Encapsulation	Supported	Supported	Poor support	Supported
Binding (early or late)	Early	Both	Late	Late



OXFORD  
HIGHER EDUCATION

# Object Oriented Programming with C++



Reema Thareja

OXFORD  
HIGHER EDUCATION

# Object Oriented Programming with C++

Reema Thareja

# Chapter Two

## Basics of C++ Programming

# INTRODUCTION TO C++

- C++ is a general purpose programming language developed by Bjarne Stroustrup in 1979 at Bell Labs. Similar to C programming, C++ is also considered as an intermediate level language because it includes the features of both high-level and low-level languages.
- C++ is a very popular programming language that can be implemented on a wide variety of hardware and operating system platforms.
- It is a powerful language for high-performance applications, including writing operating systems, system software,

# INTRODUCTION TO C++

- application software, device drivers, embedded software, high-performance client and server applications, software engineering, graphics, games, and animation software.
- C++ is an object oriented programming (OOP) language and facilitates design, reuse, and maintenance for complex software.

# HISTORY OF C++

- In 1979, **Bjarne Stroustrup** started working on ‘C with Classes’, with an aim to integrate OOP features such as classes, inheritance, inline functions, and default arguments with the C language.
- This new language was easily portable and fast, provided low-level functionality, and included OOP concepts.
- In 1983, the name of the language was changed to C++, where ++ refers to adding new features in the C language. At this time, some more features such as virtual functions, function overloading, references with the & symbol, const keyword, and single-line comments were added.
- C++ was again updated to include protected members, static members, and multiple inheritance in 1990.
- The latest version of C++ known as C++11 was released in 2011. It added new features such as support for regular expression, a new C++ time library, atomics support, a standard threading library, a new for loop syntax (similar to for each loop), the auto keyword, new container classes, and better support for unions and array-initialization.

# STRUCTURE OF C++ PROGRAM

- The preprocessor directives contain special instructions that indicate how to prepare the program for compilation.
- Global declaration section is used to declare data (variables), functions, and structures that have global scope.
- The class declaration and method definition section can be considered as a part of global declaration section that is used to declare classes the execution of a C++ program begins at this function as the operating system automatically calls it main().
- The method definition section is optional and required only if there are functions other than class methods in the program

Preprocessor's directive section
Global declaration section
Class declaration and method definition section
Main function
Method definition section

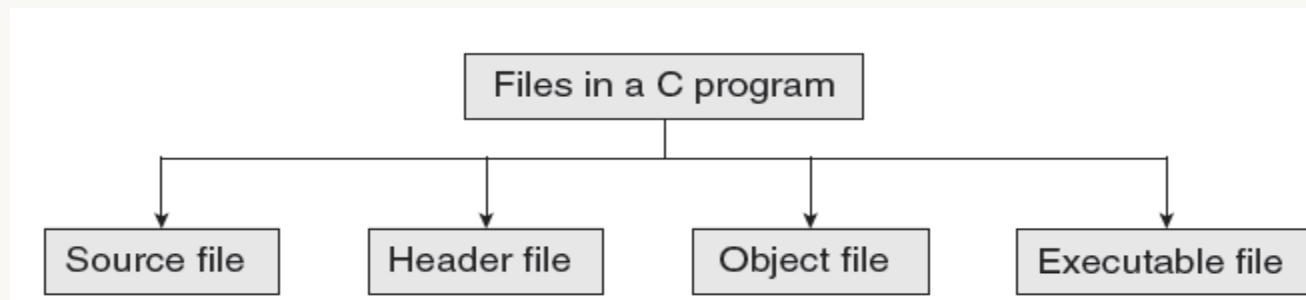
# Writing the first C++ Program

```
1.// My first program
2. #include<iostream.h>
3. main()
4. {
5.     cout<<"\n Welcome to the World of C++";
6.     return 0;
7. }
```

Escape sequence	Purpose	Escape sequence	Purpose
\a	Audible signal	\?	Question mark
\b	Backspace	\\"	Back slash
\t	Tab	\'	Single quote
\n	Newline	\"	Double quote
\v	Vertical tab	\0	Octal constant
\f	New page\Clear screen	\x	Hexadecimal constant
\r	Carriage return		

# Files used in C++ Program

- The source code file contains the source code of the program. The file extension of any C++ source code file is .cpp .
- There are functions which are used without coding. These functions are already written and compiled for ease of programmers. We just need to link the desired library function's code with our code.
- Such functions provided by all C++ compilers are included in standard header files.



# Files used in C++ Program

- Object files are generated by the compiler after processing the source code file. Object files contain a compact binary code of the function definitions
- The binary executable file is generated by the linker. The linker links the various object files to produce a binary file that can be directly executed.

Header files	Description
string.h	Used for string handling functions
graphics.h	Used for graphics functions
dos.h	Used for DS functions
time.h	Used for manipulating time and date
stdlib.h	Used for functions such as random numbers and converting numbers to text
stdio.h	Used for standard input and output functions
math.h	Used for mathematical functions
complex.h	Used for complex math functions
alloc.h	Used for memory management functions
conio.h	Used for clearing the screen
assert.h	Used for adding diagnostics that helps in program debugging

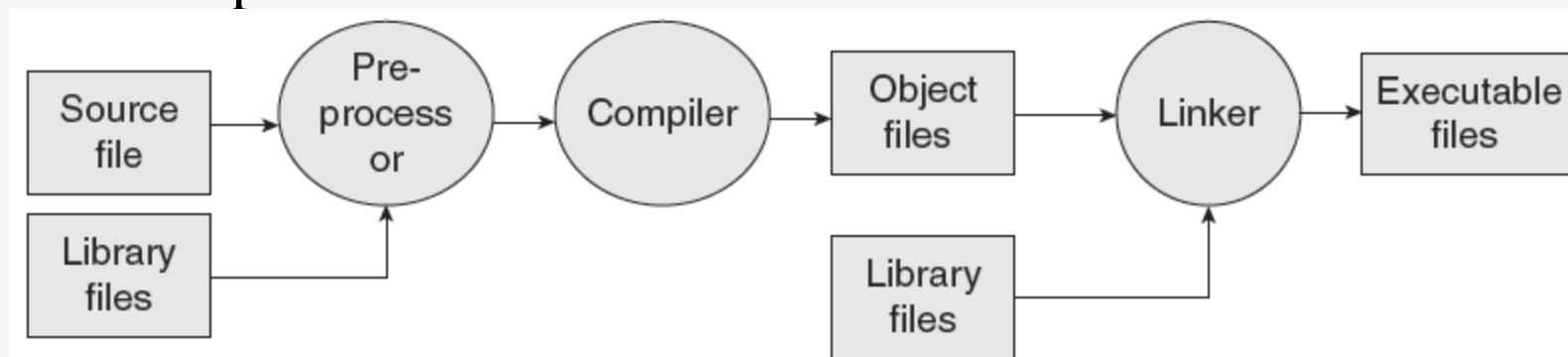
# Using comments

- Comments are used to explain what a program does.
- They are non-executable statements.
- C++ supports two types of commenting.
- // is used to comment a single statement. This is known as a line comment.
- /\* is used to comment multiple statements. A /\* is ended with \*/ and all statements that lie within these characters are commented. This type of comment is known as block comment.

```
/* Author: Reema Thareja
Description: To print "Welcome to the world of C++" on the screen */
#include<iostream.h>
int main()
{
    cout<<"\n Welcome to the world of C++ "; //prints message
    return 0; // returns a value 0 to the operating system
```

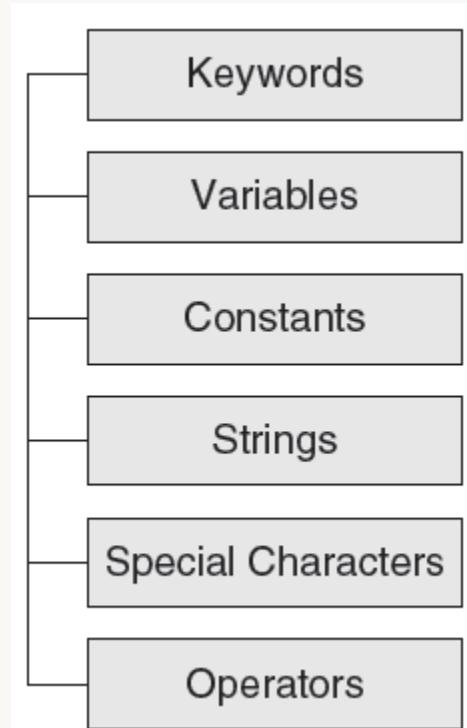
# COMPILING AND EXECUTING C++ PROGRAMS

- The programming process starts with creating a source file that consists of the statements of the program written in C++ language.
- The compiler translates the source code into an object code the object file is processed with another special program called a linker.
- The output of the linker is an executable or runnable file.



# TOKENS IN C++

- Tokens are the basic buildings blocks in C++ language. You may think of a token as the smallest individual unit in a C++ program.
- This means that a program is constructed using a combination these tokens.
- There are six main types of tokens in C++.



# CHARACTER SET

- Computer languages also use a character set that defines the fundamental raw material used to represent information.
- In C++, a character means any alphabet, digit, or special symbols used to represent information. These characters when combined together forms a token that acts as the basic building block of a C++ program. The character set of C++ can, therefore, be given as:
- Alphabets include both lower case (a – z) as well as upper case (A – Z) characters.

# CHARACTER SET

- Digits include numerical digits from 0 to 9.
- Special characters include symbols like ~, @, %, ^, &, \*, {, }, <. >, =, \_, +, -, \$, /, , (, ), \, ;, :, [, ], ‘, “, ?, .. !,.
- White space characters are used to print a blank space on the screen.
- Escape sequences have already been discussed in Section 2.8. They include- \\, \', \", \n, \a, \0, \?.

# KEYWORDS

- C++ has a set of reserved words, often known as keywords that cannot be used as an identifier. All keywords are basically a sequence of characters that have a fixed meaning.
- The meaning of a keyword cannot be changed by the programmer. Conventionally, all keywords must be written in lowercase.

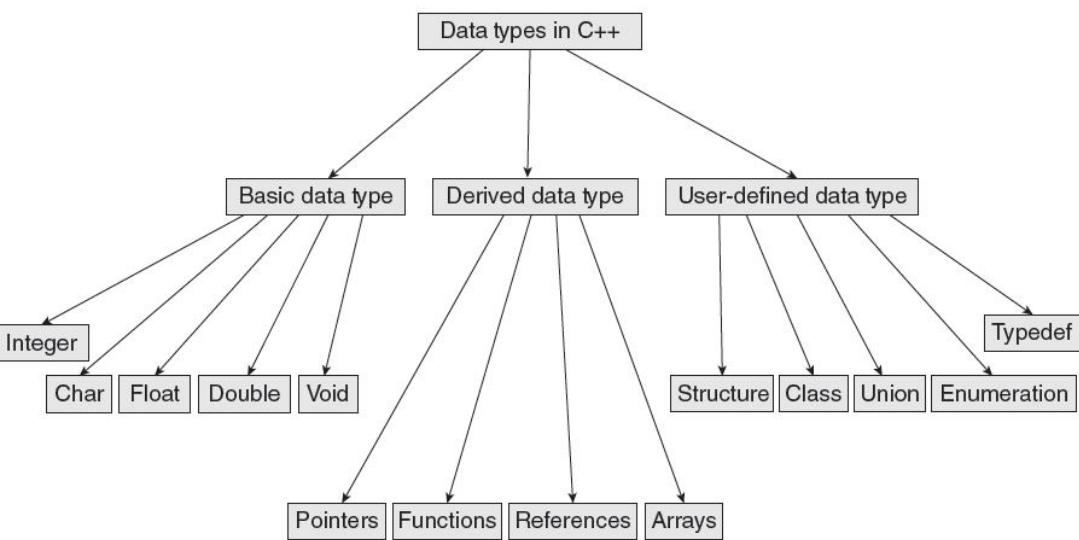
auto	break	case	char	const	continue	default	Do
double	else	enum	extern	Float	for	goto	if
int	long	register	return	Short	signed	sizeof	static
struct	switch	typedef	union	unsigned	void	volatile	while
asm	new	template	catch	operator	this	class	private
throw	delete	protected	try	friend	public	virtual	inline

# IDENTIFIER

- Identifiers, as the name suggests, helps us to identify data and other objects in the program. Identifiers are basically the names given to program elements such as variables, arrays, and functions.
- An identifier may consist of an alphabet, digit, or an underscore.
- The name cannot include any special characters or punctuation marks, except the underscore "\_" .There cannot be two successive underscores.
- Keywords cannot be used as identifiers.
- The case of alphabetic characters that form the identifier name is significant. The identifier name must begin with an alphabet or an underscore.
- Identifiers can be of any reasonable length.

# DATA TYPES IN C++

- Data are used to represent information.



Data type	Size in bytes	Range
Char	1	-128 to 127
unsigned char	1	0 to 255
signed char	1	-128 to 127
Int	2	-32768 to 32767
unsigned int	2	0 to 65535
signed short int	2	-32768 to 32767
signed int	2	-32768 to 32767
short int	2	-32768 to 32767
unsigned short int	2	0 to 65535
long int	4	-2147483648 to 2147483647
unsigned long int	4	0 to 4294967295
signed long int	4	-2147483648 to 2147483647
Float	4	3.4E-38 to 3.4E+38
Double	8	1.7E-308 to 1.7E+308
long double	10	3.4E-4932 to 1.1E+4932

# VARIABLES

- Variable is defined as a meaningful name given to the data storage location in computer memory.
- When using a variable, we actually refer to address of the memory where the data is stored.
- Numeric variables can be used to store either integer values or floating point values.
- Character variables can include any letter from the alphabet or from the ASCII chart and numbers 0 – 9 inserted between single quotes.
- In C++, a number that is put in single quotes is not the same as a number without them.

# Declaring Variables

- Each variable to be used in the program must be declared. To declare a variable, specify the data type of the variable, followed by its name.
- In C++, variables are declared at three basic places as follows:
- First, when a variable is declared inside a function it is known as a local variable.
- Second, when a variable is declared in the definition of function parameters, it is known as formal parameter (we will study this in Chapter 4).
- Third, when the variable is declared outside all functions, it is known as a global variable.

```
data_type variable_name;
```

```
int emp_num;  
float salary;  
char grade;  
double balance_amount;  
unsigned short int acc_no;
```

# Initializing Variables

```
int emp_num = 7;  
float salary = 5000;  
char grade = 'A';  
double balance_amount = 10000000;
```

# Reference Variables

- Reference variable is a new feature added to C++. A reference variable basically assigns an alternative name (alias) to an existing variable.
- For example, if we want a variable num to be a reference to the variable n , then n and num can be interchangeably used to represent that variable.
- This is same as if someone calls us by our nick name or by our formal name, both names refer to the same person.
- Therefore, if we write

```
data_type & reference_name = variable_name  
  
int n = 10;  
int &num = n;
```

Then, both n and num are of type integers and refer to the same data. Therefore,

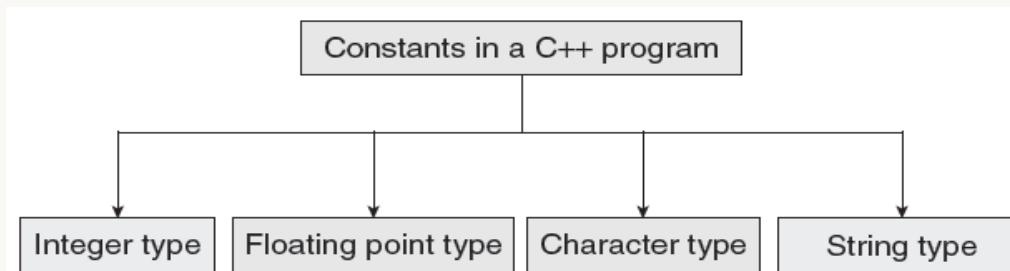
```
cout<<num; // will print 10    and    cout<<n; // will print 10
```

Therefore, if we write num = 100; Then

```
cout<<num; // will print 100   and   cout<<n; // will print 100
```

# CONSTANTS

- Constants are identifiers whose value does not change. While variables can change their value at any time, constants can never change their value.
- Constants are used to define fixed values such as Pi or the charge on an electron so that their value does not get changed in the program even by mistake.
- A constant is an explicit data value specified by the programmer.
- The value of the constant is known to the compiler at the compile time.



# Declaring Constants

- Rule 1 Constant names are usually written in capital letters to visually distinguish them from other variable names which are normally written in lower case characters.
- Rule 2 No blank spaces are permitted in between the # symbol and define keyword.
- Rule 3 Blank space must be used between #define and constant name and between constant name and constant value.
- Rule 4 #define is a preprocessor compiler directive and not a statement. Therefore, it does not end with a semi-colon.

```
const data_type const_name = value;
```

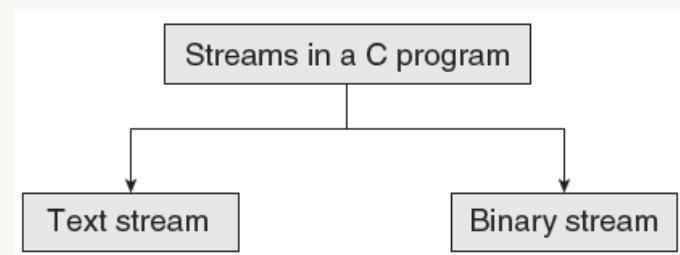
The const keyword specifies that the value of pi cannot change.

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

```
#define PI 3.14159  
#define service_tax 0.12
```

# Streams

- A stream involves transfer of information as a sequence of bytes. A stream acts in two ways. It is the source as well as the destination of data. C++ programs input data and output data from a stream.
- Streams are associated with a physical device such as the monitor or with a file stored on the secondary memory.
- In a text stream, the sequence of characters is divided into lines, with each line being terminated.
- With a new-line character (`\n`) . On the other hand, a binary stream contains data values using their memory representation.



# Cascading of Input or Output Operators

- We can use the << operator multiple times in the same line. This is called cascading.
- Similar to cout, cin can also be cascaded. For example,

```
cout<<"\n Enter the marks obtained in English and Hindi";  
cin>>marks Hindi >> marks English;
```

# Reading and Writing Characters and Strings

```
char grade;  
cin.get(grade); //The value for grade is read  
OR  
grade = cin.get(); // A character is read and assigned to grade
```

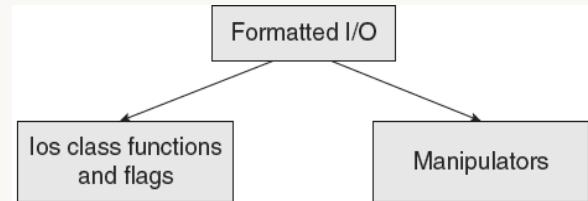
```
string mesg;  
cin>>mesg;
```

```
string name;  
cin.getline(name,20);  
cout<<"\n Welcome, "<<name;
```

# Formatted Input and Output Operations

Table 2.7 Functions for Input/Output

Function	Purpose	Syntax	Usage	Result	Comment
width()	Specifies field size (no. of columns) to display the value	cout.width(w)	cout.width(6);cout<<1239;	2 3   9	It can specify field width for only one value
precision()	Specifies no. of digits to be displayed after the decimal point of a float value	cout.precision(d)	cout.precision(3);cout<<1.234567;	1.234	It retains the setting in effect until it is reset
fill()	Specify a character to fill the unused portion of a field	cout.fill(ch)	Cout.fill('#');cout.width(6);cout<<1239;	# #   2 3   9	It retains the setting in effect until it is reset



```
#include<iostream.h>
#define PI 3.14159
main()
{
    cout.precision(3);
    cout.width(10);
    cout.fill('#');
    cout<<PI;
}
```

**OUTPUT**  
#####3.142

# Formatting with Flags

- The setf() is a member function of the ios class that is used to set flags for formatting output.
- The syntax for setf function that stands for set flags can be given as cout.setf(flag, bit-field)
- Here, flag defined in the ios class specifies how the output should be formatted and bit-field is a constant (defined in ios ) that identifies the group to which the formatting flag belongs to.
- There are two types of setf()—one that takes both flag and bit-fields and the other that takes only the flag .

Table 2.8 Types of setf()

Flag	Bit-field	Usage	Purpose	Comment
ios :: left	ios :: adjustfield	cout.setf(ios :: left, ios :: adjustfield)	For left justified output	If no flag is set, then by default, the output will be right justified.
ios :: right	ios:: adjustfield	cout.setf(ios :: right, ios :: adjustfield)	For right justified output	
ios :: internal	ios:: adjustfield	cout.setf(ios :: internal, ios :: adjustfield)	Left-justify sign or base indicator, and right-justify rest of number	
ios :: scientific	ios:: floatfield	cout.setf(ios :: scientific, ios :: floatfield)	For scientific notation	If no flag is set then any of the two notations can be used for floating point numbers depending on the decimal point
ios :: fixed	ios :: floatfield	cout.setf(ios :: fixed, ios :: floatfield)	For fixed point notation	
ios :: dec	ios :: basefield	cout.setf(ios :: dec, ios :: basefield)	Displays integer in decimal	If no flag is set, then the output will be in decimal.
ios :: oct	ios :: basefield	cout.setf(ios :: oct, ios :: basefield)	Displays integer in octal	
ios :: hex	ios :: basefield	cout.setf(ios :: hex, ios :: basefield)	Displays integer in hexadecimal	
ios :: showbase	Not applicable	cout.setf(ios :: showbase)	Show base when displaying integers	
ios :: showpoint	Not applicable	cout.setf(ios :: showpoint)	Show decimal point when displaying floating point numbers	
ios :: showpos	Not applicable	cout.setf(ios :: showpos)	Show + sign when displaying positive integers	
ios :: uppercase	Not applicable	cout.setf(ios :: uppercase)	Use uppercase letter when displaying hexadecimal (OX) or exponential numbers (E)	

```
#include<iostream.h>
#include<math.h>
main()
{
    int num;
    cout<<"\n Enter a number : ";
    cin>>num;
    // To display hexadecimal of 64
    cout.setf(ios::left );
    cout<<"\n Hexadecimal of" <<num;
    // To display ****40
    cout.fill("*");
    cout.unsetf(ios::left);
    cout.width(15);
    cout.setf(ios::hex);
    cout<<num;
    // To display octal of 64
    cout.setf(ios::left);
    cout.unsetf(ios::hex);
    cout<<"\n Octal of " <<num;
    // To display *****100
    cout.unsetf(ios::left);
    cout.width(20);
    cout.setf(ios::oct);
    cout<<num;
    // To Display Square root of
    cout.setf(ios::left | ios :: showpoint | ios::showpos);
    cout.unsetf(ios::oct);
    cout<<"\n Square Root of " <<num;
    // *****+8.000
    cout.unsetf(ios::left);
    cout.precision(3);
    cout.width(20);
    cout<<sqrt(num);
}
```

# Formatted Output Using Manipulators

- C++ has a header file iomanip.h that contains certain manipulators to format the output.
- Though you can do the same formatting using ios class flags , formatting with manipulators is easier to learn and convenient to use.

Manipulator	Purpose	Usage	Result	Alternative
setw(w)	Specifies field size (number of columns) to display the value	cout<<setw(6) <<1239;	1 2 3 9	width()
setprecision(d)	Specifies number of digits to be displayed after the decimal point of a float value	cout<<setprecision(3) <<1.234567;	1.235	precision()
setfill(c)	Specify a character to fill the unused portion of a field	cout<<setfill('#') <<setwidth(6) <<1239;	# # 1 2 3 9	fill()

## Programming

**Tip:** endl is a manipulator that adds a newline. Instead of using “\n”, you can also write coutt<<endl;.

```
#include<iostream.h>
#include>iomanip.h>
#include>math.h>
main()
{   int num;
    cout<<"\n Enter a number : ";
    cin>>num;
    // To display HexaDecimal of 64
    cout<<"\n HexaDecimal of "<<num;
    cout<<setw(10)<<setfill('*')<<hex<<num;
    // To display octal of 64
    cout<<"\n Octal of "<<dec<<num;
    cout<<setw(20)<<setfill('*')<<oct<<num;
    // To display Square root of
    cout<<"\n Square root of "<<dec<<num;
    cout.set(ios::showpoint | ios::showpos);
    cout<<setw(20)<<setfill('*')<<setprecision(3)<<dec<<sqrt(num);
}
```

## OUTPUT

```
Enter a number : 64
HexaDecimal of 64*****40
Octal of 64*****100
Square Root of +64*****+8.000
```

# OPERATORS IN C++

- An operator is defined as a symbol that specifies the mathematical, logical, or relational operation to be performed.
- C++ language supports a lot of operators to be used in expressions. These operators can be categorized into the following major groups:
  - Arithmetic operators
  - Equality operators
  - Unary operators
  - Bitwise operators
  - Comma operator
  - Relational operators
  - Logical operators
  - Conditional operators
  - Assignment operators
  - Sizeof operator

# Arithmetic Operators

Operation	Operator	Syntax	Comment	Result
Multiply	*	a * b	result = a * b	27
Divide	/	a / b	result = a / b	3
Addition	+	a + b	result = a + b	12
Subtraction	-	a - b	result = a - b	6
Modulus	%	a % b	result = a % b	0

```
#include<iostream.h>
int main()
{
    float c = 20.0;
    cout("\n Result = %f", <<c%5);
    // WRONG. Modulus operator is being applied to a float operand
    return 0;
}
```

# Relational Operators

Operator	Meaning	Example
<	Less than	3 < 5 gives 1
>	Greater than	7 > 9 gives 0
<=	Less than or equal to	100 <= 100 gives 1
>=	Greater than equal to	50 >= 100 gives 0

# Equality Operators

Operator	Meaning
==	Returns 1 if both operands are equal, 0 otherwise
!=	Returns 1 if operands do not have the same value, 0 otherwise

# Logical Operators

Table 2.13 Truth table of logical AND

A	B	A && B
0	0	0
0	1	0
1	0	0
1	1	1

Table 2.14 Truth table of logical OR

A	B	A    B
0	0	0
0	1	1
1	0	1
1	1	1

Table 2.15 Truth table of logical NOT

A	!A
0	1
1	0

**Program 2.7 Write a program that demonstrates the use of relational operators.**

```
#include<iostream.h>
int main()
{
    int x=10, y=20;
    cout<<"\n "<<x<< " < " << y<< " = " << (x<y);
    cout<<"\n "<<x<< " == " << y<< " = " << (x==y);
    cout<<"\n "<<x<< " ! " << y<< " = " << (x!=y);
    cout<<"\n "<<x<< " > " << y<< " = " << (x>y);
    cout<<"\n "<<x<< " >= " << y<< " = " << (x>=y);
    cout<<"\n "<<x<< " <= " << y<< " = " << (x<=y);
    return 0;
}
```

**OUTPUT**

10 < 20 = 1 (true because 10 is less than 20 is true)

10 == 20 = 0 (false because 10 is not equal to 20)

10 != 20 = 1 (true because 10 is not equal to 20)

10 > 20 = 0 (false because 10 is not greater than 20)

10 >= 20 = 0 (false because 10 is neither greater nor equal than 20)

10 <= 20 = 1 (true because 10 is less than 20)

# Unary Minus (-)

- Unary minus operator is strikingly different from the binary arithmetic operator that operates on two operands and subtracts the second operand from the first operand. When an operand is preceded by a minus sign, the unary operator negates its value.
- For example, if a number is positive, it becomes negative when preceded with a unary minus operator. Similarly, if the number is negative, it becomes positive after applying the unary minus operator.

```
int a, b = 10;  
a = -(b);
```

# Increment Operator (++) and Decrement Operator (--)

- The increment operator is a unary operator that increases the value of its operand by 1 .
- Similarly, the decrement operator decreases the value of its operand by 1. For example,  $--x$  is equivalent to writing  $x = x - 1$ .
- The increment/decrement operators have two variants—prefix or postfix. In a prefix expression( $++x$  or  $--x$ ) , the operator is applied before an operand is fetched for computation; thus, the altered value is used for the computation of the expression in which it occurs.

# Increment Operator (++) and Decrement Operator (--)

- On the contrary, in a postfix expression ( $x++$  or  $x--$ ) , an operator is applied after an operand is fetched for computation.
- Therefore, the unaltered value is used for the computation of the expression in which it occurs.

```
int x = 10, y;  
y = x++;  
is equivalent to writing  
y = x;  
x = x + 1;  
whereas, y = ++x;  
is equivalent to writing  
x = x + 1;  
y = x;
```

# Conditional Operators

The conditional operator or the ternary (?:) is just similar to an if ... else statement that can be within expressions. Such an operator is useful in situations in which there are two or more alternatives for an expression. The syntax of the conditional operator is as follows:

```
exp1 ? exp2 : exp3
```

exp1 is evaluated first. If it is true, then exp2 is evaluated and becomes the result of the expression; otherwise, exp3 is evaluated and it becomes the result of the expression. For example,

```
large = ( a > b ) ? a : b
```

The conditional operator is used to find largest of two given numbers. First, exp1, that is  $a>b$  is evaluated. If a is greater than b, then large = a, else large = b. Hence, large is equal to either a or b but not both.

## **Program 2.10 Write a program to find largest of two numbers using ternary operator**

```
#include<iostream.h>
int main()
{
    int num1, num2, large;
    cout<<"\n Enter the two numbers : ";
    cin>>num1>>num2;
    large = num1>num2?num1:num2;
    cout<<"\n The largest number is : "<< large;
    return 0;
}
```

# Bitwise Operators

```
10101010 & 01010101 = 11111111
```

```
int a = 10, b = 20, c = 0;  
c = a|b
```

```
10101010 ^ 01010101 = 11111111
```

have  $x = 0001\ 1101$ , then

```
x >> 1 gives result = 0000\ 1110.
```

Similarly, if we have  $x = 0001\ 1101$ , then

```
x << 4 gives result = 0000\ 0001,
```

```
~10101011 = 01010100
```

# Assignment Operators

if we have,

```
int x = 2, y = 3, sum = 0;  
sum = x + y;  
then sum = 5.
```

The assignment operator has right-to-left associativity, so the expression

```
a = b = c = 10;
```

is evaluated as

```
(a = (b = (c = 10))));
```

# Comma Operator

- The comma operator in C++ takes two operands. It works by evaluating the first and discarding its value, and then evaluates the second and returns the value as the result of the expression.
- Comma separated operands, when chained together, are evaluated in a left-to-right sequence with the rightmost value yielding the result of the expression.
- Among all the operators, the comma operator has the lowest precedence.
- Therefore, when a comma operator is used, the entire expression evaluates to the value of the right expression.

```
int a=2, b=3, x=0;  
x = (++a, b+=a);  
Now, the value of x = 6.
```

# Size of Operator

- The sizeof is a unary operator used to calculate the sizes of data types. It can be applied to all data types.
- When using this operator, the keyword ‘ sizeof ’ is followed by a type name, variable, or expression.
- The operator returns the size of the variable, data type, or expression in bytes. Therefore, the sizeof operator is used to determine the amount of memory space that the variable, expression, or data type will take.

sizeof(char) returns 1, that is the size of a character data type

If we have,

```
int a = 10;  
unsigned int result;  
result = sizeof(a);
```

# Size of Operator

- When a type name is used, it is enclosed in parentheses; however, in case of variable names and expressions, they can be specified with or without parentheses.
- A sizeof expression returns an unsigned value that specifies the size the space in bytes required by the data type, variable, or expression.

# Type conversion and type casting

- Type conversion or typecasting of variables refers to changing a variable of one data type into another. While type conversion is done implicitly, casting has to be done explicitly by the programmer.

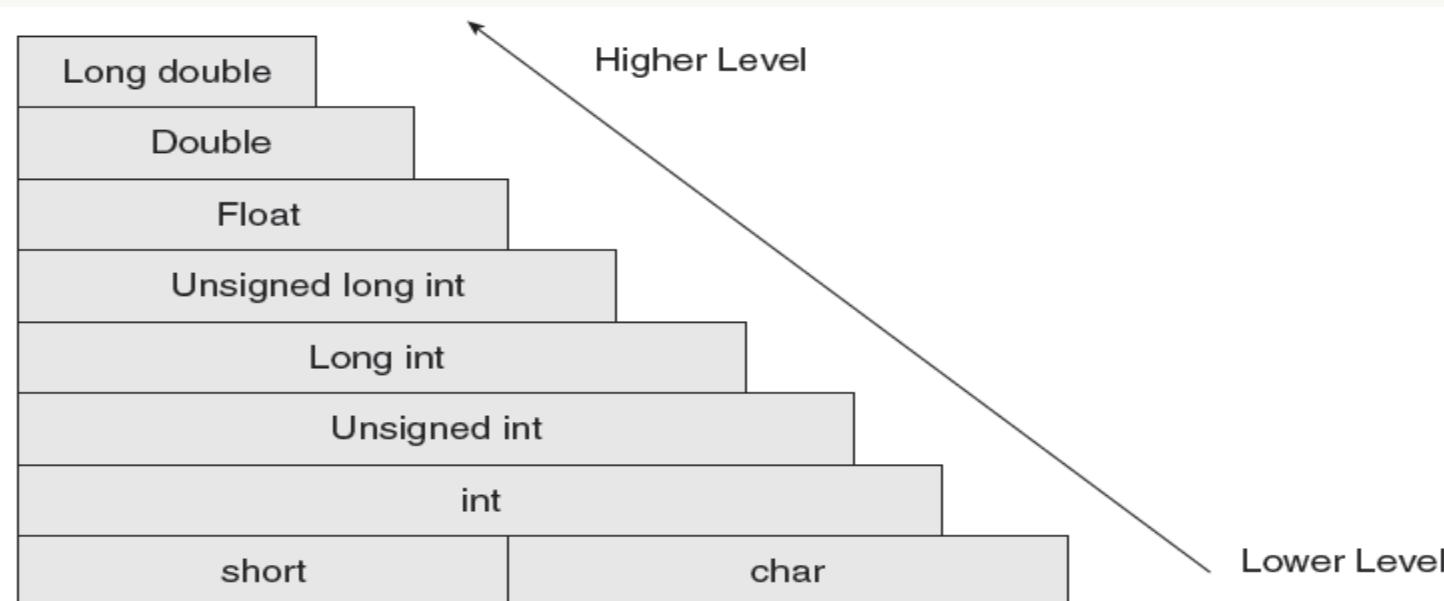


Figure 2.13 Conversion hierarchy of data types

# Type Casting

- Type casting an arithmetic expression tells the compiler to represent the value of the expression in a certain format.
- It is done when the value of a higher data type has to be converted into the value of a lower data type.
- However, this cast is under the programmer's control and not under the compiler's control. The general syntax for type casting is **destination\_variable\_name=destination\_data\_type(source\_variable\_name);**

```
float salary = 10000.00;
int sal;
sal = int (salary);
```



OXFORD  
HIGHER EDUCATION

# Object Oriented Programming with C++



Reema Thareja

OXFORD  
HIGHER EDUCATION

# Object Oriented Programming with C++

Reema Thareja

# Chapter Three

## Decision Control and Looping Statements

# DECISION CONTROL STATEMENTS

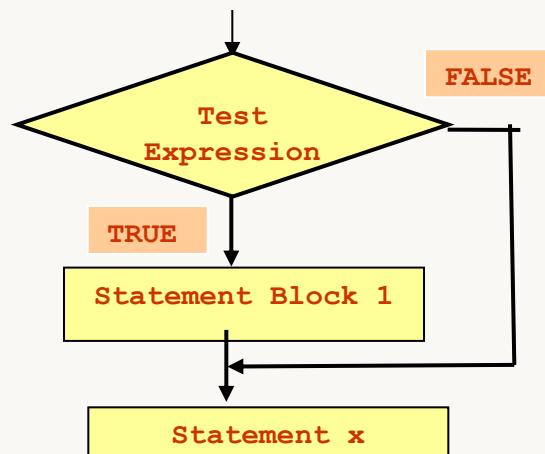
- Decision control statements are used to alter the flow of a sequence of instructions.
- These statements help to jump from one part of the program to another depending on whether a particular condition is satisfied or not.
- These decision control statements include:
  - » If statement
  - » If else statement
  - » If else if statement
  - » Switch statement

# IF STATEMENT

- If statement is the simplest form of decision control statements that is frequently used in decision making. The general form of a simple if statement is shown in the figure.
- First the test expression is evaluated. If the test expression is true, the statement of if block (statement 1 to n) are executed otherwise these statements will be skipped and the execution will jump to statement x.

## SYNTAX OF IF STATEMENT

```
if (test expression)
{
    statement 1;
    .....
    statement n;
}
statement x;
```

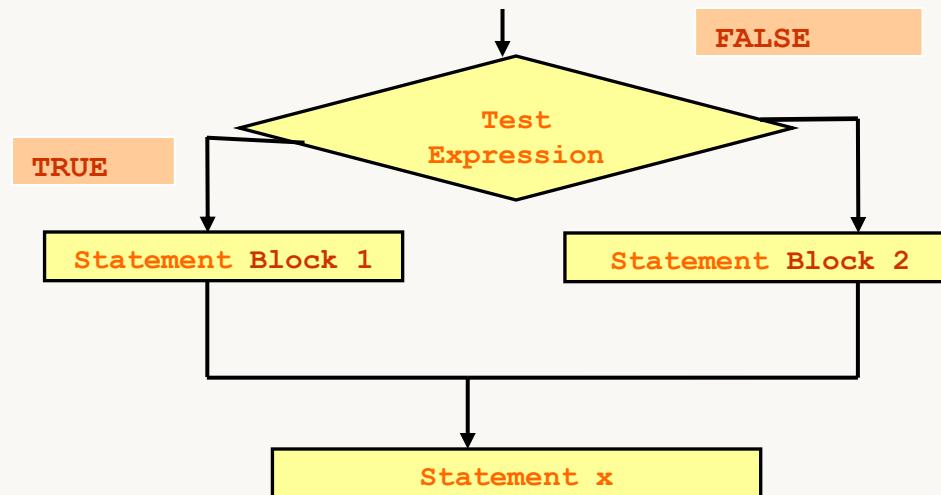


# IF ELSE STATEMENT

- In the if-else construct, first the test expression is evaluated. If the expression is true, statement block 1 is executed and statement block 2 is skipped. Otherwise, if the expression is false, statement block 2 is executed and statement block 1 is ignored. In any case after the statement block 1 or 2 gets executed the control will pass to statement x. Therefore, statement x is executed in every case.

SYNTAX OF IF STATEMENT

```
if (test expression)
{
    statement_block 1;
}
else
{
    statement_block 2;
}
statement x;
```



# PROGRAMS TO DEMONSTRATE THE USE OF IF AND IF-ELSE STATEMENTS

**Example 3.1** To check and increment a number if it is positive

```
#include <iostream.h>
int main()
{    int x = 10;                      // Initialize the value of x
    if (x > 0)                      // Test the value of x
        x++;                         // Increment the value of x if it is > 0
    cout<<"\n x = "<<x;           // Print the value of x
}
```

## OUTPUT

```
x = 11
```

**Program 3.4 Write a program to find whether the given number is even or odd**

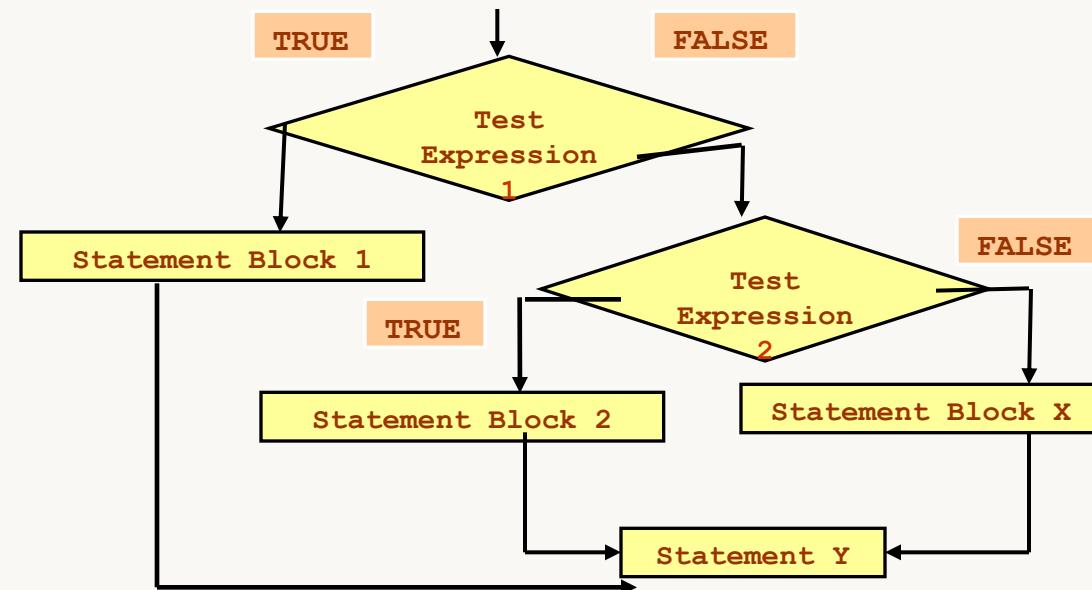
```
#include<iostream.h>
main()
{    int num;
    cout<<"\n Enter any number : ";
    cin>>num;
    if(num%2 == 0)
        cout<<"\n"<<num<<" is an even number";
    else
        cout<<"\n"<<num<<" is an odd number";
}
```

# IF ELSE IF STATEMENT

- C language supports if else if statements to test additional conditions apart from the initial test expression. The if-else-if construct works in the same way as a normal if

## SYNTAX OF IF-ELSE STATEMENT

```
if ( test expression 1)
{
    statement block 1;
}
else if ( test expression 2)
{
    statement block 2;
}
.....
else if (test expression N)
{
    statement block N;
}
else
{
    Statement Block X;
}
Statement Y;
```



# SWITCH CASE

- A switch case statement is a multi-way decision statement. Switch statements are used:  
When there is only one variable to evaluate in the expression  
When many conditions are being tested for
- Switch case statement advantages include:  
Easy to debug, read, understand and maintain  
Execute faster than its equivalent if-else construct

```
switch(grade)
{
    case 'A':
        cout<<"\n Excellent";
        break;
    case 'B':
        cout<<"\n Good";
        break;
    case 'C':
        cout<<"\n Fair";
        break;
    default:
        cout<<"\n Invalid Grade";
        break;
}
```

### Example 3.2 To test whether a number entered by the user is negative, positive or equal to zero

**Programming Tip:** Keep the logical expressions simple and short. For this, you may use nested if statements.

```
#include<iostream.h>
main()
{
    int num;
    cout<<"\n Enter any number : ";
    cin>>num;
    if(num == 0)
        cout<<"\n The value is equal to zero";
    else if(num > 0)
        cout<<"\n The number is positive";
    else
        cout<<"\n The number is negative";
}
```

#### OUTPUT

Enter any number: 0  
The number is zero

### Example 3.4 Usage of switch case statement

```
#include<iostream.h>
void main()
{
char grade = 'C';
switch(grade)
{   case 'O':      cout<<"\n Outstanding";
                break;
    case 'A':      cout<<"\n Excellent";
                break;
    case 'B':      cout<<"\n Good";
                break;
    case 'C':      cout<<"\n Fair";
                break;
    case 'F':      cout<<"\n Fail";
                break;
    default:       cout<<"\n Invalid Grade";
                break;
}
}
```

#### OUTPUT

Fair

# ITERATIVE STATEMENTS

- Iterative statements are used to repeat the execution of a list of statements, depending on the value of an integer expression. In this section, we will discuss all these statements.

While loop

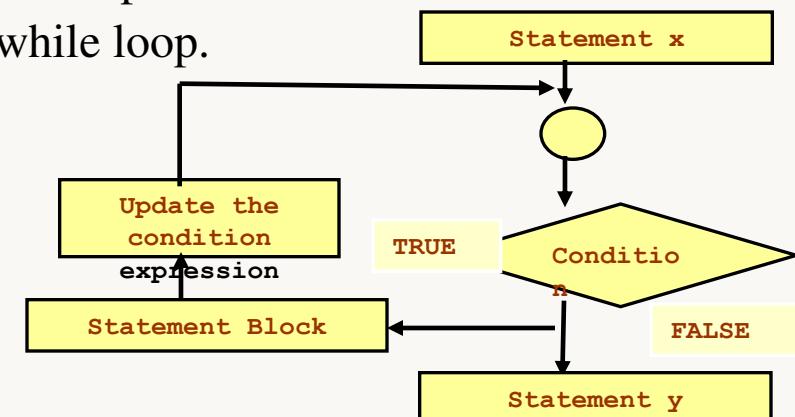
Do-while loop

For loop

## WHILE LOOP

- The while loop is used to repeat one or more statements while a particular condition is true.
- In the while loop, the condition is tested before any of the statements in the statement block is executed.
- If the condition is true, only then the statements will be executed otherwise the control will jump to the immediate statement outside the while loop block.
- We must constantly update the condition of the while loop.

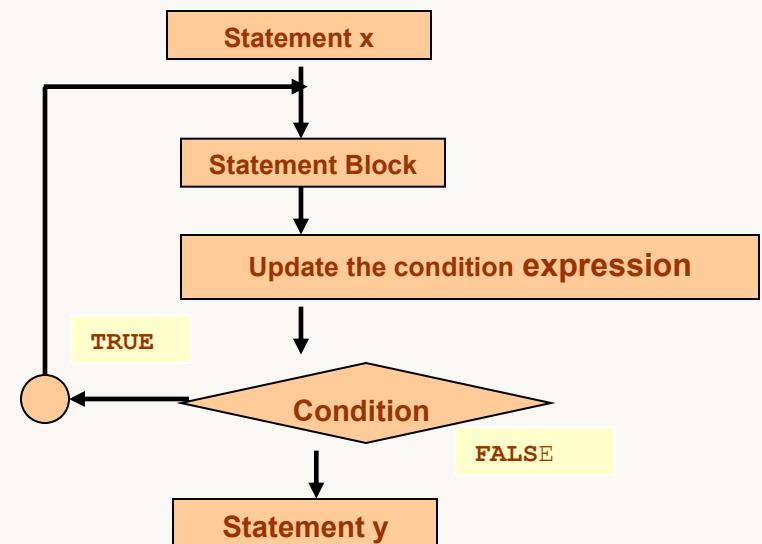
```
while (condition)
{
    statement_block;
}
statement x;
```



# DO WHILE LOOP

- The do-while loop is similar to the while loop. The only difference is that in a do-while loop, the test condition is tested at the end of the loop.
- The body of the loop gets executed at least one time (even if the condition is false).
- The do while loop continues to execute whilst a condition is true. There is no choice whether to execute the loop or not. Hence, entry in the loop is automatic there is only a choice to continue it further or not.
- The major disadvantage of using a do while loop is that it always executes at least once, so even if the user enters some invalid data, the loop will execute.
- Do-while loops are widely used to print a list of options for a menu driven program.

```
Statement x;  
do  
{  
    statement_block;  
} while (condition);  
statement y;
```



# Program to demonstrate the use of while loop and do while loop

## Example 3.6 To print first 10 numbers using a while loop

```
#include<iostream.h>
int main()
{   int i = 0; // initialize loop variable
    while(i <= 10) // test the condition
    {
        // execute the loop statements
        cout<<i;
        i = i + 1; // condition updated
    }
}
```

### OUTPUT

```
0 1 2 3 4 5 6 7 8 9 10
```

#### Programming

**Tip:** Do not forget to place a semicolon at the end of the do-while statement.

## Program 3.23 Write a program to calculate the average of first n numbers.

```
#include<iostream.h>
int main()
{ int n, i = 0, sum = 0;
    float avg = 0.0;
    cout<<"\n Enter the value of n : ";
    cin>>n;
    do
    {
        sum = sum + i;
        i = i + 1;
    } while(i <= n);
    avg = sum/n;
    cout<<"\n The sum of first n numbers = "<<sum;
    cin>>"\n The average of first "<<n<<" numbers = "<<avg;
}
```

### OUTPUT

```
Enter the value of n : 18
The sum of first n numbers = 171
The average of first %d numbers = 9.00
```

# FOR LOOP

- For loop is used to repeat a task until a particular condition is true.
- The syntax of a for loop

```
for (initialization; condition; increment/decrement/update)
```

```
{  
    statement block;  
}  
Statement Y;
```

- When a for loop is used, the loop variable is initialized only once.
- With every iteration of the loop, the value of the loop variable is updated and the condition is checked. If the condition is true, the statement block of the loop is executed else, the statements comprising the statement block of the for loop are skipped and the control jumps to the immediate statement following the for loop body.
- Updating the loop variable may include incrementing the loop variable, decrementing the loop variable or setting it to some other value like,  $i += 2$ , where  $i$  is the loop variable.

# PROGRAM FOR FOR-LOOP

**Example 3.8** To print the first n numbers using a for loop

```
#include<iostream.h>
int main()
{   int i, n;
    cout<<"\n Enter the value of n :";
    cin>>n;
    for(i = 0;i <= n;i++)
        cout<<"\t "<<i;
}
```

## OUTPUT

Enter the value of n : 10

1      2      3      4      5      6      7      8      9      10

## **BREAK STATEMENT**

- The break statement is used to terminate the execution of the nearest enclosing loop in which it appears.
- When compiler encounters a break statement, the control passes to the statement that follows the loop in which the break statement appears. Its syntax is quite simple, just type keyword break followed with a semi-colon.  
`break;`
- In switch statement if the break statement is missing then every case from the matched case label to the end of the switch, including the default, is executed.

## **CONTINUE STATEMENT**

- The continue statement can only appear in the body of a loop.
- When the compiler encounters a continue statement then the rest of the statements in the loop are skipped and the control is unconditionally transferred to the loop-continuation portion of the nearest enclosing loop. Its syntax is quite simple, just type keyword continue followed with a semi-colon.  
`continue;`
- If placed within a for loop, the continue statement causes a branch to the code that updates the loop variable. For example,  
`int i;  
for(i=0; i<= 10; i++)  
{       if (i==5)  
            continue;  
    cout<<i;  
}`

# GOTO STATEMENT

- The **goto** statement is used to transfer control to a specified label.
- Here label is an identifier that specifies the place where the branch is to be made. Label can be any valid variable name that is followed by a colon (:).
- Note that label can be placed anywhere in the program either before or after the goto statement. Whenever the goto statement is encountered the control is immediately transferred to the statements following the label.
- Goto statement breaks the normal sequential execution of the program.
- If the label is placed after the goto statement then it is called a forward jump and in case it is located before the goto statement, it is said to be a backward jump.

```
int num, sum=0;
read:      // label for go to statement
    cout<<"\n Enter the number. Enter 999 to end : ";
    cin>>num;
    if (num != 999)
    {
        if(num < 0)
            goto read; // jump to label- read
        sum += num;
        goto read;      // jump to label- read
    }
    cout<<"\n Sum of the numbers entered by the user is = "<<sum;
```



OXFORD  
HIGHER EDUCATION

# Object Oriented Programming with C++



Reema Thareja

OXFORD  
HIGHER EDUCATION

# Object Oriented Programming with C++

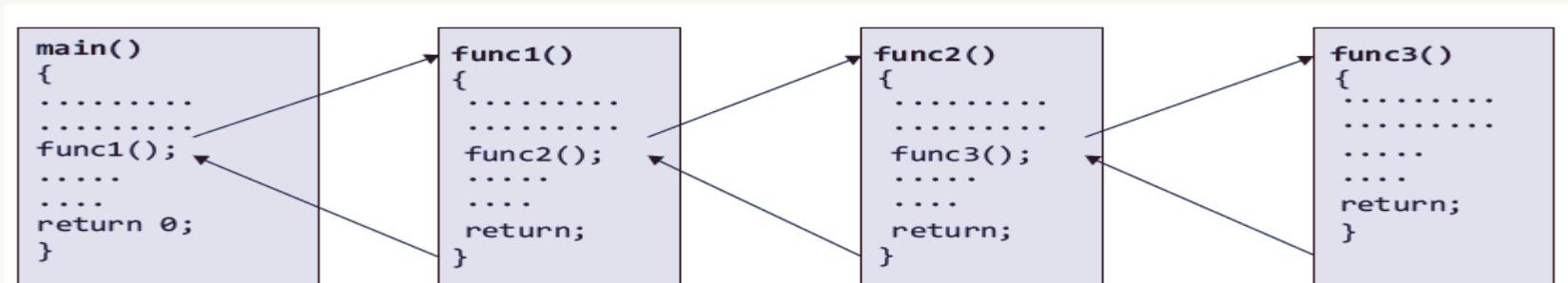
Reema Thareja

# Chapter Four

## Functions

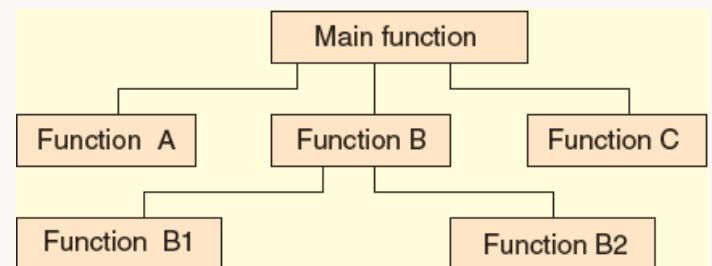
# INTRODUCTION

- C++ enables programmers to break up a program into segments commonly known as **functions**.
- Each function can be written more or less independently of the others. Every function in the program is supposed to perform a well-defined task.
- It is not necessary that the main() can call only one function. It can call as many functions as it wants and as many times as it wants.
- Any function can call any other function



# NEED FOR FUNCTIONS

- Dividing the program into separate well-defined functions facilitates each function to be written and tested separately.
- Understanding, coding, and testing multiple separate functions are far easier than doing the same for one huge function.
- All libraries in C++ contain a set of functions that programmers are free to use in their programs. These functions have been prewritten and pretested.
- When a big program is broken into comparatively smaller functions, different programmers working on that project can divide the workload by writing different functions.
- Like C++ libraries, programmers can also make their functions and use them from different points in the main program or any other program that needs its functionalities.



# USING FUNCTIONS

- A function,  $f$  , that uses another function  $g$ , is known as the **calling function** and  $g$  is known as the **called function**.
- The inputs that the function takes are known as **arguments** or **parameters**.
- When a called function returns some result back to the calling function, it is said to return that result.
- The calling function may or may not pass parameters to the called function.
- If the called function accepts the arguments, the calling function will pass parameters, otherwise, it will not do so.
- Function declaration is a declaration statement that identifies a function with its name, a list of arguments that it accepts, and the type of data it returns.

# USING FUNCTIONS

- Function definition consists of a function header that identifies the function, followed by the body of the function containing the executable code for that function.

# FUNCTION DECLARATION OR FUNCTION PROTOTYPE

```
return_data_type function_name(data_type variable1, data_type variable2,...);
```

- `function_name` is a valid name for the function.
- `return_data_type` specifies the data type of the value that will be returned to the calling function as a result of the processing performed by the called function.
- `data_type variable1, data_type variable2, ...` is a list of variables of specified data types.
- After the declaration of every function, there is a **semicolon**.
- The name of the function is global.

## Function declaration

```
char convert_to_uppercase (char ch);
```

Return Data Type

```
float avg (int a, int b);
```

Function Name

```
int find_largest (int a, int b, int c);
```

Data Type of Variable

```
double multiply(float a, float b);
```

Variable I

```
void swap (int a, int b);
```

## Use of the function

Converts a character to upper case. The function receives a character as an argument, converts it into upper case, and returns the converted character back to the calling program.

Calculates average of two numbers a and b received as arguments. The function returns a floating point value.

Finds the largest of three numbers a, b, and c received as arguments. An integer value which is the largest number of the three numbers is returned to the calling function.

Multiplies two floating point numbers a and b that are received as arguments and returns a double value.

Swaps or interchanges the value of integer variables a and b received as arguments. The function returns no value; therefore, the data type is void.

```
void print(void);
```

The function is used to print information on screen. The function neither accepts any value as argument nor returns any value. Therefore, the return type is void and the argument list contains void.

# FUNCTION DEFINITION

The syntax of a function definition can be given as

```
return_data_type function_name(data_type variable1, data_type variable2,...)
{
    .....
    statements
    .....
    return( variable);
}
```

While `return_data_type function_name(data_type variable1, data_type variable2,...)` are known as the function header, the rest of the portion comprising program statements within `{ }` is the function body which contains the code to perform the specific task.

The function header is same as that of function declaration. The only difference between the two terms is that a function header is not followed by a semicolon. The list of variables in the function header is known as the formal parameter list. The parameter list may have zero or more parameters of any data type. The function body contains instructions to perform the desired computation in a function.

# FUNCTION CALL

A function call statement has the following syntax:

```
function_name(variable1, variable2, ...);
```

- Function name and the number and type of arguments in the function call must be the same as that given in the function declaration and function header of the function definition.
- If, by mistake, the parameters passed to a function are more than specified to accept, then the extra arguments will be discarded.
- If, by mistake, the parameters passed to a function are less than specified to accept, then the unmatched argument will be initialized to some garbage value (i.e. arbitrary meaningless value).
- Names, and not the types, of variables in function declaration, function call, and header of function definition may vary.
- Arguments may be passed in the form of expressions to the called function.

### Example 4.1 To add two integers using functions

```
#include<iostream.h>
int sum(int a, int b);    // FUNCTION DECLARATION
int main()
{   int num1, num2, total = 0;
    cout<<"\n Enter two numbers : ";
    cin>>num1>>num2;
    total = sum(num1, num2);      // FUNCTION CALL
    cout<<"\n Total = "<<total;
}
// FUNCTION DEFINITION
int sum ( int a, int b)      // FUNCTION HEADER
{
    // FUNCTION BODY
    int result;
    result = a + b;
    return result;
}
```

#### OUTPUT

```
Enter the two numbers : 20    30
Total = 50
```

# RETURN STATEMENT

- The return statement is used to terminate the execution of a function and return control to the calling function.
- When the return statement is encountered, the program execution resumes in the calling function at the point immediately following the function call.
- A return statement may or may not return a value to the calling function.

```
return <expression>;
```

## Using more than one return statement

```
#include<iostream.h>
int check_relation(int a, int b);           // FUNCTION DECLARATION
main()
{   int a = 3, b = 5, res;
    res = check_relation(a, b);             // FUNCTION CALL
    if(res == 0)                          // Test the returned value
        cout<<"\n EQUAL";
    if(res == 1)
        cout<<"\n a is greater than b";
    if(res == -1)
        cout<<"\n a is less than b";
}
int check_relation(int a, int b)           // FUNCTION DEFINITION
{   if(a == b)
    return 0;
if(a > b)
    return 1;
else if (a < b)
    return -1;
}
```

### OUTPUT

a is less than b

# PASSING PARAMETERS TO THE FUNCTION

- When a function is called, the calling function may have to pass some values to the called function.
- There are two ways in which arguments or parameters can be passed to the called function.
- They include the following:
  - **Call-by-value**, in which values of the variables are passed by the calling function to the called function. The programs that we have written so far call the function using call-by-value method of passing parameters.
  - **Call-by-address**, in which the address of the variables is passed by the calling function to the called function.
  - **Call-by-reference** , in which a reference of the variable is passed by the calling function to the called function.

# Call-by-Value

- In this method, the called function creates new variables to store the value of the arguments passed to it.
- Therefore, the called function uses a copy of the actual arguments to perform its intended task.
- If the called function is supposed to modify the value of the parameters passed to it, then the change will be reflected only in the called function.
- In the calling function, no change will be made to the value of the variables.

```
#include<iostream.h>
void add(int n); // FUNCTION DECLARATION
int main()
{   int num = 2;
    cout<<"\n The value of num before calling the function = "<<num;
    add(num); // FUNCTION CALL
    cout<<"\n The value of num after calling the function = "<<num;
}
void add(int n) // FUNCTION DEFINITION
{   n = n + 10;
    cout<<"\n The value of num in the called function = "<<n;
}
```

## OUTPUT

The value of num before calling the function = 2

---

The value of num in the called function = 20

The value of num after calling the function = 2

# Call-by-Address

- The call-by-pointer or call-by-address method of passing arguments to a function copies the address of an argument into the formal parameter.
- The function then uses the address to access the actual argument. This means that any changes made to the value stored at a particular address will be reflected in the calling function also.
- The called function uses pointers to access the data. In other words, the **dereferencing operator** is used to access the variables in the called function.

```
#include<iostream.h>
void add( int *n);
int main()
{   int num = 2;
    cout<<"\n The value of num before calling the function = "<<num;
    add(&num);
    cout<<"\n The value of num after calling the function = "<<num;
}
void add( int *n)
{   *n = *n + 10;
    cout<<"\n The value of num in the called function = "<<n;
}
```

### OUTPUT

The value of num before calling the function = 2

The value of num in the called function = 12

The value of num after calling the function = 12

# Call-by-Reference

- When the calling function passes arguments to the called function using call-by-value method, the only way to return the modified value of the argument to the caller is by using the return statement explicitly.
- A better option when a function can modify the value of the argument is to pass arguments using call-by-reference technique.
- In call-by-reference, we declare the function parameters as references rather than normal variables.
- When this is done, any changes made by the function to the arguments it received are visible by the calling program.

```
#include<iostream.h>
void add( int &n);
int main()
{   int num = 2;
    cout<<"\n The value of num before calling the function = "<<num;
    add(&num);
    cout<<"\n The value of num after calling the function = "<<num;
}
void add(int &n)
{   n = n + 10;
    cout<<"\n The value of num in the called function = "<<n;
}
```

### OUTPUT

```
The value of num before calling the function = 2
The value of num in the called function = 12
The value of num after calling the function = 12
```

# DEFAULT ARGUMENTS

- When a function is specified with default arguments, the function can be called with **missing arguments**.
- When a function is called with a missing argument, the function assigns a **default value** to the parameter.
- This default value is specified by the programmer in the function declaration statement.
- While specifying the default values during function declaration, the programmer must keep the following in mind.
- Only trailing arguments can have default values; therefore, specify them from right to left.
- No argument specified in the middle can have default values.

int my_func(int a, int b, int c = 10);	//Correct
int my_func(int a, int b = 5, int c = 10);	//Correct
int my_func(int a = 1, int b = 5, int c = 10);	//Correct
int my_func(int a, int b = 5, int c = 10);	//Wrong

**Programming Tip:** Arguments explicitly specified in function call override the default values given during function declaration.

### Program 4.9 Write a program to calculate the volume of a cuboid using default arguments.

```
#include<iostream.h>
int Volume( int length, int width = 3, int height = 4 );
int main()
{ cout<<"\n Volume = "<<Volume(4, 6, 2);
  cout<<"\n\n Volume = "<<Volume(4, 6);
  cout<<"\n\n Volume = "<<Volume(4);
}
int Volume(int length, int width, int height)
{   cout<<"\n Length = "<<length<<" Width = "<<width " and Height
    = "<<height;
  return length * width * height;
}
```

### OUTPUT

```
Length = 4 Width = 6 and Height = 2
Volume = 48
Length = 4 Width = 6 and Height = 4
Volume = 96
Length = 4 Width = 3 and Height = 4
Volume = 48
```

# RETURN BY REFERENCE

- This allows a function to be used on the left side of an assignment statement.

The following are the key points to remember while returning by reference:

- The function should not return a local variable by reference as it will go out of scope immediately as soon as the function ends. This means the following code will generate a compiler error.

```
int &my_func(int num)
{   int x = num*2;
    return x;      //ERROR, x is a local variable
}
```

- The function may return a reference to a static variable. Therefore, the code given here is permissible in C++.

```
int &my_func()
{   static int x;
    return x;
}
```

- Variables passed by reference can be returned by reference.
- Return by reference is extensively used to return structure variables and objects of classes.

```
#include<iostream.h>
int &greater(int &x, int &y)
{  if(x>y)
    return x;
else
    return y;
}
main()
{  int num1, num2, large;
cout<<"\n Enter two numbers : ";
cin>>num1>>num2;
cout<<"\n Two numbers are : "<<num1<<" "<<num2;
large = greater(num1, num2);
cout<<"\n Large = "<<large;
greater(num1,num2) = -1;
cout<<"\n Two numbers are : "<<num1<<" "<<num2;
}
```

### OUTPUT

```
Enter two numbers : 5 2
Two numbers are : 5 2
Large = 5
Two numbers are : -1 2
```

# PASSING CONSTANTS AS ARGUMENTS

- When parameters are passed by reference parameter, the called function may intentionally or inadvertently modify the actual parameters.
- However, at times, the programmer may strictly want the actual parameters not to be modified by the called function.
- In such cases, a **constant parameter** must be passed.
- Therefore, using the const keyword allows programmers to achieve performance benefits while ensuring that the actual parameter is not modified.

```
#include<iostream.h>
void add_2(int const &x)
{   x = x + 2;    // ERROR, cannot modify a constant
    cout<<"\n The numbers is now : "<<x;
}
main()
{   int num1;
    cout<<"\n Enter a number : ";
    cin>>num1;
    add_2(num1);
}
```

### OUTPUT

Error

# VARIABLES SCOPE

- In C++, all constants and variables have a **defined scope**.
- By scope, we mean the accessibility and visibility of variables at different points in the program.
- A variable or a constant in C++ has four types of scope—block, function, file, and program scope.

## Block Scope

- A statement block is a group of statements enclosed within an opening and closing curly brackets ({}).
- If a variable is declared within a statement block, as soon as the control exits that block, the variable will cease to exist.
- Such a variable also known as a local variable is said to have a block scope.

### Example 4.10 Code which illustrates the block scope concept

```
#include <iostream.h>
int main()
{   int x = 10;
    int i=0;
    cout<< "\n The value of x outside the while loop is "<<x;
    while (i<3)
    {
        int x = i;
        cout<<"\n The value of x inside the while loop is "<<x;
        i++;
    }
    cout<<"\n The value of x outside the while loop is "<< x;
}
```

#### OUTPUT

```
The value of x outside the while loop is 10
The value of x inside the while loop is 0
The value of x inside the while loop is 1
The value of x inside the while loop is 2
The value of x outside the while loop is 10
```

# VARIABLES SCOPE

## Function Scope

- Function scope indicates that a variable is active and visible from the beginning to the end of a function.
- In C++, only the `goto` label has function scope.

## Scope of the Program

- If you want the functions to be able to access some variables which are not passed to them as arguments, declare those variables outside any function blocks. Such variables are commonly known as **global variables**.
- Global variables are those variables that can be accessed from any point in the program.

# VARIABLES SCOPE

## File Scope

- When a global variable is accessible until the end of the file, the variable is said to have file scope.

- To allow a variable to have file scope, declare that variable with the static keyword before specifying its data type as follows:

```
static int x = 10;
```

- A global static variable can be used anywhere from the file in which it is declared but it is not accessible by any other files. Such variables are useful when the programmer writes his own header files.

### **Example 4.11** Code to illustrate the concept of program scope

```
#include<iostream.h>
int x = 10;
void print();
int main()
{   cout<<"\n The value of x in the main() = "<<x;
    int x = 2;
    cout<<"\n The value of local variable x in the main() = "<<x;
    print();
}
void print()
{   cout<<"\n The value of x in the print() = "<<x;
}
```

#### **OUTPUT**

```
The value of x in the main() = 10
The value of local variable x in the main() = 2
The value of x in the print() = 10
```

# STORAGE CLASSES

- The storage class of a variable defines the scope or visibility and life time of variables and/or functions declared within a C++ program.

<storage\_classSpecifier> <data type> <variable name>

Feature	Storage class			
	Auto	Extern	Register	Static
Accessibility	Accessible within the function or block in which it is declared	Accessible within all program files that are a part of the program	Accessible within the function or block in which it is declared	Local Accessible within the function or block in which it is declared  Global: Accessible within the program in which it is declared
Storage	Main memory	Main memory	CPU register	Main memory
Existence	Exists when the function or block in which it is declared is entered. Ceases to exist when the control returns from the function or the block in which it was declared	Exists throughout the execution of the program	Exists when the function or block in which it is declared is entered. Ceases to exist when the control returns from the function or the block in which it was declared	Local Retains value between function calls or block entries  Global Preserves value in program files
Default value	Garbage	Zero	Garbage	Zero

# INLINE FUNCTIONS

- The major difference between an ordinary function and an **inline** function is that when an inline function is called, the compiler places a copy of its code at each point of call.
- As in case of an ordinary function, the compiler does not have to jump to the called function.
- This saves the function call overhead and results in faster execution of the code.
- We can make a function inline by taking care of two aspects as follows:
  - First, write the keyword **inline** before the function name.
  - Second, define that function before any calls are made to it.

# INLINE FUNCTIONS

- However, the programmer must not forget that keyword `inline` just makes a request to the compiler to make the function inline (and place its code at each point of call).
- The compiler may ignore the request if the function has too many lines.

## Example 4.16 To find the greater number using an inline function

**Programming Tip:** main() cannot be used as an inline function.

```
#include<iostream.h>
inline int greater(int x, int y) // Function definition
{ return (x > y)? x : y; // Function returning greater of the two nos
}
int main( )
{
    int num1, num2;
    cout << "\n Enter two numbers : ";
    cin>>num1>>num2;
    cout<<"\n The greater number is : "<<greater(num1,num2);      // Function call
}
```

### OUTPUT

```
Enter two numbers : 5 9
The greater number is : 9
```

# ADVANTAGES

- An inline function generates faster code as it saves the time required to execute function calls.
- Small inline functions (three lines or less) create less code than the equivalent function call as the compiler does not have to generate code to handle function arguments and a return value.
- Inline functions are subject to code optimizations that are usually not available to normal functions as the compiler does not perform inter-procedural optimizations.
- Inline function avoids function call overhead. As a result, we need to save variables and other program parameters on the **system stack**.

# ADVANTAGES

- Since there are no function calls, the overhead of returning from a function is also avoided.
- It allows the compiler to apply intra-procedural optimization

# DISADVANTAGES

- Size of the program increases.
- Large programs may take longer time to be executed.
- At times, the program may not fit in the cache memory.
- There may be a problem in making efficient use of CPU registers if the inline function has many register variables.
- If the code of the inline function is modified, the entire program needs to be re-compiled.
- Inline functions should not be used for designing embedded systems due to memory size constraints.

# COMPARISON OF INLINE FUNCTIONS WITH MACROS

- Macro invocations skip the job of type checking; this is a must-to-do work in function calls.
- Macros cannot return a value while a function can return a value.
- Macros use mere textual substitution which can give unintended results due to inaccurate reevaluation of arguments and order of operations.
- Debugging of compiler errors in case of macros is more difficult than debugging functions.
- All constructs cannot be expressed using macros; however, with functions, they can be expressed with ease.
- Macros have a slightly difficult syntax while the syntax of writing function is similar to that of a normal function.

# FUNCTION OVERLOADING

- Function overloading, also known as method overloading, is a feature in C++ that allows creation of several methods with the same name but with different parameters.
- For example, print(), print(int), and print("Hello") are overloaded methods.
- While calling print() , no arguments are passed to the function; however, when calling print(int) and print("Hello") , an integer and a string arguments are passed to the called function.
- Function overloading is a type of **polymorphism**.

# FUNCTION OVERLOADING

- Basically, there are two types of polymorphism:
- **Compile time (or static) polymorphism** and **run-time (or dynamic)** polymorphism.
- Function overloading falls in the category of static polymorphism which calls a function using the best match technique or overload resolution.

# MATCHING FUNCTION CALLS WITH OVERLOADED FUNCTIONS

When an overloaded function is called, one of the following cases occurs:

- **Case 1:** A direct match is found, and there is no confusion in calling the appropriate overloaded function.
- **Case 2:** If a match is not found, a linker error will be generated. However, if a direct match is not found, then, at first, the compiler will try to find a match through the type conversion or type casting.
- **Case 3:** If an ambiguous match is found, that is, when the arguments match more than one overloaded function, a compiler error will be generated. This usually happens because all standard conversions are treated equal.

```
void print(int); // Function declaration  
print('R'); // Function call
```

### **Example 4.19** To demonstrate the ambiguity in function call

```
#include<iostream.h>
void print(int n){ cout<<n;}
void print(char c){ cout<<c; }
void print(float f){ cout<<f; }
main()
{    print(5);    //Function call
     print (98.7);
     print('R');
}
```

#### **OUTPUT**

Error

### Example 4.20 To compute the volume of different shapes using function overloading concept

```
#include<iostream.h>
int volume(int side)
{   return side*side*side; // cube
}
float volume(float radius, float height)
{   return 3.14+radius*radius*height; //cylinder
}
long int volume(int length, int breadth, int height)
{   return length*breadth*height; //cuboid
}
main()
{   int s,l,b,height;
    float r, h;
    cout<<"\n Enter the side of the cube : ";
    cin>>s;
    cout<<"\n Volume of cube with side "<<s<<" = "<<volume(s);
    cout<<"\n \n Enter the radius and height of the cylinder : ";
    cin>>r>>h;
    cout<<"\n Volume of cylinder with radius "<<r<<" and height "<<h<<" = "<<volume(r,h);
    cout<<"\n Enter the length, breadth and height of the cuboid : ";
    cin>>l>>b>>height;
    cout<<"\n Volume of cuboid with length "<<l<<" breadth "<<b<<" and height
    "<<height<<" = "<<volume(l,b,height);
}
```

#### OUTPUT

```
Enter the side of the cube : 3
Volume of cube with side 3 = 27
Enter the radius and height of the cylinder : 3 4
Volume of cylinder with radius 3 and height 4 = 39.13
Enter the length, breadth and height of the cuboid : 2 3 4";
Volume of cuboid with length 2 breadth 3 and height 4 = 24
```

# Functions that Cannot be Overloaded

- Functions that differ only in the return type. For example, the program given here will give compile time error.

```
#include<iostream.h>
int my_func()      { return 1; }
char my_func()     { return 'E'; }
main()
{ int num = my_func();
  char c = my_func();
}
```

- Parameter declarations that differ only in a pointer \* versus an array [] are equivalent. A program with the following declarations will give a compilation error as both declarations are equivalent. The reason why they are equivalent will be clear in the chapter on Pointers.

```
int my_func(int *ptr);
int my_func(int ptr[]);
```

- If parameters in two functions differ only in the presence or absence of const and/or volatile, then those functions are considered to be equivalent. For example, the program having following declarations will not compile.

```
int my_func(int n);
int my_func(const int n);
```

- Using typedef does not introduce a new type, therefore, the following two function declarations are equivalent and this cannot be overloaded. (Use of typedef will be studied in chapter on Structures).

```
typedef int integer;
int my_func(int n);
int my_func(integer n);
```

- Function declarations that differ only in their default arguments are equivalent. Therefore, the program having function declarations given below will not compile.

```
int my_func(int n);
int my_func(int n = 10);
```

- Function declarations that differ only in a reference parameter and a normal parameter cannot be overloaded. Therefore, the program having function declarations given here will not compile.

```
int my_func(int n);
int my_func(int &n);
```

# RECURSIVE FUNCTIONS

- A recursive function is defined as a function that calls itself to solve a smaller version of its task until a final call is made which does not require a call to itself.
- Every recursive solution has two major cases which are given as follows:
  - **Base case**, in which the problem is simple enough to be solved directly without making any further calls to the same function.
  - **Recursive case**, in which first the problem at hand is divided into simpler sub parts.
- Second, the function calls itself but with sub parts of the problem obtained in the first step.
- Third, the result is obtained by combining the solutions of simpler sub-parts.

Problem	Solution
5!	5 X 4 X 3 X 2 X 1!
= 5 X 4!	= 5 X 4 X 3 X 2 X 1
= 5 X 4 X 3!	= 5 X 4 X 3 X 2
= 5 X 4 X 3 X 2!	= 5 X 4 X 6
= 5 X 4 X 3 X 2 X 1!	= 5 X 24
	= 120

## Example 4.21 To calculate the factorial of a number recursively

### Programming

**Tip:** Every recursive function must have at least one base case. Otherwise, the recursive function will generate an infinite sequence of calls, thereby resulting in an error condition known as an infinite stack.

```
#include<iostream.h>
int Fact(int); // FUNCTION DECLARATION
main()
{ int num;
    cout<<"\n Enter the number : ";
    cin>>num;
    cout<<"\n Factorial of "<<num<<" = "<<Fact(num); // FUNCTION CALL
}
int Fact( int n) // FUNCTION DEFINITION
{ if(n==1)
    return 1;
    return (n * Fact(n-1)); // FUNCTION CALL
}
```

### OUTPUT

```
Enter the number : 5
Factorial of 5 = 120
```

# ADVANTAGES

- Recursive solutions often tend to be shorter and simpler than non-recursive ones.
- Code is clearer and easier to use.
- Recursion represents like the original formula to solve a problem.
- Follows a divide and conquer technique to solve problems.
- In some (limited) instances, recursion may be more efficient.

# DISADVANTAGES

- For some programmers and readers, recursion is a difficult concept.
- Recursion is implemented using system stack.
- If the stack space on the system is limited, recursion to a deeper level will be difficult to implement.
- Aborting a recursive process in midstream is slow and sometimes nasty.
- Using a recursive function takes more memory and time to execute as compared to its non-recursive counter part.
- It is difficult to find bugs, particularly, when using global variables

# FUNCTIONS WITH VARIABLE NUMBER OF ARGUMENTS

- In case we need a function that should accept a variable number of arguments, we have to design a function that accepts the data-type and/or number of arguments at the run-time while execution and not during compilation.
- Let us take a sample program that accepts any number of values and then return the average.
- To write such a program, we must include the cstdarg header file to use the following macros:
  - `va_list` that stores the list of arguments.
  - `va_start` to initialize the list.
  - It is a macro from which we can begin reading arguments from it. It accepts two arguments— `va_list` and a last named

# FUNCTIONS WITH VARIABLE NUMBER OF ARGUMENTS

argument. After the last named argument, the number of arguments read is stored.

- `va_arg` returns the next argument in the list. It helps in accessing the individual arguments in the list. For this, you must just need to specify the type of argument to retrieve.
- `va_end` is used to clean up the variable argument list once we are done with it.
- A function that is supposed to accept variable number of arguments must be declared in a special way.
- Instead of writing the last argument, you must put an ellipsis (like ‘...’). Therefore, `int my_func( int x, ... );` means that `my_func` accepts any number of arguments.

### Example 4.22 Using variable number of arguments

```
#include<stdarg.h>
#include<iostream.h>
float avg(int argc, ...)
{
    va_list args;
    int i, sum = 0;
    va_start(args, argc);
    for(i = 0;i < argc;i++)
        sum += va_arg(args, int); // Adds next value in args to sum
    va_end(args);
    return( (float) sum/argc);
}

main()
{
    float result;
    result = avg(7,8,9,10,4,5,10,10,10,9,7,8,9,10,10);
    cout.precision(2);
    cout<<"\n AVERAGE = "<<result;
}
```

#### OUTPUT

AVERAGE = 8.00

**FREE**  
ONLINE RESOURCES  
For Teachers and Students

OXFORD  
HIGHER EDUCATION

# Object Oriented Programming with C++



Reema Thareja

OXFORD  
HIGHER EDUCATION

# Object Oriented Programming with C++

Reema Thareja

# Chapter Five

## Arrays

# INTRODUCTION

- An array is a collection of similar data elements.
- These data elements have the same data type.
- The elements of the array are stored in consecutive memory locations and are referenced by an index (also known as the subscript).
- Declaring an array means specifying three things:
  - The data type- what kind of values it can store ex, int, char, float
  - Name- to identify the array
  - The size- the maximum number of values that the array can hold
- Arrays are declared using the following syntax.

type name[size];

1 <sup>st</sup> element	2 <sup>nd</sup> element	3 <sup>rd</sup> element	4 <sup>th</sup> element	5 <sup>th</sup> element	6 <sup>th</sup> element	7 <sup>th</sup> element	8 <sup>th</sup> element	9 <sup>th</sup> element	10 <sup>th</sup> element
marks[0]	marks[1]	marks[2]	marks[3]	marks[4]	marks[5]	marks[6]	marks[7]	marks[8]	marks[9]

# ACCESSING ELEMENTS OF THE ARRAY

To access all the elements of the array, you must use a loop. That is, we can access all the elements of the array by varying the value of the subscript into the array. But note that the subscript must be an integral value or an expression that evaluates to an integral value.

## CALCULATING THE ADDRESS OF ARRAY ELEMENTS

```
int i, marks[10];
for(i=0;i<10;i++)
    marks[i] = -1;
```

Address of data element,  $A[k] = BA(A) + w(k - \text{lower\_bound})$

Here, A is the array

k is the index of the element of which we have to calculate the address

BA is the base address of the array A.

w is the word size of one element in memory, for example, size of int is 2.

99	67	78	56	88	90	34	85
Marks[0] marks[7] 1000	marks[1] 1002	marks[2] 1004	marks[3] 1006	marks[4] 1008	marks[5] 1010	marks[6] 1012	marks[7] 1014

$$\begin{aligned}\text{Marks}[4] &= 1000 + 2(4 - 0) \\ &= 1000 + 2(4) = 1008\end{aligned}$$

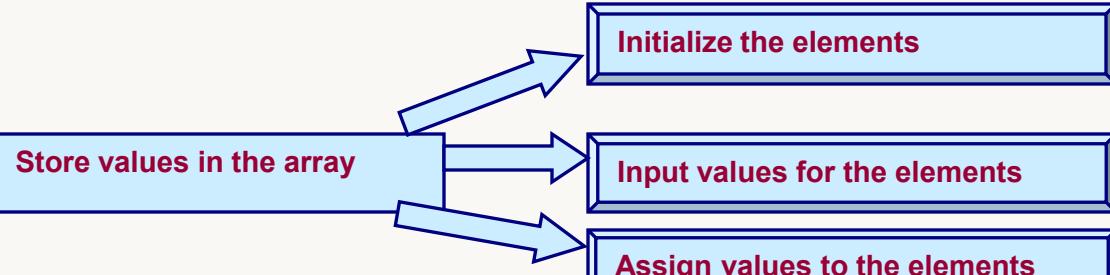
# STORING VALES IN ARRAYS

## Initialization of Arrays

Arrays are initialized by writing,  
type array\_name[size]={list of values};  
**int marks[5]={90, 82, 78, 95, 88};**

## Inputting Values

```
int i, marks[10];
for(i=0;i<10;i++)
    scanf("%d", &marks[i]);
```



## Assigning Values

```
int i, arr1[10], arr2[10];
for(i=0;i<10;i++)
    arr2[i] = arr1[i];
```

## CALCULATING THE LENGTH OF THE ARRAY

Length = upper\_bound – lower\_bound + 1

Where, upper\_bound is the index of the last element  
and lower\_bound is the index of the first element in the array

99	67	78	56	88	90	34	85
Marks[0]	marks[1]	marks[2]	marks[3]	marks[4]	marks[5]	marks[6]	marks[7]]

Here, lower\_bound = 0, upper\_bound = 7

Therefore, length = 7 – 0 + 1 = 8

# WRITE A PROGRAM TO READ AND DISPLAY N NUMBERS USING AN ARRAY

## Program 5.1 Write a program to read and display n numbers using an array.

```
#include<iostream.h>
int main()
{   int i = 0, n, arr[20];
    cout<< "\n Enter the number of elements : ";
    cin>>n;
    for(i = 0;i < n;i++)
    {   cout<<"\n Arr["<<i<<"] = ";
        cin>>arr[i];
    }
    cout<<"\n The array elements are \n";
    for(i = 0;i < n;i++)
        cout<<"\t Arr["<<i<<"] = "<<arr[i];
}
```

### OUTPUT

```
Enter the number of elements : 5
1 2 3 4 5
The array elements are
Arr[0] = 1 Arr[1] = 2 Arr[2] = 3 Arr[3] = 4 Arr[4] = 5
```

# INSERTING AN ELEMENT IN THE ARRAY

Algorithm to insert a new element to the end of the array.

```
Step 1: Set upper_bound = upper_bound + 1  
Step 2: Set A[upper_bound] = VAL  
Step 3: EXIT
```

The algorithm INSERT will be declared as INSERT( A, N, POS, VAL).

```
Step 1: [INITIALIZATION] SET I = N  
Step 2: Repeat Steps 3 and 4 while I >= POS  
Step 3:           SET A[I + 1] = A[I]  
Step 4:           SET I = I - 1  
               [End of Loop]  
Step 5: SET N = N + 1  
Step 6: SET A[POS] = VAL  
Step 7: EXIT
```

Calling **INSERT (Data, 6, 3, 100)** will lead to the following processing in the array

45	23	34	12	56	20	20
----	----	----	----	----	----	----

Data[0] Data[1] Data[2] Data[3] Data[4] Data[5] Data[6]

45	23	34	12	12	56	20
----	----	----	----	----	----	----

Data[0] Data[1] Data[2] Data[3] Data[4] Data[5] Data[6]

45	23	34	12	56	56	20
----	----	----	----	----	----	----

Data[0] Data[1] Data[2] Data[3] Data[4] Data[5] Data[6]

45	23	34	100	12	56	20
----	----	----	-----	----	----	----

Data[0] Data[1] Data[2] Data[3] Data[4] Data[5] Data[6]

# DELETING AN ELEMENT FROM THE ARRAY

Algorithm to delete an element from the end of the array

```
Step 1: Set upper_bound = upper_bound - 1  
Step 2: EXIT
```

The algorithm DELETE will be declared as DELETE( A, N, POS).

```
Step 1: [INITIALIZATION] SET I = POS  
Step 2: Repeat Steps 3 and 4 while I <= N - 1  
Step 3:           SET A[I] = A[I + 1]  
Step 4:           SET I = I + 1  
               [End of Loop]  
Step 5: SET N = N - 1  
Step 6: EXIT
```

45	23	34	12	56	20
Data[0]	Data[1]	Data[2]	Data[3]	Data[4]	Data[5]

45	23	12	56	56	20
Data[0]	Data[1]	Data[2]	Data[3]	Data[4]	Data[5]

Calling DELETE (Data, 6, 2) will lead to the following processing in the array

45	23	12	12	56	20
Data[0]	Data[1]	Data[2]	Data[3]	Data[4]	Data[5]

45	23	12	56	20
Data[0]	Data[1]	Data[2]	Data[3]	Data[4]

45	23	12	56	20
Data[0]	Data[1]	Data[2]	Data[3]	Data[4]

# LINEAR SEARCH

```
LINEAR_SEARCH(A, N, VAL, POS)

Step 1: [INITIALIZE] SET POS = -1
Step 2: [INITIALIZE] SET I = 0
Step 3:     Repeat Step 4 while I<N
Step 4:             IF A[I] = VAL, then
                        SET POS = I
                        PRINT POS
                        Go to Step 6
                    [END OF IF]
                [END OF LOOP]
Step 5: PRINT "Value Not Present In The Array"
Step 6: EXIT
```

## BINARY SEARCH

BEG = lower\_bound and END = upper\_bound

MID = (BEG + END) / 2

If VAL < A[MID], then VAL will be present in the left segment of the array. So,  
the value of END will be changed as, END = MID – 1

If VAL > A[MID], then VAL will be present in the right segment of the array. So,  
the value of BEG will be changed as, BEG = MID + 1

```

BINARY_SEARCH(A, lower_bound, upper_bound, VAL, POS)

Step 1: [INITIALIZE] SET BEG = lower_bound, END = upper_bound, POS = -1
Step 2: Repeat Step 3 and Step 4 while BEG <= END
Step 3:           SET MID = (BEG + END)/2
Step 4:           IF A[MID] = VAL, then
                    POS = MID
                    PRINT POS
                    Go to Step 6
                IF A[MID] > VAL then;
                    SET END = MID - 1
                ELSE
                    SET BEG = MID + 1
                [END OF IF]
            [END OF LOOP]
Step 5: IF POS = -1, then
        PRINTF "VAL IS NOT PRESENT IN THE ARRAY"
    [END OF IF]
Step 6: EXIT

```

int A[] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10};  
and VAL = 9, the algorithm will proceed in the following manner.

BEG = 0, END = 10, MID = (0 + 10)/2 = 5

Now, VAL = 9 and A[MID] = A[5] = 5

A[5] is less than VAL, therefore, we will now search for the value in the later half of the array. So, we change the values of BEG and MID.

Now, BEG = MID + 1 = 6, END = 10, MID = (6 + 10)/2 = 16/2 = 8

Now, VAL = 9 and A[MID] = A[8] = 8

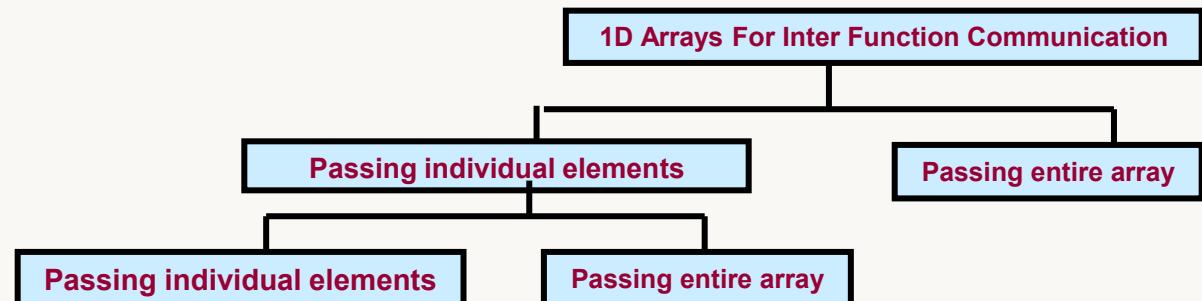
A[8] is less than VAL, therefore, we will now search for the value in the later half of the array. So, again we change the values of BEG and MID.

Now, BEG = MID + 1 = 9, END = 10, MID = (9 + 10)/2 = 9

Now VAL = 9 and A[MID] = 9.

Now VAL = 9 and A[MID] = 9.

# ONE DIMENSIONAL ARRAYS FOR INTER FUNCTION COMMUNICATION



## Passing data values

<pre>main() {     int arr[5] ={1, 2, 3, 4, 5};     func(arr[3]); }</pre>	<pre>void func(int num) {     cout&lt;&lt;num; }</pre>
--	--

## Passing addresses

<pre>main() {     int arr[5] ={1, 2, 3, 4, 5};     func(&amp;arr[3]); }</pre>	<pre>void func(int *num) {     cout&lt;&lt;num; }</pre>
---	---

## Passing the entire array

<pre>main() {     int arr[5] ={1, 2, 3, 4, 5};     func(arr); }</pre>	<pre>void func(int arr[5]) {     int i;     for(i=0;i&lt;5;i++)         cout&lt;&lt;arr[i]; }</pre>
---	---

# TWO DIMENSIONAL ARRAYS

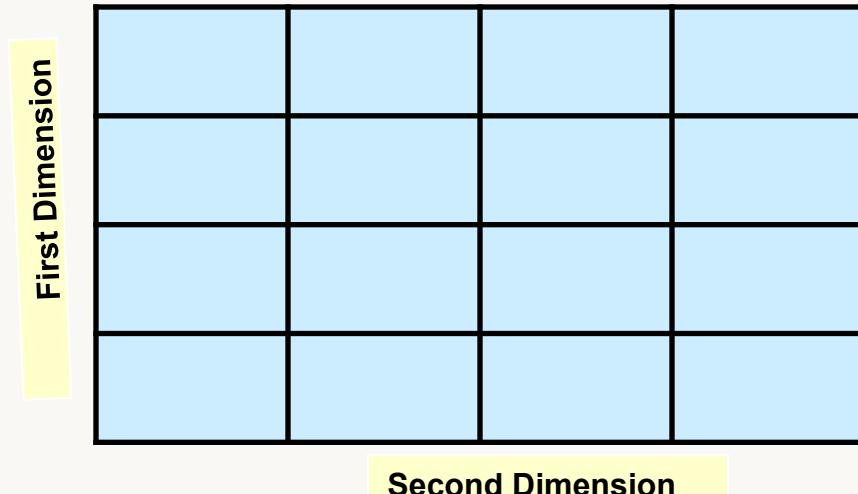
- A two dimensional array is specified using two subscripts where one subscript denotes row and the other denotes column.
- C looks a two dimensional array as an array of a one dimensional array.

A two dimensional array is declared as:

```
data_type array_name[row_size][column_size];
```

Therefore, a two dimensional mXn array is an array that contains  $m \times n$  data elements and each element is accessed using two subscripts, i and j where  $i \leq m$  and  $j \leq n$

```
int marks[3][5]
```



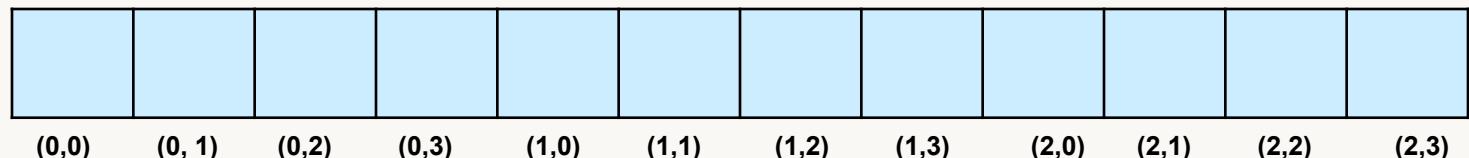
Rows/Columns	Col 0	Col 1	Col 2	Col 3	Col 4
Row 0	Marks [0] [0]	Marks [0] [1]	Marks [0] [2]	Marks [0] [3]	Marks [0] [4]
Row 1	Marks [1] [0]	Marks [1] [1]	Marks [1] [2]	Marks [1] [3]	Marks [1] [4]
Row 2	Marks [2] [0]	Marks [2] [1]	Marks [2] [2]	Marks [2] [3]	Marks [2] [4]

Two Dimensional Array

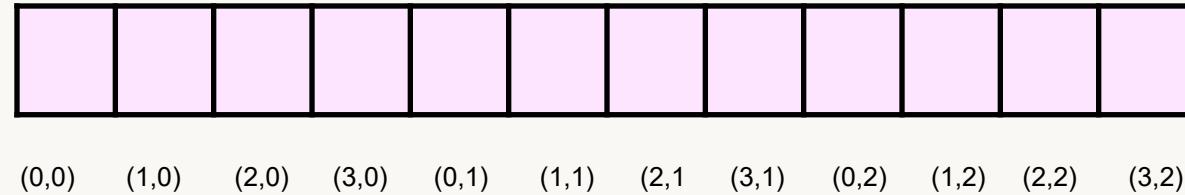
# MEMORY REPRESENTATION OF A TWO DIMENSIONAL ARRAY

There are two ways of storing a 2-D array can be stored in memory. The first way is row major order and the second is column major order.

In the row major order the elements of the first row are stored before the elements of the second and third row. That is, the elements of the array are stored row by row where n elements of the first row will occupy the first nth locations.



However, when we store the elements in a column major order, the elements of the first column are stored before the elements of the second and third column. That is, the elements of the array are stored column by column where n elements of the first column will occupy the first nth locations.



Address( $A[I][J]$ ) = Base\_Address +  $w\{M ( J - 1 ) + ( I - 1 )\}$ , if the array elements are stored in column major order.  
And, Address( $A[I][J]$ ) = Base\_Address +  $w\{N ( I - 1 ) + ( J - 1 )\}$ , if the array elements are stored in row major order.

Where, w is the number of words stored per memory location

m, is the number of columns

n, is the number of rows

I and J are the subscripts of the array element

# TWO DIMENSIONAL ARRAYS CONTD..

- A two dimensional array is initialized in the same was as a single dimensional array is initialized. For example,

```
int marks[2][3]={90, 87, 78, 68, 62, 71};  
int marks[2][3]={{90,87,78},{68, 62, 71}};
```

## Program 5.20 Write a program to print the elements of a 2D array.

```
#include<iostream.h>  
main()  
{  int arr[2][2] = {12, 34, 56,32};  
    int i, j;  
    for(i = 0;i < 2;i++)  
    {   cout<<"\n";  
        for(j = 0;j < 2;j++)  
            cout<<"\t"<<arr[i][j];  
    }  
}
```

## OUTPUT

```
12    34  
56    32
```

# TWO DIMENSIONAL ARRAYS FOR INTER FUNCTION COMMUNICATION



There are three ways of passing parts of the two dimensional array to a function. First, we can pass individual elements of the array. This is exactly same as we passed element of a one dimensional array.

## Passing a row

```
main()
{
    int arr[2][3]= { {1, 2, 3}, {4, 5, 6} };
    func(arr[1]);
}

void func(int arr[])
{
    int i;
    for(i=0;i<5;i++)
        cout<<arr[i] * 10;
}
```

## Passing the entire 2D array

To pass a two dimensional array to a function, we use the array name as the actual parameter. (The same we did in case of a 1D array). However, the parameter in the called function must indicate that the array has two dimensions.

# PROGRAM ILLUSTRATING PASSING ENTIRE ARRAY TO A FUNCTION

**Programming Tip:** A compiler error will be generated if you omit the array size in the parameter declaration for any array dimension other than the first.

**Program 5.27 Write a menu driven program to read and display an  $m \times n$  matrix. Find the sum, transpose, and product of two  $m \times n$  matrices.**

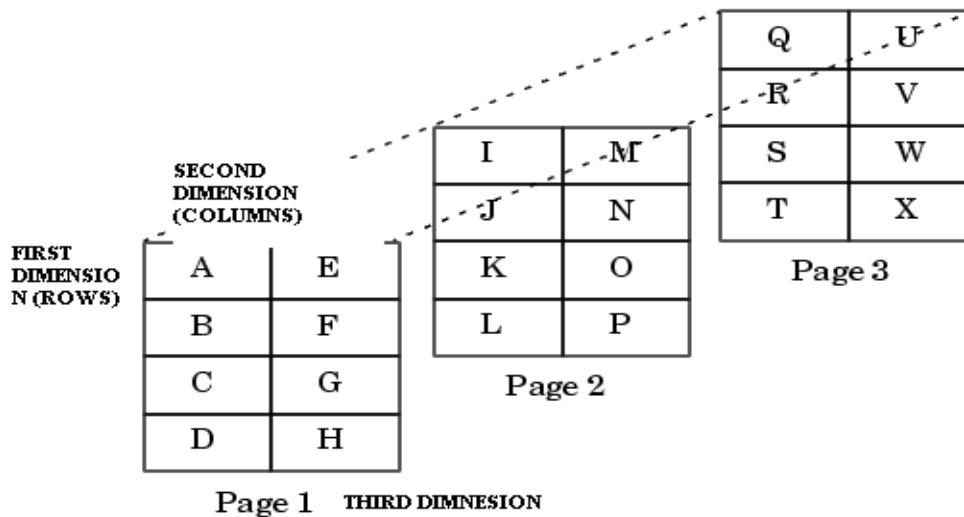
```
#include<iostream.h>
void read_matrix(int mat[][][], int, int);
void sum_matrix(int mat1[][][], int mat2[][][], int, int);
void mul_matrix(int mat1[][][], int mat2[][][], int, int);
void transpose_matrix(int mat2[][][], int, int);
void display_matrix(int mat[5][5], int r, int c);
int main()
{int option, row, col;
 int mat1[5][5], mat2[5][5];
 do
 { cout<<"\n ***** MAIN MENU *****";
 cout<<"\n 1. Read the two matrices";
 cout<<"\n 2. Add the matrices";
 cout<<"\n 3. Multiply the matrices";
 cout<<"\n 4. Transpose the matrix";
 cout<<"\n 5. EXIT";
 cout<<"\n\n Enter your option : ";
 cin>>option;
 switch(option)
 { case 1 : cout<<"\n Enter the number of rows and columns of the matrix : ";
 cin>>row>>col;
 read_matrix(mat1, row, col);
 cout<<"\n Enter the second matrix : ";
 read_matrix(mat2, row, col);
 break;
 case 2 : sum_matrix(mat1, mat2, row, col);
 break;
 case 3 : if(col == row)
 mul_matrix(mat1, mat2, row, col);
 else
 cout<<"\n To multiply two matrices, number of columns in the
 first matrix must be equal to number of rows in the second matrix";
 break;
 case 4:
 transpose_matrix(mat1, row, col);
 transpose_matrix(mat2, row, col);
 break;
 }
 }while(option != 5);
}
void read_matrix(int mat[5][5], int r, int c)
{ int i, j;
 for(i = 0;i < r;i++)
 { for(j = 0;j < c;j++)
 { cout<<"\n mat["<<i<<"]["<<j<<"] = ";
 cin>>mat[i][j];
 }
 }
```

# MULTI DIMENSIONAL ARRAYS

- A multi dimensional array is an array of arrays.
- Like we have one index in a single dimensional array, two indices in a two dimensional array, in the same way we have n indices in a n-dimensional array or multi dimensional array.
- Conversely, an n dimensional array is specified using n indices.
- An n dimensional  $m_1 \times m_2 \times m_3 \times \dots \times m_n$  array is a collection  $m_1 * m_2 * m_3 * \dots * m_n$  elements.
- In a multi dimensional array, a particular element is specified by using n subscripts as  $A[I_1][I_2][I_3]\dots[I_n]$ , where,

$I_1 \leq M_1 I_2 \leq M_2$

$I_3 \leq M_3 \dots I_n \leq M_n$



# PROGRAM TO READ AND DISPLAY A 2X2X2 ARRAY

**Program 5.28 Write a program to fill a square matrix with value zero on the diagonals, 1 on the upper right triangle and -1 on the lower left triangle.**

```
#include<iostream.h>
void read_matrix(int mat[5][5], int, int);
void sum_matrix(int mat1[5][5], int mat2[5][5], int, int);
void mul_matrix(int mat1[5][5], int mat2[5][5], int, int);
void transpose_matrix(int mat2[5][5], int, int);
void display_matrix(int mat[5][5], int r, int c);
int main()
{   int option, row, col;
    int mat1[5][5], mat2[5][5];
    do
    {   cout<<"\n ***** MAIN MENU *****";
        cout<<"\n 1. Read the two matrices";
        cout<<"\n 2. Add the matrices";
        cout<<"\n 3. Multiply the matrices";
        cout<<"\n 4. Transpose the matrix";
        cout<<"\n 5. EXIT";
        cout<<"\n\n Enter your option : ";
        cin>>option;
        switch(option)
        {   case 1:cout<<"\n Enter the number of rows and columns of the matrix : ";
            cin>>row>>col;
            read_matrix(mat1, row, col);
            cout<<"\n Enter the second matrix : ";
            read_matrix(mat2, row, col);
            break;
        case 2 : sum_matrix(mat1, mat2, row, col);
            break;
        case3 : if(col == row)
            mul_matrix(mat1, mat2, row, col);
            else
                cout<<"\n To multiply two matrices, number of columns in the
                first matrix must be equal to number of rows in the second matrix";
            break;
        case 4:
            transpose_matrix(mat1, row, col);
            transpose_matrix(mat2, row, col);
            break;
        }
    }while(option != 5);
}
void read_matrix(int mat[5][5], int r, int c)
{   int i, j;
    for(i = 0;i < r;i++)
    {   for(j = 0;j < c;j++)
        {   cout<<"\n mat["<<i<<"]["<<j<<"] = ";
            cin>>mat[i][j];
        }
    }
}
void sum_matrix(int mat1[5][5], int mat2[5][5], int r, int c)
{   int i, j, sum[5][5];
    for(i = 0;i < r;i++)
```

# SPARSE MATRIX

- Sparse matrix is a matrix that has many elements with a value zero.
- In order to efficiently utilize the memory, specialized algorithms and data structures that take advantage of the sparse structure of the matrix should be used. Otherwise, execution will slow down and the matrix will consume large amounts of memory.
- There are two types of sparse matrices. In the first type of sparse matrix, all elements above the main diagonal have a value zero. This type of sparse matrix is also called a (lower) triagonal matrix. In a lower triangular matrix,  $A_{i,j} = 0$  where  $i < j$ .
- An  $n \times n$  lower triangular matrix A has one non zero element in the first row, two non zero element in the second row and likewise,  $n$  non zero elements in the  $n$ th row.

1					
5	3				
2	7	-1			
3	1	4	2		
-9	2	-8	1	7	

- In an upper triangular matrix  $A_{i,j} = 0$  where  $i > j$ .
- An  $n \times n$  upper triangular matrix A has  $n$  non zero element in the first row,  $n-1$  non zero element in the second row and likewise, 1 non zero elements in the  $n$ th row.

1	2	3	4	5	
	3	6	7	8	
		-1	9	1	
			9	3	
				7	

# SPARSE MATRIX

- In the second variant of a sparse matrix, elements with a non-zero value can appear only on the diagonal or immediately above or below the diagonal. This type of matrix is also called a tridiagonal matrix.
- In a tridiagonal matrix,  $A_{i,j} = 0$  where  $|i - j| > 1$ . Therefore, if elements are present on
- the main diagonal the, it contains non-zero elements for  $i=j$ . In all there will be  $n$  elements
- diagonal below the main diagonal, it contains non zero elements for  $i=j+1$ . In all there will be  $n-1$  elements
- diagonal above the main diagonal, it contains non zero elements for  $i=j-1$ . In all there will be  $n-1$  elements

4	1			
5	1	2		
	9	3	1	
	4	2	2	
		5	1	9
			6	7

**FREE**  
ONLINE RESOURCES  
For Teachers and Students

OXFORD  
HIGHER EDUCATION

# Object Oriented Programming with C++



Reema Thareja

OXFORD  
HIGHER EDUCATION

# Object Oriented Programming with C++

Reema Thareja

# Chapter Six

## Strings

# INTRODUCTION

- A string is a null-terminated character array. This means that after the last character, a null character ('\0') is stored to signify the end of the character array.
- The general form of declaring a string is

```
char str[size];
```

- For example if we write,

```
char str[] = "HELLO";
```

We are declaring a character array with 5 characters namely, H, E, L, L and O. Besides, a null character ('\0') is stored at the end of the string. So, the internal representation of the string becomes- HELLO'\0'. Note that to store a string of length 5, we need 5 + 1 locations (1 extra for the null character).

The name of the character array (or the string) is a pointer to the beginning of the string.

str[0]	1000	H
str[1]	1001	E
str[2]	1002	L
str[3]	1003	L
str[4]	1004	O
str[5]	1005	\0

## Reading Strings

If we declare a string by writing

```
char str[100];
```

then, `str` can be read in four ways which are explained as follows:

- using `cin`
- using `gets()` function
- using `getchar()`, `getch()` or `getche()` function repeatedly
- using `getline()`
- The string can be read using `cin` by writing

```
cin>>str;
```

- The next method of reading a string is by using `gets()` function. The string can be read by writing

```
gets(str);
```

```
i = 0;
ch = getchar();           // Get a character
while(ch != '*')
{   str[i] = ch;         // Store the read character in str
    i++;
    ch = getchar();       // Get another character
}
str[i] = '\0';           // terminate str with null character
```

```
cin.getline(str,'*'); // string will be read until * is entered
cin.getline(str,10); // 10 characters will be read
```

## Writing Strings

The string can be displayed on the screen using three ways:

- using cout function
- using puts() function
- using putchar() and putch() function repeatedly
- The string can be displayed using cout by writing

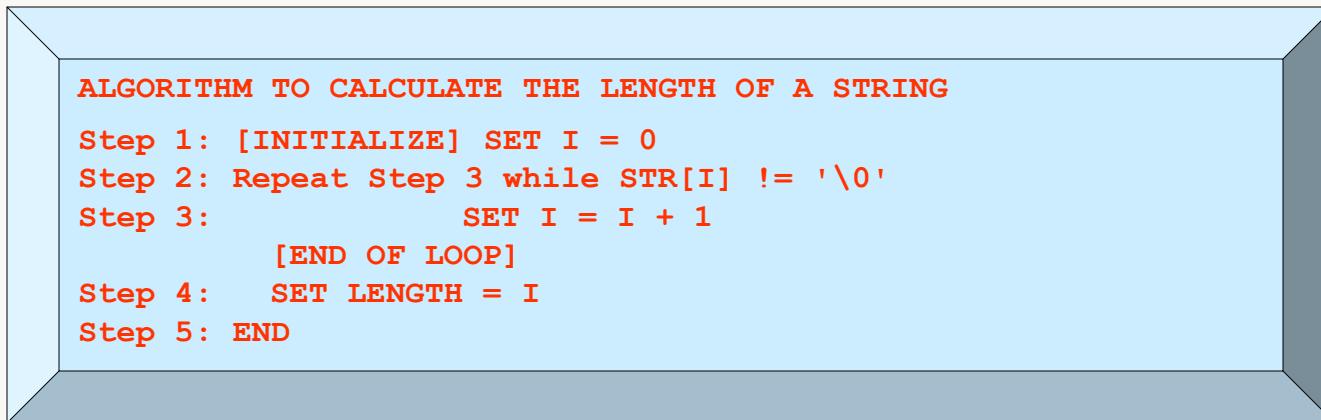
```
cout<<str;
```

```
puts(str);
```

```
i = 0;
while(str[i] != '\0')
{
    putchar(str[i]); // Prints the character on the screen one by one
    i++;
}
```

# LENGTH

- The number of characters in the string constitutes the length of the string.
- For example, LENGTH("C PROGRAMMING IS FUN") will return 20. Note that even blank spaces are counted as characters in the string.
- LENGTH('0') = 0 and LENGTH("") = 0 because both the strings does not contain any character.



## CONVERTING CHARACTERS OF A STRING INTO UPPER CASE

- In memory the ASCII code of a character is stored instead of its real value. The ASCII code for A-Z varies from 65 to 91 and the ASCII code for a-z ranges from 97 to 123. So if we have to convert a lower case character into upper case, then we just need to subtract 32 from the ASCII value of the character.

#### ALGORITHM TO CONVERT THE CHARACTERS OF STRING INTO UPPER CASE

```
Step1:      [Initialize] SET I=0
Step 2: Repeat Step 3 while STR[I] != '\0'
Step 3:          IF STR[I] > 'a' AND STR[I] < 'z'
                    SET Upperstr[I] = STR[I] - 32
                ELSE
                    SET Upperstr[I] = STR[I]
            [END OF IF]
        [END OF LOOP]
Step 4: SET Upperstr[I] = '\0'
Step 5: EXIT
```

## CONCATENATING TWO STRINGS TO FORM A NEW STRING

- IF S1 and S2 are two strings, then concatenation operation produces a string which contains characters of S1 followed by the characters of S2.

#### ALGORITHM TO CONCATENATE TWO STRINGS

```
1. Initialize I = 0 and J=0
2. Repeat step 3 to 4 while I <= LENGTH(str1)
3.     SET new_str[J] = str1[I]
4.     Set I =I+1 and J=J+1
    [END of step2]
5. SET I=0
6. Repeat step 6 to 7 while I <= LENGTH(str2)
7.     SET new_str[J] = str2[I]
8.     Set I =I+1 and J=J+1
    [END of step5]
9. SET new_str[J] = '\0'
10. EXIT
```

## **Program 6.2 Write a program to convert characters of a string in upper case.**

```
#include<iostream.h>
#include<stdio.h>
int main()
{   char str[100], upper_str [100];
    int i = 0;
    cout<<"\n Enter the string:";
    gets(str);
    while(str[i] != '\0')
    {   if(str[i]>='a' && str[i]<='z')
        upper_str[i] = str[i] - 32;
        else
            upper_str[i] = str[i];
        i++;
    }
    upper_str[i] = '\0';
    cout<<"\n The string converted into upper case is : ";
    puts(upper_str);
}
```

### **OUTPUT**

Enter the string: hello

# APPENDING

- Appending one string to another string involves copying the contents of the source string at the end of the destination string. For example, if S1 and S2 are two strings, then appending S1 to S2 means we have to add the contents of S1 to S2. so S1 is the source string and S2 is the destination string. The appending operation would leave the source string S1 unchanged and destination string  $S2 = S2 + S1$ .

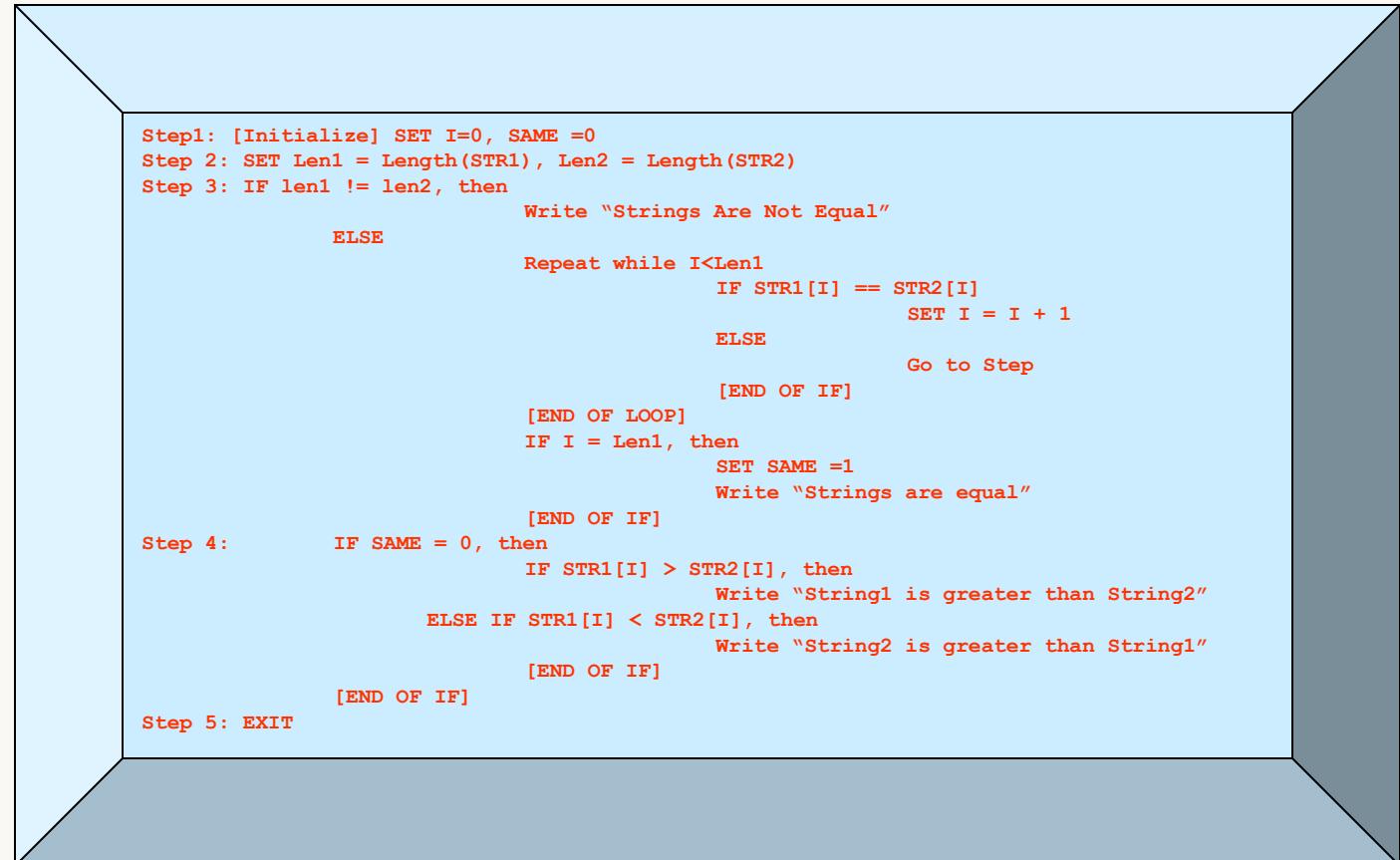
## ALGORITHM TO APPEND A STRING TO ANOTHER STRING

```
Step 1: [Initialize] SET I =0 and J=0
Step 2: Repeat Step 3 while Dest_Str[I] != '\0'
Step 3:         SET I + I + 1
          [END OF LOOP]
Step 4: Repeat Step 5 to 7 while Source_Str[J] != '\0'
Step 5:         Dest_Str[I] = Source_Str[J]
Step 6:         SET I = I + 1
Step 7:         SET J = J + 1
          [END OF LOOP]
Step 8: SET Dest_Str[J] = '\0'
Step 9: EXIT
```

# COMPARING TWO STRINGS

If S1 and S2 are two strings then comparing two strings will give either of these results-

- S1 and S2 are equal
- S1>S2, when in dictionary order S1 will come after S2
- S1<S2, when in dictionary order S1 precedes S2



## **Program 6.6 Write a program to compare two strings.**

```
#include <iostream.h>
#include <conio.h>
#include <string.h>
main()
{   char str1[50], str2[50];
    int i=0, len1 = 0, len2 = 0, same = 0;
    clrscr();
    cout<<"\n Enter the first string: ";
    cin.getline(str1, 50);
    cout<<"\n Enter the second string: ";
    cin.getline(str2, 50);
    len1 = strlen(str1);
    len2 = strlen(str2);
    if(len1 == len2)
    {   while(i<len1)
        {   if(str1[i] == str2[i])
            i++;
            else break;
        }
        if(i==len1)
        {   same=1;
            cout<<"\n The two strings are equal";
        }
    }
    if(len1!=len2)
        cout<<"\n The two strings are not equal";
    if(same == 0)
    {   if(str1[i]>str2[i])
        cout<<("\n String1 is greater than string2";
        else if(str1[i]<str2[i])
            cout<<"\n String2 is greater than string1";
    }
    getch();
    return 0;
}
```

# REVERSING A STRING

- If S1= “HELLO”, then reverse of S1 = “OLLEH”. To reverse a string we just need to swap the first character with the last, second character with the second last character, so on and so forth.

```
ALGORITHM TO REVERSE A STRING

Step1: [Initialize] SET I=0, J= Length(STR)
Step 2: Repeat Step 3 and 4 while I< Length(STR)
Step 3:      SWAP( STR(I) , STR(J))
Step 4:      SET I = I + 1, J = J - 1
           [END OF LOOP]
Step 5: EXIT
```

## EXTRACTING A SUBSTRING FROM LEFT

- In order to extract a substring from the main string we need to copy the content of the string starting from the first position to the nth position where n is the number of characters to be extracted.
- For example, if S1 = “Hello World”, then Substr\_Left(S1, 7) = Hello W

```
ALGORITHM TO EXTRACT N CHARCTERS FROM RIGHT OF A STRING

Step 1: [Initialize] SET I=0, J = Length(STR) - N + 1
Step 2: Repeat Step 3 while STR[J] != '\0'
Step 3:      SET Substr[I] = STR[J]
Step 4:      SET I = I + 1, J = J + 1
           [END OF LOOP]
Step 5: SET Substr[I] =' \0'
Step 6: EXIT
```

# EXTRACTING A SUBSTRING FROM RIGHT OF THE STRING

- In order to extract a substring from the right side of the main string we need to first calculate the position. For example, if S1 = “Hello World” and we have to copy 7 characters starting from the right, then we have to actually start extracting characters from the 5th position. This is calculated by, total number of characters – n + 1.
- For example, if S1 = “Hello World”, then Substr\_Right(S1, 7) = o World

## ALGORITHM TO EXTRACT N CHARCTERS FROM RIGHT OF A STRING

```
Step 1: [Initialize] SET I=0, J = Length(STR) - N + 1
Step 2: Repeat Step 3 while STR[J] != '\0'
Step 3:   SET Substr[I] = STR[J]
Step 4:   SET I = I + 1, J = J + 1
          [END OF LOOP]
Step 5: SET Substr[I] =' \0'
Step 6: EXIT
```

# EXTRACTING A SUBSTRING FROM THE MIDDLE OF A STRING

To extract a substring from a given string requires information about three things

- the main string
- the position of the first character of the substring in the given string
- maximum number of characters/length of the substring

For example, if we have a string-

str[] = “Welcome to the world of programming”; then,

SUBSTRING(str, 15, 5) = world

## Algorithm to extract substring from a given text

```
Step 1: [INITIALIZE] Set I=M, J = 0
Step 2: Repeat steps 3 to 6 while str[I] != '0' and N>=0
Step 3:         SET substr[J] = str[I]
Step 4:         SET I = I + 1
Step 5:         SET J = J + 1
Step 6:         SET N = N - 1
        [END of loop]
Step 7: SET substr[J] = '\0'
Step 8: EXIT
```

# INSERTION

The insertion operation inserts a string S in the main text, T at the kth position. The general syntax of this operation is: INSERT(text, position, string). For ex, INSERT("XYZXYZ", 3, "AAA") = "XYZAAAXYZ"

## Algorithm to insert a string in the main text

```
Step 1: [INITIALIZE] SET I=0, J=0 and K=0
Step 2: Repeat steps 3 to 4 while text[I] != '0'
Step 3: IF I = pos, then
        Repeat while str[K] != '\0'
            new_str[j] = str[k]
            SET J=J+1
            SET K = K+1
        [END OF INNER LOOP]
    ELSE
        new_str[[J] = text[I]
        SET J = J+1
Step 4: SET I = I+1
[END OF OUTER LOOP]
Step 5: SET new_str[J] = '\0'
Step 6: EXIT
```

# INDEXING

- Index operation returns the position in the string where the string pattern first occurs. For example,
- INDEX("Welcome to the world of programming", "world") = 15
- However, if the pattern does not exist in the string, the INDEX function returns 0.

Algorithm to find the index of the first occurrence of a string within a given text

```
Step 1: [Initialize] SET I=0 and MAX = LENGTH(text) - LENGTH(str) +1
Step 2: Repeat Steps 3 to 6 while I <= MAX
Step 3:  Repeat step 4 for K = 0 To Length(str)
Step 4:          IF str[K] != text[I + K], then GOTO step 6
                  [END of inner loop]
Step 5:          SET INDEX = I. Goto step 8
Step 6:          SET I = I+1
                  [END OF OUTER LOOP]
Step 7: SET INDEX = -1
Step 8: EXIT
```

# DELETION

- The deletion operation deletes a substring from a given text. We write it as, DELETE(text, position, length)
- For example, DELETE("ABCDXXXABCD", 5, 3) = "ABCDABCD"

Algorithm to delete a substring from a text

```
Step 1: [INITIALIZE] SET I=0 and J=0
Step 2: Repeat steps 3 to 6 while text[I] != '\0'
Step 3:           IF I=M, then
                    Repeat Step 4 while N>=0
                        SET I = I+1
                        SET N = N - 1
                    [END of inner loop]
                [END OF IF]
Step 4: SET new_str[J] = text[I]
Step 5: SET J= J + 1
Step 6: SET I = I + 1
        [ END of outer loop]
Step 7: SET new_str[J] = '\0'
Step 8: EXIT
```

# REPLACEMENT

- Replacement operation is used to replace the pattern P<sub>1</sub> by another pattern P<sub>2</sub>. This is done by writing, REPLACE(text, pattern1, pattern2)
- For example, (“AAABBBCCC”, “BBB”, “X”) = AAAXCCC
- (“AAABBBCCC”, “X”, “YYY”) = AAABBCC
- Note in the second example there is no change as ‘X’ does not appear in the text.

```
Algorithm to replace a pattern P1 with another pattern P2 in the given text  
TEXT
```

```
Step 1: [INITIALIZE] SET Pos = INDEX(TEXT, P1)  
Step 2: SET TEXT = DELETE(TEXT, Pos, LENGTH(P1))  
Step 3: INSERT(TEXT, Pos, P2)  
Step 4: EXIT
```

## **Program 6.12 Write a program to replace a pattern with another pattern in the text.**

```
#include <iostream.h>
#include <conio.h>
main()
{   char str[200], pat[20], new_str[200], rep_pat[100];
    int i=0, j=0, k, n=0, copy_loop=0, rep_index=0;
    clrscr();
    cout<<"\n Enter the string: ";
    cin.getline(str,200);
    cin.ignore();
    cout<<"\n Enter the pattern: ";
    cin.getline(pat, 20);
    cin.ignore();
    cout<<"\n Enter the replace pattern: ";
    cin.getline(rep_pat, 100);
    while(str[i]!='\0')
    {
        j=0,k=i;
        while(str[k]==pat[j] && pat[j]!='\0')
        {
            k++;
            j++;
        }
        if(pat[j]=='\0')
        {
            copy_loop=k;
            while(rep_pat[rep_index] !='\0')
            {
                new_str[n] = rep_pat[rep_index];
                rep_index++;
                n++;
            }
        }
        new_str[n] = str[copy_loop];
        i++;
        copy_loop++;
        n++;
    }
    new_str[n]='\0';
    cout<<("\n The new string is: ");
    puts(new_str);
    getch();
    return 0;
}
```

### **OUTPUT**

```
Enter the string: How ARE you?
Enter the pattern: ARE
Enter the pattern: are
The new string is: How are you?
```

# ARRAY OF STRINGS

- Now suppose that there are 20 students in a class and we need a string that stores names of all the 20 students. How can this be done? Here, we need a string of strings or an array of strings. Such an array of strings would store 20 individual strings. An array of string is declared as,  
char names[20][30];
- Here, the first index will specify how many strings are needed and the second index specifies the length of every individual string. So here, we allocate space for 20 names where each name can be maximum 30 characters long.
- Let us see the memory representation of an array of strings. If we have an array declared as,  
char name[5][10] = {"Ram", "Mohan", "Shyam", "Hari", "Gopal"};

<b>Name[0]</b>	R	A	M	'\0	,						
<b>Name[1]</b>	M	O	H	A	N	'\0	,				
<b>Name[2]</b>	S	H	Y	A	M	'\0	,				
<b>Name[3]</b>	H	A	R	I	'\0	,					
<b>Name[4]</b>	G	O	P	A	L	'\0	,				

### **Program 6.13 Write a program to read and print the names of n students of a class.**

```
#include<stdio.h>
#include<iostream.h>
main()
{   char names[5][10];
    int i, n;
    cout<<"\n Enter the number of students : ";
    cin>>n;
    for(i = 0;i<n;i++)
    {   cout<<"\n Enter the name of "<<i+1<<"th student : ";
        cin.ignore();
        cin.getline(names[i],10);
    }
    cout<<"\n Names of the students are : \n";
    for(i = 0;i<n;i++)
        puts(names[i]);
}
```

#### **OUTPUT**

```
Enter the number of students : 3
Enter the name of 1th student : Aditya
Enter the name of 2dth student : Goransh
Enter the name of 3dth student : Sarthak
Names of the students are : Aditya Goransh Sarthak
```

**FREE**  
ONLINE RESOURCES  
For Teachers and Students

OXFORD  
HIGHER EDUCATION

# Object Oriented Programming with C++



Reema Thareja

OXFORD  
HIGHER EDUCATION

# Object Oriented Programming with C++

Reema Thareja

# Chapter Seven

## Pointers

# UNDERSTANDING THE COMPUTER'S MEMORY

- Every computer has a primary memory. All our data and programs need to be placed in the primary memory for execution.
- The primary memory or RAM (Random Access Memory which is a part of the primary memory) is a collection of memory locations (often known as cells) and each location has a specific address. Each memory location is capable of storing 1 byte of data
- Generally, the computer has three areas of memory each of which is used for a specific task. These areas of memory include- stack, heap and global memory.
- **Stack-** A fixed size of stack is allocated by the system and is filled as needed from the bottom to the top, one element at a time. These elements can be removed from the top to the bottom by removing one element at a time. That is, the last element added to the stack is removed first.
- **Heap-** Heap is a contiguous block of memory that is available for use by the program when need arise. A fixed size heap is allocated by the system and is used by the system in a random fashion.
- When the program requests a block of memory, the dynamic allocation technique carves out a block from the heap and assigns it to the program.
- When the program has finished using that block, it returns that memory block to the heap and the location of the memory locations in that block is added to the free list.

# UNDERSTANDING THE COMPUTER'S MEMORY

- **Global Memory-** The block of code that is the main() program (along with other functions in the program) is stored in the global memory. The memory in the global area is allocated randomly to store the code of different functions in the program in such a way that one function is not contiguous to another function. Besides, the function code, all global variables declared in the program are stored in the global memory area.
- **Other Memory Layouts-** C provides some more memory areas like- text segment, BSS and shared library segment.
- The text segment is used to store the machine instructions corresponding to the compiled program. This is generally a read-only memory segment
- BSS is used to store un-initialized global variables
- Shared libraries segment contains the executable image of shared libraries that are being used by the program.

# INTRODUCTION

- Every variable in C has a name and a value associated with it. When a variable is declared, a specific block of memory within the computer is allocated to hold the value of that variable. The size of the allocated block depends on the type of the data.

```
int x = 10;
```

- When this statement executes, the compiler sets aside 2 bytes of memory to hold the value 10. It also sets up a symbol table in which it adds the symbol x and the relative address in memory where those 2 bytes were set aside.
- Thus, every variable in C has a value and an also a memory location (commonly known as address) associated with it. Some texts use the term *rvalue* and *lvalue* for the value and the address of the variable respectively.
- The rvalue appears on the right side of the assignment statement and cannot be used on the left side of the assignment statement. Therefore, writing  $10 = k;$  is illegal.

# DECLARING POINTER VARIABLES

- Actually pointers are nothing but memory addresses.
- A pointer is a variable that contains the memory location of another variable.
- The general syntax of declaring pointer variable is

```
data_type *ptr_name;
```

Here, data\_type is the data type of the value that the pointer will point to. For example:

```
int *pnum;      char *pch;      float *pfnum;  
  
int x= 10;  
  
int *ptr = &x;
```

The '\*' informs the compiler that ptr is a pointer variable and the int specifies that it will store the address of an integer variable.

The & operator retrieves the lvalue (address) of x, and copies that to the contents of the pointer ptr.

# DE-REFERENCING A POINTER VARIABLE

pointer constant; therefore, it cannot be changed in the program code.

## Example 7.2

Illustrating the use of pointer variable

```
#include<iostream.h>
int main()
{    int num, *pnum;
    pnum = &num;
    cout<<"\n Enter the number : ";
    cin>>num;
    cout<<"\n The number that was entered is : "<<*pnum;
    cout<<"\n The address of number in memory is : %p", &num;
}
```

## OUTPUT

```
Enter the number : 10
The number that was entered is : 10
The address of number in memory is : FFDC
```

# POINTER EXPRESSIONS AND POINTER ARITHMETIC

- Pointer variables can also be used in expressions. For ex,

```
int num1=2, num2= 3, sum=0, mul=0, div=1;  
  
int *ptr1, *ptr2;  
  
ptr1 = &num1, ptr2 = &num2;  
  
sum = *ptr1 + *ptr2;  
  
mul = sum * *ptr1;  
  
*ptr2 +=1;  
  
div = 9 + *ptr1/*ptr2 - 30;
```

- We can add integers to or subtract integers from pointers as well as to subtract one pointer from the other.
- We can compare pointers by using relational operators in the expressions. For example  $p1 > p2$  ,  $p1==p2$  and  $p1!=p2$  are all valid in C.

When using pointers, unary increment (++) and decrement (--) operators have greater precedence than the dereference operator (\*). Therefore, the expression

$*ptr++$  is equivalent to  $*(ptr++)$ . So the expression will increase the value of ptr so that it now points to the next element.

In order to increment the value of the variable whose address is stored in ptr, write  $(*ptr)++$

# NULL POINTERS

A *null pointer* which is a special pointer value that is known not to point anywhere. This means that a NULL pointer does not point to any valid memory address.

To declare a null pointer you may use the predefined constant NULL,

```
int *ptr = NULL;
```

You can always check whether a given pointer variable stores address of some variable or contains a null by writing,

```
if ( ptr == NULL)  
{     Statement block;  
}
```

Null pointers are used in situations if one of the pointers in the program points somewhere some of the time but not all of the time. In such situations it is always better to set it to a null pointer when it doesn't point anywhere valid, and to test to see if it's a null pointer before using it.

# GENERIC POINTERS

- A generic pointer is pointer variable that has void as its data type.
- The generic pointer, can be pointed at variables of any data type.
- It is declared by writing

```
void *ptr;
```

- You need to cast a void pointer to another kind of pointer before using it.
- Generic pointers are used when a pointer has to point to data of different types at different times. For ex,

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

```
int main()
```

```
{     int x=10;
```

```
     char ch = 'A';
```

```
     void *gp;
```

```
     gp = &x;    cout<<"\n Generic pointer points to the integer value = "<< *(int*)gp;
```

```
     gp = &ch;   cout<<"\n Generic pointer now points to the character <<*(char*)gp);
```

```
return 0;
```

```
} OUTPUT:
```

- **Generic pointer points to the integer value = 10**
- **Generic pointer now points to the character = A**

# PASSING ARGUMENTS TO FUNCTION USING POINTERS

The calling function sends the addresses of the variables and the called function must declare those incoming arguments as pointers. In order to modify the variables sent by the caller, the called function must dereference the pointers that were passed to it. Thus, passing pointers to a function avoid the overhead of copying data from one function to another.

## **Program 7.4 Write a program to add two integers using functions.**

```
#include<iostream.h>
void sum ( int *a, int *b, int *t);
int main()
{
    int num1, num2, total;
    cout<<"\n Enter the two numbers : ";
    cin>>num1>>num2;scanf("%d", &num1);
    sum(&num1, &num2, &total);
    cout<<"\n Total = "<<total;
}
void sum ( int *a, int *b, int *t)
{
    *t = *a + *b;
}
```

## **OUTPUT**

```
Enter the two numbers : 2  3
Total = 5
```

# POINTERS AND ARRAYS

```
int arr[5] = {1, 2, 3, 4, 5};  
int *ptr = &arr[0];  
ptr++;  
Cout<<"\n The value of the second element of the array is "<<*ptr;
```

## Example 7.10 To display an array of given numbers from 1 to 9

```
#include<iostream.h>  
main()  
{    int arr[]={1,2,3,4,5,6,7,8,9};  
    int *ptr1, *ptr2;  
    ptr1 = arr; // ptr1 pointing to the first element of the array  
    ptr2 = &arr[8]; // ptr2 pointing to the last element of the array  
    while(ptr1<=ptr2)  
    {        cout<<" " <<*ptr1;  
        ptr1++;  
    }  
}
```

### OUTPUT

```
1 2 3 4 5 6 7 8 9
```

# POINTERS AND TWO DIMENSIONAL ARRAY

- Individual elements of the array mat can be accessed using either:

`mat[i][j]` or `*(*(mat + i) + j)` or `*(mat[i]+j);`

- See pointer to a one dimensional array can be declared as,

```
int arr[]={1,2,3,4,5};
```

```
int *parr;
```

```
parr=arr;
```

- Similarly, pointer to a two dimensional array can be declared as,

```
int arr[2][2]={{1,2},{3,4}};
```

```
int (*parr)[2];
```

```
parr=arr;
```

## Example 7.16

To illustrate the use of a pointer to a 2D array

```
#include<iostream.h>
main()
{
    int arr[2][2]={{1,2},{3,4}};
    int i, (*parr)[2];
    parr=arr;
    for(i=0;i<2;i++)
    {
        for(int j=0;j<2;j++)
            cout<<" "<<(*parr+i))[j];
    }
}
```

## OUTPUT

```
1 2 3 4
```

# POINTERS AND STRINGS

- Now, consider the following program that prints a text.
- #include<stdio.h>
- main()
- { char str[] = "Oxford";  
char \*pstr = str;  
cout<<"\n The string is : ";  
while( \*pstr != '\0'  
{ cout<<\*pstr;  
pstr++;  
}  
}
- In this program we declare a character pointer \*pstr to show the string on the screen. We then "point" the pointer pstr at str. Then we print each character of the string in the while loop. Instead of using the while loop, we could have straight away used the function puts(), like  
puts(pstr);
- Consider here that the function prototype for puts() is:
- int puts(const char \*s); Here the "const" modifier is used to assure the user that the function will not modify the contents pointed to by the source pointer. Note that the address of the string is passed to the function as an argument.

# POINTERS AND STRINGS

- Consider another program which reads a string and then scans each character to count the number of upper and lower case characters entered

## Example 7.12

### Printing a text using pointers

```
#include<iostream.h>
int main()
{
    char str[] = "Oxford";
    char *pstr = str;
    cout<<"\n The string is : ";
    while( *pstr != '\0')
    {
        cout<<*pstr;
        pstr++;
    }
}
```

#### OUTPUT

The string is : Oxford

# ARRAY OF POINTERS

An array of pointers can be declared as

```
int *ptr[10]
```

The above statement declares an array of 10 pointers where each of the pointer points to an integer variable. For example, look at the code given below.

```
int *ptr[10];  
  
int p=1, q=2, r=3, s=4, t=5;  
  
ptr[0]=&p;  
  
ptr[1]=&q;  
  
ptr[2]=&r;  
  
ptr[3]=&s;  
  
ptr[4]=&t
```

Can you tell what will be the output of the following statement?

```
cout<<"\n"<< *ptr[3];
```

Yes, the output will be 4 because ptr[3] stores the address of integer variable s and \*ptr[3] will therefore print the value of s that is 4.

# POINTER TO FUNCTION

- This is a useful technique for passing a function as an argument to another function.
- In order to declare a pointer to a function we have to declare it like the prototype of the function except that the name of the function is enclosed between parentheses () and an asterisk (\*) is inserted before the name.

```
/* pointer to function returning int */  
int (*func)(int a, float b);
```

- If we have declared a pointer to the function, then that pointer can be assigned to the address of the right sort of function just by using its name.
- When a pointer to a function is declared, it can be called using one of two forms:

```
(*func)(1,2); OR func(1,2);
```

## Example 7.18

To illustrate the use of a pointer to a function

```
#include <iostream.h>  
void print(int n);  
void (*fp)(int); // function pointer declaration  
main()  
{    fp = print;  
    (*fp)(10); // initializing function pointer declaration  
    fp(20);  
}  
void print(int value)  
{    cout<<value;  
}
```

## OUTPUT

```
10  
20
```

# PASSING A FUNCTION POINTER AS AN ARGUMENT TO A FUNCTION

A function pointer can be passed as a function's calling argument. This is done when you want to pass a pointer to a callback function.

**Example 7.20**

Passing a pointer to a function

```
#include<iostream.h>
#include <stdio.h>
int add(int, int);
int subt(int, int);
int operate(int (*operate_fp) (int, int), int, int);
main()
{
    int result;
    result = operate(add, 9, 7);
    cout <<"\n Addition = "<< result;
    result = operate(sub, 9, 7);
    cout <<"\n Subtraction = "<< result;

    return 0;
}
int add (int a, int b)
{
    return (a + b);
}
int subtract (int a, int b)
{
    return (a - b);
}
int operate(int (*operate_fp) (int, int), int a, int b)
{
    int result;
    result = (*operate_fp) (a,b);
    return result;
}

OUTPUT
Addition = 16
Subtraction = 2
```

# POINTERS TO POINTERS

- You can use pointers that point to pointers. The pointers in turn, point to data (or even to other pointers). To declare pointers to pointers just add an asterisk (\*) for each level of reference.

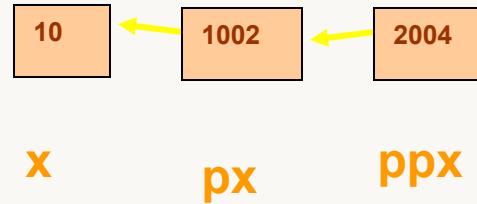
For example, if we have:

```
int x=10;  
int *px, **ppx;  
  
px=&x;  
  
ppx=&px;
```

Now if we write,

```
cout<<"\n "<<**ppx;
```

Then it would print 10, the value of x.



**FREE**  
ONLINE RESOURCES  
For Teachers and Students

OXFORD  
HIGHER EDUCATION

# Object Oriented Programming with C++



Reema Thareja

OXFORD  
HIGHER EDUCATION

# Object Oriented Programming with C++

Reema Thareja

# Chapter Eight

## Structure, Union, and Enumerated Data Types

# INTRODUCTION

- A structure is same as that of records. It stores related information about an entity.
- Structure is basically a user defined data type that can store related information (even of different data types) together.
- A structure is declared using the keyword struct followed by a structure name. All the variables of the structures are declared within the structure. A structure type is defined by using the given syntax.
- **struct struct-name**

```
{    data_type var-name;
    data_type var-name;
    ...
};

struct student
{
    int r_no;
    char name[20];
    char course[20];
    float fees;
};
```
- The structure definition does not allocates any memory. It just gives a template that conveys to the C compiler how the structure is laid out in memory and gives details of the member names. Memory is allocated for the structure when we declare a variable of the structure. For ex, we can define a variable of student by writing  
`struct student stud1;`

# TYPEDEF DECLARATIONS

- When we precede a struct name with `typedef` keyword, then the struct becomes a new type. It is used to make the construct shorter with more meaningful names for types already defined by C or for types that you have declared. With a `typedef` declaration, becomes a synonym for the type.
- For example, writing
- `typedef struct student`
- `{`
- `int r_no;`
- `char name[20];`
- `char course[20];`
- `float fees;`
- `};`
- Now that you have preceded the structure's name with the keyword `typedef`, the `student` becomes a new data type. Therefore, now you can straight away declare variables of this new data type as you declare variables of type `int`, `float`, `char`, `double`, etc. to declare a variable of structure `student` you will just write,
- `student stud1;`

# INITIALIZATION OF STRUCTURES

- Initializing a structure means assigning some constants to the members of the structure.
- When the user does not explicitly initializes the structure then C automatically does that. For int and float members, the values are initialized to zero and char and string members are initialized to the '\0' by default.
- The initializers are enclosed in braces and are separated by commas. Note that initializers match their corresponding types in the structure definition.
- The general syntax to initialize a structure variable is given as follows.
- **struct struct\_name**
- {  
    **data\_type member\_name1;**  
    **data\_type member\_name2;**  
    **data\_type member\_name3;**  
    .....  
};  
**struct\_var = {constant1, constant2, constant 3,...};**
- OR
- **struct struct\_name**
- {  
    **data\_type member\_name1;**  
    **data\_type member\_name2;**  
    **data\_type member\_name3;**  
    .....  
};  
**struct struct\_name struct\_var = {constant1, constant2, ....};**
- **struct student stud1 = {01, "Rahul", "BCA", 45000};**

# ACCESSING THE MEMBERS OF A STRUCTURE

- Each member of a structure can be used just like a normal variable, but its name will be a bit longer. A structure member variable is generally accessed using a ‘.’ (dot operator).
- The syntax of accessing a structure a member of a structure is:  
`struct_var.member_name`
- For ex, to assign value to the individual data members of the structure variable **Rahul**, we may write,

```
stud1.r_no = 01;  
strcpy(stud1.name, "Rahul");  
stud1.course = "BCA";  
stud1.fees = 45000;
```

- We can assign a structure to another structure of the same type. For ex, if we have two structure variables **stu1** and **stud2** of type **struct student** given as
- `struct student stud1 = {01, "Rahul", "BCA", 45000};`
- `struct student stud2;`
- Then to assign one structure variable to another we will write,
- `stud2 = stud1;`

**Program 8.1 Write a program using structures to read and display the information about a student.**

```
#include<iostream.h>
int main()
{   struct student
    {   int roll_no;
        char name[20];
        float fees;
    };
    struct student stud1;
    cout<<"\n Enter the roll number : ";
    cin>>stud1.roll_no;
    cout<<"\n Enter the name : ";
    cin.ignore();
    cin.getline(stud1.name, 20);
    cout<<"\n Enter the fees : ";
    cin>>stud1.fees;
    cout<<"\n *****STUDENT'S DETAILS *****";
    cout<<"\n ROLL No. = "<<stud1.roll_no;
    cout<<"\n NAME. = "<<stud1.name;
    cout<<"\n FEES = "<<stud1.fees;
}
```

**OUTPUT**

```
Enter the roll number : 01
Enter the name : Rahul
Enter the fees : 45000
*****STUDENT'S DETAILS *****
ROLL No. = 01
NAME. = Rahul
FEES = 45000.00
```

# NESTED STRUCTURES

- A structure can be placed within another structure. That is, a structure may contain another structure as its member. Such a structure that contains another structure as its member is called a nested structure.

- **typedef struct**

- {       **char first\_name[20];**  
         **char mid\_name[20];**  
         **char last\_name[20];**

**}NAME;**

**typedef struct**

- {   **int dd;**  
      **int mm;**  
      **int yy;**

**}DATE;**

**struct student stud1;**

- **stud1.name.first\_name = "Janak";**  
**stud1.name.mid\_name = "Raj";**

- **stud1.name.last\_name = "Thareja";**

- **stud1.course = "BCA";**

- **stud1.DOB.dd = 15;**

- **stud1.DOB.mm = 09;**

- **stud1.DOB.yy = 1990;**

- **stud1.fees = 45000;**

#### **Program 8.4 Write a program to read and display information of a student using structure within a structure.**

```
#include<iostream.h>
int main()
{   struct DOB
    {   int day;
        int month;
        int year;
    };
    struct student
    {   int roll_no;
        char name[20];
        float fees;
        struct DOB date;
    };
    student stud1;
    cout<<"\n Enter the roll number : ";
    cin>>stud1.roll_no;
    cout<<"\n Enter the name : ";
    cin.ignore();
    cin.getline(stud1.name, 20);
    cout<<"\n Enter the fees : ";
    cin>>stud1.fees;
    cout<<"\n Enter the DOB : ";
    cin>>stud1.date.day>>stud1.date.month>>stud1.date.year;
    cout<<"\n *****STUDENT'S DETAILS*****";
    cout<<"\n ROLL No. = "<< stud1.roll_no;
    cout<<"\n NAME. = "<<stud1.name;
    cout<<"\n FEES. = "<< stud1.fees;
    cout<<"\n DOB = "<<stud1.date.day<<" - "<<stud1.date.month<<" - "
                                <<stud1.date.year;
}
```

# ARRAYS OF STRUCTURES

- The general syntax for declaring an array of structure can be given as,
- `struct struct_name struct_var[index];`
- `struct student stud[30];`
- Now, to assign values to the  $i^{\text{th}}$  student of the class, we will write,
- `stud[i].r_no = 09;`
- `stud[i].name = "RASHI";`
- `stud[i].course = "MCA";`
- `stud[i].fees = 60000;`

## **Program 8.5 Write a program to read and display information of all the students in the class.**

```
#include<iostream.h>
int main()
{   struct student
    {   int roll_no;
        char name[20];
        float fees;
        char DOB[80];
    };
    struct student stud[50];
    int n, i;
    cout<<"\n Enter the number of students : ";
    cin>>n;
    for(i=0;i<n;i++)
    {   cout<<"\n Enter the roll number : ";
        cin>>stud[i].roll_no;
        cout<<"\n Enter the name : ";
        cin.ignore();
        cin.getline(stud[i].name, 20);
        cout<<"\n Enter the fees : ";
        cin>>stud[i].fees;
        cout<<"\n Enter the DOB : ";
        cin>>stud[i].DOB;
    }
    for(i=0;i<n;i++)
    {   cout<<"\n *****DETAILS OF "<<i+1<<"STUDENT*****";
        cout<<"\n ROLL No. = "<< stud[i].roll_no;
        cout<<"\n NAME. = "<<stud[i].name;
        cout<<"\n FEES = "<< stud[i].fees;
        cout<<"\n DOB = "<<stud[i].DOB;
    }
}
```

# Passing Individual Structure Members to a Function

- To pass any individual member of the structure to a function we must use the direct selection operator to refer to the individual members for the actual parameters. The called program does not know if the two variables are ordinary variables or structure members.
- ```
#include<stdio.h>
typedef struct
{
    int x;
    int y;
}POINT;
void display(int, int);
main()
{
    POINT p1 = {2, 3};
    display(p1.x, p1.y);
    return 0;
}
void display( int a, int b)
{
    cout<<a<<b;
}
```

# PASSING A STRUCTURE TO A FUNCTION

- When a structure is passed as an argument, it is passed using call by value method. That is a copy of each member of the structure is made. No doubt, this is a very inefficient method especially when the structure is very big or the function is called frequently. Therefore, in such a situation passing and working with pointers may be more efficient.
- The general syntax for passing a structure to a function and returning a structure can be given as, struct struct\_name func\_name(struct struct\_name struct\_var);
- The code given below passes a structure to the function using call-by-value method.

```
#include<stdio.h>
typedef struct
{
    int x;
    int y;
}POINT;
void display(POINT);
main()
{
    POINT p1 = {2, 3};
    display(p1);
    return 0;
}
void display( POINT p)
{
    cout<<p.x<<p.y;
}
```

# PASSING STRUCTURES THROUGH POINTERS

- C allows to create a pointer to a structure. Like in other cases, a pointer to a structure is never itself a structure, but merely a variable that holds the address of a structure. The syntax to declare a pointer to a structure can be given as
- ```
struct struct_name
{
    data_type member_name1;
    data_type member_name2;
    .....
}*ptr;
```
- OR
- ```
struct struct_name *ptr;
```
- For our student structure we can declare a pointer variable by writing
- ```
struct student *ptr_stud, stud;
```
- The next step is to assign the address of stud to the pointer using the address operator (&). So to assign the address, we will write
- ```
ptr_stud = &stud;
```
- To access the members of the structure, one way is to write
- ```
/* get the structure, then select a member */
(*ptr_stud).roll_no;
```
- An alternative to the above statement can be used by using ‘pointing-to’ operator (->) as shown below.
- ```
/* the roll_no in the structure ptr_stud points to */
ptr_stud->roll_no = 01;
```

# Write a program using pointer to structure to initialize the members in the structure.

```
#include<iostream.h>
struct student
{
    int r_no;
    char name[20];
    char course[20];
    float fees;
};
main()
{
    struct student *ptr_stud1;
    struct student stud1 = {01, "Rahul", "BCA", 45000};
    ptr_stud1 = &stud1;
    cout<<"\n DETAILS OF STUDENT";
    cout<<"\n -----";
    cout<<"\n ROLL NUMBER = "<< ptr_stud1->r_no;
    cout<<"\n NAME = "<< ptr_stud1->name;
    cout<<"\n COURSE = "<<ptr_stud1->course;
    cout<<"\n FEES = "<< ptr_stud1->fees;
}
```

## OUTPUT

DETAILS OF STUDENT

-----  
ROLL NUMBER = 01  
NAME = Rahul  
COURSE = BCA  
FEES = 45000.00

# SELF REFERENTIAL STRUCTURES

- Self referential structures are those structures that contain a reference to data of its same type. That is, a self referential structure in addition to other data contains a pointer to a data that is of the same type as that of the structure. For example, consider the structure node given below.
- struct node
- {
- int val;
- struct node \*next;
- };
- Here the structure node will contain two types of data- an integer val and next that is a pointer to a node. You must be wondering why do we need such a structure? Actually, self-referential structure is the foundation of other data structures.

# UNION

- Like structure, a union is a collection of variables of different data types. The only difference between a structure and a union is that in case of unions, you can only store information in one field at any one time.
- To better understand union, think of it as a chunk of memory that is used to store variables of different types. When a new value is assigned to a field, the existing data is replaced with the new data.
- Thus unions are used to save memory. They are useful for applications that involve multiple members, where values need not be assigned to all the members at any one time.

## DECLARING A UNION

- The syntax for declaring a union is same as that of declaring a structure.

```
• union union-name  
• {      data_type var-name;  
        data_type var-name;  
        ...  
};
```

Again, the **typedef** keyword can be used to simplify the declaration of union variables.

- The most important thing to remember about a union is that the size of an union is the size of its largest field. This is because a sufficient number of bytes must be reserved to store the largest sized field.

## ACCESSING A MEMBER OF A UNION

- A member of a union can be accessed using the same syntax as that of a structure. To access the fields of a union, use the dot operator(.). That is the union variable name followed by the dot operator followed by the member name.

## INITIALIZING UNIONS

- It is an error to initialize any other union member except the first member
- A striking difference between a structure and a union is that in case of a union, the field fields share the same memory space, so fresh data replaces any existing data. Look at the code given below and observe the difference between a structure and union when their fields are to be initialized.

```
#include<stdio.h>
typedef struct POINT1
{
    int x, y;
};

typedef union POINT2
{
    int x;
    int y;
};

main()
{
    POINT1 P1 = {2,3};
    // POINT2 P2 ={4,5}; Illegeal with union

    POINT2 P2;
    P2. x = 4;
    P2.y = 5;
    printf("\n The co-ordinates of P1 are %d and %d", P1.x, P1.y);
    printf("\n The co-ordinates of P2 are %d and %d", P2.x, P2.y);
    return 0;
}

OUTPUT
The co-ordinates of P1 are 2 and 3
The co-ordinates of P2 are 5 and 5
```

# ARRAYS OF UNION VARIABLES

- Like structures we can also have array of union variables. However, because of the problem of new data overwriting existing data in the other fields, the program may not display the accurate results.
- ```
#include <stdio.h>
union POINT
{
    int x, y;
};
main()
{
    int i;
    union POINT points[3];
    points[0].x = 2;           points[0].y = 3;
    points[1].x = 4;           points[1].y = 5;
    points[2].x = 6;           points[2].y = 7;
    for(i=0;i<3;i++)
        cout<<"\n Co-ordinates of Points"<< points[i].x<< points[i].y;
    return 0;
}
```
- **OUTPUT**
- **Co-ordinates of Points[0] are 3 and 3**
- **Co-ordinates of Points[1] are 5 and 5**
- **Co-ordinates of Points[2] are 7 and 7**

# UNIONS INSIDE STRUCTURES

- union can be very useful when declared inside a structure. Consider an example in which you want a field of a structure to contain a string or an integer, depending on what the user specifies. The following code illustrates such a scenario.

## Example 8.4

Consider an example in which you want a field of a structure to contain a string or an integer, depending on what the user specifies. The following code illustrates such a scenario.

```
#include <iostream.h>
struct student
{
    union
    {
        char name[20];
        int roll_no;
    };
    int marks;
};

main()
{
    struct student stud;
    char choice;
    cout<<"\n You can enter the name or roll number of the student";
    cout<<"\n Do you want to enter the name? (Yes or No) : ";
    cin>>choice;
    if(choice=='y' || choice=='Y')
    { cout<<"\n Enter the name : ";
        Cin.getline(stud.name, 20);
    }
    else
    { cout<<"\n Enter the roll number : ";
        cin>>stud.roll_no;
    }
    cout<<"\n Enter the marks : ";
    cin>>stud.marks;
    if(choice=='y' || choice=='Y')
        cout<<"\n Name : "<<stud.name;
    else
        cout<<"\n Roll Number : "<<stud.roll_no;
        cout<<"\n Marks : "<<stud.marks;
}
```

## OUTPUT

```
You can enter the name or roll number of the student
Do you want to enter the name? (Yes or No) Y
Enter the name : Saisha
Enter the marks : 85
Name : Saisha
Marks : 85
```

# ENUMERATED DATA TYPES

- The enumerated data type is a user defined type based on the standard integer type.
- An enumeration consists of a set of named integer constants. That is, in an enumerated type, each integer value is assigned an identifier. This identifier (also known as an enumeration constant) can be used as symbolic names to make the program more readable.
- To define enumerated data types, enum keyword is used.
- Enumerations create new data types to contain values that are not limited to the values fundamental data types may take. The syntax of creating an enumerated data type can be given as below.

```
enum enumeration_name { identifier1, identifier2, ..... , identifiern };
```

- Consider the example given below which creates a new type of variable called COLORS to store colors constants.
- **enum COLORS {RED, BLUE, BLACK, GREEN, YELLOW, PURPLE, WHITE};**

In case you do not assign any value to a constant, the default value for the first one in the list - RED (in our case), has the value of 0. The rest of the undefined constants have a value 1 more than its previous one. So in our example,

- **RED = 0, BLUE = 1, BLACK = 2, GREEN = 3, YELLOW = 4, PURPLE = 5, WHITE =6**
- If you want to explicitly assign values to these integer constants then you should specifically mention those values as shown below.
- **enum COLORS {RED = 2, BLUE, BLACK = 5, GREEN = 7, YELLOW, PURPLE , WHITE = 15};**
- As a result of the above statement, now
- **RED = 2, BLUE = 3, BLACK = 5, GREEN = 7, YELLOW = 8, PURPLE = 9, WHITE = 15**

# ENUM VARIABLES

- The syntax for declaring a variable of an enumerated data type can be given as,  
`enumeration_name variable_name;`
- So to create a variable of COLORS, we may write:  
`enum COLORS bg_color;`
- Another way to declare a variable can be as illustrated in the statement below.
- `enum COLORS {RED, BLUE, BLACK, GREEN, YELLOW, PURPLE, WHITE}bg_color, fore_color;`

## USING THE TYPEDEF KEYWORD

- C permits to use `typedef` keyword for enumerated data types. For ex, if we write  
`typedef enum COLORS color;`
- Then, we can straight-away declare variables by writing  
`color forecolor = RED;`

## ASSIGNING VALUES TO ENUMERATED VARIABLES

- Once the enumerated variable has been declared, values can be stored in it. However, an enumerated variable can hold only declared values for the type. For example, to assign the color black to the background color, we will write,  
`bg_color = BLACK;`
- Once an enumerated variable has been assigned a value, we can store its value in another variable of the same type as shown below.  
`enum COLORS bg_color, border_color;`  
`bg_color = BLACK;`  
`border_color = bg_color;`

# ENUMERATION TYPE CONVERSION

- Enumerated types can be implicitly or explicitly cast. For ex, the compiler can implicitly cast an enumerated type to an integer when required.
- However, when we implicitly cast an integer to an enumerated type, the compiler will either generate an error or warning message.
- To understand this, answer one question. If we write:
- `enum COLORS{RED, BLUE, BLACK, GREEN, YELLOW, PURPLE, WHITE};`
- `enum COLORS c;c = BLACK + WHITE;`
- Here, c is an enumerate data type variable. If we write, `c = BLACK + WHITE`, then logically, it should be  $2 + 6 = 8$ ; which is basically a value of type int. However, the left hand side of the assignment operator is of the type enum COLORS. SO the statement would complain an error.
- To remove the error, you can do either of two things. First, declare c to be an int.
- Second, cast the right hand side in the following manner. :
- `c = enum COLORS(BLACK + WHITE);`

## **COMPARING ENUMERATED TYPES**

- C also allows using comparison operators on enumerated data type. Look at the following statements which illustrate this concept.
- ```
bg_color = (enum COLORS)6;
```
- ```
if(bg_color == WHITE)
```
- `fore_color = BLUE;`
- `fore_color = BLUE;`
- ```
if(bg_color == fore_color)
```
- `cout<<"\n NOT VISIBLE";`
- Since enumerated types are derived from integer type, they can be used in a switch-case statement.
- ```
enum {RED, BLUE, BLACK, GREEN, YELLOW, PURPLE, WHITE}bg_color;
```
- ```
switch(bg_color)
```
- {
- `case RED:`
- `case BLUE:`
- `case GREEN:`
- `cout<<"\n It is a primary color";`
- `break;`
- `case default:`
- `cout<<"\n It is not a primary color";`
- `break;`
- }

## **INPUT/OUTPUT OPERATIONS ON ENUMERATED TYPES**

- Since enumerated types are derived types, they cannot be read or written using formatted I/O functions available in C. When we read or write an enumerated type, we read/write it as an integer. The compiler would implicitly do the type conversion as discussed earlier.
- `enum COLORS(RED, BLUE, BLACK, GREEN, YELLOW, PURPLE, WHITE);`
- `enum COLORS c;`
- `Cin>>c;`
- `Cout<<"\n Color = "<<c;`



OXFORD  
HIGHER EDUCATION

# Object Oriented Programming with C++



Reema Thareja

OXFORD  
HIGHER EDUCATION

# Object Oriented Programming with C++

Reema Thareja

# Chapter Nine

## Classes and Objects

# INTRODUCTION

Classes form the building blocks of the object-oriented programming paradigm. They simplify the development of large and complex projects and help produce software which is easy to understand, modular, re-usable, and easily expandable. Syntactically, classes are similar to structures, the only difference between a structure and a class is that by default, members of a class are private, while members of a structure are public.

Although there is no general rule as when to use a structure and when to use a class, generally, C++ programmers use structures for holding data and classes to hold data and functions.

# SPECIFYING A CLASS

- A class is the basic mechanism to provide data encapsulation. Data encapsulation is an important feature of object-oriented programming paradigm.
- It binds data and member functions in a single entity in such a way that data can be manipulated only through the functions defined in that class.
- The process of specifying a class consists of two steps—class declaration and function definitions (Fig.).

# Class Declaration

```
class class_name  
{ private:  
    data declarations;  
    function declarations;  
public:  
    data declarations;  
    function declarations;  
};
```

(a)

```
class Rectangle  
{ private:  
    float length;  
    float breadth;  
public:  
    void get_data();  
    void show_data();  
    float area();  
};
```

(b)

The keyword `class` denotes that the `class_name` that follows is user-defined data type. The body of the class is enclosed within curly braces and terminated with a semicolon, as in structures. Data members and functions are declared within the body of the class.

# Visibility Labels

- Data and functions are grouped under two sections :-Private and Public data.
- **Private Data**:- All data members and member functions declared private can be accessed only from within the class.
- They are strictly not accessible by any entity—function or class—outside the class in which they have been declared. In C++, data hiding is implemented through the private visibility label.
- **Public Data**:- and functions that are public can be accessed from outside the class.
- By default, members of the class, both data and function, are private. If any visibility label is missing, they are automatically treated as private members—private and public.

# Defining a Function Inside the Class

- In this method, function declaration or prototype is replaced with function definition inside the class.
- Though it increases the execution speed of the program, it consumes more space.
- A function defined inside the class is treated as an inline function by default, provided they do not fall into the restricted category of inline functions.

```
class rectangle
{ private:
    float length;
    float breadth;
public:
    void get_data()
    { cout<<"\n Enter the length and breadth of the rectangle: ";
        cin>>length>>breadth;
    }
    void show_data();
    float area();
};
```

## Example:- Function inside the class

# Defining a Function Outside the Class

- `class_name::` and `function_name` tell the compiler that scope of the function is restricted to the `class_name`. The name of the `::` operator is scope resolution operator.
- The importance of the `::` is even more prominent in the following cases.
- When different classes in the same program have functions with the same name. In this case, it tells the compiler which function belongs to which class to restrict the non-member functions of the class to use its private members.
- It allows a member function of the class to call another member function directly without using the dot operator.

```
return_type function_name(list of arguments)
{
FUNCTION BODY
}
```

(a)

Scope resolution operator

```
return_type class_name :: function_name(list of arguments)
{
FUNCTION BODY
}
```

(b)

```
float Rectangle :: area(void)
{   return length * breadth;
}
```

Example:-Defining a function outside the class

# CREATING OBJECTS

- To use a class, we must create variables of the class also known as objects.
- The process of creating objects of the class is called class instantiation.
- Memory is allocated only when we create object(s) of the class.
- The syntax of defining an object (variable) of a class is as follows:-`class_name object_name;` .

```
class Rectangle
{ private:
    float length;
float breadth;
public:
    void get_data();
    void show_data();
    float area();
}rect1, rect2, rect3;
```

```
Rectangle rect;
```

## Example:- Object Creation.

# ACCESSING OBJECT MEMBERS

- There are two types of class members—private and public.
- While private members can be only accessed through public members, public members, on the other hand, can be called from main() or any function outside the class.
- The syntax of calling an object member is as follows:-  
`object_name.function_name(arguments);`

```
class Sample
{  private:
    int a;
void func1()
    {SOME CODE}
public:
    int b;
void func2();
    {SOME CODE}
};
main()
{  Sample s;
    s.a = 1;          //Wrong, a is private, cannot be accessed in main()
    s.func1();        //Wrong, func1() is a private function
    s.b = 2;          //Right, b is a public data
    s.func2();        //Right, func2() is a public function
    a = 3;            //Wrong, a is private data
    b = 4;            //Wrong b cannot be accessed without object name
    func2();          //Wrong can't be accessed without object name and '.'
}
```

## Example:- Calling of object.

# Static Data Members

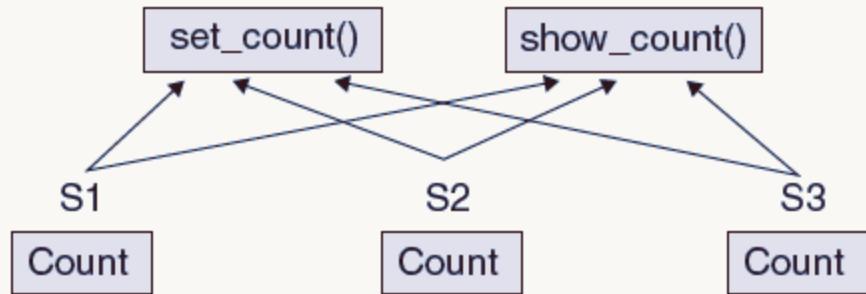
- When a data member is declared as static, the following must be noted:-
- Irrespective of the number of objects created, only a single copy of the static member is created in memory.
- All objects of a class share the static member.
- All static data members are initialized to zero when the first object of that class is created.
- Static data members are visible only within the class but their lifetime is the entire program.

# Static Data Members

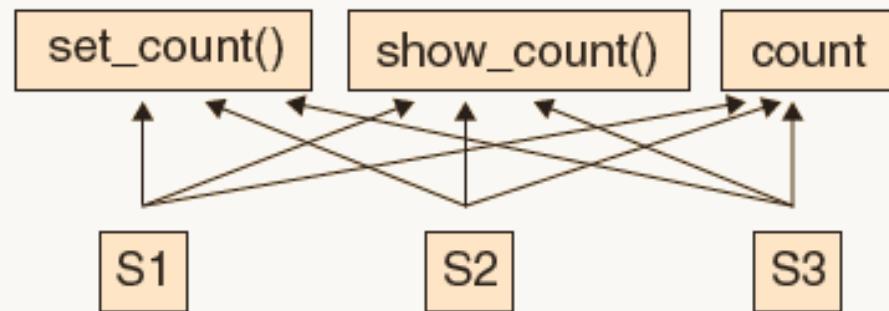
- **Relevance:-** Static data members are usually used to maintain values that are common for the entire class. For example, to keep a track of how many objects of a particular class has been created.
- **Place of Storage:-** Although static data members are declared inside a class, they are not considered to be a part of the objects. Consequently, their declaration in the class is not considered as their definition. A static data member is defined outside the class. This means that even though the static data member is declared in class scope, their definition persist in the entire file. A static member has a file scope.

# Static Data Members contd.

- However, since a static data member is declared inside the class, they can be accessed only by using the class name and the scope resolution operator.



**Figure 9.7** Memory allocation for the class sample having non-static data members



**Figure 9.8** Memory allocation for the class Sample having static data member

**Example:- Static data members**

# Static Member Functions

- It can access only the static members—data and/or functions—declared in the same class.
- It cannot access non-static members because they belong to an object but static functions have no object to work with.
- Since it is not a part of any object, it is called using the class name and the scope resolution operator.
- As static member functions are not attached to an object, the this pointer does not work on them.
- A static member function cannot be declared as virtual function.
- A static member function can be declared with const, volatile type qualifiers.

```
#include<iostream.h>
class ID_Generator
{ private:
    static int next_ID;           // next_ID is a static data member
public:
    static int GenNextID()        // GenNextID is a static member function
    {   cout<<"\n NEXT ID = "<<next_ID++; }
};
int ID_Generator :: next_ID = 1;
main()
{   for(int i=0;i<5;i++)
    ID_Generator :: GenNextID();
}
```

## OUTPUT

```
NEXT ID = 1
NEXT ID = 2
NEXT ID = 3
NEXT ID = 4
NEXT ID = 5
```

# Static Object

- To initialize all the variables of an object to zero, we have two techniques:-
- Make a constructor and explicitly set each variable to 0. We will read about it in the next chapter.
- To declare the object as static, when an object of a class is declared static, all its members are automatically initialized to zero.

# ARRAY OF OBJECTS

- Just as we have arrays of basic data types, C++ allows programmers to create arrays of user-defined data types as well.
- We can declare an array of class student by simple writing `student s[20];` // assuming there are 20 students in a class.
- When this statement gets executed, the compiler will set aside memory for storing details of 20 students.
- This means that in addition to the space required by member functions, `20 * sizeof(student)` bytes of consecutive memory locations will be reserved for objects at the compile time.
- An individual object of the array of objects is referenced by using an index, and the particular member is accessed using the dot operator. Therefore, if we write, `s[i].get_data()` then the statement when executed will take the details of the *i th student*.

# OBJECTS AS FUNCTION ARGUMENTS

- Pass-by-value In this technique, a copy of the actual object is created and passed to the called function.
- Therefore, the actual (object) and the formal (copy of object) arguments are stored at different memory locations.
- This means that any changes made in formal object will not be reflected in the actual object.
- Pass-by-reference In this method, the address of the object is implicitly passed to the called function.
- Pass-by-address In this technique, the address of the object is explicitly passed to the called function.

```
#include<iostream.h>
class myClass
{ private:
    int num;
public:
    void get_data()
    {   cout<<"\n Enter a number : ";
        cin>>num;
    }
    void set_data(myClass o2)      // function accepts object
    {   num = o2.num;
    }
    // function accepts address of object
    void set_data(int a, myClass &o2)
    {   num = o2.num;
    }
    void set_data(myClass *o2) // function accepts pointer
    {   num = o2->num;
    }
    void show_data()
    {   cout<<"\n Num = "<<num;
    }
};
main ()
{   myClass o1, o2, o3, o4;
    int a=0;
    o2.get_data();
    o1.set_data(o2);           //call by value
    o3.set_data(&o2);         //call by pointer
    o4.set_data(a, o2);       //call by reference
    o1.show_data();
    o2.show_data();
    o3.show_data();
}
```

# RETURNING OBJECTS

- You can return an object using the following three methods:-
- Returning by value which makes a duplicate copy of the local object.
- Returning by using a reference to the object. In this way, the address of the object is passed implicitly to the calling function.
- Returning by using the ‘this pointer’ which explicitly sends the address of the object to the calling function.

```
Complex Complex :: add(Complex &c2)
{   Complex c3;
    c3.real = real + c2.real;
    c3.imag = imag + c2.imag;
    return c3;
}
```

```
Complex &Complex :: compare(Complex &c2)
{   if(real < c2.real)
    return c2;
    else if(real == c2.real)
    {   if(imag < c2.imag)
        return c2;
        else
        return *this;
    }
}
```

## Example:- Returning an object.

# this POINTER

- C++ uses the `this` keyword to represent the object that invoked the member function of the class.
- **this pointer** is an implicit parameter to all member functions and can, therefore, be used inside a member function to refer to the invoking object.

```
#include<iostream.h>
class Sample
{ private:
    int a;
public:
void set_data(int num)
{   a = num;   }
void show_data()
{   cout<<a; }
};
main()
{   Sample s;
    int num = 10;

    s.set_data(num);
    s.show_data();
}
```

## OUTPUT

10

# CONSTANT MEMBER FUNCTION

- Constant member functions are used when a particular member function should strictly not attempt to change the value of any of the class's data member.

**return\_type function\_name(arguments) const**

- Note that the keyword const should be suffixed in function header during declaration and definition.

```
#include<iostream.h>
class Employee
{ private:
    int emp_no;
    float sal;
public:
    void set_data(int e, float s);
    void show_data() const; // constant member function, cannot modify data values
};
void Employee :: set_data(int e, float s)
{   emp_no = e;
    sal = s;
}
void Employee :: show_data() const
{   cout<<"\n EMP NO. = "<<emp_no<< " \n Salary = "<<sal;
}
main()
{   Employee e;
    e.set_data(1, 10000);
    e.show_data();
}
```

# CONSTANT PARAMETERS

- When we pass constant objects as parameters, then members of the object—whether private or public—cannot be changed.
- This is even more important when we are passing an object to a non-member function as in,

void increment(Employee e, float amt)

```
{ e.sal += amt;  
}
```

- Hence, to prevent any changes by non-member of the class, we must pass the object as a constant object. The function will then receive a read-only copy of the object and will not be allowed to make any changes to it. To pass a const object to the increment function, we must write:-

**void increment(Employee const e, float amt);**

**void increment(Employee const &e, float amt);**

```
#include<iostream.h>
class Array
{ private:
    int *arr; // arr is a pointer to integer
    int size;
public:
    void get_data(int n);
    int get_sum();
    void display_data();
};
void Array :: get_data(int n)
{   size = n;
    arr = new int [size];      //dynamically allocate memory for array
    for(int i=0;i<size;i++)
        cin>>arr[i];
}
void Array :: display_data()
{   for(int i=0;i<size;i++)
    cout<<"\t"<<arr[i];
    cout<<"\n Sum of elements = "<<get_sum();
    cout<<"\n Average of numbers = "<<float(get_sum())/size;
}
int Array :: get_sum()
{   int sum = 0;
    for(int i=0;i<size;i++)
        sum += arr[i];
    return sum;
}
main()
{   Array a; //a is an object of class Array
    int n;
    cout<<"\n Enter the number of elements : ";
    cin>>n;
    a.get_data(n);
    a.display_data();
}
```

# LOCAL CLASSES

- A local class is the class which is defined inside a function. Till now, we were creating global classes since these were defined above all the functions, including main().
- However, as the name implies, a local class is local to the function in which it is defined and therefore is known or accessible only in that function.
- There are certain guidelines that must be followed while using local classes, which are as follows:-
- Local classes can use any global variable but along with the scope resolution operator.

# LOCAL CLASSES

- Local classes can use static variables declared inside the function.
- Local classes cannot use automatic variables.
- Local classes cannot have static data members.
- Member functions must be defined inside the class.
- Private members of the class cannot be accessed by the enclosing function

```
#include<iostream.h>
int NUM;
main()
{   class Sample
    {   private:
        int a;
        public:
Programming
Tip: Making an
object of a local
class outside
the function
in which it is
defined will
result in a
compilation
error.
            void get_data()
            {       cout<<"\n Enter the value of a : ";
                    cin>>a;
                    cout<<"\n Enter the value of global variable : ";
                    cin>>::NUM;
            }
            void show_data()
            {       cout<<"\n Class Private Data Member = "<<a;
                    cout<<"\n Global Variable = "<<::NUM;
            }
        };
        Sample s;
        s.get_data();
        s.show_data();
    }
```

# NESTED CLASSES

- C++ allows programmers to create a class inside another class. This is called nesting of classes.
- When a class B is nested inside another class A, class B is not allowed to access class A's members. The following points must be noted about nested classes:-
  - A nested class is declared and defined inside another class.
  - The scope of the inner class is restricted by the outer class.
  - While declaring an object of inner class, the name of the inner class must be preceded with the name of the outer class.

```
#include<iostream.h>
class A
{ public:
    class B
    { private:
        int num;
    public:
        void large(int x, int y)
        {   num = x;
            if(num<y)
                num = y;
        }
        void show_data()
        {   cout<<"\n Large = "<<num;      }
    };
main()
{   A::B b;
    b.large(100, 200);
    b.show_data();
}
```

## OUTPUT

Large = 200

# COMPLEX OBJECTS ( OBJECT COMPOSITION)

- Complex objects are objects that are built from smaller or simpler objects
- In OOP, it is used for objects that have a has-a relationship to each other. For ex, a car has-a metal frame, has-an engine, etc.
- **Benefits**
- Each individual class can be simple and straightforward.
- A class can focus on performing one specific task.
- The class is easier to write, debug, understand, and usable by other programmers.

# COMPLEX OBJECTS ( OBJECT COMPOSITION)

- While simpler classes can perform all the operations, the complex class can be designed to coordinate the data flow between simpler classes.
- It lowers the overall complexity of the complex object because the main task of the complex object would then be to delegate tasks to the sub-objects, who already know how to do them.

```
#include<iostream.h>
class One
{ private:
    int num;
public:
    void set(int i)
    {   num = i;
    }
    int get()
    {   return num;
    }
};
class Two
{ public:
    One O;
    void show()
    {   cout<<"\n Number = "<<O.get();
    }
};
void main()
{   Two T;
    T.O.set(100);
    T.show();
}
```

## OUTPUT

Number = 100

# EMPTY CLASSES

- An empty class is one in which there are no data members and no member functions. These type of classes are not frequently used. However, at times, they may be used for exception handling, discussed in a later chapter.
- The syntax of defining an empty class is as follows:- **class class\_name{};** .

```
#include<iostream.h>
class Empty{};
main()
{   Empty e;
    cout<<"\n Size of empty class = "<<sizeof(Empty);
    cout<<"\n Size of empty class's object = "<<sizeof(e);
    cout<<"\n Address of empty class's object = "<<hex<<&e;
}
```

#### OUTPUT

```
Size of empty class = 1
Size of empty class's object = 1
Address of empty class's object = 0x844ffee4c
```

## Example:- Empty Classes

# FRIEND FUNCTION

- A friend function of a class is a non-member function of the class that can access its private and protected members.
- To declare an external function as a friend of the class, you must include function prototype in the class definition. The prototype must be preceded with keyword friend.
- The template to use a friend function can be given as follows:

**class class\_name**

{ -----

-----

**friend return\_type function\_name(list of arguments);**

# FRIEND FUNCTION contd.

```
-----  
};  
return_type function_name(list of arguments)  
{ -----  
-----  
}
```

- Friend function is a normal external function that is given special access privileges.
- It is defined outside that class' scope. Therefore, they cannot be called using the ‘.’ or ‘->’ operator. These operators are used only when they belong to some class.

# FRIEND FUNCTION

- While the prototype for friend function is included in the class definition, it is not considered to be a member function of that class.
- The friend declaration can be placed either in the private or in the public section.
- A friend function of the class can be member and friend of some other class.
- Since friend functions are non-members of the class, they do not require this pointer. The keyword friend is placed only in the function declaration and not in the function definition.
- A function can be declared as friend in any number of classes.
- A friend function can access the class's members using directly using the object name and dot operator followed by the specific member.

```
#include<iostream.h>
class Distance
{ private:
    int meters;
    int kms;
    friend float convert_meters(Distance &d);
public:
    void get_data()
    { cout<<"\n Enter kms and metres : ";
      cin>>kms>>meters;
    }
};
float convert_meters(Distance &d)
{ return ((d.kms * 1000) + d.meters);
}
main()
{ Distance d;
  d.get_data();
  cout<<"\n Distance in meters = "<<convert_meters(d);
}
```

## Example:- Friend function

# FRIEND CLASS

- A friend class is one which can access the private and/or protected members of another class.
- To declare all members of class A as friends of class B, write the following declaration in class A.  
friend class B;

```
class B;  
class A  
{   friend class B;  
    data members  
    member functions  
};  
class B  
{   data members  
    member functions  
};
```

# FRIEND CLASS

- Similar to friend function, a class can be a friend to any number of classes.
- When we declare a class as friend, then all its members also become the friend of the other class.
- You must first declare the friend becoming class (forward declaration).
- A friendship must always be explicitly specified. If class A is a friend of class B, then class B can access private and protected members of class B. Since class B is not declared a friend of class A, class A cannot access private and protected members of class B.

# FRIEND CLASS contd.

- A friendship is not transitive. This means that friend of a friend is not considered a friend unless explicitly specified. For example, if class A is a friend of class B and class B is a friend of class C, then it does not imply—unless explicitly specified—that class A is a friend of class C.

# BIT-FIELDS IN CLASSES

- It allow users to reserve the exact amount of bits required for storage of values.
- C++ facilitates users to store integer members into memory spaces smaller than the compiler would ordinarily allow. These space-saving members are called bit fields. In addition to this, C++ permits users to explicitly declare width in bits.
- The syntax for specifying a bit field can be given as follows:

```
class Test
{   unsigned short a: 2;
    unsigned short b: 1;
    int c: 7;
    unsigned short d: 4;
};
```

**type:-specifier declarator: constant-expression**

```
#include<iostream.h>
#define MATHS 0
#define COMPUTER 1
#define ENGLISH 2
#define IP 1
#define DU 2
#define JNU 3
#define DTU 4
#define AU 5
class Choice
{ private:
    unsigned course : 2;
    unsigned university : 3;
public:
    void set_data()
    {   course = COMPUTER;
        university = DTU;
    }
    void show_data()
    {   cout<<"\n COURSE = "<<course;
        cout<<"\n UNIVERSITY = "<<university;
    }
};
main()
{   Choice c;
    cout<<"\n Size of Choice's object = "<<sizeof(c);
    c.set_data();
    c.show_data();
}
```

# Declaring and Assigning Pointer to Data Members of a Class

Recollect that the syntax for declaring a pointer to normal variable is as follows:

```
data_type *ptr_var;
```

The syntax for declaring a pointer to a class data member is as follows:

```
data_type class_name :: *ptr_var;
```

The syntax for assigning an address to a normal pointer variable is

```
ptr_var = &variable;
```

Similarly, the syntax for assigning the address of a class's data member to a pointer pointing to it is

```
ptr_var = &class_name :: data_member_name;
```

We can declare and assign pointer variables in the same line by writing

```
data_type *ptr_var = &variable;
```

Similarly, we can declare and assign a pointer to a class data member by writing

```
data_type class_name :: *ptr_var = &class_name :: data_member_name;
```

# Accessing Data Members Using Pointers

To access a normal variable of a class we write

`obj.data_member_name;` or `obj_ptr->data_member_name`, if we have a pointer to object.

Similarly, to access the same data member using a pointer to the data member, we have to dereference the pointer and write

`obj.*ptr_var;` or `obj_ptr->*ptr_var;`

```
#include<iostream.h>
class Test
{ public:
    int x;
    void show_data();
};
void Test :: show_data()
{ cout<<"\n x = "<<x;      }
main()
{ Test t; // create object
    int Test :: *ptr = &Test :: x; //create pointer to member
    t.*ptr = 20;      // access member through pointer
    t.show_data();
    Test *tp = new Test; // dynamically allocate memory for object
    tp->*ptr = 80; //using pointer to member
    tp->show_data();
}
```

# Pointer to Member Functions

Pointers can also be used to point to member functions of a class. The syntax for declaring and assigning a pointer to member function can be given as follows:

```
return_type(class_name :: *ptr_name)(arg_type) :: & clas_name :: function_name;
```

We can also separate declaration and assignment by writing,

```
return_type(class_name :: *ptr_name)(arg_type);  
ptr_name = & clas_name :: function_name;
```

The general format to call a function using a pointer to member function is

```
(object_name.*ptr_name)(args);
```

```
#include<iostream.h>  
class Test  
{ public:  
    void show_msg();  
};  
void Test :: show_msg()  
{     cout<<"\n Hello World !!!";      }  
main()  
{     void (Test :: *fp)(void);  
     fp = &Test :: show_msg;  
     Test t;  
     (t.*fp)();  
}
```



OXFORD  
HIGHER EDUCATION

# Object Oriented Programming with C++



Reema Thareja

OXFORD  
HIGHER EDUCATION

# Object Oriented Programming with C++

Reema Thareja

# Chapter Ten

## Constructors and Destructors

# CONSTRUCTOR

A constructor is a special member function of a class which is automatically invoked at the time of creation of an object to initialize or construct the values of data members of the object. The name of the constructor is the same as that of the class to which it belongs.

- A constructor must be declared in the public section.
  - It should not be explicitly called because a constructor is automatically invoked when an object of a class is created.
  - A constructor can never return any value; therefore, unlike a normal function, a constructor does not have any return value (not even void ).
  - A constructor cannot be inherited and virtual.
  - The address of a constructor cannot be referred to in programs.
- Therefore, pointers and references do not work with constructors.

# CONSTRUCTOR

- A **constructor** cannot be declared as static, volatile, or const .
- Like a normal function, a constructor function can also be overloaded.
- Like a normal function, a constructor function can also have default arguments.
- Like a normal class member function, a constructor can be either defined inside the class or outside the class.

# Dummy Constructor (Do Nothing Constructor)

In Chapter 9, we had been writing programs without any constructor. In such cases, the C++ run time mechanism calls a dummy constructor which does not perform any action. Here, action means does not initialize any data member and thus, the variables acquire a garbage (irrelevant) value.

## Example:- Dummy constructor

```
#include<iostream.h>
class Numbers
{ private:
    int x;
public:
    void show_data()
    { cout<<"\n x = "<<x; }
};

main()
{ Numbers N; // Dummy constructor is called
N.show_data();
}
```

### OUTPUT

```
x = 3983 (garbage value)
```

# Default Constructor

- A constructor that does not take any argument is called a default constructor.
- The default constructor simply allocates storage for the data members of the object.

## Example:- Default constructor

```
#include<iostream.h>
class Numbers
{ private:
    int x;
public:
    Numbers()      // Default Constructor
    {   x = 0;   }
    void show_data()
    {   cout<<"\n x = "<<x;   }
};

main()
{   Numbers N;      // Default constructor is called
    N.show_data();
}
```

### OUTPUT

```
x = 0
```

# Parameterized Constructor

- A constructor that accepts one or more parameters is called a parameterized constructor.
- The program code given here uses a parameterized constructor to initialize the data member of the class.

## Example:- Parameterized Constructor

```
#include<iostream.h>
class Numbers
{ private:
    int x;
public:
    Numbers(int i)      // Parameterized Constructor
    {   x = i;    }      // equivalent to writing this->x = i;
    void show_data()
    {   cout<<"\n x = "<<x;    }
};
main()
{   Numbers N(10);      // Parameterized constructor is called
    N.show_data();
}
```

### OUTPUT

x = 10

# Copy Constructor

- A copy constructor takes an object of the class as an argument and copies data values of members of one object into the values of members of another object.
- Since it takes only one argument, it is also known as a one argument constructor.
- The primary use of a copy constructor is to create a new object from an existing one by initialization.
- For this, the copy constructor takes a reference to an object of the same class as an argument.

## Example:- Copy constructor

```
#include<iostream.h>
class Numbers
{ private:
    int x;
public:
    Numbers(Numbers & N)      // Copy Constructor
    { x = N.x; }
    Numbers(int i)
    { x = i; }
    void show_data()
    { cout<<"\n x = "<<x; }
};

main()
{ Numbers N1(20);           // Parameterized Constructor called
    Numbers N2(N1);         // Copy constructor is called
    N2.show_data();
    Numbers N3 = N1;         // Copy constructor called
    N3.show_data();
}
```

### OUTPUT

```
x = 20
x = 20
```

# Dynamic Constructor

- Dynamic constructors, as the name suggests, are those constructors in which memory for data members is allocated dynamically.
- Dynamic constructor enables the program to allocate the right amount of memory to data members of the object during execution.
- This is even more beneficial when the size of data members is not the same each time the program is executed.
- The memory allocated to the data members is released when the object is no longer required and when the object goes out of scope.

```
#include<iostream.h>
class Array
{ private:
    int *arr;
    int n;
public:
    Array()
    {   n = 0;      }
    Array(int);
    void show_data();
};
Array :: Array(int num)
{   n = num;
    arr = new int [n]; // memory allocated for array dynamically
    cout<<"\n Enter the elements : ";
    for(int i=0;i<n;i++)
        cin>>arr[i];
}
void Array :: show_data()
{   for(int i=0;i<n;i++)
    cout<<" " <<arr[i];
}
main()
{   int size;
    cout<<"\n Enter the size of the array : ";
    cin>>size;
    Array Arr(size); //calls constructor and allocates memory
    Arr.show_data();
}
```

# CONSTRUCTOR WITH DEFAULT ARGUMENTS

- Like other functions, constructors can also have default arguments.
- When an object of a class is created, the C++ compiler calls the suitable constructor for initializing that object.

## Programming

**Tip:** A compiler error will be generated if you try to return any value from a constructor or a destructor. You can call a constructor from a destructor.

```
#include<iostream.h>
class Student
{private:
    int roll_no;
    int marks;
public:
    Student()
    {   roll_no = 0;
        marks = 0;
    }
    Student(int r, int m = 0)
    {   roll_no = r;
        marks = m;
    }
    void show_data()
    {   cout<<"\n ROLL NO. = "<<roll_no;
        cout<<"\t MARKS = "<<marks;
    }
};
main()
{   Student S1;           // default constructor called
    S1.show_data();
    Student S2(3);
    S2.show_data();
    Student S3(05, 98);
    S3.show_data();
}
```

# CONSTRUCTOR OVERLOADING

When a class has multiple constructors, they are called overloaded constructors. Some important features of overloaded constructors are as follows:

- They have the same name; the names of all the constructors is the name of the class.
- Overloaded constructors differ in their signature with respect to the number and sequence of arguments passed.
- When an object of the class is created, the specific constructor is called.

# DESTRUCTORS

- Like a constructor, a destructor is also a member function that is automatically invoked.
- However, unlike the constructor which constructs the object, the job of destructor is to destroy the object.
- For this, it de-allocates the memory dynamically allocated to the variable(s) or perform other clean up operations.

# DESTRUCTORS

- The name of the destructor is also the same as that of the class. However, the destructor's name is preceded by the tilde symbol ‘~’.
- A destructor is called when an object goes out of scope.
- A destructor is also called when the programmer explicitly deletes an object using the delete operator.
- Like a constructor, a destructor is also declared in the public section.
- The order of invoking a destructor is the reverse of invoking a constructor.
- Destructors do not take any argument and hence cannot be overloaded.

# DESTRUCTORS

- A destructor does not return any value.
- A destructor must be specifically defined to free (de-allocate) the resources such as memory and files opened that have been dynamically allocated in the program.
- The address of a destructor cannot be accessed in the program.
- An object with a constructor or a destructor cannot be used as a member of a union.
- Constructors and destructors cannot be inherited. On Inheritance , unlike constructors, destructors can be virtual.

**Programming**  
**Tip:** Deleting  
an object more  
than once is a  
serious error.

```
#include<iostream.h>
class Sample
{ private:
    int x;
public:
    Sample(int n)
    { x = n;
        cout<<"\n Constructor Called for object with value : "<<x;
    }
    ~Sample()
    { cout<<"\n Destructor Called for object with value : "<<x;
    }
};
main()
{ Sample S1(1);
    Sample S2(2);
    Sample S3(3);
}
```

## OUTPUT

```
Constructor Called for object with value : 1
Constructor Called for object with value : 2
Constructor Called for object with value : 3
Destructor Called for object with value : 3
Destructor Called for object with value : 2
Destructor Called for object with value : 1
```

# OBJECT COPY

- Shallow copy In this method, all the members of A will be copied into members of B . If the member holds a memory address, it copies the memory address, and if the member holds a value of primitive type it copies the value of the primitive type.
- The disadvantage of this method is that if the memory address of a B 's data member is modified, then the memory address that A 's member point also gets modified. Moreover, it just copies the address, it neither allocates memory nor copies value.
- Note that the default copy constructor and default assignment operators use the shallow copy, which is also known as a member-wise copy.

# OBJECT COPY

- In a shallow copy, C++ copies each member of the class individually using the assignment operator. It is the best method for simple classes that do not contain any dynamically allocated memory.

# DEEP COPY

- **Deep copy :**In this method, the data is actually copied. Therefore, gives different result from shallow copy. The main advantage is that A and B do not depend on each other. However, it is a slow method.
- **Lazy copy :**This method has advantages of both the previous methods.
- During initial copying, it uses shallow copy and also uses a counter to track how many objects share the data.
- When an attempt is made to modify the object, it can determine if the data is shared and can then go for deep copy, if necessary.

# CONSTANT OBJECTS

- The syntax for defining a constant object is as follows:  
`class_nameconstobject_name(parameter);`
- Constant object can be initialized only by a constructor. This means that a constant object is initialized during its creation.
- Data members of the constant object cannot be modified by any member function. This means that member function can only read the values of data members and not modify them.
- Constant objects behave as read only object. Their data members are read only members.

## Example:- Constant Objects

```
#include<iostream.h>
#include<string.h>
class Quotation
{ private:
    char *quote;
    char *author;
public:
    Quotation(char *q, char *a)
    {   quote = new char[strlen(q) + 1];
        strcpy(quote, q);
        author = new char[strlen(a) + 1];
        strcpy(author, a);
    }
    Voidshow_details()const
    {   cout<<"\n"<<quote<<" - "<<author;
    }
};
main()
{   Quotation constQT("Jai Hind", "NetajiSubhash Chandra Bose");
    QT.show_details();
}
```

### OUTPUT

Jai Hind - NetajiSubhash Chandra Bose

# ANONYMOUS OBJECTS

- A nameless object is created by using the following syntax:  
`class_name (arguments if any);`
- Anonymous objects are primarily used to pass or return objects to/from functions without having to create temporary objects unnecessarily. However, an important point to note here is that anonymous objects can only be passed or returned by value.

## Example:- Anonymous Objects

```
#include<iostream.h>
class Number
{ private:
    int x;
public:
Number()
{ x = 0; }
Number(int n)
{   x = n;
}
int return_data()
{   return x;
}
friend Number add(Number &, Number &);
};
Number add(Number &N1, Number &N2)
{   return Number(N1.x + N2.x);
}
main()
{   Number N1(2);
    Number N2(5);
    cout<<"\n x = "<<add(N1, N2).return_data();
}
```

### OUTPUT

X = 7

# ANONYMOUS CLASSES

- Anonymous class also called a nameless class is a class that is defined without any name. A class can be defined as anonymous when the class has a very short body only one instance of the class is needed. However, there are some constraints that must be kept in mind while using anonymous classes.
- These restrictions are as follows:-
- Anonymous classes cannot have a constructor.
- Anonymous classes cannot have a destructor.
- Anonymous classes cannot be passed as arguments to functions.
- Anonymous classes cannot be used to return values from functions.

## Example:-Anonymous Class

```
#include<iostream.h>
typedef class           // class name missing
{ private:
    int x;             //data member
public:
    voidget_data()     //member function
    { cout<<"\n Enter the value of x : ";
      cin>>x;
    }
    voidshow_data()
    { cout<<"\n x = "<<x;
    }
}Number;
main()
{ Number N1; // object of a nameless class. Number is used as an alternate name
   for nameless class as we have used typedef
   N1.get_data();
   N1.show_data();
}
```

### OUTPUT

```
Enter the value of x : 7
x = 7
```



OXFORD  
HIGHER EDUCATION

# Object Oriented Programming with C++



Reema Thareja

OXFORD  
HIGHER EDUCATION

# Object Oriented Programming with C++

Reema Thareja

# Chapter Eleven

# Operator Overloading and Type Conversions

# INTRODUCTION

- The utility of operators such as +, =, \*, /, >, <, and so on is predefined in any programming language.
- Programmers can use them directly on built-in data types to write their programs.
- However, these operators do not work for user-defined types such as objects.
- Therefore, C++ allows programmers to redefine the meaning of operators when they operate on class objects.
- This feature is called *operator overloading*

# Another Form of Polymorphism

- Like function overloading, operator overloading is also a form of compile time polymorphism.
- Operator overloading is, therefore, less commonly known as operator adhoc polymorphism .
- Since different operators have different implementations depending on their arguments. Operator overloading is generally defined by the language, the programmer, or both.

# Advantages

- Operator overloading enables programmers to use notation closer to the target domain.
- For example, to add two matrices, we can simply write  $M1 + M2$ , rather than writing `M1.add(M2)`.
- With operator overloading, a similar level of syntactic support is provided to user-defined types as provided to the built-in types.
- In scientific computing where computational representation of mathematical objects is required, operator overloading provides great ease to understand the concept.
- Operator overloading makes the program clear.
- For example, the statement `div(mul(M1, M2), add(M1,M2))`; can be better written as  $M1 * M2 / M1+M2$

# SYNTAX

```
class class_name
{-----
    public:
        return_type operator op(arguments if any)
        {
            -----
            //Function Body
            -----
        }
    -----
};

};
```

```
Complex operator +(Complex &c2)
{
    Complex Temp;
    Temp.real = real + c2.real;
    Temp.imag = imag + c2.imag;
    return Temp;
}
```

# OPERATORS THAT CAN AND CANNOT BE OVERLOADED

The exceptional operators that cannot be overloaded are as follows:

- Scope resolution operator ()::
- Member selection operator (.)
- Member selection through a pointer to a function (.\* )
- Ternary operator (?:)

# Important Points

- Operator overloading cannot change the operation performed by an operator.
- The two operators—the assignment operator (`=`) and the address operator (`&`)—need not be overloaded.
- Operator overloading cannot alter the precedence and the associativity of operators.
- New operators such as like `**`, `<>`, `|&`, and so on cannot be created.
- When operators such as `&&`, `||`, and, are overloaded, they lose their special properties of short-circuit evaluation and sequencing.
- Overloaded operators cannot have default arguments.

# Important Points

- All overloaded operators except the assignment operator are inherited by derived classes.
- Number of operands cannot be changed.
- Overloading an operator that is not associated with the scope of a class is not permissible.

# IMPLEMENTING OPERATOR OVERLOADING

| Member function                                                                                                                                                                                                                                                                                                                                                                                                                              | Friend function                                                                                                                                                                                                                                                                                                                                      |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <ul style="list-style-type: none"><li>Number of explicit parameters is reduced by one, as the calling object is implicitly supplied as an operand.</li><li>Unary operators take no explicit parameters.</li><li>Binary operators take one explicit parameter.</li><li>Left-hand operand has to be the calling object.</li><li><code>obj2 = obj1 + 10;</code> is permissible but <code>obj2 = 10 + obj1;</code> is not permissible.</li></ul> | <ul style="list-style-type: none"><li>Number of explicit parameters is more.</li><li>Unary operators take only one parameter.</li><li>Binary operators take two parameters.</li><li>Left-hand operand need not be an object of the class.</li><li><code>obj2 = obj1 + 10;</code> as well as <code>obj2 = 10 + obj1;</code> is permissible.</li></ul> |

# Using a member function to overload Unary Operator

```
return_type operator op ()
```

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

{ private:
    int x;
public:
    Number()
    { x = 0; }
    Number(int n) //parameterized constructor
    { x = n; }
    void operator -() // operator overloaded function
    { x = -x; }
    void show_data()
    { cout<<"\n x = "<<x; }
};

main()
{ Number N(7); // create object
  N.show_data();
  -N;           // invoke operator overloaded function
  N.show_data();
}
```

## OUTPUT

```
x = 7
x = -7
```

# Returning Object

```
#include<iostream.h>
class Number
{ private:
    int x;
public:
    Number()
    {   x = 0;      }
    Number(int n)
    {   x = n;      }
    Number operator -()      // operator overloading function returns an object
    {   Number temp;
        temp.x = -x;
        return temp;      //object returned
    }
    void show_data()
    {   cout<<"\n x = "<<x;      }
};
main()
{
    Number N1(-10), N2;
    N2 = -N1;      // return value assigned to another object
    N2.show_data();
}
```

# Using a Friend Function to Overload a Unary Operator

- The function will take one operand as an argument.
- This operand will be an object of the class.
- The function will use the private members of the class only with the object name.
- The function may take the object by using value or by reference.
- The function may or may not return any value.
- The friend function does not have access to the this pointer.

## Example:- Use of friend function to overload a unary operator

```
#include<iostream.h>
class Number
{ private:
    int x;
public:
    Number()
    {   x = 0;   }
    Number(int n)
    {   x = n;   }
    void show_data()
    {   cout<<"\n x = "<<x;   }
    friend Number & operator-(Number &); //friend function declared
};
Number & operator -(Number &N)
{   N.x = -N.x;           //use objectname with data member
    return N;             //return object
}
main()
{   Number N1(100), N2;
    N2 = -N1;           // overloaded operator function called
    N2.show_data();
}
```

### OUTPUT

x = -100

# OVERLOADING BINARY OPERATORS

- Binary means two and binary operators mean operators that work with two operands. Like unary operators, binary operators such as +, -, \*, /, =, <. >, !, %, ^, &&, ||, <<, and >> can also be overloaded.
- The syntax of overloading a binary operator using a member function can be given as :

return\_type operator op(arguments)

- The syntax to overload a binary operator using a friend function is :

friend return\_type operator op(arg1, arg2)

```
friend return_type operator op(arg1, arg2)
```

## Program 11.8 Write a program to add two arrays using classes and operator overloading.

**Programming Tip:** Operator overloaded function can be invoked only by an object of the class.

```
#include<iostream.h>
class Array
{
    private:
        int arr[10];
        int size;
    public:
        Array();
        Array(int);
        void show_data();
        Array operator+(Array &);

    };
Array :: Array()
{
    for(int i=0;i<10;i++)
        arr[i] = 0;
    size = 0;
}
Array :: Array(int n)
{
    size = n;
    cout<<"\n Enter the array elements ";
    for(int i=0;i<size;i++)
        cin>>arr[i];
}
void Array :: show_data()
{
    for(int i=0;i<size;i++)
        cout<<" "<<arr[i];
}
Array Array :: operator+(Array &A)
{
    Array Temp;
    Temp.size = size;
    for(int i=0;i<size;i++)
        Temp.arr[i] = arr[i] + A.arr[i];
    return Temp;
}
main()
{
    int n;
    cout<<"\n Enter the size of the arrays : ";
    cin>>n;
    Array A1(n), A2(n), A3;
    A3 = A1 + A2;
    cout<<"\n The resultant array is ";
    A3.show_data();
}
```

mming  
w  
te

# Overloading New and Delete Operators

- To allow users to allocate memory in a customized way.
- To allow users to debug the program and keep track of memory allocation and de-allocation in their programs.
- To allow users to perform additional operations while allocating or de-allocating memory.
- The syntax for overloading the new operator can be given as follows:

**void\* operator new(size\_t size);**

- `size_t` , which specifies the number of bytes of memory to be allocated.
- The return type of the overloaded new must be `void*` . The overloaded function returns a pointer to the beginning of the block of memory allocated.

The syntax for overloading the new operator can be given as follows:

```
void* operator new(size_t size);
```

Similarly, the syntax for the overloaded delete operator can be given as

```
void operator delete(void*);
```

```
#include<iostream.h>
class Array
{
    private:
        int *arr;
    public:
        void * operator new(size_t size)
        {   void *parr = ::new int[size]; // dynamicallyallocatingspaceusingoverloadednewoperator
            return parr;
        }
        void operator delete(void *parr)
        {   ::delete parr; } // dynamicallyde-allocatingspaceusingoverloadeddeleteoperator
            void get_data();
            void show_data();
};
void Array :: get_data()
{   cout<<"\n Enter the elements : ";
    for(int i=0;i<5;i++)
        cin>>arr[i];
}
void Array :: show_data()
{   cout<<"\n The array is : ";
    for(int i=0;i<5;i++)
        cout<<" "<<arr[i]
}
main()
{   Array *A = new Array; // calls the overloaded new operator
    A->get_data();
    A->show_data();
    delete A;           // calls the overloaded delete operator
}
```

# Overloading [] operator

The subscript operator—[ ]—is used to access array elements and can be overloaded to enhance its functionality with classes. The syntax of overloading the subscript operator can be given as follows:

Identifier[expression]

where `identifier` is the object of the class. The syntax is interpreted as

`identifier.operator[](expression)`

From the syntax, it is clear that the subscript operator is a binary operator in which the first operator is an object of the class and the second operand is an integer index.

**Note** The overloaded operator [ ] ( ) must be defined as a non-static member function of a class.

```
#include<iostream.h>
class Array
{
    private:
        int arr[10];
    public:
        Array();
        void get_data();
        void show_data();
        int& operator[](int i);
};
Array :: Array()
{
    for(int i=0;i<10;i++)
        arr[i] = 0;
}
void Array :: get_data()
{
    cout<<"\n Enter the array elements : ";
    for(int i=0;i<10;i++)
        cin>>arr[i];
}
void Array :: show_data()
{
    cout<<"\n The Array is : ";
    for(int i=0;i<10;i++)
        cout<< " <<arr[i];
}
int& Array :: operator[](int i)
{
    return arr[i];      //returns value at specified index
}
main()
{
    Array A;      //create object
    A.get_data();  //invokes member function
    A.show_data();
    cout<<"\n Modified Array Elements Are : ";
    for(int i=0;i<10;i++)
        cout<< " <<A[i] * 2; //invokes overloaded [] operator
}
```

**Programming**

**Tip:**

Overloaded

new operators

# Overloading () for a two-dimensional matrix

```
#include<iostream.h>
#include<stdlib.h>
class Matrix
{
    private:
        int arr[2][2];
    public:
        Matrix();
        void get_data();
        void show_data();
        int& operator()(int i, int j); // ()operator overloaded function declaration
};
Matrix :: Matrix()
{
    for(int i=0;i<2;i++)
        for(int j=0;j<2;j++)
            arr[i][j] = 0;
}
void Matrix :: get_data()
{
    cout<<"\n Enter the Matrix elements : ";
    for(int i=0;i<2;i++)
        for(int j=0;j<2;j++)
            cin>>arr[i][j];
}
void Matrix :: show_data()
{
    cout<<"\n The Matrix is : ";
    for(int i=0;i<2;i++)
    {
        cout<<"\n";
        for(int j=0;j<2;j++)
            cout<<" " <<arr[i][j];
    }
}
int& Matrix :: operator()(int i, int j) // overloaded () operator
{
    if(i<0 || i>9 || j<0 || j>9)
    {
        cout<<"\n Matrix index out of bounds";
        exit(1);
    }
    else
        return arr[i][j];
}
main()
{
    Matrix M;
    M.get_data();

    M.show_data();
    cout<<"\n Modified Matrix Elements Are : ";
    for(int i=0;i<2;i++)
    {
        for(int j=0; j<2;j++)
            cout<<" " <<M(i,j)*2; //invoking overloaded operator function
    }
}
```

# Overloading Class Member Access Operator ( -> )

- C++ allows programmers to control class member access by overloading the member access operator (>).
- The -> operator is a unary operator as it takes only one operand, that is the object of the class. The syntax for overloading the -> operator can be given as,  
where class\_ptr is a pointer of class type.
- Before overloading the class member access operator, the overloaded -> operator function must be a non-static member function of the class.

```
class_ptr *operator->()
```

# Example

```
#include<iostream.h>
class Sample
{    public:
        int num;
        Sample(int n)
        {    num = n;    }
        Sample * operator->(void)    //overloaded operator ->function
        {    return this;    }
};
main()
{    Sample *ptr = new Sample(10);
    cout<<"\n Number = "<<ptr->num;    // using normal object pointer
    Sample S(20);
    cout<<"\n Number = "<<S->num;    //using overloaded -> operator
}
```

## OUTPUT

```
Number = 10
Number = 20
```

# Overloading Input and Output Operators

- cin and cout are defined in class iostream.
- While cin is an object of istream class, cout is an object of the ostream class. We will read more on the relationship between istream, ostream, iostream, cin, and cout later in chapter File Handling.
- The insertion and extraction of operators will be overloaded by using a friend function because we need to call these functions without any object. The insertion and extraction operators must return the value of the left operand.
- The ostream or istream object—so that multiple << or >> operators may be used in the same statement. The syntax for overloading the >> function can be given as

```
friend ostream & operator << (ostream & output , My_class&obj)
{
    // code
}
```

```
#include<iostream.h>
class Date
{
    private:
        int dd, mm, yy;
    public:
        friend istream & operator >> (istream & input , Date &D)
        {   input>>D.dd>>D.mm>>D.yy;
            return input;
        }
        friend ostream & operator << (ostream & output , Date &D)
        {   cout<<D.dd<<" - "<<D.mm<<" - "<<D.yy;
            return output;
        }
};
main()
{
    Date D;
    cout<<"\n Enter the Date : ";
    cin>>D; //overloaded >> is invoked
    cout<<"\n DATE : ";
    cout<<D; //overloaded << is invoked
}
```

## OUTPUT

```
Enter the Date : 17 2 2007
DATE : 17 - 2 - 2007
```

# Conversion from Basic to Class Type

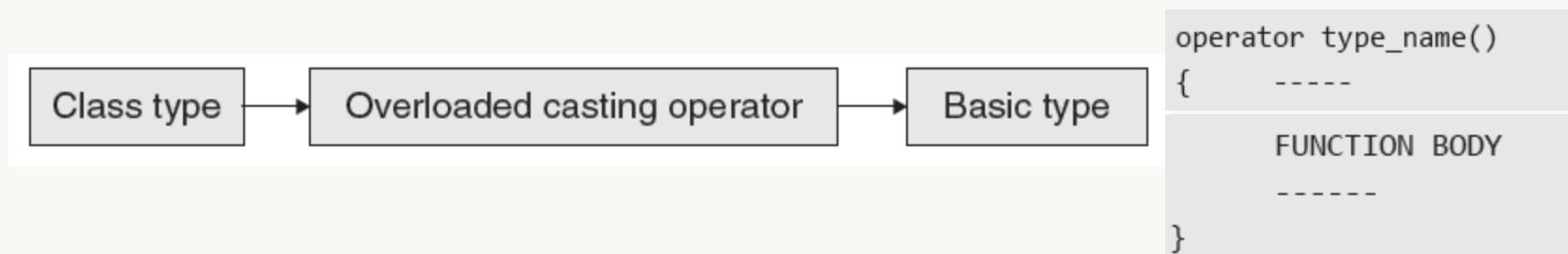
```
String(char *s)
{      strcpy(str, s);
}
```



```
String S1 = name; or
String S1(name);
```

# Conversion from Class to Basic Data Type

- The function can be declared and/or defined within the class.
- The function converts the class type into the type\_name . For example, operator int() will convert a class type to int .
- The function does not have any return type.
- The function does not take any argument.
- The casting operator is a unary operator that takes just one argument that is the object of the class that invokes this function.



```
#include<iostream.h>
class Time
{
    private:
        int h, m;
    public:
        Time()
        {   h = m = 0;   }
        Time(int t) // converting int to class
        {   h = t / 60;
            m = t % 60;
        }
        void get_data()
        {   cin>>h>>m;   }
        void show_data()
        {   cout<<h<<" hrs "<<m<<" mins";   }
        operator int() //converting class to int
        {   int t = h*60 + m;
            return t;
        }
};
main()
{
    int min;
    cout<<"\n Enter the minutes : ";
    cin>>min;
    Time T1;
    T1 = min;           //constructor called
    T1.show_data();
    cout<<"\n Enter the number of hrs and mins : ";
    T1.get_data();
    min = T1;           //casting operator called
    cout<<"\n Total Minutes = "<<min;
}
```

# Conversion from Class to Class Type

- When performing conversion in the source class, a typecast operator is used. The syntax of such an operator function can be given as

```
Obj_Dest = Obj_Source;
```

where operator is the keyword and typename may be a built-in or user-defined type. typename usually refers to the destination type.

```
operator typename()
```

**Program 11.21 Write a program to convert a rectangle into a square. Perform the conversion in the source class.**

**Programming Tip:** When performing conversion in the destination class, a single argument constructor is used.

```
#include<iostream.h>
class Square;
class Rectangle
{
    private:
        int length, breadth;
    public:
        Rectangle()
        {   length = breadth = 0;   }
        Rectangle(int l, int b)
        {   length = l;
            breadth = b;
        }
        void show_data()
        {   cout<<"\n Length = "<<length<<" and Breadth = "
<<breadth;
        }
};
class Square
{
    private:
        int side;
    public:
        Square(int s)
        {   side = s;   }
        operator Rectangle()
        {   Rectangle R(side, side);;
            return R;
        }
};
main()
{
    Rectangle R(10, 20);
    R.show_data();
    Square S(50);
    R = S;
    R.show_data();
}
```

# Conversion in Destination Class

- When performing conversion in the destination class, a single argument constructor is used.
- The argument is an object of the source class which is passed to the destination class for conversion.
- The syntax for such a constructor function can be given as

**Program 11.20 Write a program to convert a square into a rectangle. Perform the conversion in the destination class.**

```
#include<iostream.h>
class Rectangle;
class Square
{
    private:
        int side;
    public:
        Square(int s)
        {   side = s;   }
        int get_side()
        {   return side;   }
};
class Rectangle
{
    private:
        int length, breadth;
    public:
        Rectangle(int l, int b)
        {
            length = l;
            breadth = b;
        }
        Rectangle(Square S)
        {
            int sqr_side = S.get_side();
            length = breadth = sqr_side;
        }
        void show_data()
        {
            cout<<"\n Length = "<<length<< " and Breadth = "<<breadth;
        }
}
main()
{
    Rectangle R(10, 20);
    R.show_data();
    Square S(50);
    R = S;
    R.show_data();
}
```



OXFORD  
HIGHER EDUCATION

# Object Oriented Programming with C++



Reema Thareja

OXFORD  
HIGHER EDUCATION

# Object Oriented Programming with C++

Reema Thareja

# Chapter Twelve

## Inheritance and Run-Time Polymorphism

# Inheritance

- The technique of creating a new class from an existing class is called inheritance.
- The old or existing class is called the **base class** and the new class is known as the **derived class** or **sub-class**.
- The derived classes are created by first inheriting the data and methods of the base class and then adding new specialized data and functions in it.
- During the process of inheritance, the base class remains unchanged.

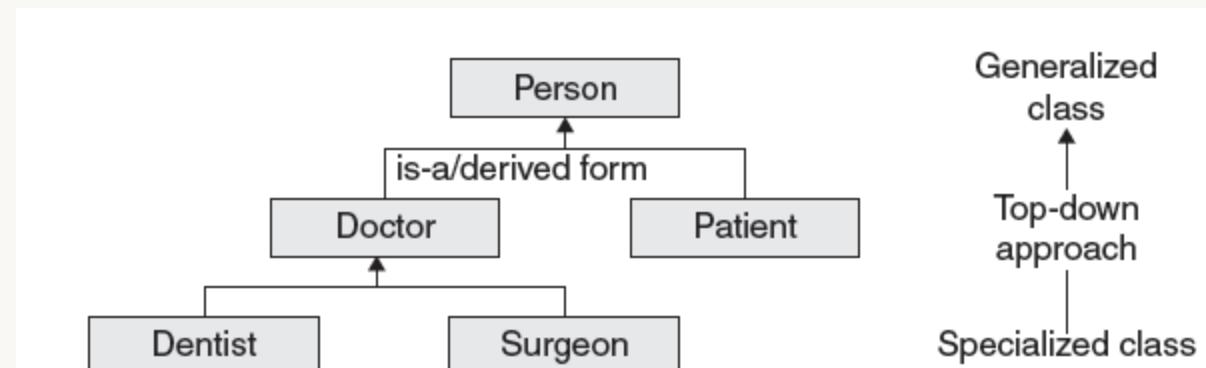


Figure 12.1 Is-a relationship between classes

# DEFINING DERIVED CLASSES

A class can be derived from one or more base classes using the following syntax:

```
class derived_class_name : [access-specifier] base_class_name
{   -----
    -----
    -----
};
```

According to the syntax, the `derived_class_name` is derived from the `base_class_name`, thereby inheriting some or all of its members. The access-specifier, also known as the visibility-mode, is optional and if present, can be either `public`, `private`, or `protected`. If no access-specifier is written, then by default, the class will be derived in `private` mode.

# ACCESS SPECIFIERS

- **Private** is the highest level of data hiding. When a base class is privately inherited by a derived class, then public members of the base class become private members of the derived class.
- The private members cannot be inherited but the derived class can access them using public member functions of the base class.
- This means that the object of a privately inherited class cannot access the inherited members.

# ACCESS SPECIFIERS contd.

- **Public** is the lowest and the most open level of data hiding.
- When a base class is publicly inherited by a derived class, then public members of the base class become public members of the derived class.
- The private members cannot be inherited but the derived class can access them using public member functions of the base class.

# ACCESS SPECIFIERS contd.

- A protected specifier lies between private and public .
- A data member or a member function declared as protected can be accessed by the class in which it is defined—as in private —and the class which is immediately derived from it.
- No other class except these two can access the protected members of a class.

| Specifier<br>or class | Same class | Derived<br>class | Any other<br>class | Friend<br>function | Friend<br>class |
|-----------------------|------------|------------------|--------------------|--------------------|-----------------|
| Private               | Yes        | No               | No                 | Yes                | Yes             |
| Protected             | Yes        | Yes              | No                 | Yes                | Yes             |
| Public                | Yes        | Yes              | Yes                | Yes                | Yes             |

# Inheriting Protected Members

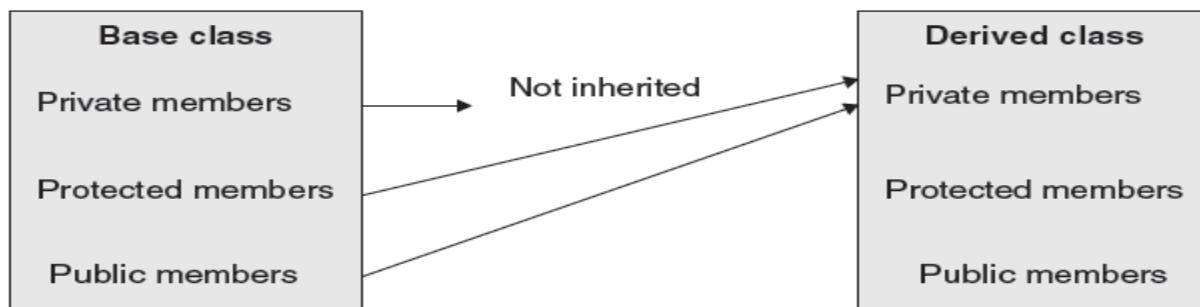


Figure 12.2 A class derived in private mode

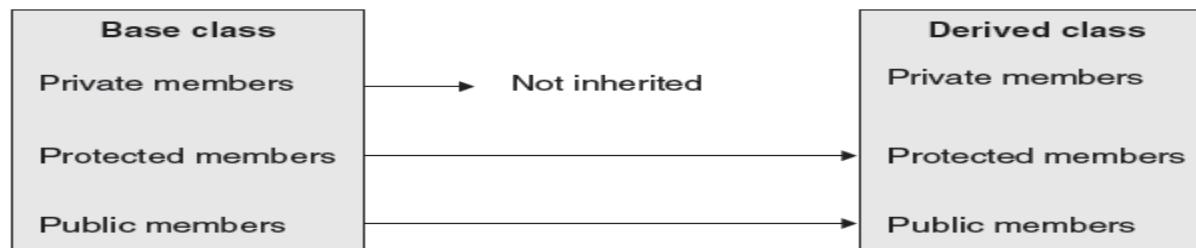


Figure 12.3 A class derived in public mode

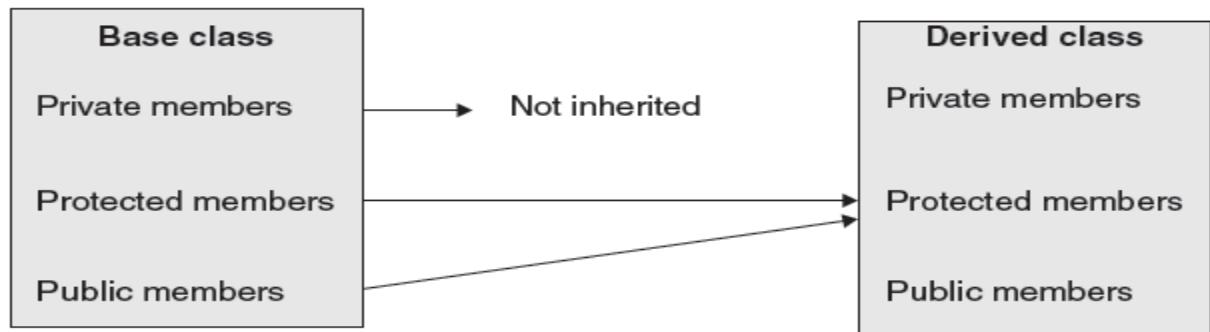
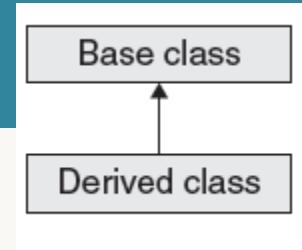


Figure 12.4 A class derived in protected mode

# SINGLE INHERITANCE

```
#include<iostream.h>
class Student
{ private:
    int roll_no;
protected:
    char course[10];
public:
    void get_rno()
    { cout<<"\n Enter the roll number : ";
      cin>>roll_no;
    }
    void show_rno()
    { cout<<"\n ROLL NO : "<<roll_no;
    }
};
class Result : public Student
{ private:
    int marks[3];
public:
    void get_data();
    int total();
    void display()
    { show_rno();
      cout<<"\n COURSE : "<<course;
      cout<<"\n TOTAL MARKS : "<<total();
    }
};
void Result :: get_data()
{ get_rno();
  cout<<"\n Enter the course : ";
  cin.ignore();
  cin.getline(course,10);
  cout<<"\n Enter marks in three subjects : ";
  for(int i=0;i<3;i++)
    cin>>marks[i];
}
int Result :: total()
{     int tot_marks = 0;
  for(int i=0;i<3;i++)
    tot_marks += marks[i];
  return tot_marks;
}
main()
{ Result R;
  R.get_data();
  R.display();
  //R.get_rno();                                //Ok
  //R.roll_no = 12;                             // not allowed
  //strcpy(R.course, "BCA") ;                   // not allowed
}
```



# CONSTRUCTORS AND DESTRUCTORS IN DERIVED CLASSES

- If the base class constructor does not take any argument, the derived class may not have a **constructor**.
- If the base class has a constructor with one or more arguments, then the derived class must have a constructor function to pass the arguments to the base class constructor.
- It is the derived class's responsibility to pass arguments to the base class because in the `main()` , we will create objects only of the derived class and not of the base class.
- If both the derived class and base class has constructors, the base class constructor is executed first and then the constructor in the derived class is executed.

# CONSTRUCTORS AND DESTRUCTORS IN DERIVED CLASSES contd.

- The order of execution of **destructors** is just the reverse of constructor.
- First, the destructor of derived class is called and then the constructor of base class is called.

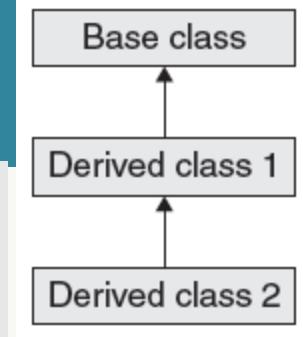
# Invoking Constructors with Arguments

```
class Student
{ private:
    int roll_no;
protected:
    char course[10];
public:
    Student(int rno, char *c)
    {   roll_no = rno;
        strcpy(course, c);
    }
    void show_rno()
    {   cout<<"\n ROLL NO : "<<roll_no;
    }
};
class Result : protected Student
{ private:
    int marks[3];
public:
    Result(int rno, char *c, int m1, int m2, int m3) : Student(rno, c)
    {   marks[0] = m1;
        marks[1] = m2;
        marks[2] = m3;
    }
    void get_data();
    int total();
    void display()
    {   show_rno();
        cout<<"\n COURSE : "<<course;
        cout<<"\n TOTAL MARKS : "<<total();
    }
};

-----  
derived_class_name(arg_list1, arg_list2):base(arg_list1)
{
    -----
    // initialize derived class members with arg_list2
    -----
}
```

# MULTI-LEVEL INHERITANCE

```
class Student
{   protected:
    int roll_no;
    char name[20];
    char course[10];
public:
    void get_data()
    {   cout<<"\n Enter roll number : ";
        cin>>roll_no;
        cout<<"\n Enter name : ";
        cin.ignore();
        cin.getline(name, 20);
        cout<<"\n Enter course : ";
        cin.getline(course, 10);
    }
    int get_rno()
    {   return roll_no;
    }
    char * get_name()
    {   return name;
    }
    char * get_course()
    {   return course;
    }
};
class Marks : public Student
{   protected:
    int marks[3];
public:
    void get_marks()
    {   cout<<"\n Enter marks in three subjects : ";
        cin>>marks[0]>>marks[1]>>marks[2];
    }
    int total()
    {   return (marks[0] + marks[1] + marks[2]);
    }
};
class Result : public Marks
{   public:
    void display()
    {   cout<<"\n ROLL NUMBER : "<<get_rno();
        cout<<"\n NAME : "<<get_name();
        cout<<"\n COURSE : "<<get_course();
        cout<<"\n TOTAL MARKS : "<<total();
    }
};
```



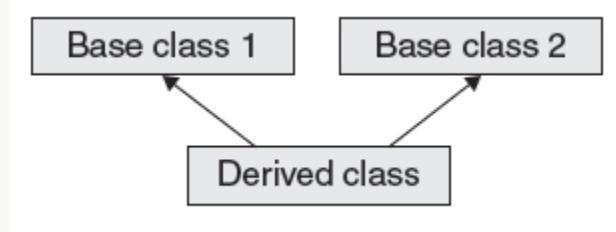
# CONSTRUCTOR IN MULTI-LEVEL INHERITANCE

```
Student(int r, char *n, char *c)
{   roll_no = r;
    strcpy(name, n);
    strcpy(course, c);
    cout<<"\n In STUDENT's constructor - Base Class";
}
```

```
Marks(int r, char *n, char *c, int m1, int m2, int m3):Student(r,n,c)
{   marks[0] = m1;
    marks[1] = m2;
    marks[2] = m3;
    cout<<"\n In MARK's Constructor - Intermediate Class";
}
```

```
Result(int r, char *n, char *c, int m1, int m2, int m3) : Marks(r, n, c, m1,
m2, m3)
{   cout<<"\n In RESULT's Constructor - Derived Class";   }
~Result()
{   cout<<"\n In RESULT's Destructor - Derived Class";   }
void display()
{   cout<<"\n ROLL NUMBER : "<<get_rno();
    cout<<"\n NAME : "<<get_name();
    cout<<"\n COURSE : "<<get_course();
    cout<<"\n TOTAL MARKS : "<<total();
}
};
```

# MULTIPLE INHERITANCE



```
class derived_class_name : visibility mode base1_
    class_name, visibility mode base1_class_
```

```
name, ..... visibility mode basen_class_name
{
    -----
}
```

# CONSTRUCTORS AND DESTRUCTORS IN MULTIPLE INHERITANCE

```
Marks(int m1, int m2, int m3)
{   marks[0] = m1;
    marks[1] = m2;
    marks[2] = m3;
    cout<<"\n In MARK's Constructor - Second Base Class";
}
```

```
Student(int r, char *n, char *c)
{   roll_no = r;
    strcpy(name, n);
    strcpy(course, c);
    cout<<"\n In STUDENT's constructor - First Base Class";
}
```

```
Result(int r, char *n, char *c, int m1, int m2, int m3) : Student(r, n, c),
Marks(m1, m2, m3)
{   cout<<"\n In RESULT's Constructor - Derived Class"; }
```

# AMBIGUITY IN MULTIPLE INHERITANCE

- In multiple inheritance, when a class inherits features from more than one base class, there may be a case when two or more classes have a function with the same name.
- In such cases, if the object of the derived class tries to access that function, then it will result in ambiguity as it will not be clear to the compiler which base's class member function should be invoked.
- The ambiguity simply means the state when the compiler is confused.

```
#include<iostream.h>
class B1
{ public:
    void show_data()
    { cout<<"\n In Base Class 1";
    }
};
class B2
{ public:
    void show_data()
    { cout<<"\n In Base Class 2";
    }
};
class D : public B1, public B2
{ public:
    D()
    { cout<<"\n In Derived Class Constructor";
    }
};
main()
{ D derived_obj;
    derived_obj.show_data(); // which one to call?
}
```

# Solution for the Ambiguity Problem

The ambiguity can be resolved by using the scope resolution operator to specify the class whose member function has to be invoked. For example, in `main()`, we should write

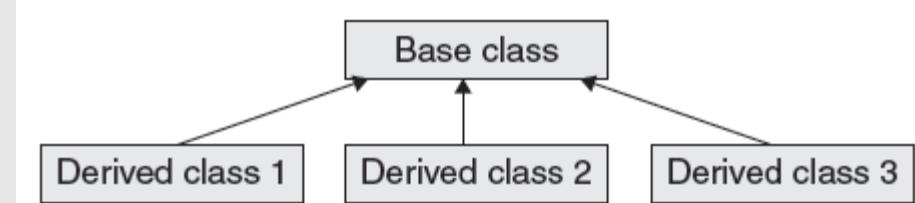
```
Derived_obj.B1::show_data(); or derived_obj.B2::show_data();
```

We can also define a function in the derived class as `void display()` which would call its base class `show_data()` function by writing

```
classD
{
public:
    void display()
    {
        B1::show_data();
        B2::show_data();
    }
};
```

# HIERARCHICAL INHERITANCE

```
class Student
{ protected:
    int roll_no;
    char name[20];
public:
    void get_Stud_Details()
    { cout<<"\n Enter the roll number : ";
      cin>>roll_no;
      cout<<"\n Enter the name : ";
      cin.ignore();
      cin.getline(name, 20);
    }
};
class Academic : public Student
{ protected:
    int marks;
    char grade;
public:
    void get_Acad_Details()
    { get_Stud_Details();
      cout<<"\n Enter the marks : ";
      cin>>marks;
      cout<<"\n Enter the grade : ";
      cin>>grade;
    }
    void show_Acad_Details()
    { cout<<"\n ROLL NUMBER : "<<roll_no;
      cout<<"\n NAME : "<<name;
      cout<<"\n MARKS : "<<marks;
      cout<<"\n GRADE : "<<grade;
    }
};
class Accounts : public Student
{ protected:
    float fees;
    char DUES;
public:
    void get_Accounts_Details()
    { get_Stud_Details();
      cout<<"\n Enter the fees : ";
      cin>>fees;
      cout<<"\n Is there any dues lest (Y/N) : ";
      cin>>DUES;
    }
    void show_Accounts_Details()
    { cout<<"\n ROLL NUMBER : "<<roll_no;
```



# CONSTRUCTORS AND DESTRUCTORS IN HIERARCHICAL INHERITANCE

- Conceptually, hierarchical inheritance is similar to single inheritance; hence, the order of execution of constructors and destructors will be same as specified in case of single inheritance.
- First, the constructor of the base class would be invoked followed by that of derived class .
- Since we have more than one derived classes from the same base class, the constructor of the base class would be invoked multiple times—once for each derived object.

# CONSTRUCTORS AND DESTRUCTORS IN HIERARCHICAL INHERITANCE contd.

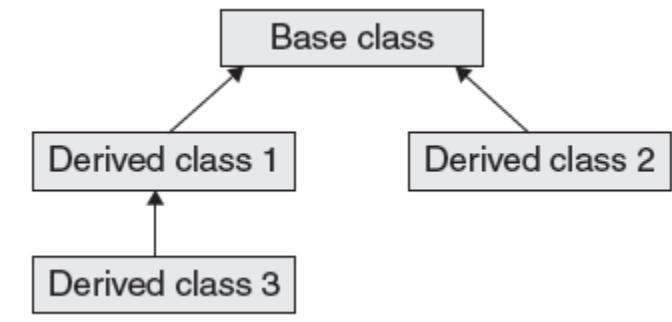
```
Student(int rno, char *n)
{   roll_no = rno;
    strcpy(name, n);
}
```

```
Academic(int rno, char *n, int m, char g):Student(rno, n) {   marks = m;
grade = g;
}
```

```
Accounts(int rno, char *n, float f, char d):Student(rno, n)
{   fees = f;
    DUES = d;
}
```

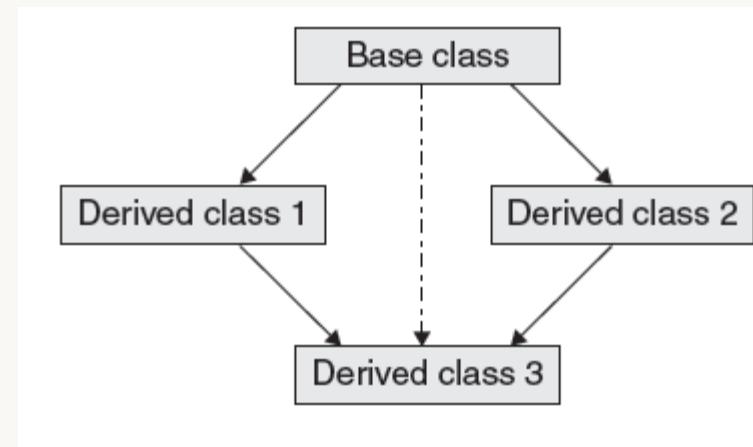
# HYBRID INHERITANCE

```
class Marks
{   protected:
    int marks[3];
public:
    void get_test_marks()
    {   cout<<"\n Enter the marks in three subjects : ";
        cin>>marks[0]>>marks[1]>>marks[2];
    }
    int total_test_marks()
    {   return (marks[0] + marks[1] + marks[2]);
    }
};
class Activities
{   protected:
    int acts[2];
public:
    void get_act_marks()
    {   cout<<"\n Enter marks in two activities : ";
        cin>>acts[0]>>acts[1];
    }
    int total_act_marks()
    {   return (acts[0] + acts[1]);
    }
};
class Student : public Marks, public Activities
{   protected:
    int roll_no;
    char name[20];
public:
    void get_details()
    {   cout<<"\n Enter the roll number : ";
        cin>>roll_no;
        cout<<"\n Enter the name : ";
        cin.ignore();
        cin.getline(name, 20);
        get_test_marks();
        get_act_marks();
    }
};
class Result : public Student
{   public:
    void get_data()
    {   get_details();   }
    void show_data()
    {   cout<<"\n ROLL NUMBER : "<<roll_no;
        cout<<"\n NAME : "<<name;
        cout<<"\n TOTAL MARKS : "<<total_test_marks() + total_act_marks();
    }
};
```



# MULTI-PATH INHERITANCE

- Problem in Multi-path Inheritance or **Diamond Problem**
- The derived class inherits the members of the base class (grandparent) twice, through parent1 ( Derived\_Class1 ) and parent 2 ( Derived\_Class2 ).
- This results in ambiguity because a duplicate set of members is created. This ambiguous situation must be avoided.



# VIRTUAL BASE CLASSES

- The solution to the problem of ambiguity in multi-path inheritance is by making the common base class or grandparent class into a virtual base class .
- This is done by using the keyword **virtual** while declaring the direct or intermediate base classes or parent classes.
- The keyword **virtual** ensures that only one copy of the base class (grandparent) is inherited, irrespective of the number of inheritance paths that exist between the virtual base class and the derived class.

```
class derived_class : virtual public base class
{   -----
    -----
};

OR

class derived_class : public virtual base class
{   -----
    -----
};
```

# OBJECT SLICING

- The object of a derived class can be assigned to an object of its base class; however, the same rule does not apply for the object of a base class. This process is called **object slicing**.
- Object slicing occurs because when the derived class object is assigned to an object of the base class, and the additional features of the derived class are sliced off from the objects of base class.

```
#include<iostream.h>
class Base
{ protected:
    int a;
public:
    Base(int x)
    { a = x; }
    virtual void show()
    { cout<<"\n In Base : a = "<<a;
    }
};

class Derived : public Base
{ private:
    int b;
public:
    Derived(int x, int y) : Base(x)
    { b = y; }
    virtual void show()
    { cout<<"\n In Derived : a = "<<a<<" and b = "<<b;
    }
};

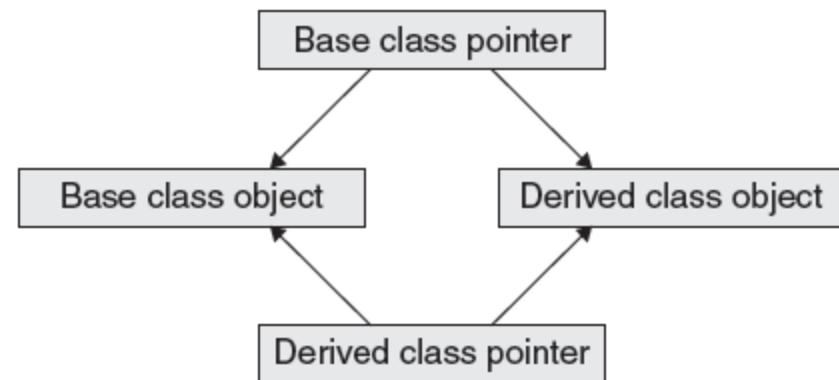
main()
{ Base B(5);
  Derived D(2,3);
  B = D;          // derived object is assigned to base class object
  B.show();        //additional features cannot be accessed
}
```

## OUTPUT

In Base : a = 2

# POINTERS TO DERIVED CLASS

- We can declare pointers to direct to either the base class or to the derived classes.
- Pointers to object of base class are compatible with pointers to object of derived class.
- Therefore, a single base class pointer can point to objects of the base as well as to the objects of the derived classes.
- However, this statement does not hold true for the reverse as a pointer to derived class.



```
#include<iostream.h>
class Base
{ public:
    void print()
    { cout<<"\n PRINT - BASE CLASS";
    }
};

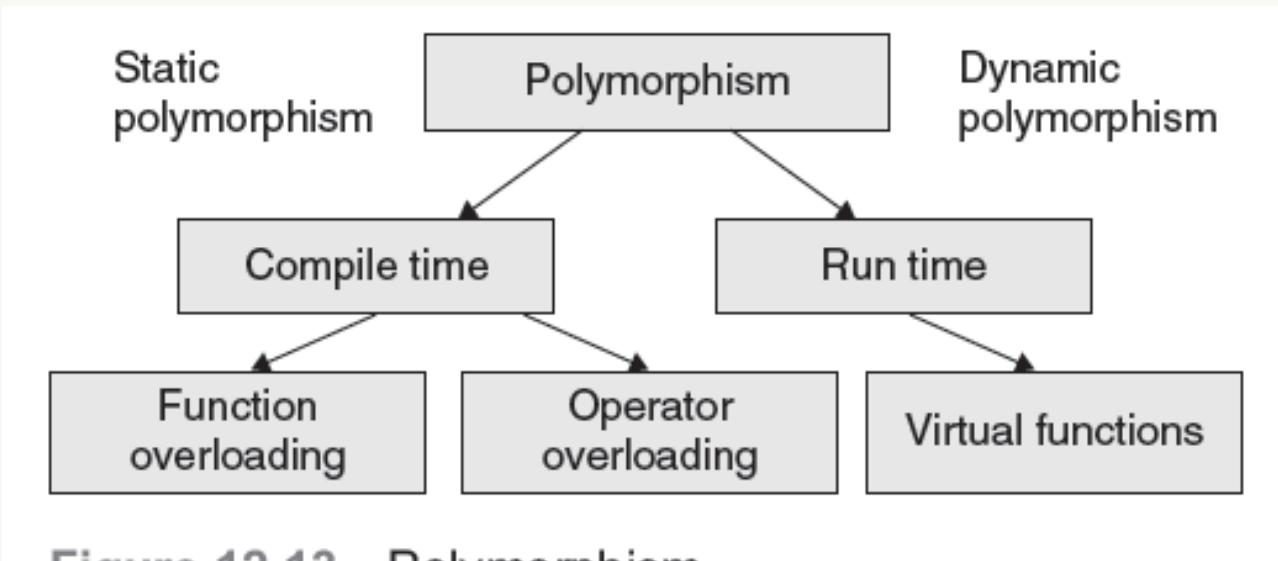
class Derived : public Base
{ public:
    void print()
    { cout<<"\n PRINT - DERIVED CLASS";
    }
};

main()
{ Base B, *bptr;
  Derived D, *dptr;
  bptr = &B;      // Base class ptr points to base class object
  bptr->print();
  bptr = &D;      // Base class ptr points to derived class object
  bptr->print();
  dptr = &D;      // Derived class ptr points to derived class object
  dptr->print();
}
```

## OUTPUT

PRINT - BASE CLASS  
PRINT - BASE CLASS  
PRINT - DERIVED CLASS

# RUN-TIME POLYMORPHISM



# VIRTUAL FUNCTIONS

- In the code, the function `show()` has the same prototype in both the classes; therefore, to print values of objects of both the classes, we may use the class **resolution operator** that specifies the class while invoking the function.
- However, this does not involve run-time polymorphism.
- Therefore, in C++, we have virtual functions to implement dynamic polymorphism.
- When we have the same function name in the base as well as the derived class, the function in the base class is declared as virtual .
- To declare a virtual function, precede the function prototype with the keyword `virtual`.

# VIRTUAL FUNCTIONS contd.

- The syntax for defining a virtual function is as follows:

```
virtual return_type function_name (arguments)
{   -----
}
```

```
class Base
{   -----
    public:
        void show()
        {   -----  }
};
class Derived : public Base
{-----
    public:
        void show()
        {   -----  }
};
```

# Run-time Polymorphism through Virtual Functions

- To implement dynamic polymorphism we use a **pointer** to the base class to refer to all the derived objects.
- At run time, the appropriate function will be invoked based on the object is currently under consideration.
- Here, the object under consideration is the one which is pointed to by the base pointer.

```
#include<iostream.h>
class Base
{ public:
    void print()
    { cout<<"\n PRINT - BASE CLASS";
    }
    virtual void show()
    { cout<<"\n SHOW - BASE CLASS";
    }
};
class Derived : public Base
{ public:
    void print()
    { cout<<"\n PRINT - DERIVED CLASS";
    }
    void show()
    { cout<<"\n SHOW - DERIVED CLASS";
    }
};
main()
{ Base B, *bptr;
 Derived D;
 bptr = &B;      // Base class ptr points to base class object
 bptr->print();
 bptr->show();
 bptr = &D;      // Base class ptr points to derived class object
 bptr->print();
 bptr->show();
}
```

## OUTPUT

```
PRINT - BASE CLASS
SHOW - BASE CLASS
PRINT - BASE CLASS
SHOW - DERIVED CLASS
```

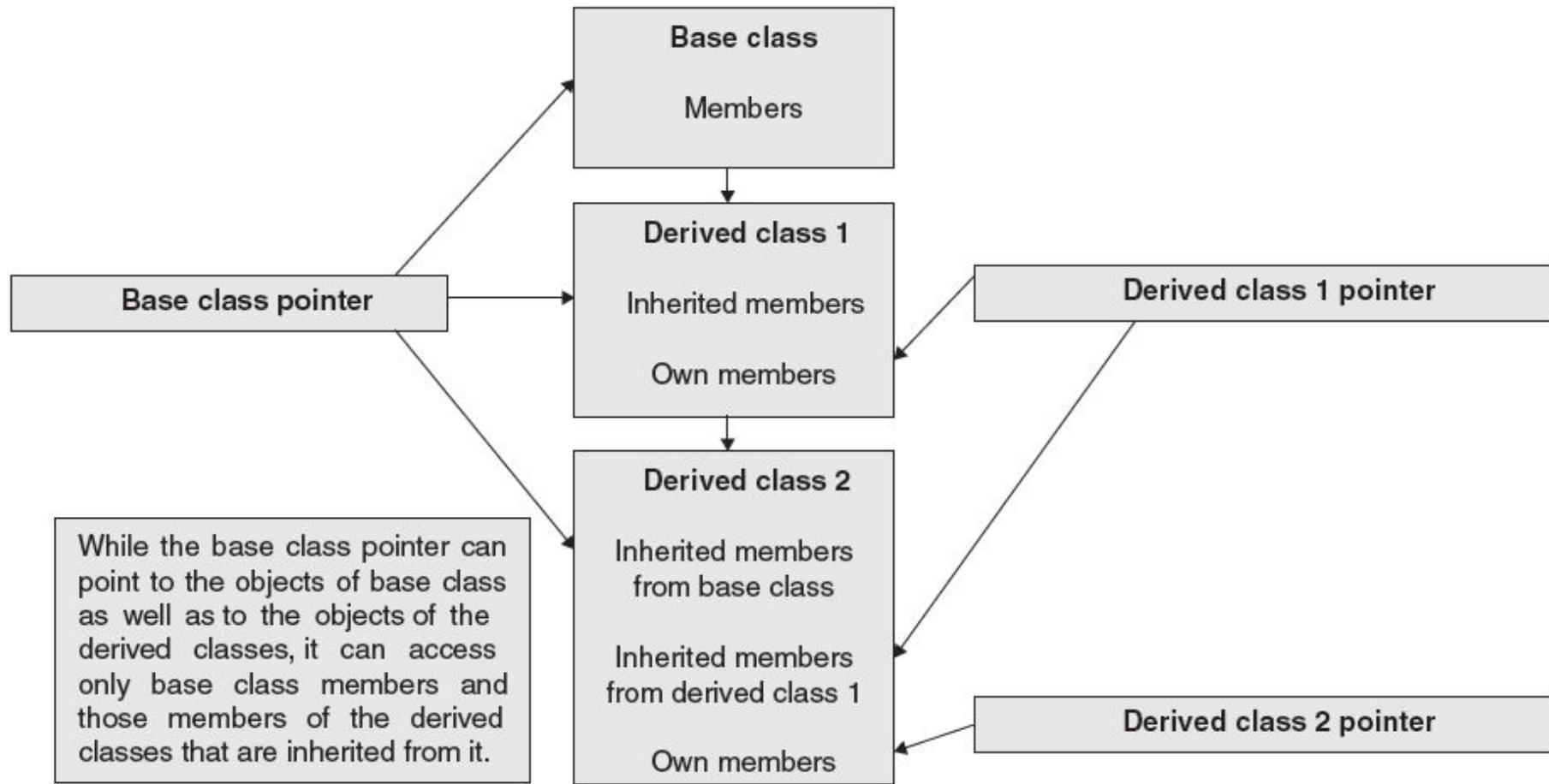


Figure 12.14 Base class pointers and derived class pointers

# PURE VIRTUAL FUNCTIONS

- As a good programming habit, we must redefine the virtual functions in the derived classes because function in the base class just gives a template for the derived classes and therefore has not much functionality to implement.
- Therefore, it is recommended to declare such virtual functions as empty, or without any definition.
- These do-nothing functions that have no definition are called pure virtual functions.
- To declare a virtual function, the prototype of the function is preceded with the keyword `virtual` and followed by `= 0;`
- For example, to declare a pure virtual function `show()` ; we need to write `virtual void show() = 0;`

```
#include<iostream.h>
#include<conio.h>
class Base
{ public:
    void print()
    { cout<<"\n PRINT - BASE CLASS";
    }
    virtual void show() = 0;
};
class Derived : public Base
{ public:
    void print()
    { cout<<"\n PRINT - DERIVED CLASS";
    }
    void show()
    { cout<<"\n SHOW - DERIVED CLASS";
    }
};
main()
{ Base *bptr;
    Derived D;
    bptr = &D;      // Base class ptr points to derived class object
    bptr->print();
    bptr->show();
}
```

## OUTPUT

PRINT - BASE CLASS  
SHOW - DERIVED CLASS

reserved.

# ABSTRACT BASE CLASSES

- An abstract class is a class that is specifically defined to lay a foundation for other classes that exhibits a common behavior or similar characteristics.
- It is primarily used only as a base class for inheritance.
- Since an abstract class is an incomplete class, users are not allowed to create its object.
- To use such a class, programmers must derive it, keeping in mind that they must implement all pure virtual functions specified in that class.

# ABSTRACT BASE CLASSES contd.

- **Features**

- Abstract base classes have at least one pure virtual function
- Classes inheriting abstract classes must either define all pure virtual functions or

must themselves become an abstract class.

- There can be no object of an abstract class.
- Pointers or references to abstract classes can be created.
- An abstract class can have other data members and member functions in addition to the pure virtual functions.

```
#include<iostream.h>
class sample
{
public:
    virtual void example()=0;
};
class Ex1:public sample
{
public:
    void example()
    {
        cout<<"ubuntu";
    }
};
class Ex2:public sample
{
public:
    void example()
    {
        cout<<" is awesome";
    }
};
void main()
{
    sample *ptr[2];
    Ex1 e1;
    Ex2 e2;
    ptr[0]= &e1;
    ptr[1]= &e2;
    ptr[0]->example();
    ptr[1]->example();
}
```

## OUTPUT

Ubuntu is awesome

# VTABLES

- When a class has a virtual function, the compiler creates a VTABLE , in which the address of the virtual function is inserted.
- When the object of that class is created, the compiler inserts a hidden pointer called **VPTR** or virtual pointer to point to the **VTABLE** for that object.
- The VTABLE is created for each class that has a virtual function and for the classes derived from it.
- Dynamic binding is accomplished with the help of VTABLE and VPTR .
- VTABLE is an array of pointers to virtual functions.
- VTABLE is created by the compiler at the compile time.

# VTABLES contd.

- During run time, the appropriate virtual function is called by fetching the address of the virtual function from the VTABLE
- A VTABLE is shared amongst all the objects belonging to the same class.
- The compiler secretly inserts a hidden code in the constructor of each class having a virtual function that initializes the VPTR of its objects with the address of the VTABLE .
- For pure virtual functions, a null pointer is stored in the VTABLE of the abstract class.
- The VPTR is inherited to all the derived classes.

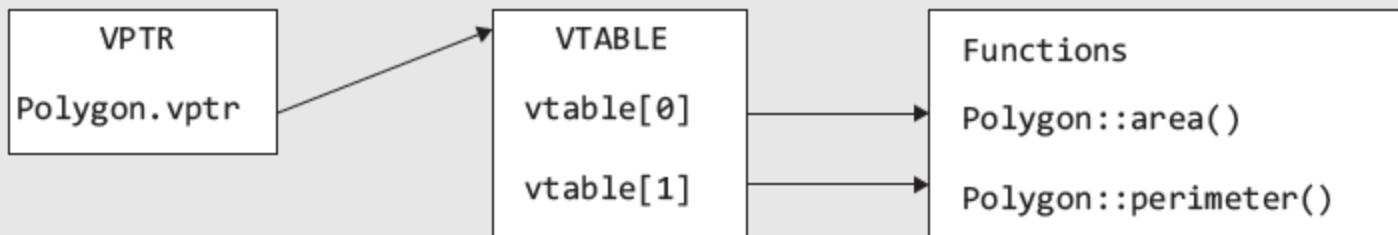
```
class Polygon
{    public:
        virtual void area();
        virtual void
perimeter();
};


```

```
Polygon.vptr = address of
Polygon.vtable

Polygon.vtable[0] = &Polygon::area()

Polygon.vtable[1]    =
&Polygon::perimeter()
```



```

#include<iostream.h>
class Base
{ public:
    virtual void print()
    { cout<<"\n Base :: print()";
    }
    virtual void show()
    { cout<<"\n Base :: show()";
    }
};
class Derived : public Base
{ public:
    void show()
    { cout<<"\n Derived :: show()";
    }
};
main()
{ Base *b = new Derived(); /*b is assigned Derived class's VPTR
    b->show();
    b->print();
}

```

### OUTPUT

Derived :: show()  
Base :: print()

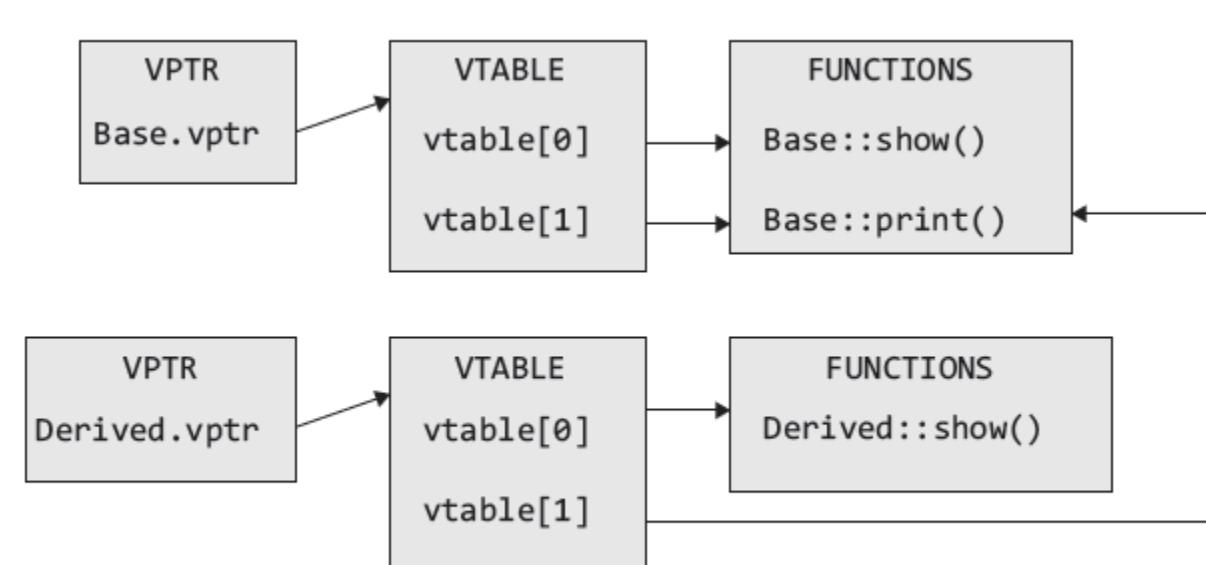


Figure 12.16 Virtual table

# NO VIRTUAL CONSTRUCTOR

- C++ is a static-typed language which means that the C++ compiler must be aware of the class type to create the object during the compile time of the program.
- While calling a constructor, if the VTABLE had not been created, there is an answer as to how will the call to virtual constructor be resolved.
- Moreover, in C++, object creation and object type are tightly coupled but a virtual constructor decouples object creation from its type.
- The choice of object to be created is decided dynamically by the compiler, based on the name of constructor used along with the new keyword.
- Hence, there is no need of virtual constructor.

# Virtual Destructor

```
#include<iostream.h>
class Base
{ public:
    Base()
    { cout<<"\n In Base Class Constructor";
    }
    ~Base()
    { cout<<"\n In Base Class Destructor";
    }
};
class Derived : public Base
{ public:
    Derived()
    { cout<<"\n In Derived Class Constructor";
    }
    ~Derived()
    { cout<<"\n In Derived Class Destructor";
    }
};
main()
{ Base *bptr = new Derived();
 delete bptr;
}
```

## OUTPUT

```
In Base Class Constructor
In Derived Class Constructor
In Base Class Destructor
```

# Virtual Destructor

A virtual destructor ensures that the destructor for any class that is derived from the base class will be called. Consider the output of the same program with the destructor declared as

```
virtual~Base()
{   cout<<"\n In Base Class Destructor";
}
```

## OUTPUT

```
In Base Class Constructor
In Derived Class Constructor
In Derived Class Destructor
In Base Class Destructor
```

According to the output, both constructors and destructors are called in the right sequence. To conclude, the guiding principle is that if a class has one or more virtual functions, then that class should also have a virtual destructor.



OXFORD  
HIGHER EDUCATION

# Object Oriented Programming with C++



Reema Thareja

OXFORD  
HIGHER EDUCATION

# Object Oriented Programming with C++

Reema Thareja

# Chapter Thirteen

## File Handling

# INTRODUCTION

- A file is a collection of data stored on a secondary storage device like hard disk
- File is basically used because real-life applications involve large amounts of data

# STREAMS IN C++

- A stream is a logical interface to the devices that are connected to the computer.
- In C++, the standard streams are termed as pre-connected input and output channels between a text terminal and the program (when it begins execution).
- The three standard streams in C++ language are **console input (cin)**, **console output (cout)**, and **console error (cerr)**.

# STREAMS contd.

- **Console input ( cin ) or standard input ( stdin )**
- **Standard input** is the stream from which the program receives its data.
- The program requests transfer of data using the read operation.
- **Console output ( cout ) or standard output ( stdout )**
- **Standard output** is the stream where a program writes its output data.
- **Console error ( cerr ) or standard error ( stderr )**
- **Standard error** is basically an output stream used by programs to report error messages or diagnostics.
- **Console log ( clog )** :It is a fully buffered version of cerr .clog which is mainly used to report problems.

# CLASSES FOR FILE STREAM OPERATIONS

- ifstream , ofstream , and fstream are derived from fstream base and from iostream.h .
- Classes that manage the disk files are declared in **fstream.h** . Therefore, fstream.h must be included in any program that uses files.
- **filebuf** It is used to set the file buffer for read and write operations. It contains open() and close() as its member.

# CLASSES FOR FILE STREAM OPERATIONS

**fstream base:** It provides functions common to the file streams. This file is the base class for:

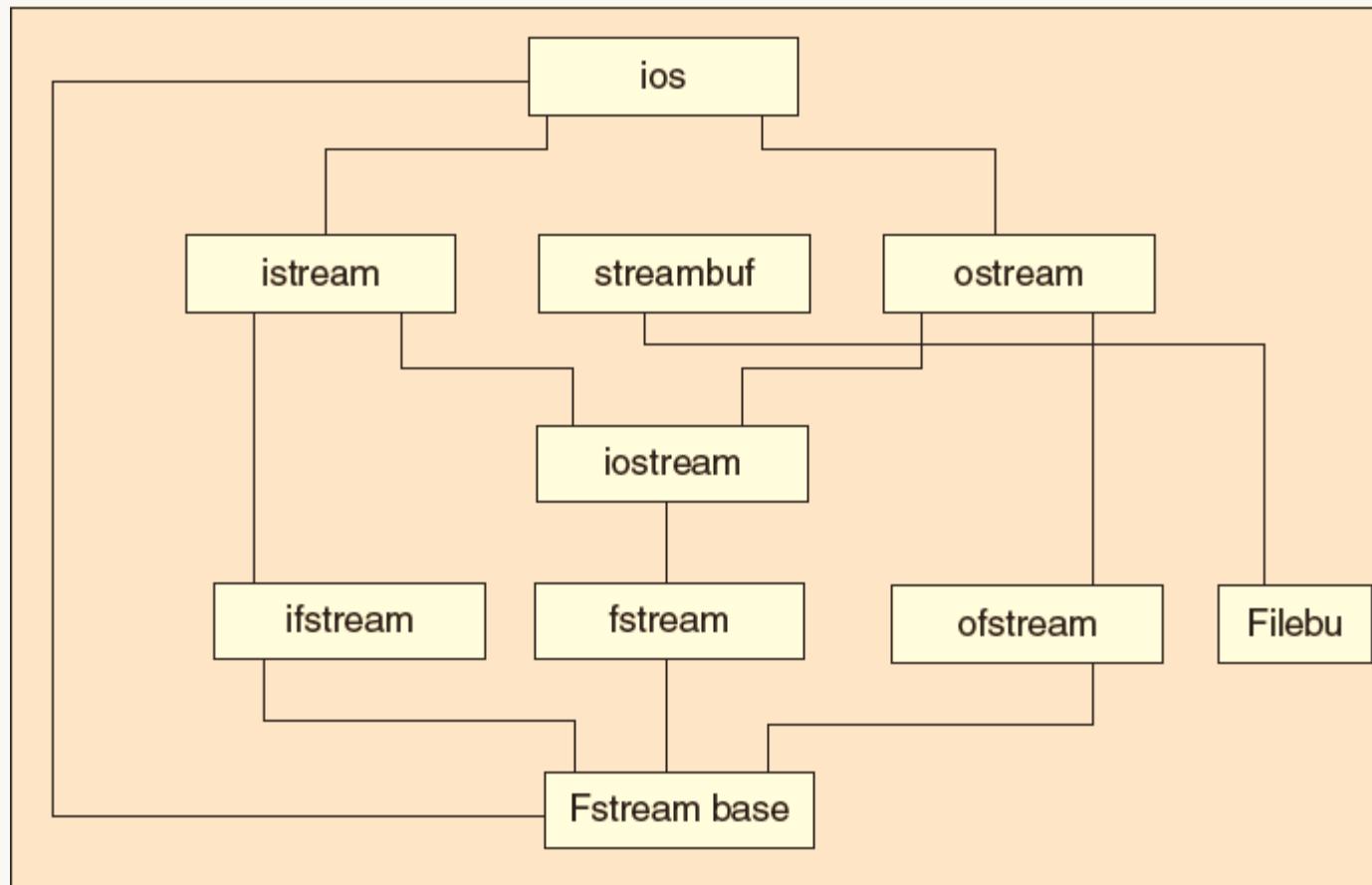
**ifstream , ofstream , and fstream** , and contains open() and close() as its members.

- **Ifstream:** It provides functions for data input. Therefore, the open() defined as its member opens a stream in the input mode by default. Moreover, ifstream inherits functions such as get() , getline() ,read(), seekg() , and tellg() functions from the istream class.
- **Ofstream:** It provides functions for data output. Therefore, the open() defined as its member, opens a stream in the output mode by default. Moreover, ofstream inherits functions like

# CLASSES FOR FILE STREAM OPERATIONS

- `put()` , `write()` ,`seekp()` , and `tellp()` functions from the `ostream` class.
- **Fstream:** It provides functions for both input and output operations.

# CLASSES FOR FILE STREAM OPERATIONS



# OPENING AND CLOSING OF FILES

- Before we perform I/O operations on a file, the file must be first opened to get a handle. The file handle serves as a pointer to the file. However, to obtain this handle, the user must specify the name of the file, data type, purpose, and the opening method.
- The data type and the structure of the file stream is defined using stream classes;
- `ifstream` , `ofstream` , or `fstream` , depending on the purpose of its usage, depending on whether the file is being opened for reading data or writing.
- For example, if we know that we are opening the file only to read data, the data type of the file can be `ifstream` .

# OPENING AND CLOSING OF FILES

- If we are opening for writing data, the file must be opened using ostream class.
- If the file is being opened for simultaneous read and write operations, then it must be opened using fstream class.

# Opening Files using Constructors

1. Create a file stream object. For example, ofstream is used to create the output stream and ifstream is used to create the input stream.
2. Initialize the file stream object by specifying the file name.

The syntax for opening a file using a constructor function is as follows:

```
streamclass_name file_objectname ("filename");
```

If the file is in the current directory, then only the filename needs to be specified; if the file is not in the current directory, then the complete pathname must be given. For example, to open a file students.txt for reading data, we write as

```
ifstream inpFile("student.txt");
```

Now, inpFile is an object of class ifstream that will be used to manage the input stream. Name of the object can be any valid identifier in C++. To open the employee.txt file with output stream, you must write,

```
ofstream outFile("employee.txt");
```

```
#include<iostream.h>
#include<fstream.h>
#include<conio.h>
void main()
{   char name[10];
    float sal;
    ofstream outFile("Employee"); // create an object and open the file using constructor
    for(int i=0;i<3;i++)
    {   cout<<"\n Enter the name and salary of Employee "<<i+1<<" : ";
        cin>>name>>sal;
        outFile<<"\n"<<name<<"\t"<<sal; // write to file
    }
    outFile.close(); // close the file after writing data
    ifstream inpFile("Employee"); // open the file for reading data
    for(i=0;i<3;i++)
    {   inpFile>>name; // read from file
        inpFile>>sal;
        cout<<"\n EMPLOYEE "<<i+1<<" : ";
        cout<<name<<"\t"<<sal;
    }
    inpFile.close();
    getch();
}
```

# Opening Files using Member function

- A file can also be opened explicitly by using open() , defined as member function of the class

```
file_stream_class stream_object ("file_name"); or  
file_stream_class stream_object;  
stream_object.open("file_name")
```

# Test for errors

- When we try to open a file, it may open successfully. But this may not always be true. For ex, if the user tries to open a non-existent file in read-mode or tries to open a read-only file in write mode, then the file operation definitely fails.
- Therefore, it is always recommended to test for errors by using the **operator !** with an instance of the ifstream, ofstream, or fstream .
- The operator ! is overloaded and returns a non-zero value if an error(s) is encountered and a zero otherwise.
- The statements given here open a file using the open() and check if the file was opened successfully.

```
ifstream inpFile("employee.txt");
if(! inpFile)
{    cerr<<"\n File could not be opened";
    exit(1);
}
```

```
#include<iostream.h>
#include<fstream.h>
#include<conio.h>
#include<stdlib.h>
void main()
{
    char name[10];
    float sal;
    ofstream outFile;
    outFile.open("Employee");
    if(!outFile)
    {
        cout<<"\n File Could Not Be Opened";
        exit(1);
    }
    for(int i=0;i<3;i++)
    {
        cout<<"\n Enter the name and salary of Employee "<<i+1<<" : ";
        cin>>name>>sal;
        outFile<<"\n"<<name<<"\t"<<sal;
    }
    outFile.close();
    ifstream inpFile("Employee"); //open file for reading
    if(!inpFile) // check if file was opened successfully
    {
        cout<<"\n File Could Not Be Opened";
        exit(1);
    }
    for(i=0;i<3;i++)
    {
        inpFile>>name;
        inpFile>>sal;
        cout<<"\n EMPLOYEE "<<i+1<<" : ";
        cout<<name<<"\t"<<sal;
    }
    inpFile.close(); // close file
    getch();
}
```

# DETECTING THE END-OF-FILE

- In real-world programs, we may not know the number of records stored in the file.
- We need to analyze whether we have reached the end of the file or not. Therefore, in our file handling programs, we need to continuously check for end-of-file condition to prevent reading data further from the file.

| <b>While(ifstream_object_name)</b>                                                                                                                                                                                                                                                                | <b>While(ifstream_object_name.eof()!=0)</b>                                                                                                                                                                                                                                             |
|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <ul style="list-style-type: none"><li>• Will return 0 if any error occurs and stop reading.</li><li>• Can stop reading even if any other (except EOF) occurs.</li><li>• <b>USAGE</b><br/><code>while(inpFile)<br/>{          getline(name, 20);<br/>          cout&lt;&lt;name;}</code></li></ul> | <ul style="list-style-type: none"><li>• Will return 1 if EOF is encountered and stop reading.</li><li>• Will read till EOF is not reached.</li><li>• <b>USAGE</b><br/><code>while(inpFile.eof()!=0)<br/>{          getline(name, 20);<br/>          cout&lt;&lt;name;}</code></li></ul> |

# FILE MODES

- Until now, we had been opening files in the default mode. Basically, the constructor function and the
- `open()` used to open a file needs two parameters such as **filename and the file mode**.
- However, we had provided only the first parameter, which is the filename. For the second parameter, the default values were used.
- The file mode is used to specify the purpose for which the file is being opened.

# FILE MODES

Table 13.3 File modes

| Class    | Mode     | Meaning    |                                                              |
|----------|----------|------------|--------------------------------------------------------------|
| ifstream | ios::in  | Read only  | <code>stream_object_name.open("filename", file_mode);</code> |
| ofstream | Ios::out | Write only |                                                              |

Table 13.4 Additional file modes

| File mode                                                                | Meaning                                                                                                                                                                                                                                                                                               |
|--------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| ios::app                                                                 | Appends data at the end of the file. This means any data written in this file is not deleted; rather the new data will be inserted at the end of the file. Sets the initial position at the end of the file. If this flag is not set, then the default initial position is the beginning of the file. |
| ios::ate<br>ios::binary<br>ios::nocreate<br>ios::noreplace<br>ios::trunc | Opens a binary file. If not set, by default a text file is opened. Open() fails if the file does not exist. Open() fails if the file already exists. If an already existing file is opened for writing, its contents are deleted or truncated and the new data is written.                            |

# FILE MODES

- When a file is opened in **ios::out** mode, it is automatically opened using the **ios::trunc** mode
- **ios::app** as well as **ios::ate** moves the file pointer to the end of the file when it is opened. If the file does not exist, it creates the file.
- **ios::app** allows new data to be written only at the end of the file.
- **ios::ate** allows data to be added or modified anywhere in the file.
- The fstream class does not provide a default mode, so it must be explicitly specified while opening files with fstream object.
- For ifstream or ofstream class objects, the file mode need not

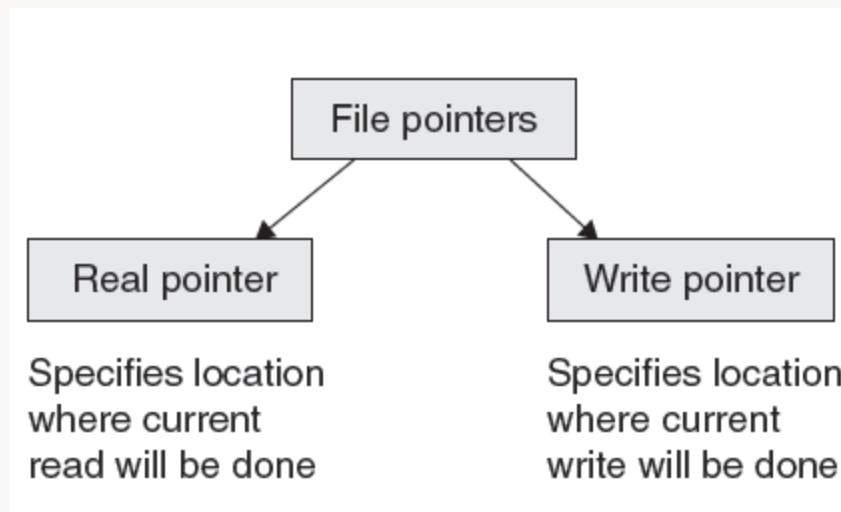
# FILE MODES

- be specified as default values can be used.
- Programmers can combine two modes using the bitwise OR operator. For example, out- File("Employee", **ios::ate | ios::nocreate**) would open the Employee file for adding or modifying existing data. If the file does not exist, the open() will fail but will not create the file named Employee.

# FILE POINTERS AND THEIR MANIPULATORS

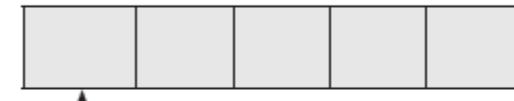
- The two types of file pointers are **read pointer** and the **write pointer**.
- While the read pointer is called, the get pointer specifies a location from where the current read operation is initiated.
- The write or the put pointer specifies a location from where the current write operation is initiated.
- Once the read or write operation is completed, the appropriate pointer is automatically advanced.

# FILE POINTERS AND THEIR MANIPULATORS



Read pointer in READ mode

Figure 13.5 5ios::in mode



Write pointer in WRITE MODE

Figure 13.6 ios::out mode



Write pointer in APPEND MODE

Figure 13.7 ios::app mode

# Manipulating File Pointers

Table 13.5 Manipulation functions

| Function | Member of | Utility                                         | Example                | Effect                                                  |
|----------|-----------|-------------------------------------------------|------------------------|---------------------------------------------------------|
| seekg()  | ifstream  | Moves the get pointer to the specified position | inpFile.<br>seekg(10); | Moves the get pointer to byte 10.                       |
| seekp()  | ofstream  | Moves the put pointer to the specified position | outFile.<br>seekg(10); | Moves the put pointer to byte 10.                       |
| tellg()  | ifstream  | Gives the current position of the get pointer   | inpFile.<br>tellg();   | Returns 10 if the get pointer is at byte 10 in the file |
| tellp()  | ofstream  | Gives the current position of the put pointer   | outFile.<br>tellp();   | Returns 10 if the put pointer is at byte 10 in the file |

# Specifying the Offset

- There is another variant of seekg() and seekp() functions. It can also accept two arguments— **the offset and the reference position .**
- The syntax for a two argument seekg() or seekp() is
- **inpFile.seekg(10, ios::beg)**— moves to 10 bytes starting from the beginning of the file
- **inpFile.seekg(10, ios::cur)**— moves to 10 bytes starting from the current position of the file
- **inpFile.seekg(0, ios::end)**— moves to the end of the file
- **inpFile.seekg(0, ios::beg)**— moves to the beginning of the file

# Specifying the Offset

- **inpFile.seekg(0, ios::cur)**— stays at the current location of the file pointer
- **inpFile.seekg(-10, ios::cur)**— moves 10 bytes backward starting from the current position of the file pointer
- **inpFile.seekg(-10, ios::end)**— moves 10 bytes backward from the end of the file

# Specifying the Offset

```
seekg(offset, reference_postion);  
seekp(offset, reference_postion);
```

Table 13.6 Reference position

| Reference_position | Location in file                |
|--------------------|---------------------------------|
| ios::beg           | Beginning of the file           |
| ios::cur           | Current location of the pointer |
| ios::end           | End of the file                 |

# ASCII Text Files

- A text file is a stream of characters that can be sequentially processed by a computer in forward direction. Hence, a text file is usually opened for only one kind of operation such as reading, writing or appending at any given time.
- As text files only process characters, they can only read or write data one character at a time in a line.
- A text file is not a C++ string, and is not terminated by a null character.
- When a text file is used, there are actually two representations of data—internal or external.
- For example, an int value will be represented as two or four bytes of memory internally but externally, the int value will be represented as a string of characters representing its decimal or hexadecimal values.

# Binary Files

- A binary file is a file which may contain any type of data, encoded in binary form for computer storage and processing purposes. Like a text file, a binary file is a collection of bytes.
- A binary file does not require any special processing of the data and each byte of data is transferred to or from the disk unprocessed.
- C++ places no constructs on the file and it may be read from, or written to, in any manner the programmer wants.
- While text files can be processed sequentially, binary files, on the other hand, can be either processed sequentially or randomly depending on the needs of the application.

|                                                                                                         |                                                                                                                                                                                     |                                                                   |   |   |   |
|---------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------|---|---|---|
| <table border="1"><tr><td>123</td></tr></table>                                                         | 123                                                                                                                                                                                 | <table border="1"><tr><td>1</td><td>2</td><td>3</td></tr></table> | 1 | 2 | 3 |
| 123                                                                                                     |                                                                                                                                                                                     |                                                                   |   |   |   |
| 1                                                                                                       | 2                                                                                                                                                                                   | 3                                                                 |   |   |   |
| <p>In binary format, the number 123 occupies only two bytes (since an integer is allocated 2 bytes)</p> | <p>In ASCII format, the number 123 occupies three bytes. Each digit is treated as a single character. Since a character requires one byte of storage space is stored in 3 bytes</p> |                                                                   |   |   |   |

Figure 13.8 Binary and ASCII format

# get() and put()

- The get() reads a single character from the associated stream and the put() is used to write a single character to the associated stream.

```
#include<fstream.h>
#include<string.h>
void main()
{   char comment[80];
    char ch;
    cout<<"\n Enter your feedback about this college : ";
    cin.getline(comment,80);
    int len = strlen(comment);
    fstream FILE;
    FILE.open("Feedback", ios::in | ios::out);
    for(int i=0;i<len;i++)
        FILE.put(comment[i]);
    FILE.seekg(0);
    cout<<"\n USER'S FEEDBACK : ";
    for(i = 0;i<len;i++)
    {   FILE.get(ch);
        cout<<ch;
    }
    FILE.close();
}
```

### Program 13.3 Write a program to count the number of characters and number of lines in a file.

```
#include<fstream.h>
#include<string.h>
#include<stdlib.h>
main()
{   fstream FILE;
    int ch, no_of_characters=0, no_of_lines=0;
    char filename[20];
    cout<<"\n Enter the filename : ";
    cin.getline(filename,20);
    FILE.open(filename, ios::in);
    if(!FILE)
    {   cout<<"\n Error Opening The File";
        exit(1);
    }
    while(FILE.eof()==0)
    {   ch = FILE.get();
        if(ch=='\n')
            no_of_lines++;
        no_of_characters++;
    }
    if(no_of_characters > 0 )
        cout<<"\n In the file "<<filename<<" there are "
<<no_of_lines<<" lines and "<<no_of_characters<<" characters";
    else
        cout<<"\n File is empty";
    FILE.close();
}
```

# read() and write() Functions

- The **get()** reads a single character from the associated stream and the **put()** is used to write a single character.
- **read()** and **write()** functions take two arguments—**the address of the variable** that has to be written and **the size of that variable** in bytes. to the associated stream.

The syntax of **read()** and **write()** is as follows:

```
ifstream_object_namen.read((char *)&var, sizeof(var));  
ofstream_object_namen.write((char *)&var, sizeof(var));
```

```
#include<fstream.h>
void main()
{   int marks[]={98, 67, 89, 100, 45,65, 51, 78, 12, 43};
    int arr[10]={0};
    fstream FILE; //create object
    FILE.open("temp", ios::out|ios::binary); //open file for writing
    FILE.write((char *)marks, sizeof(marks)); //write data
    FILE.close(); //close file
    FILE.open("temp",ios::in|ios::binary); //open for reading
    FILE.read((char *)arr, sizeof(marks)); //read data
    cout<<"\n MARKS OBTAINED : ";
    for(int i = 0;i<10;i++)
        cout<<"\t"<<arr[i]; //display data
    FILE.close(); //close file
}
```

# ERROR HANDLING DURING FILE OPERATIONS

- Open a non-existent file in read mode.
- Open a read-only in ios::out mode.
- Open a file with invalid name.
- Read beyond the end-of-file.
- Write more data in a file stored on disk when no sufficient disk space is available.
- Manipulate data stored in an unopened file.
- No file is connected with the stream object.
- Media errors which may occur while reading or writing data
- Class ios has a member function called that records information on the status of a file that is currently being used.

|        |        |        |        |            |                   |          |             |
|--------|--------|--------|--------|------------|-------------------|----------|-------------|
| Unused | Unused | Unused | Unused | Hard error | Invalid operation | R/w fail | End-of-file |
|--------|--------|--------|--------|------------|-------------------|----------|-------------|

# ERROR HANDLING DURING FILE OPERATIONS

Table 13.7 Functions for accessing status of file stream

| Function  | Utility                                                                                                                                                                                                            | Usage                                     |
|-----------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------|
| eof()     | Returns a non-zero value (true) if end-of-file is encountered; otherwise, returns 0.                                                                                                                               | while(FILE.eof()==0)<br>{ // Process }    |
| fail()    | Returns true if read/write operation has failed; otherwise, returns false.                                                                                                                                         | If(FILE.fail())<br>{ // File not opened } |
| bad()     | Returns true if either an invalid operation is attempted or any unrecoverable error has occurred.                                                                                                                  | if(FILE.bad())<br>{ // Error }            |
| good()    | Returns false if it is possible to recover from error and continue operation.                                                                                                                                      | if(FILE.good())<br>{ // Error }           |
| clear()   | Returns true if no error has occurred. In other words, if all the above functions are false. Therefore, when FILE.good() returns true, it means that everything is fine and further operations can be carried out. | FILE.clear();<br>FILE.rdstate();          |
| rdstate() | Clears the error states and proceeds further processing with file.<br>Reads the status state data member defined in class ios.                                                                                     |                                           |

# ACCEPTING COMMAND LINE ARGUMENTS

- Command-line arguments are given after the name of a program in command-line operating systems such as DOS or Linux, and are passed into the program
- The main() can accept two arguments:-
- The first argument is an integer value that specifies number of command line arguments.
- The second argument is a full list of all of the command line arguments.
- The full declaration of main() can be given as follows:

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

The full declaration of `main()` can be given as follows:

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

# ACCEPTING COMMAND LINE ARGUMENTS

- The **integer argc** specifies the number of arguments passed into the program from the command line, including the name of the program.
- The array of character pointers, **argv** contains the list of all the arguments. **argv[0]** is the **name of the program**, or an empty string if the name is not available. **argv[1] to argv[argc – 1]** specifies the command line argument.
- In the C++ program, every element in the argv can be used as a string.am from the operating system.

```
#include<iostream.h>
int main(int argc, char *argv[])
{
    int i;
    cout<<"\n Number of arguments passed = "<<argc;
    for (i=0; i<argc; i++)
        cout<<"\n argv["<<i<<"] = "<<argv[i];
    return 0;
}
```

## OUTPUT

```
Number of arguments passed = 3
argv[0] = exercise
argv[1] = first
argv[2] = second
```



OXFORD  
HIGHER EDUCATION

# Object Oriented Programming with C++



Reema Thareja

OXFORD  
HIGHER EDUCATION

# Object Oriented Programming with C++

Reema Thareja

# Chapter Fourteen

## Templates

# INTRODUCTION

- Templates allow programmers to write functions and classes to be defined with any type or of a general type.
- Until now, we have been writing non-template functions in which the function parameters are declared to be of a particular type. While calling the function, the arguments passed to the function must strictly match the data types of the parameters.
- However, with a template function, that function can be called with any compatible type.
- Templates act as a model of function or a class that can be used to generate functions or classes. During compilation of a program that has templates, the C++ compiler generates one or more functions or classes based on the specified template.

# INTRODUCTION

- Therefore, through templates, C++ supports the concept of generic programming.
- Generic programming is a technique of programming in which a general code is written first.
- The code is instantiated only when the need arises for specific types that are provided as parameters.

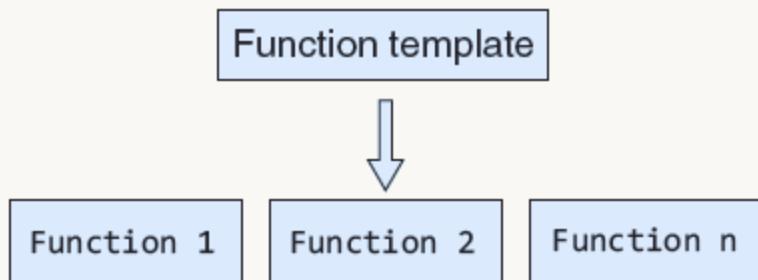


Figure 14.1(a) Generated functions

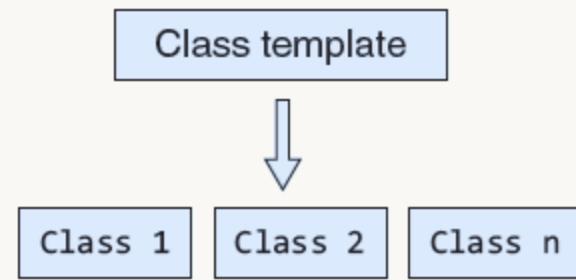


Figure 14.1(b) Generated classes

# FUNCTION TEMPLATES

- This enables programmers to write functions without having to specify the exact type(s) of some or all of the variables.
- Instead, we define the function using placeholder types (i.e. for which no data type is specified), called template type parameters.

The diagram illustrates the syntax of a function template. It consists of three main parts: a 'Keyword' box containing 'template< class type, ....>', a 'Template data types' box containing 'return\_type Func\_name(Type arg1, ....)', and a 'Function prototype using arguments with template types' box containing 'Body of Template Function'. Arrows point from each box to its corresponding part in the template code.

```
template< class type, ....>
{
    return_type Func_name(Type arg1, ....)
    {
        -----
        Body of Template Function
        -----
    }
}
```

Figure 14.2 Syntax of a function template

```
#include<iostream.h>
template<class Type>
Type square(Type num)
{   return num*num;
}
main()
{   int num1 = 3;
    cout<<"\n Square of "<<num1<<" = "<<square(num1);//int is replaced with Type
    float num2 = 5.6;
    cout<<"\n Square of "<<num2<<" = "<<square(num2);//float is replaced with Type
    double num3 = 123.456;
    cout<<"\n Square of "<<num3<<" = "<<square(num3);//double is replaced with Type
}
```

## OUTPUT

Square of 3 = 9  
Square of 5.6 = 31.59999299  
Square of 123.456 = 15241.383936

# Generic Data Types

- Every template data type must be used in the function declaration and/or definition. Failing to do so results in an error and will be treated as an invalid template.
- For example, a template definition like the one given here is an error.

```
template<class Type1>
int square(intnum)// Error: Type 1 is not used as an argument
{   return num*num;
}
```

- Using only some and not all generic types as function arguments also results in an error.
- Therefore, the given definition will also generate an error.

# Overloaded Function Templates

- Like normal functions, template functions can also be overloaded. In such cases, the template function has the same name but different number or type of arguments.
- When the compiler encounters an overloaded template function, it uses the following rules to select a suitable template.
- If an exact match is found using an ordinary function, the compiler calls it.
- If an exact match is not found, it looks for a function template from which an exact match can be generated. If found, it calls for the same.

```
template<class Type1, class Type 2>
int square(Type1 num)//Error: Type 2 is not used as an argument
{
    Type 2 res = num * num;
    return res;
}
```

## Example 14.2      Template function overloading

```
#include<iostream.h>
template<class Type>
void display(Type num)
{   cout<<"\n"<<num;
}
template<class Type1, class Type2>
void display(Type1 mes, Type2 num)
{   cout<<"\n"<<mes<<num;
}
main()
{   intnum = 5;
    char *m = "The number is : ";

    display(num);
    display(m, num);
}
```

### OUTPUT

5

The number is : 5

# Recursive Template Functions

- C++ also allows programmers to call nested or recursive template functions. For example,
- consider the program code given here that finds factorial of a number.

```
#include<iostream.h>
template<class Type>
Type fact (Type num)
{
    if(num==1)
        return 1;
    else
        return (num*fact(num-1));
}
main()
{
    int num1 = 5;
    cout<<"\n Factorial of " <<num1<< " = "<<fact(num1);
    longint num2 = 10;
    cout<<"\n Factorial of " <<num2<< " = "<<fact(num2);
}
```

# Function Templates with User-defined Types

- C++ allows programmers to use user-defined types in function template.
- For example, a template use the display() to print an integer, char, string, a structure, etc., to show its contents.
- For example, display() prints the contents of all its individual data members on screen
- If no match can be found, ordinary overloading resolution for the function is tried.
- If no match is still found, an error is generated.
- In case more than one match is found, then an ambiguity error is generated.

```
template<class Type1>
void display(Type1 &stud)
{   cout<<"\n Roll Number : "<<stud.rno;
    cout<<"\n Name : "<<stud.name;
    cout<<"\n MARKS : "<<stud.marks;
}
```

# CLASS TEMPLATE

- Class templates specify how individual classes can be constructed so that every individual class supports similar operations on different data types.
- Therefore, templates enable programmers to create abstract classes that define the behavior of the class without actually knowing what data type will be handled by the class operations.
- Defining a class template is similar to defining an ordinary class except the two differences:-
- The template class is prefixed with template <class Type> that tells the compiler that a template is being declared.
- The use of Type in declaring members of the class.

# CLASS TEMPLATE

The syntax for declaring a class template is as follows:

```
template<class Type1, class Type2, ....>
class class_name
{
    .....
    // Body of the class
    .....
};
```

```
template<class Type>
class Array
{   private:
    Type *arr;
    int size;
public:
    Array(int n)
    {   arr = new Type [n];
        size = n;
    }
    void read();
    void print();
};
```

A class created from a template is called a template class. The syntax for creating an object of the template class is as follows:

```
class_name<Type>object_name(arguments...);
```

Therefore, to declare objects of the template class Array, we may write

```
Array <int>A(5);
Array <char>B(7);
```

# Key Points to Know About Class Templates

- C++ allows programmers to specify default types for the template class data types. The syntax for specifying default data types can be given as follows:

```
template<class Type1, class Type2 = int>
class class_name
{
    .....
// Body of the class
.....
};
```

- To define a member function of a template class outside the class, it must be treated as a function template and the following syntax must be used.

```
template<class Type>
return_type class_name<Type> :: function_name(args....)
{
    .....
    // Function Body
    .....
};
```

# CLASS TEMPLATES AND FRIEND FUNCTION

```
template<class Type>
class Myclass
{  private:
    .....
public:
    .....
    friend return_type func(Myclass <Type> Obj)
    {   //code
    }
};
```

# TEMPLATES AND STATIC VARIABLES

- All objects of the class share the static variable. Similarly, in function templates, each instantiation of the function has its own copy of static variables.
- Let us realize this point with the help of a program.

```
#include <iostream.h>
#include<conio.h>
static int i = 0;
template <class Type>
void fun(Type data)
{
    i++;
    cout<<"\n Value = " << data;
    cout<<"\n The function has been called " << i << " times ";
}
void main()
{
    fun<int>(10);
    fun<float>(20);
    fun<double>(1.1);
    fun<char>('A');
    fun<char*>( "Hello ");
    getch();
}
```

# CLASS TEMPLATES AND INHERITANCE

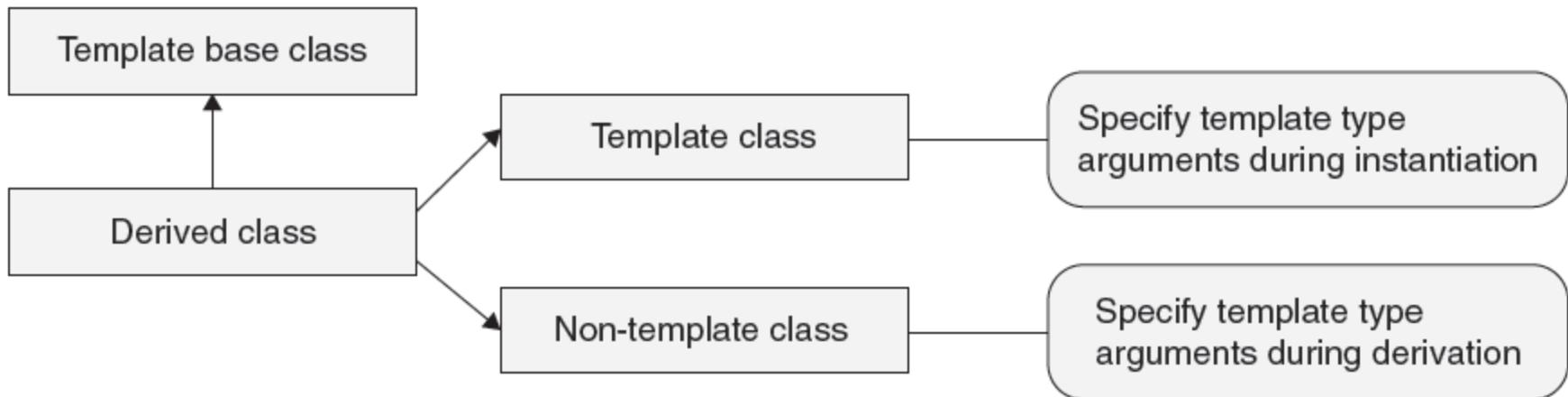
- Deriving a class template from a base class (template class).
- Deriving a class template from a base class which is a template class and adding more template members to it.
- Deriving a class template from a base class which is a template class but disallowing the derived class and its derivatives to have template features.
- Deriving a non-template base class and adding some template members to it.
- The syntax for declaring a derived class from base class which is a template class can be given as follows:

```

template<class Type1, class Type 2....>
class Base
{
    -----
    // Body of the class
    -----
};

template<class Type1, class Type2 ....>
class Derived : public Base <Type1, Type2, ....>
{
    -----
    // Body of the class
    -----
};

```



**Figure 14.3** Class template and inheritance

```

class Derived : public Base <int, float, ....>
{
    -----
    // Body of the class
    -----
};

```

# CLASS TEMPLATE WITH OPERATOR OVERLOADING

```
template <class Type>
class Complex
{ private:
    Type real, imag;
public:
    void get_data();
    void show_data();
    Complex operator+(Complex &);
};

template <class Type>
void Complex <Type> :: get_data()
{ cout<<"\n Enter the real and imaginary parts : ";
  cin>>real>>imag;
}

template <class Type>
void Complex <Type> :: show_data()
{ cout<<real << " + "<<imag<<"i ";
}

template <class Type>
Complex <Type> Complex <Type> :: operator + (Complex <Type>&C)
{ real += C.real;
  imag += C.imag;
  return *this;
}
```

# ADVANTAGES OF TEMPLATES

- A template not only enhances code reusability but also makes the code short and easy to maintain.
- A single template can handle different types of parameters.
- Testing and debugging efforts are reduced.
- For non-templated functions or classes, the compiler has to compile all n copies. Since they are all a part of the source-code. However, with a template, the compiler would compile only for required set of data-types.
- This means that compilation would be faster if number of different data-types actually required during implementation is less than.

# DISADVANTAGES OF TEMPLATES

- Some compilers provide little support for templates. This reduces the code portability.
- Many compilers do not have clear instructions on when and how to detect a template definition error.
- Even when the compiler detects an error, it takes more time and effort to debug the code and resolve the error.
- Since a separate code is generated for each template type, frequent use of templates can result in larger executables.
- Templates are usually defined in the headers, which exposes the code to the entire program, thereby defeating the concept of information hiding



OXFORD  
HIGHER EDUCATION

# Object Oriented Programming with C++



Reema Thareja

OXFORD  
HIGHER EDUCATION

# Object Oriented Programming with C++

Reema Thareja

# Chapter Fifteen

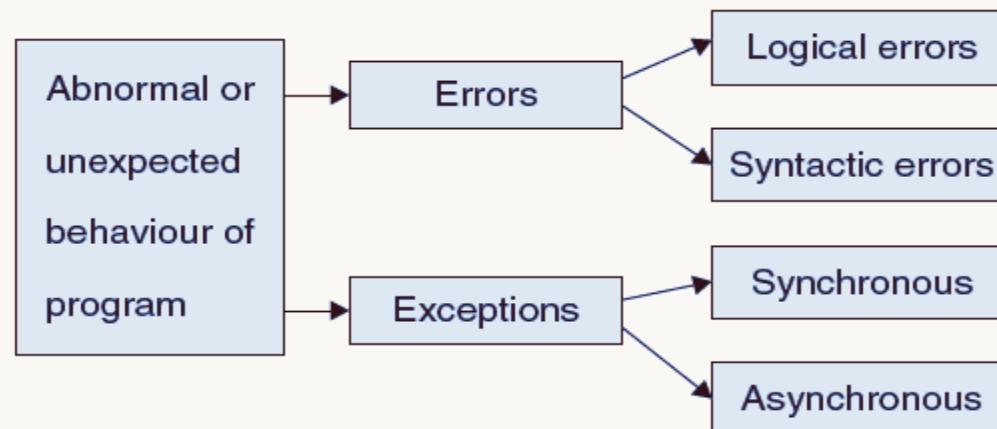
## Exception Handling

# INTRODUCTION

- The two common types of errors that we often encounter are **syntax errors** and **logic errors**.
- While logic errors occur due to poor understanding of problem and its solution, syntax errors, on the other hand, arise due to poor understanding of the language.
- However, such errors can be detected by exhausting and debugging and testing procedures.
- However, many a time, we come across some peculiar problems which are often categorized as **exceptions**.

# INTRODUCTION

- Exceptions are run-time anomalies or unusual conditions such as divide by zero, accessing arrays out of its bounds, running out of memory or disk space, overflow ,and underflow that a program may encounter during execution.
- To handle such exceptional situations, a new feature of exception handling has been added to ANSI C++ that helps the programmers to identify and effectively deal with these run-time errors.



# EXCEPTION HANDLING

- Exceptions provide a way to react to exceptional conditions in programs by transferring program control to special functions called **handlers**.
- The main aim of exception handling is to detect and report an exceptional condition(s), so that appropriate action can be taken to deal with it.
- **Try block** This block is a group of statements that may generate an exception.
- When an exception is generated, it is thrown using the keyword `throw`. The **throw** keyword invokes the exception handling routine.

```
-----  
try  
{ -----  
    throw exception;  
-----  
}  
catch(type arg)  
{ -----  
-----  
}
```

# EXCEPTION HANDLING

- Catch block This block catches the exception thrown from the try block and handles it in the way specified by the statements in the catch block.

# EXCEPTION HANDLING

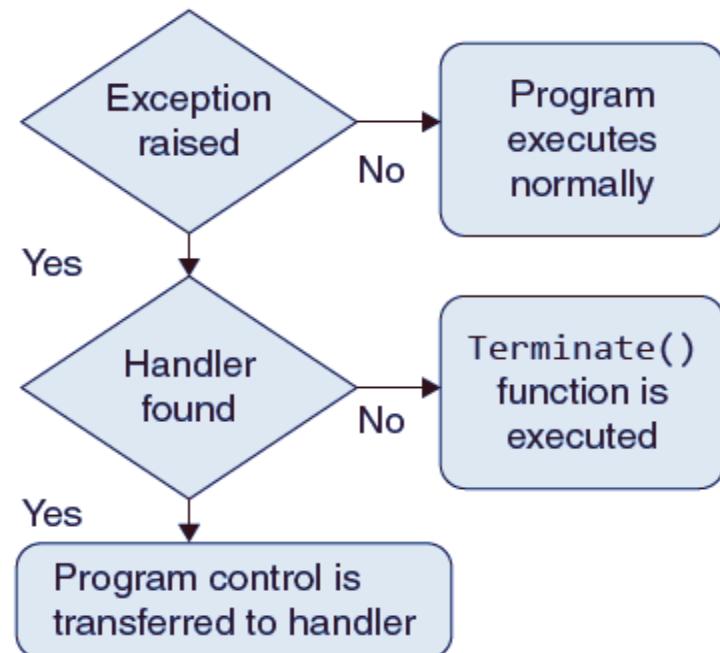
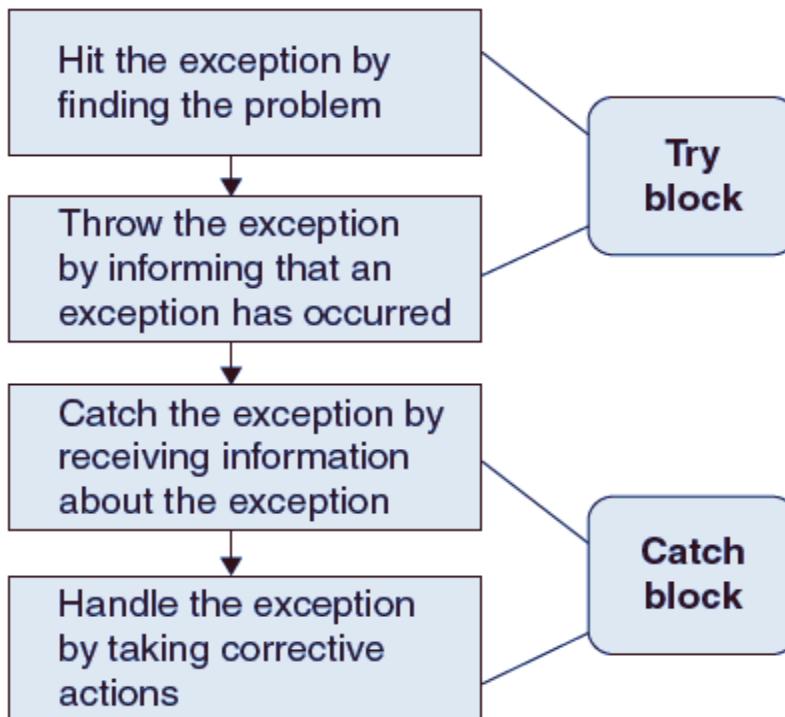


Figure 15.2 Handling exceptions

Figure 15.3 Program flow in case of exception

Exceptions can be thrown by writing as follows:

```
throw exception;  
throw (exception);  
throw;
```

```
#include<iostream.h>
main()
{ int num1, num2;
  cout<<"\n Enter two numbers : ";
  cin>>num1>>num2;
  try
  { if(num2!=0)// no exception is thrown
    { float res = (float) num1/num2;
      cout<<"\n RESULT = "<<res;
    }
    else
      throw(num2); //exception thrown
  }
  catch(int num2) // exception caught
  { cout<<"\n Divide by zero error encountered"; // exception handled
  }
  cout<<"\n EXITING MAIN()";
}
```

**Programming Tip:** A rethrown exception cannot be caught in the same function.

## OUTPUT

```
Enter two numbers :9 3
RESULT = 3
EXITING MAIN()
SECOND CASE
Enter two numbers :9 0
Divide by zero error encountered
EXITING MAIN()
```

# Multiple Catch Statements

- It is possible to associate more than one catch statement with a single try block. This is usually done when a program segment has more than one condition to throw as an exception.
- In such cases, when an exception is thrown, the exception handlers are searched to find an appropriate match.
- The first catch block that matches the type of the exception thrown is executed.
- After execution, the program control goes to the first statement after the last catch

```
try
{
    -----
    throw exception;
    -----
}

catch(type1 arg1)
{
    -----
    -----
}

catch(type2 arg2)
{
    -----
    -----
}

}

-----
catch(typenargn)
{
    -----
    -----
}
```

# Multiple Catch Statements

- block for that try block.
- This means that all other catch blocks are ignored.
- However, if no match is found, then the program is terminated using the **default abort()**.

```
#include<iostream.h>
#include<conio.h>
main()
{ intnum;
    cout<<"\n Enter a positive number : ";
    cin>>num;
    try
    { if(num==0)
        throw ("Zero");// exception 1
    else if(num<0)
        thrownum; //exception 2
    else
        cout<<"\n NUMBER = "<<num;
    }
    catch(intnum)
    { cout<<"\n"<<num<<" is negative"; // exception 2 caught
    }
    catch(char* msg)
    { cout<<"\n The number is "<<msg; // exception 1 caught
    }
    cout<<"\n EXITING MAIN()";
getche();
}
```

## OUTPUT

Enter a positive number : 0

The number is ZERO

# Catch all Exceptions

- In large software programs, often, it is difficult to anticipate all types of possible exceptional conditions.
- Therefore, the programmer may not be able to write a different handler (**catch block**) for every individual type of exception.
- In such situations, a better idea is to write a handler that would catch all types of exceptions.
- This is something similar to the default case in a switch statement.

```
catch(...)  
{  
    // code to handle all exceptions  
}
```



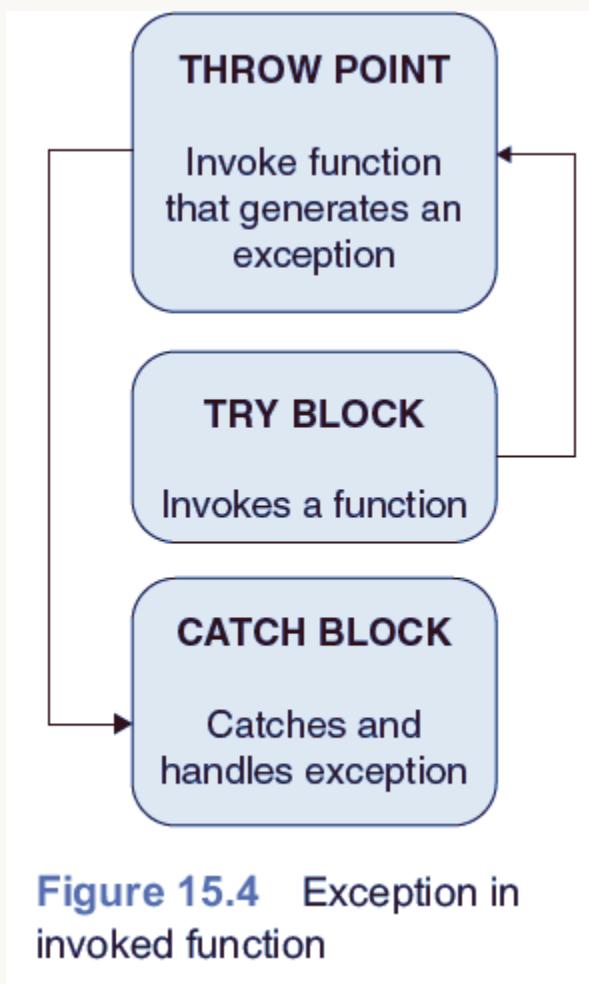
Catch all exceptions

```
#include<iostream.h>
#include<conio.h>
main()
{ intnum;
  cout<<"\n Enter a number (-1, 0, 1): ";
  cin>>num;
  try
  { if(num==0)
      throw ("Zero"); // exception 1
    else if(num== -1)
      throw num; // exception 2
    else if(num == 1)
      cout<<"\n NUMBER = "<<num;
  }
  else
    throw(float)num; // exception 3
}
catch(intnum) //caught exception 2
{ cout<<"\n"<<num<<" is negative";
}
catch(char* msg) // caught exception 1
{ cout<<"\n The number is "<<msg;
}
catch(...) // caught exception 3
{ cout<<"\n No specific code executed";
}
cout<<"\n EXITING MAIN()";
getche();
}
```

### **OUTPUT**

```
Enter a number (-1, 0, 1): 56
No specific code executed
```

# Exceptions in Invoked Function



```
typefunction_name(arg list)
{
    -----
    ...
    -----
}

try
{
    -----
    function_name(); // function call
    -----
}

catch(type arg)
{
    -----
    // Code to handle exception
    -----
}
```

Figure 15.4 Exception in invoked function

```
#include<iostream.h>
#include<conio.h>
void divide(int a, int b)
{ if(b==0)
    throw ("Divide by zero exception");
else
{ float res = (float)a/b;
    cout<<"\n RESULT = "<<res;
}
}
main()
{ int num1, num2;
cout<<"\n Enter the two numbers : ";
cin>>num1>>num2;
try
{ divide(num1, num2);
}
catch(char *mesg)
{ cout<<"\n"<<mesg;
}
cout<<"\n EXITING MAIN()";
}
```

### **OUTPUT**

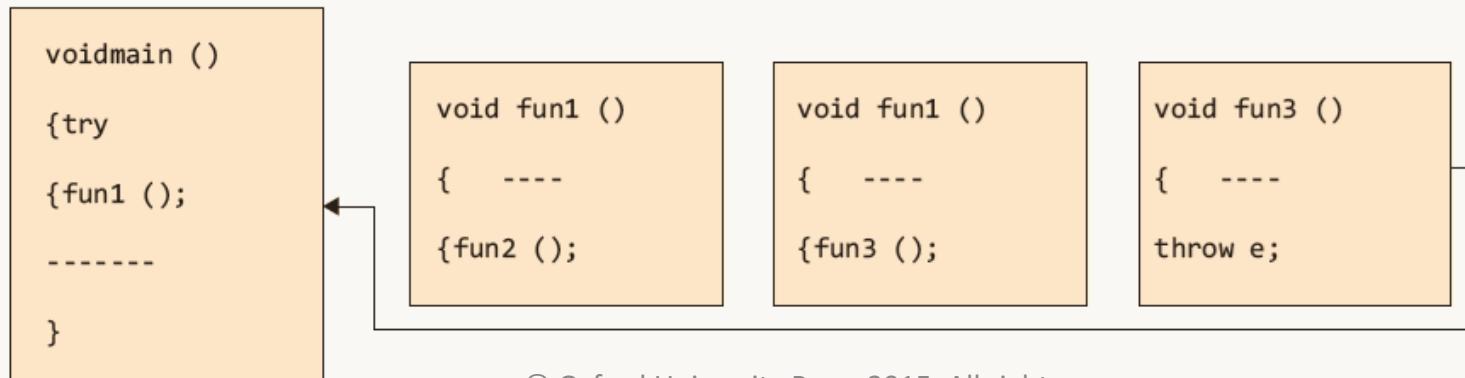
Enter the two numbers : 9 0

Divide by zero exception

EXITING MAIN()

# Stack Unwinding

- **Stack unwinding**, which is related to exception handling, is the process of removing function entries from function call stack at run time.
- When an exception is thrown, the function call stack is searched for an exception handler.
- In this searching process, all the entries before the function with exception handler are removed from the function call stack.
- Therefore, exception handling performs stack unwinding if the thrown exception is not handled in same function.



# Rethrowing Exception

- In some situations, the exception handler in the catch block may decide to re-throw the exception without processing it.
- To re-throw an exception, the syntax is as follows:
- `throw;`
- A throw statement without any exception explicitly mentioned causes the current exception to be thrown to the next enclosing try/catch block which catches it and handles it in the specified manner.

```
#include<iostream.h>
#include<conio.h>
void divide(int a, int b)
{ try
{ if(b==0)
    throw ("Divide by zero exception");
    else
    { float res = (float)a/b;
      cout<<"\n RESULT = "<<res;
    }
}
catch(char *mesg)
{ cout<<"\n In TRY CATCH block of called function";
  cout<<"\n RE THROWING EXCEPTION";
throw;
}
}
main()
{ int num1, num2;
  cout<<"\n Enter the two numbers : ";
  cin>>num1>>num2;
  try
  { divide(num1, num2);
  }
  catch(char *mesg)
  { cout<<"\n In TRY CATCH block of MAIN()";
    cout<<"\n"<<mesg;
  }
  cout<<"\n EXITING MAIN()";
getche();
}
```

# Restricting the Exceptions that can be Thrown

```
typefunction_name(arg list) throw (type list)
{ .....
.....
#include<iostream.h>
#include<conio.h>
voidcheck_number(inti) throw(int)
{ if(i==0)
    throw(i); // exception 1
else if(i== -1)
    throw (float)i; // exception 2 occurs but not thrown
else throw (char)i; // exception 3 occurs but not thrown
}
void main()
{ intnum;
cout<<"\n Enter a number (-1,0 or 1) : ";
cin>>num;
try
{ check_number(i);
}
catch(int)
{ cout<<"\n Caught an integer";
}
catch(float)
{ cout<<"\n Caught a floating point number";
}
catch(char)
{ cout<<"\n Caught a character";
}
cout<<"\n Exiting MAIN()";
getche();
}
```

List of exceptions  
that can be raised

## OUTPUT

```
Enter a number (-1,0 or 1) : 1
Abnormal Termination
```

# Catching Class Type as Exceptions

**Programming Tip:** A return in catch block forces return from the function that contains the catch block.

```
#include<iostream.h>
#include<conio.h>
#include<string.h>
class Exception
{ private:
    int errno;
    char *err_msg;
public:
    Exception(int en, char* em)
    { _errno = en;
        intlen = strlen(em);
        err_msg = new char[len+1];
        strcpy(err_msg, em);
    }
    void display()
    { cout<<"\n ERROR NO. = "<<errno<<"\t ERROR = "<<err_msg;
    }
};
void divide(int a, int b)
{ if(b==0)
    throw Exception(0, "Divide by Zero Error"); //throwing class type exception
else
{ float res = (float)a/b;
    cout<<"\n RESULT = "<<res;
}
}
main()
{ int num1, num2;
    cout<<"\n Enter the two numbers : ";
    cin>>num1>>num2;
    try
    { divide(num1, num2);
    }
    catch(Exception e) //catching class type exception
    { e.display();
    }
    cout<<"\n EXITING MAIN()";
getche();
}
```

# EXCEPTIONS IN CONSTRUCTORS

- The possibility of an exception while initializing the object of a class cannot be ignored.
- The key concern here is that when an exception occurs while the constructor has not yet fully executed, then it must have had reserved memory for some data members.
- The allocated memory cannot be freed using destructor because the moment the exception occurs, the program control goes out of the object's context. This may lead to memory **leak problems**.
- To prevent such memory leak problems, the exception handling mechanism must be implemented in the constructor itself.
- This will enable programmers to free the memory allocated for data members before handling the exception.

# EXCEPTIONS IN DESTRUCTORS

- As in the case of constructor, the exception handling routine must be defined in the destructor.
- This is important because an exception may be raised during the execution of the destructor.
- This may result in memory leak problems when the destructor is unable to release all the reserved space before the exception occurred.
- This feature helps programmers to write secure operations.

**Programming**  
**Tip:** Function calls that you anticipate might produce an exception must be enclosed in braces and preceded by the keyword try.

```
#include<iostream.h>
#include<conio.h>
class Divide
{ private:
    int *num1, *num2;
public:
    Divide()
    { try
    {
        num1 = new int();
        num2 = new int();
        cout<<"\n Enter the two numbers : ";
        cin>>*num1>>*num2;

        if(*num2==0)
            throw("Divide by zero error will be generated"); //exception 1 from constructor
    }
    catch(char *mesg)//exception caught
    { delete num1;
      delete num2;
      throw; //rethrowing exception
    }
}
~Divide()
{ try
{ delete num1;
  delete num2;
// throw;
}
catch(...)
{ cout<<"\n Exception generated in destructor";
}
}
float division()
{ return (float)*num1/ *num2;
}
};

main()
{ try
{ Divide D;
  float res = D.division();
  cout<<"\n RESULT = "<<res;
}
catch(char *mesg) // not executed since class exception handler is invoked
{ cout<<mesg;
}
cout<<"\n EXITING MAIN()";
getche();
}
```

# EXCEPTIONS IN OPERATOR OVERLOADED FUNCTIONS

```
#include<iostream.h>
#include<conio.h>
class Complex
{ private:
    int real, imag;
public:
    class Error{};
    Complex(int r, int i)
    {   real = r;
        imag = i;
    }
    void display()
    {   cout<<real<<" + "<<imag<<"i";
    }
    Complex& operator+=(Complex &c)
    {   if(real == 0 &&c.real == 0)
        throw Error();
        else
        {   real += c.real;
            imag += c.imag;
            return *this;
        }
    }
};

main()
{   Complex c1(0,2), c2(0,4);
    try
    {
        c1+=c2;
        c1.display();
    }
    catch(Complex :: Error)
    {
        cout<<"\n Add Zero Exception";
    }
    cout<<"\n Exiting MAIN()";
    getch();
}
```

# EXCEPTIONS AND INHERITANCE

```
#include<iostream.h>
#include<conio.h>
class Base {};
class Derived: public Base
{};
void main()
{ try
{ throw Derived();
}
catch (Base B)
{
    cout<<"\n Base Class Exception Caught";
}
catch (Derived D)
{
    cout<<"\n Derived Class Exception Caught";
}
getch();
}
```

## OUTPUT

Base Class Exception Caught

# EXCEPTIONS AND TEMPLATES

```
#include<iostream.h>
#include<conio.h>
#include<stdlib.h>
#define MAX 10
template <class Type>
void insert(Type arr[], Type num, int &n)

{ if(n==MAX)
    throw "OUT OF BOUND EXCEPTION"; // Exception thrown
else
{ arr[n++] = num;
}
}

void main()
{ int arr[10], i, num, n;
cout<<"\n Enter the number of elements : ";
cin>>n;
cout<<"\n Enter the array elements : ";
for(i=0;i<n;i++)
cin>>arr[i];
cout<<"\n Enter the number to be inserted : ";
cin>>num;
try
{ insert(arr, num, n);
}
catch(char *mesg) // exception caught
{ cout<<mesg;
exit(1);
}
cout<<"\n The Array is : ";
for(i=0;i<n;i++)
cout<<"\t"<<arr[i];
getch();
}
```

# HANDLING UNCAUGHT EXCEPTIONS

- If no exception is generated, all the statements in the try block are executed and the control goes to the statement immediately following the catch block.
- If an exception occurs, it is thrown and then the following steps are taken to handle the exception.
- The control goes to the first catch block to check if the exception type matches with that of the catch block.
- If a match is not found, then the control passes to the next catch until a match is found.
- If a match is found then the body of that catch block is executed.
- In case no match is found, then the compiler looks for a catch(...). If it is present, the block is executed; otherwise, terminate() is invoked.

```
#include<iostream.h>
#include<conio.h>
class Error{};
void main()
{ try
{ cout<<"\n Throwing Exception";
  throw Error();
}
catch(int)
{ cout<<"\n It will never get executed";
}
getche();
}
```

### **OUTPUT**

Throwing Exception  
Abnormal Program Termination

```
#include<iostream.h>
#include<except.h>
#include<conio.h>
class Error{};
void My_Terminate_Func() // invoked to handle exception
{
    cout<<"\n In My Terminate Function";
    cout<<"\n Abnormal Program Termination";
    getch();
    exit(1);
}
void main()
{
    set_terminate(My_Terminate_Func);
    try
    {
        cout<<"\n Throwing Exception";
        throw Error(); // exception thrown
    }
    catch(int) //exception uncaught
    {
        cout<<"\n It will never get executed";
    }
}
```

**Programming Tip:** catch(...), if included must always be the last catch, unless it is the only catch of the clause.

```
#include<iostream.h>
#include<conio.h>
class Error{};
void main() throw(int)
{ try
  { cout<<"n Throwing Exception of class Error Type";
    throw Error();
  }
  catch(int)
  { cout<<"n Caught an Exception";
  }
  cout<<"n EXITING MAIN()";
  getch();
}
```

**Programming Tip:** A function without an exception specification allows any object to be thrown from the function.

```
#include<iostream.h>
#include<except.h>
#include<conio.h>
#include<process.h>
class Error{};
void My_Unexpected()
{   cout<<"\n In My Unexpected Function";
    cout<<"\n Program Terminating Abnormally";
    getch();
    exit(1);
}
void Func() throw(int)
{   cout<<"\n Throwing Exception of class Error Type";
    throw Error();
}
void main()
{   set_unexpected(My_Unexpected);
    try
    {   Func();
    }
    catch(...)
    {   cout<<"\n Caught an Exception";
    }
    cout<<"\n EXITING MAIN()";
}
```

# STANDARD EXCEPTIONS

| Type of exception                                                                                                                                    | Position when it was generated                                                                                                                                                                                                                                 |
|------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Logic errors <ul style="list-style-type: none"><li>• domain_error</li><li>• invalid_argument</li><li>• length_error</li><li>• out_of_range</li></ul> | Logical errors in the program <ul style="list-style-type: none"><li>• invalid function argument</li><li>• invalid arguments in C++ standard functions</li><li>• length of object exceeds maximum allowable length</li><li>• array index out of range</li></ul> |
| Run-time error <ul style="list-style-type: none"><li>• range_error</li><li>• overflow_error</li><li>• underflow_error</li></ul>                      | Error during execution <ul style="list-style-type: none"><li>• error in standard template library (STL) container</li><li>• overflow in STL container</li><li>• underflow in STL container</li></ul>                                                           |
| bad_alloc                                                                                                                                            | Memory could not be allocated                                                                                                                                                                                                                                  |
| bad_exception                                                                                                                                        | Exception types does not match type of any catch block                                                                                                                                                                                                         |
| ios_base::failure                                                                                                                                    | Error in processing external file                                                                                                                                                                                                                              |
| bad_typeid                                                                                                                                           | Error in typeid                                                                                                                                                                                                                                                |
| bad_cast                                                                                                                                             | Error in dynamic cast                                                                                                                                                                                                                                          |

# ADVANTAGES

- Exception handling helps programmers develop fault-tolerant systems.
- Exception handling separates the error handling code from the program code.
- C++ allows exceptions to be handled outside of the regular code.
- Functions can handle any exception they choose.
- With exception handling, the program does not end abruptly.

# WORD OF CAUTION

- Do not throw an exception unless absolutely necessary.
- If an exception is thrown before the execution of constructor is complete, the destructor for that object will not be called.
- Throwing an exception deep inside the nested function(s) should be avoided.
- Programmers must keep in mind that in exception handling routines, implicit type conversion does not take place for primitive types.
- For example, in the following program, "x" is not implicitly converted to int.

# WORD OF CAUTION

- A derived class exception should be caught before a base class exception.
- If an exception is raised, all objects created inside the enclosing try block are destructed before the control is transferred to catch block.



OXFORD  
HIGHER EDUCATION

# Object Oriented Programming with C++



Reema Thareja

OXFORD  
HIGHER EDUCATION

# Object Oriented Programming with C++

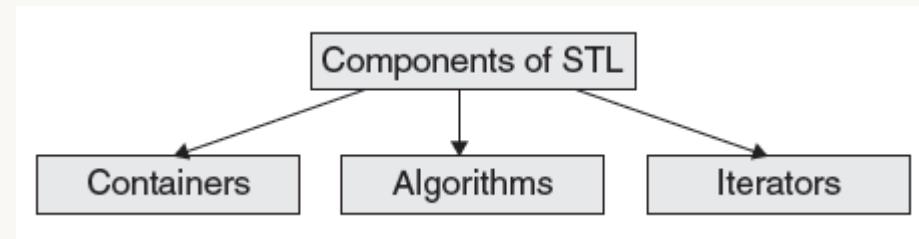
Reema Thareja

# Chapter Sixteen

## STL and New Features in C++

# STANDARD TEMPLATE LIBRARY (STL)

- STL is a collection of classes that provide templated containers, algorithms, and iterators.
- It contains common classes or algorithms so that programmers can use them without having to write and debug the classes.
- The STL components, which are now a part of C++ library, are defined in the namespace `std` .
- Therefore, all programs that make use of STL components must have the directive using `namespace std;` .



# CONTAINERS

Containers are objects that

- store data
- define the manner in which data will be stored
- are implemented using templates, and can, therefore, store data of different data types.

| Category               | Description                                                                                                                                                                                                                                                                                                                                                                       | Types                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |
|------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Sequence containers    | <ul style="list-style-type: none"> <li>• Data has a linear sequence</li> <li>• Each element is related to other elements by its position along the line</li> <li>• Elements can be accessed using iterator</li> </ul>                                                                                                                                                             | <p>Vector—Dynamic array that allows insertion/deletion at the end. Permits direct access to an element. Defined in header file as <code>&lt;vector&gt;</code>.</p> <p>List—A bi-direction linear list that allows insertion/deletion at any position. Defined in header file <code>&lt;list&gt;</code>.</p> <p>Dequeue—A double-ended queue that allows insertions and deletions at both ends. Permits direct access to any element. Defined in header file <code>&lt;deque&gt;</code>.</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |
| Associative containers | <ul style="list-style-type: none"> <li>• Supports direct access to elements using keys</li> <li>• No sequential ordering of elements</li> <li>• Data is sorted while being input</li> <li>• Data is stored using a tree structure</li> <li>• Facilitates fast searching, insertion and deletion</li> <li>• Slow for random access</li> <li>• Not efficient for sorting</li> </ul> | <p>Set and multiset—Store multiple data elements and provide functions to manipulate elements using their key values. A multiset is the same as a set, the only difference is that multiset allows duplicate data but a set does not. For example, a set or a multiset may be used to store objects of employee class that are ordered based on their Employee_ID as keys. Defined in header file <code>&lt;set&gt;</code>.</p> <p>Map and multimap—Stores data as pairs of key and value. While key is used for indexing and sorting, value is used to store data. Allows manipulation of values using keys. Map and multimap are same, the only difference is that map allows only one key for a given value but a multimap permits use of multiple keys for a value. A dictionary is an example of a multimap where, key is the word and value is the meaning of that word. Moreover, a word can have multiple meanings and multiple words can have the same meaning.</p> |
| Derived containers     | <ul style="list-style-type: none"> <li>• Created from sequence containers</li> <li>• Does not support iterators</li> <li>• Because of no iterator, they cannot be used for data manipulation</li> <li>• Support two member functions—<code>push()</code> and <code>pop()</code> for inserting and deleting elements</li> </ul>                                                    | <p>Stack—It is a last in first out (LIFO) data structure (in which data that was last inserted is first to be taken out). Insertion and deletion can be done only at one end.</p> <p>Queue—It is a First in first out (FIFO) data structure (in which data that was first inserted is first to be taken out). Insertion is done at one end and deletion from the other.</p> <p>Priority queue—The first element to be taken out is the highest priority of the element.</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |

# ALGORITHMS

- Reinforcing the concept of reusability, STL algorithms consist of functions that programmers can directly use in their code to process data stored in different containers for processing their data.
- Although each type of container has its own set of functions to perform basic operations, STL defines more than 60 algorithms to facilitate users to perform extended and complex operations.
- These STL algorithms categorized under five types.
- They are neither declared as member functions of any container class nor are defined as **friend functions**.
- They can be accessed by including the header file `<algorithm>` .

| Function                                    | Purpose                                                           |
|---------------------------------------------|-------------------------------------------------------------------|
| • <code>for_each</code>                     | Applies a piece of code to a range of elements                    |
| • <code>count/count_if</code>               | Gives the number of elements that satisfies a specific criteria   |
| • <code>equal</code>                        | Determines if two sets of elements are the same                   |
| • <code>find/find_if/find_if_not</code>     | Finds the first element that meets the given criteria             |
| • <code>find_end</code>                     | Finds the last sequence of elements in a certain range            |
| • <code>find_first_of</code>                | Searches for any one of a set of elements                         |
| • <code>adjacent_find</code>                | Finds the first two adjacent items that are equal                 |
| • <code>search</code>                       | Searches for a range of elements                                  |
| • <code>search_n</code>                     | Searches for a sequence of a specified number of similar elements |
| • <code>copy/copy_if/ copy_n</code>         | Copies elements to a new location                                 |
| • <code>copy_backward</code>                | Copies elements in backwards order                                |
| • <code>fill</code>                         | Assigns elements a certain value                                  |
| • <code>fill_n</code>                       | Assigns a value to a number of elements                           |
| • <code>remove/remove_if</code>             | Removes elements                                                  |
| • <code>remove_copy/remove_copy_if</code>   | Copies elements after removing certain elements                   |
| • <code>replace/replace_if</code>           | Replaces elements that matches the condition                      |
| • <code>replace_copy/replace_copy_if</code> | Copies elements after replacing with specified value              |
| • <code>swap</code>                         | Swaps the values of two elements                                  |
| • <code>swap_ranges</code>                  | Swaps two ranges of elements                                      |

# ITERATORS

- Iterators are **pointer-like** entities that are used to access the elements stored in the container.
- Iterators are basically used for traversing the elements of container.
- This process of traversing from one element to the next is called iterating through the container.
- All containers provide two main types of iterators as follows:
  - `container :: iterator`, which provides a read/write iterator.
  - `container :: const_iterator`, which provides a read-only iterator.

| Type          | Direction             | Access type | Function            | Ability to be saved |
|---------------|-----------------------|-------------|---------------------|---------------------|
| Input         | Forward               | Linear      | Read                | No                  |
| Output        | Forward               | Linear      | Write               | No                  |
| Forward       | Forward               | Linear      | Read and Write both | Yes                 |
| Bidirectional | Forward backward both | Linear      | Read and Write both | Yes                 |
| Random        | Forward backward both | Random      | Read and Write both | Yes                 |

# Vector

- Vectors are sequence containers that represent dynamic arrays that can change in size.
- Like arrays, vectors store its elements in contiguous memory locations.
- The main advantage of vectors is that it may grow in size when number of elements increases and shrink in size when a number of elements are deleted.

```
#include <vector>
#include <iostream>
#include <conio>
int main()
{
using namespace std;
intnum;
vector<int>vect;
cout<<"\n Enter the elements of the vector : ";
for (int i=0; i<10; i++)
{ cin>>num;
vect.push_back(num); // insert at end of array
}
cout<<"\n Elements of the vector are : \n";
for (int i=0; i<vect.size(); i++)
cout<<vect[i] << " ";
// iterator points to the first element
vector<int> :: iterator itr = vect.begin();
// inserts one element 25 at the 4th location from the beginning
vect.insert(itr+4,1,25);
//Deleting an element
vect.erase(vect.begin() + 8); // deletes the 6th element
cout<<"\n Contents after inserting an element at the 4th position and 6th
element : \n";
for (int i=0; i<vect.size(); i++)
cout<<vect[i] << " ";
getche();
}
```

#### OUTPUT

| Function    | Purpose                                               |
|-------------|-------------------------------------------------------|
| push_back() | Adds an element at the end                            |
| erase()     | Deletes elements                                      |
| insert()    | Inserts elements                                      |
| size()      | Gives the number of elements                          |
| at()        | Gives a reference to the specified element            |
| back()      | Gives a reference to the last element                 |
| begin()     | Gives a reference to the first element                |
| capacity()  | Gives the capacity of the vector                      |
| clear()     | Deletes all elements from the vector                  |
| empty()     | Checks if the vector is empty                         |
| end()       | Gives a reference to the end of the vector            |
| pop_back()  | Deletes the last element                              |
| resize()    | Changes the size of the vector                        |
| swap()      | Interchanges the values stored at specified locations |

# Deque

- A deque is a double-ended queue that allows insertions and deletions at both ends.

```
#include <iostream>
#include <deque>
#include <conio.h>
int main()
{
    using namespace std;
    int num;
    deque<int>deq;
    cout<<"\n Enter the elements in the dequeue : ";
    for (int i=0; i<5; i++)
    {
        cin>>num;
        deq.push_back(num); // insert at end of array
        cin>>num;
        deq.push_front(num); // insert at front of array
    }
    cout<<"\n The contents of the dequeue are : ";
    for (int i=0; i<deq.size(); i++)
        cout<< "\t"<<deq[i];
    getch();
}
```

## OUTPUT

```
Enter the elements in the dequeue : 1 2 3 4 5 6 7 8 9 10
The contents of the dequeue are : 10 8 6 4 2 1 3 5 7 9
```

# List

- A list is a container in which element points to the next and previous elements in the list.
- It supports only sequential or linear access. Therefore, if you want to access the fourth element, you cannot access it directly.
- You will have to start from the beginning and traverse through the list until the desired element is found.

```
#include <list>
#include <stdlib>
#include <iostream>
#include <conio>
void main()
{
using namespace std;
int num, option;
list<int> L1;
list<int> L2;
list<int> :: iterator itr;
    cout<<"\n Enter the elements in the first list : ";
    for (int i=0; i<5; i++)
    {   cin>>num;
        L1.push_back(num); // insert at end of list
    }
cout<<"\n Enter the elements for the second list : ";
    for (int i=0; i<5; i++)
    {   cin>>num;
        L2.push_back(num); // insert at end of list
    }
cout<<"\n LIST 1 is : ";
for(itr = L1.begin();itr!=L1.end();itr++)
    cout<<"\t"<<*itr;
cout<<"\n LIST 2 is : ";
for(itr = L2.begin();itr!=L2.end();itr++)
    cout<<"\t"<<*itr;
L1.reverse();
cout<<"\n LIST 1 is : ";
for(itr = L1.begin();itr!=L1.end();itr++)
    cout<<"\t"<<*itr;
L2.sort();
cout<<"\n LIST 2 is : ";
for(itr = L2.begin();itr!=L2.end();itr++)
    cout<<"\t"<<*itr;
L1.merge(L2);
cout<<"\n MERGED LIST is : ";
for(itr = L1.begin();itr!=L1.end();itr++)
    cout<<"\t"<<*itr;
getche();
}
```

| Function                  | Purpose                                            |
|---------------------------|----------------------------------------------------|
| <code>empty()</code>      | Tests whether container is empty                   |
| <code>size()</code>       | Returns size of the list                           |
| <code>max_size()</code>   | Returns maximum size of the list                   |
| <code>front()</code>      | Accesses the first element                         |
| <code>back()</code>       | Accesses the last element                          |
| <code>push_front()</code> | Inserts the element at the beginning               |
| <code>pop_front()</code>  | Deletes the first element                          |
| <code>push_back()</code>  | Inserts the element at the end                     |
| <code>insert()</code>     | Inserts elements                                   |
| <code>erase()</code>      | Deletes elements                                   |
| <code>swap()</code>       | Swaps elements                                     |
| <code>resize()</code>     | Changes the size of the list                       |
| <code>clear()</code>      | Removes all the elements                           |
| <code>remove()</code>     | Removes a specific element                         |
| <code>remove_if()</code>  | Removes element(s) that match a specific condition |
| <code>merge()</code>      | Merges two lists                                   |
| <code>unique()</code>     | Removes duplicate elements                         |
| <code>sort()</code>       | Sorts the list                                     |
| <code>reverse()</code>    | Reverses the list                                  |

# Maps

- Maps are implemented as associative array in which key is specified using the subscript operator as `Scode[“Delhi”] = 011 .`
- To insert an entry in the map, write `Scode["Chandigarh"] =072.`
- Map supports functions such as `begin()`, `clear()`, `empty()`, `end()`, `erase()`, `find()`, `insert()`, `size()`, and `swap()` .

```
#include <map>
#include <iostream>
#include <conio>
void main()
{
using namespace std;
map<char[],int>STDcode;
    char area[20];
int code;
STDcodeScode;
cout<<"\n Enter the city and its code : ";
for(int i=0;i<4;i++)
{
    cin.getline(area, 20);
    cin>>code;
Scode[area] = code;
}
Scode["Chandigarh"] = 072; //one way to insert
Scode.insert(pair<char *, int>("Bangalore", 080)); //second way to insert
intlen = Scode.size();
cout<<"\n Number of entries in map = "<<len;
cout<<"\n Contents of Map include : \n";
STDcode :: iterator itr;
for(itr=Scode.begin(); itr!=Scode.end();itr++)
    cout<<(*itr).first<<"\t"<<(*itr).second;
getche();
}
```

# STRING CLASS

- C++ provides its users a string class that has member functions to manipulate the string objects. Moreover, string is treated as a container class.
- Therefore, all the algorithms that are applicable on containers can also be used with string objects.
- To create a string object, the string class has three constructors as follows:
  - String() that creates an empty string.
  - String(const char \*str) that creates a string object from a null-terminated string.
  - String(const string &str) that creates a string object from another object.

| Functions                                                                                                                                                                | Operators                                  |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------|
| append(), assign(), at(), begin(),<br>capacity(), compare(), empty(), end(),<br>erase(), find(), insert(), length(), max_<br>size(), replace(), resize(), size(), swap() | =, +, +=, ==, !=, <, <=, >, >=, [], <<, >> |

```
if(option==1)
{   string S2(S1);
    cout<<"\n The duplicate string contains : "<<S2;
}
if(option==2)
{   string substr;
    cout<<"\n Enter the string to be inserted in the main string : ";
cin>>substr;
cout<<"\n Enter the position at which substring has to be inserted : ";
cin>>pos;
    S1.insert(pos, substr);
cout<<"\n The modified string is : "<<S1;
}
if(option==3)
{   cout<<"\n Enter the position from which substring has to be
deleted : ";
    cin>>pos;
cout<<"\n Enter the number of characters to be deleted : ";
cin>>num;
    //S1.erase(pos, num);
}
if(option==4)
{   string s2(" Welcome ");
    S1.replace(5,3,s2);      //starting from position 5 3 characters
    will be replaced with s2
cout<<"\n The modified string is : "<<S1;
}
if(option==5)
{   string S2("Hello");
    if(S1== S2)
cout<<"\n The two strings are equal";
    if(S1>S2)
cout<<"\n"<<S1<<" is greater than "<<S2;
```

# DATA TYPE BOOLEAN

- The Boolean data type is used to declare a variable that can have only one of the two values—true(1) or false (0).
- To declare a Boolean variable, the **bool** keyword is used.
- The statement given next declares a Boolean variable and initializes it.
- `booleligibleToVote = false;`
- Boolean constant(s) can be used to check if the state of a variable, expression, or a function is true or false.

```
#include <iostream>
#include <conio>
void main()
{ int age;
    booleligibleToVote = false;
    while(1)
    {   cout<<"\n Enter your age : ";
        cin>>age;
        if(age>=18)
        {   eligibleToVote = true;
            cout<<"\n Cast your vote";
        }
        else
            cout<<"\n Sorry, you are not eligible";
        if(age<=0)
            break;
    }
    getch();_
}
```

## **OUTPUT**

```
Enter your age : 33
Cast your vote
Enter your age : 7
Sorry, you are not eligible
Enter your age : 0
```

# wchar\_t DATA TYPE

- ANSI C++ supports another data type, wchar\_t that is used to store a wide character.
- This means that a variable of wchar\_t reserves two bytes or 16 bits of memory instead of one byte.
- On some systems, it may also reserve four bytes.
- The wchar\_t data type was mainly incorporated to write programs for international distribution.
- Earlier, eight bits in a character was unable to support languages that have more than 255 characters.
- Therefore, to support languages that have a wider character set like Japanese, the wchar\_t data type was introduced.

```
#include <iostream.h>
#include <conio.h>
void main()
{ wchar_t wch;
    char ch;
wch = L'R';
ch = 'R';
cout<<"\n The value of character is : " <<ch;
cout<<"\n Size of the wide char is :" <<sizeof(wch);
cout<<"\n Size of the char is :" <<sizeof(ch);
getche();
} _
```

## OUTPUT

The value of character is : R  
Size of the wide char is : 2  
Size of the char is : 1

# static\_cast Operator

- To convert one type into another
- To convert a pointer to a base class to a pointer to a derived class (although such conversions are not always safe)
- To convert enumerated data types to integers and vice versa.
- To convert from void\* to any pointer type.
- The syntax for using the static\_cast operator is as follows:
- `static_cast<type>(expression);`
- Here, type specifies the destination data type of the expression.
- For example, to convert an integer into a floating point value, we need to write,

`double result = static_cast<double>(total/5);`

# static\_cast Operator

- A static\_cast can be used to convert an integer to a character but the resulting char may not have enough bits to hold the entire integer value.
- A static\_cast is not always a safe type of conversion, as it does no run-time type checking.
- This problem is more prominent when a static cast to an ambiguous pointer is made.

```
#include <iostream.h>
class Base
{
    // class members and their definition
};
class Derived : public Base
{
    // class members and their definition
};
void main()
{   Base *b;
    Derived *d;
    b = static_cast<Base*>(d);          // Safe conversion
    d = static_cast<Derived*>(b);        // Unsafe conversion
}
```

# const\_cast Operator

- We have seen that a variable or a pointer to any data can be converted into any another type but with the exception of const, volatile, and `__unaligned` qualifiers.
- To perform conversions for these exceptional variables, the `const_cast` operator is used.
- The `const_cast` operator overrides or in simpler terms removes the variable's constant, volatile, or unaligned status to perform the cast.
- The syntax for using the `const_cast` is same as that of `static_cast` and is given as follows:

`const_cast<type>(expression);`

```
#include <iostream>
#include <conio>>
//using namespace std;
void main()
{   intnum=10;
    constint *cptr = &num;
    int *ptr = const_cast<int *>(cptr);
    cout<<"\n Number = "<<*ptr;
    getch();
}
```

## OUTPUT

Number = 10

# reinterpret\_cast Operator

- The reinterpret\_cast operator allows any pointer to be converted into any other pointer type.
- For example, with reinterpret\_cast operator, char \* can be converted into int\* and vice versa.
- It is mainly used for performing the following tasks:
- To convert any integral value into any pointer type
- To convert any pointer type into any integral value
- To convert a pointer to one class into a pointer to another class, which may be entirely unrelated to each other. Therefore, it is unsafe.
- To convert a null pointer value to the null pointer value of the destination type
- To convert an enumeration type to a pointer.
- To convert a pointer to a function into a pointer to a function of a different type.
- To convert a pointer to a member of a class into a pointer to a member of a different class.

# reinterpret\_cast Operator contd.

- The syntax for using the reinterpret\_cast operator is  
**reinterpret\_cast<<type>>(expression)**

**Note** : const\_cast operator is used to remove the constness of object pointed by a pointer.

## Drawbacks

- Misuse of the reinterpret\_cast operator can be unsafe so as a good programming habit, it should be avoided until and unless very necessary.
- The reinterpret\_cast operator cannot cast the variables with const , volatile, or unaligned qualifiers.

# reinterpret\_cast Operator

- However, an important point to note here is that the type and the type of expression must be the same.
- They must only differ in their const and volatile qualifiers.
- `const_cast` operator is applied at the compile time

```
#include <iostream>
#include <conio>
void main()
{   intnum=10;
    int *ptr = &num;
cout<<"\n Number with *ptr = "<<*ptr;
void *vptr = reinterpret_cast<void*>(ptr) ;
cout<<"\n Integer pointer is now Void *";
int *ptr2 = reinterpret_cast<int*>(vptr);
cout<<"\n Number with *ptr2 = "<<*ptr2;
```

## OUTPUT

```
Number with *ptr = 10
Integer pointer is now Void *
Number with *ptr2 = 10
```

# dynamic\_cast Operator

- The dynamic\_cast operator is used for safe conversion.
- For this, it ensures that the result of type conversion points to a valid complete object of the destination pointer type.
- This means that dynamic\_cast operator can be used for upcast as well as downcast **polymorphic classes**.
- dynamic\_cast operator is also used to perform all permissible conversions on the following:
  - pointers
  - null pointers
  - pointers to different classes (even between unrelated classes)
  - pointer of any type to a void\* pointer.

# dynamic\_cast Operator

- The syntax for the dynamic\_cast operator is

dynamic\_cast<type>(expression);

**Note** : A reinterpret\_cast operator handles conversions between unrelated types.

- The operator returns a pointer to specified type, if successful, or returns null pointer to indicate failure when it is unable to cast a pointer because it is not a complete object of the required class.
- However, if it fails to convert a reference type then it throws an exception of type bad\_cast .

```
#include <iostream.h>
#include <conio.h>
#include <except.h>
class Base
{
    virtual void show()
    {   cout<<"\n In Base Class";
    }
};
class Derived: public Base{ };
void main()
{
    try
    { Base *bptr;
        Derived *dptr;
        bptr = dynamic_cast<Base *>(dptr);
        dptr = dynamic_cast<Derived *>(bptr);
    if(dptr==0)
        cout<<"\n Null pointer on dynamic cast";
    }
    catch(...)
    { cout<<"\n Exception";
    }
    getch();
}
```

### OUTPUT

Null pointer on dynamic cast

# typeid Operator

- The typeid operator is used to determine the type of an object at the run-time.
- For this, it retrieves the actual derived type of the object referred to by a pointer or a reference.
- It is usually used when it is difficult to determine the object type with the static information provided, for example, when working with reference to a class or pointers.
- To use the typeid operator, the programmers must include the <typeinfo> header file in their programs.

# typeid Operator

- The syntax for typeid operator is  
**typeid(type); or typeid(expression)**
- If type is specified, then the typeid operator returns the type of type.
- If expression is specified, then it evaluates the expression and returns type of value.
- In both cases, the actual type is known at the compile time and thus no run time overhead is caused.
- However, if the expression is a **polymorphic object** or if it evaluates to a zero, then the actual type of the class is queried dynamically with some runtime overhead.

# typeid Operator

## Drawbacks

- The dynamic\_cast operator can be only used with pointers and references to classes.
- Let us write a small program that demonstrates the use of dynamic\_cast operator.

```
#include <iostream.h>
#include <typeinfo.h>
#include <conio.h>
int main () {
intnum = 0;
float *ptr;
cout<<"\n Type ID of num = "<<typeid(num).name();
cout<<"\n Type ID of ptr = "<<typeid(ptr).name();
getche();
return 0;
}
```

## OUTPUT

```
Type ID of num = int
Type ID of ptr = float *
```

# explicit KEYWORD

- The explicit keyword is used with a single parameter constructor to prevent implicit conversion to take place.

```
#include <iostream>
#include <conio.h>
class Sample
{
    private:
        int data;
public:
    explicit Sample(int num)
    {
        data = num;
    }
};
void main()
{
    Sample S(99); // right
    Sample S = 99; //error - cannot convert int to Sample
}
```

# MUTABLE KEYWORD

- Sometimes, we may need to modify one or more data members of class or structure from within a constant function.
- In such a situation, when we want to modify only few data members (not all), we can prefix the data member declaration with the keyword **mutable** .
- The syntax for using declaring a variable as mutable is given as follows:

**mutable data type variable\_name;**

- Mutable allows a data member to be modified from within a constant member function.
- The second use of keyword mutable is to change the value of data member of a constant object.

```
#include <iostream>
#include <string>
#include <conio.h>
class Employee
{ private:
    char name[20]; // data member
    int emp_id;
    int exp;
    mutable float salary; //mutable data member
public:
    Employee(char n[], int id, int years, float sal)
    { strcpy(name, n);
        emp_id = id;
        exp = years;
        salary = sal;
    }
    void show_data() const // constant member function
    { cout<<"\n EMPLOYEE ID : "<<emp_id;
        cout<<"\n NAME : "<< name;
        if(exp>5)
            salary += 5000; //give loyalty bonus // no other data except salary can be modified
        cout<<"\n SALARY : "<<salary;
    }
};
void main()
{
    cout<<"\n\n With mutable data on a constant function";
    Employee e1("Richa Raina", 101, 7, 80000);
    e1.show_data();
    cout<<"\n\n With mutable data on a constant object";
    const Employee e2("Aditya Roy", 297, 6, 70000); // constant object
    e2.show_data();
    getch();
}
```

# NAMESPACES

- A namespace provides additional information to differentiate variables, classes, or functions from others with the same name available in different libraries.
- This means that namespace defines a scope that allows programmers to define the context in which names are defined.
- The syntax for defining a namespace is given as follows

```
namespace namespace_name
{   // variable, class or function declaration
}
```

```
namespaceMy_space
{
    bool flag = 1;
    void show()
    {
        cout<<"\n FLAG = "<<flag;
    }
}
```

```
#include <iostream>
#include <conio.h>
namespace Myfirst_namespace
{ void show()
    { cout<<"\n In First name space";
    }
}
namespace Mysecond_namespace
{ void show()
    { cout<<"\n In Second name space";
    }
}
void main()
{ Myfirst_namespace :: show();
    using namespace Mysecond_namespace;
    show();
    getch();
}
```

### **OUTPUT**

In First name space  
In Second name space

# Nested Namespaces

- C++ allows programmers to have nested namespaces in which one namespace is defined inside another namespace.

```
namespace namespace_name1
{ // variable, class or function declaration
    namespace namespace_name2
    { // variable, class or function declaration
    }
}
```

To access a member of nested namespace, write

```
using namespace namespace_name1 :: namespace_name2; OR
namespace_name1 :: or namespace_name2 :: function_name;
```

```
#include <iostream>
#include <conio.h>
namespace Myfirst_namespace
{
    void show()
    {   cout<<"\n In First name space";
}
    NamespaceMysecond_namespace
    {
        void show()
        { cout<<"\n In Second name space";
}
    }
}
void main()
{
    Myfirst_namespace :: show();
    using namespace Myfirst_namespace :: Mysecond_namespace;
    show();
    getch();
}
```

## OUTPUT

In First name space  
In Second name space

# Unnamed Namespaces

- C++ also allows programmers to define unnamed namespaces.
- By definition, an unnamed namespace is one that is specified without any name.
- It is primarily used to define global static members that are accessible in all scopes following its declaration.
- The syntax for defining an unnamed namespace is given as follows:

```
namespace
{  // variable, class or function declaration
}
```

```
#include <iostream>
#include <conio.h>
namespace
{ int num = 10;
    void show()
    { cout<<"\n In Unnamed name space"; }

}
void main()
{ cout<<"\n NUMBER = "<<num;
    show();
    getche();
}
```

## OUTPUT

```
NUMBER = 10
In Unnamed name space
```

# OPERATOR KEYWORDS

| Operator                | Operator keyword    | Usage                         |
|-------------------------|---------------------|-------------------------------|
| <code>&amp;&amp;</code> | <code>and</code>    | <code>res = a and b</code>    |
| <code>  </code>         | <code>Or</code>     | <code>res = a or b</code>     |
| <code>!</code>          | <code>Not</code>    | <code>res = a not b</code>    |
| <code>!=</code>         | <code>not_eq</code> | <code>res = a not_eq b</code> |
| <code>&amp;</code>      | <code>Bitand</code> | <code>res = a bitand b</code> |
| <code> </code>          | <code>Bitor</code>  | <code>res = a bitorb</code>   |
| <code>^</code>          | <code>Xor</code>    | <code>res = a xor b</code>    |
| <code>&amp;=</code>     | <code>and_eq</code> | <code>res and_eq a</code>     |
| <code>!=</code>         | <code>or_eq</code>  | <code>res or_eq a</code>      |
| <code>^=</code>         | <code>xor_eq</code> | <code>res xor_eq a</code>     |

# SPECIFYING HEADER FILES

- ANSI C++ has laid out a new way of including the header files.
- Besides supporting the old way `#include<iostream.h>` it also supports the new way as `#include<iostream>`.

| Old name                      | New name                     |
|-------------------------------|------------------------------|
| <code>&lt;assert.h&gt;</code> | <code>&lt;cassert&gt;</code> |
| <code>&lt;ctype.h&gt;</code>  | <code>&lt;cctype&gt;</code>  |
| <code>&lt;float.h&gt;</code>  | <code>&lt;cfloat&gt;</code>  |
| <code>&lt;limits.h&gt;</code> | <code>&lt;climits&gt;</code> |
| <code>&lt;math.h&gt;</code>   | <code>&lt;cmath&gt;</code>   |
| <code>&lt;stdio.h&gt;</code>  | <code>&lt;cstdio&gt;</code>  |
| <code>&lt;stdlib.h&gt;</code> | <code>&lt;cstdlib&gt;</code> |
| <code>&lt;string.h&gt;</code> | <code>&lt;cstring&gt;</code> |
| <code>&lt;time.h&gt;</code>   | <code>&lt;ctime&gt;</code>   |



OXFORD  
HIGHER EDUCATION

# Object Oriented Programming with C++



Reema Thareja

OXFORD  
HIGHER EDUCATION

# Object Oriented Programming with C++

Reema Thareja

# **Chapter Seventeen**

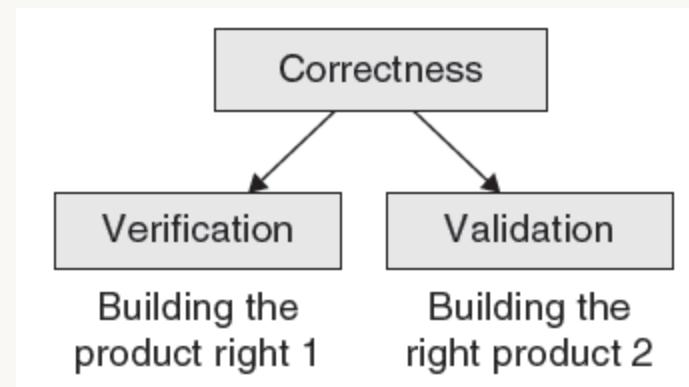
## **Object-Oriented System Analysis, Design, and Development**

# INTRODUCTION

- Systems development refers to the set of activities that need to be carried out for producing an information system.
- These activities include the following:
  - Analysis • Design • Coding • Testing • Maintenance
- Therefore, a software development methodology is a series of processes that results in the development of an application that achieves the ultimate goal based on system requirements.
- As software development process is never-ending, these activities mentioned will be continuously applied until the system is in operation.

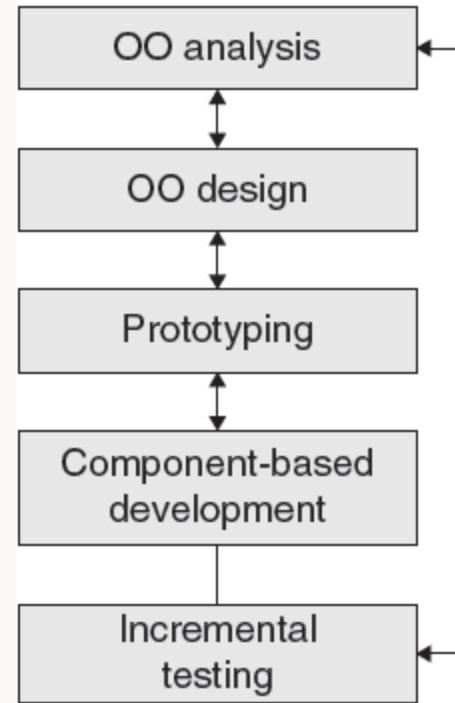
# BUILDING HIGH-QUALITY SOFTWARE

- The software process transforms the user's needs into a software solution. However, before deploying the software system in a user's environment, the software must be tested to ensure that it is free of bugs. A high-quality software product must not only satisfy user's requirements but also have minimal or no defects. It is always better to correct the errors before delivery to the users than correcting them post-delivery.
- The software system must be tested according to how it has been built or, what it should do. To ensure this, the software must be checked if it is corrected, verified, and validated.



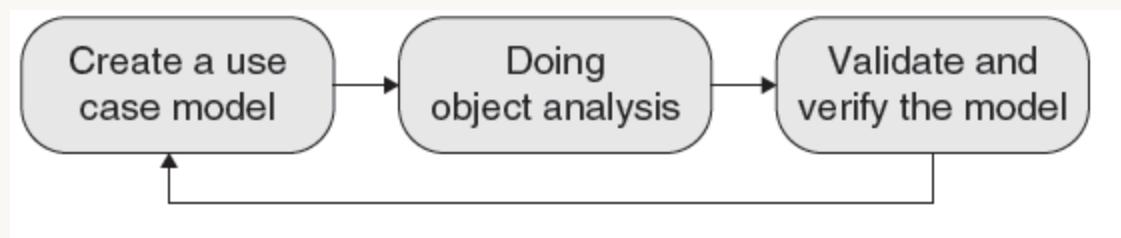
# OBJECT-ORIENTED SYSTEMS DEVELOPMENT

- OO-SDLC believes that in real world applications, a top down approach cannot be followed.
- Rather, real world solutions require a partial strict down and a partial bottom up approach.
- Another key difference between traditional SDLC and OO SDLC is that design is based on entities (called objects) and not on functions.



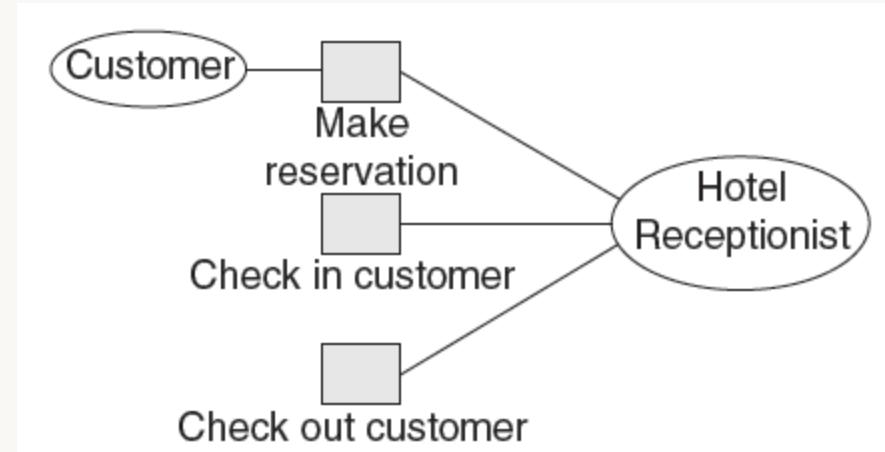
# Object-oriented Analysis

- This phase of object-oriented software development determines the system requirements and identifies classes along with their relationship to other classes in the problem domain.
- For this, the actors and the roles they play in the system are identified.
- It uses scenarios to understand the end user's requirements.
- However, to ensure that scenarios are completely documented, Ivar Jacobson came up with the concept of the use case, which is another name for a scenario that describes the user computer system interaction. The use case model.



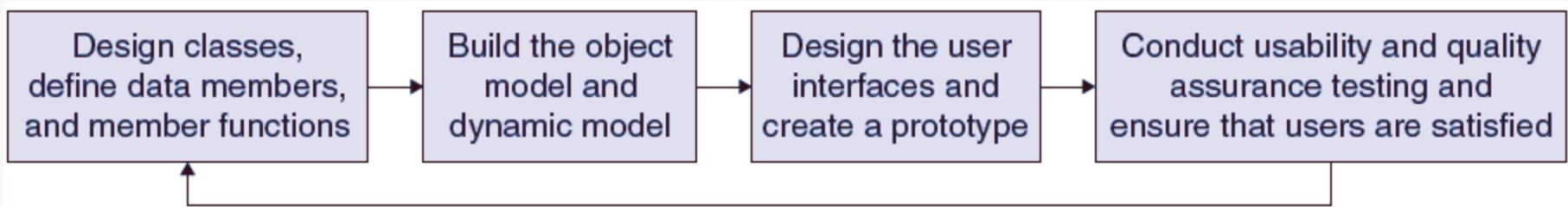
# USE CASE MODEL

- The technique of expressing these high-level processes and interactions with users in a scenario and analysing them is called use case modelling. The advantages of use case modelling are as follows:
- The use case model represents the user's view of the system or user's needs.
- The process of developing use cases is an iterative process.
- Once a use case is identified, the classes and their relationships are created to establish a workable model.



# Object-oriented Design

- This phase is used to design classes identified in the previous phase and the user interfaces.
- During the design phase, additional objects and classes that are used for implementing system requirement are identified.
- OO development is highly incremental; it enables you to start with analysis, move to its design, and refine and complete the other models of the system



# Object-oriented Design

- Some golden rules that must be followed in the design phase include the following:-
  - Gain knowledge about the existing classes.
  - Reuse existing classes rather than building new ones
  - Design a large number of simple classes, rather than designing a small number of complex classes.
  - Design member functions or methods.
  - Criticize your own work and if required, go back and refine the classes.

# Prototyping

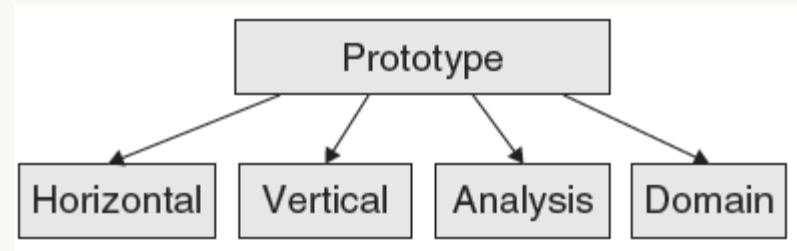
- A prototype is a version of the software which is developed quite early for specific experimental purposes.
- Enables users to comment on the usability and usefulness of the user interface design
- Enables to assess the fit between the software tools selected and the functional specification
- Can be used to define the use cases
- Simplifies use case modelling
- Gives an insight into the basic courses of action for developing use cases covered by the prototype
- Provides a means to test and refine the user interface

# Prototyping

- Gives a chance to the development team to enhance the usability of the system
- Confirms that the requirements' specification is correct
- Collects information about errors or other problems in the system that can be eliminated in the next prototype
- Gives the management and users a glimpse of what solution the software will provide Can be evaluated and tested for performance if supporting data is available.

# Types of Prototypes

- A **horizontal prototype** is used to simulate the interface.
- Such a prototype incorporates the entire user interface that will be in the complete version of the software system.
- A **vertical prototype** provides a subset of the system features with complete functionality.
- An **analysis prototype** is one that is used to explore the problem domain. It is specifically designed to demonstrate the proof of a concept.
- A **domain prototype** is used for incremental development of the complete software.

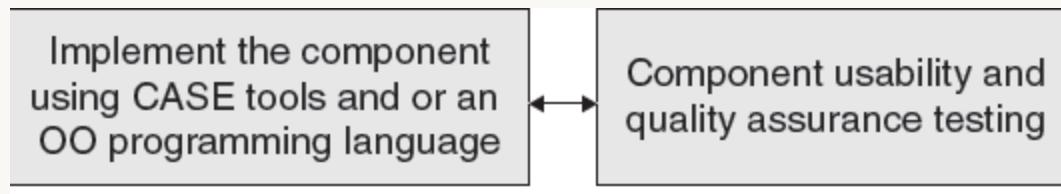


# Tools in OO Analysis and OO Design

- Class diagrams or templates Class diagrams model key entities in the problem domain and their relationships with each other. To develop a class diagram, the first step is to identify candidate classes and objects which may be people, places, things, organizations, concepts, or events.
- Next, the attributes and functions of each class are then identified and documented.
- Object diagrams They are used to model the interactions between objects.
- Object state diagrams Object state diagram models the dynamic behaviour within a single object. It shows all the possible states of an object and the allowed transition between the states.

# Implementing Component-based Development

- Files, functions, classes, executables, etc. are components of C++. We have seen that functions and classes are designed in such a way that they support reusability through the concepts of polymorphism, inheritance, containership, and templates.
- These days, the trend in software development is to build the solution from prefabricated components (or classes) that have already been tested.
- Existing components can be customized based on current requirements and delivered to the users.



# Implementing Component-based Development

- The key advantages of using a component based development model are as follows:
- Faster product development cycles
- Enhanced flexibility
- Better customization
- Reduced development time and effort
- More efficient systems
- Enables legacy integration through component wrapping

# UNIFIED MODELLING LANGUAGE

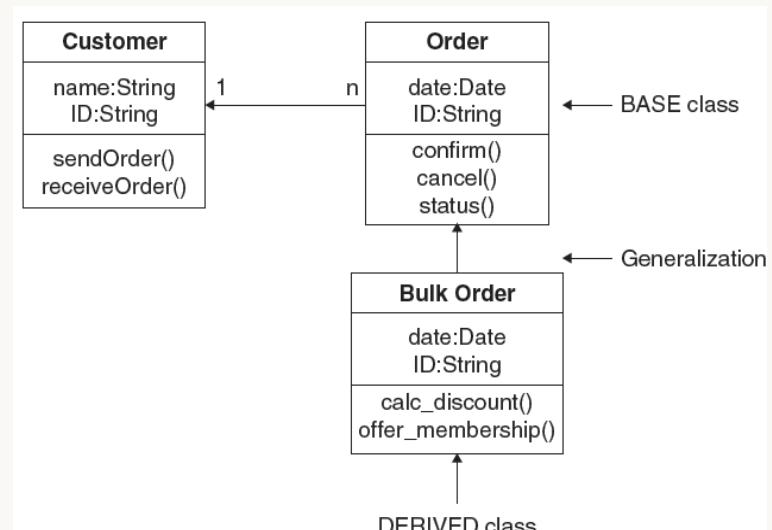
- Unified modelling language (UML) is an object-oriented system of notation that has finally evolved from the combined efforts of **Grady Booch, James Rumbaugh, and Ivar Jacobson**.
- UML is a general purpose visual modelling language that helps everybody associated with the system to visualize the system and software engineers to specify, construct, and document the software system.
- This means that UML diagrams are not only used by developers but also for business users and anybody interested to understand the system.

# UNIFIED MODELLING LANGUAGE

- With UML, any complex system can be easily understood by making some useful diagrams. A single diagram is not enough to cover all aspects of the system. Therefore, UML defines a variety of diagrams to cover most of the aspects of a system.

# Class Diagrams

- It is a static diagram that is used for visualizing, describing, and documenting different aspects of a system.
- The executable code is constructed using these diagrams, as it describes the attributes and operations of a class and also the constraints imposed on the system.
- The class diagram also known as a structural diagram shows a collection of classes, interfaces, associations, collaborations, and constraints.



# Class Diagrams

- While the class diagram represents abstract model consisting of classes and their relationships, object diagram, on the other hand, represents an instance at a particular moment, which is concrete in nature. This means that the object diagram is more close to the actual system behaviour.
- It not only helps programmers to understand the relationships clearly, but also helps in forward and reverse engineering.

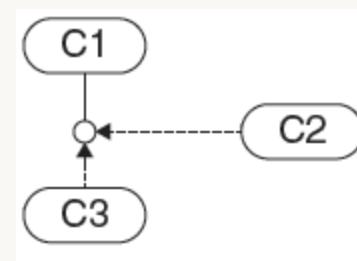
# Object Diagram

- They are derived from class diagrams to represent an instance of a class diagram. Although the basic concepts of both the diagrams are similar, object diagrams represent the static view of a system at a particular moment.
- They depict a set of objects and their relationships as an instance.



# Component Diagram

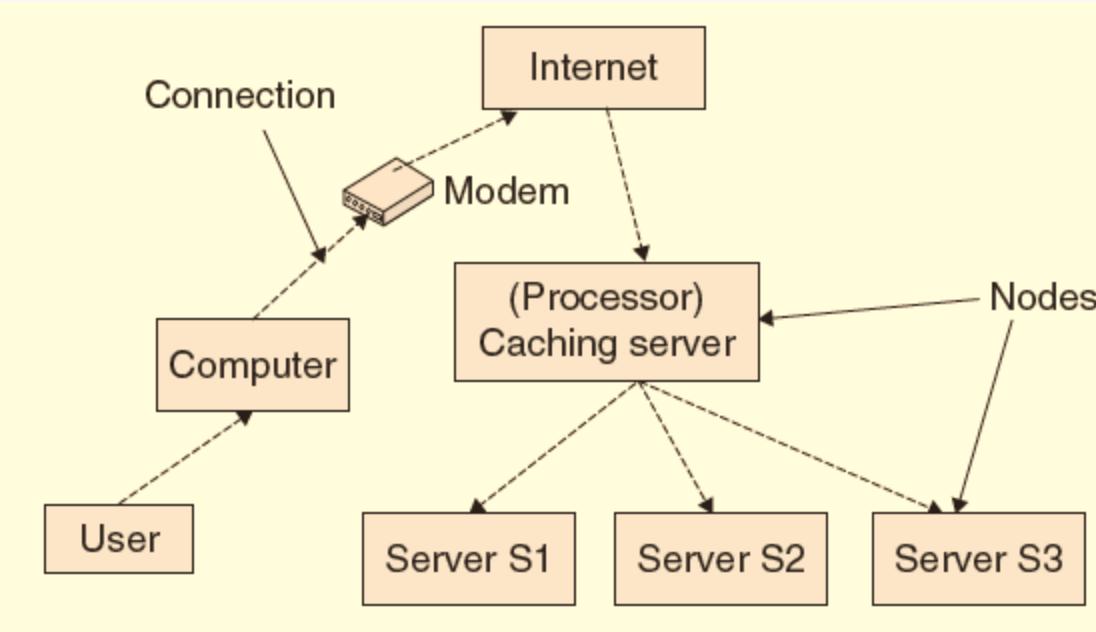
- Component diagrams are used to model physical aspects of a system like executables, libraries, files, documents, etc. which reside in a node.
- They are used to visualize the organization and relationships among components in a system and are very useful in the executable system, as they help the programmers to identify the components whose functionalities have to be used.
- A single component diagram may not be able to represent the entire system, so a collection of diagrams issued to represent the whole system.



# Deployment Diagram

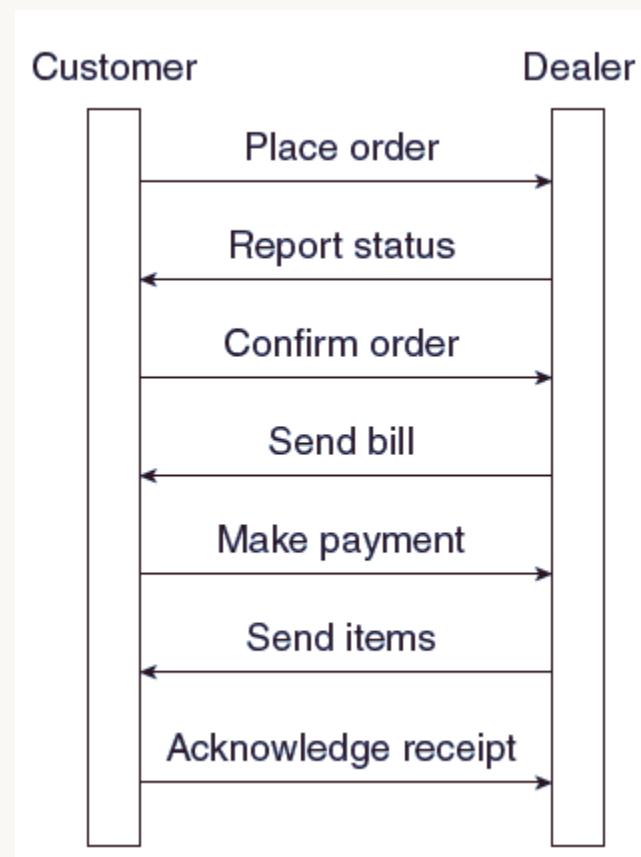
- It consists of nodes and their relationships to help the team visualize the topology of the physical components of a system where the software components are deployed.
- This means that the diagram describes the hardware components like nodes and connections where software components of the system are deployed.
- Component diagrams and deployment diagrams are quite similar and related to each other.
- While the component diagram describes the components to be used, deployment diagrams show how they are deployed in hardware.

# Deployment Diagram



# Sequence Diagram

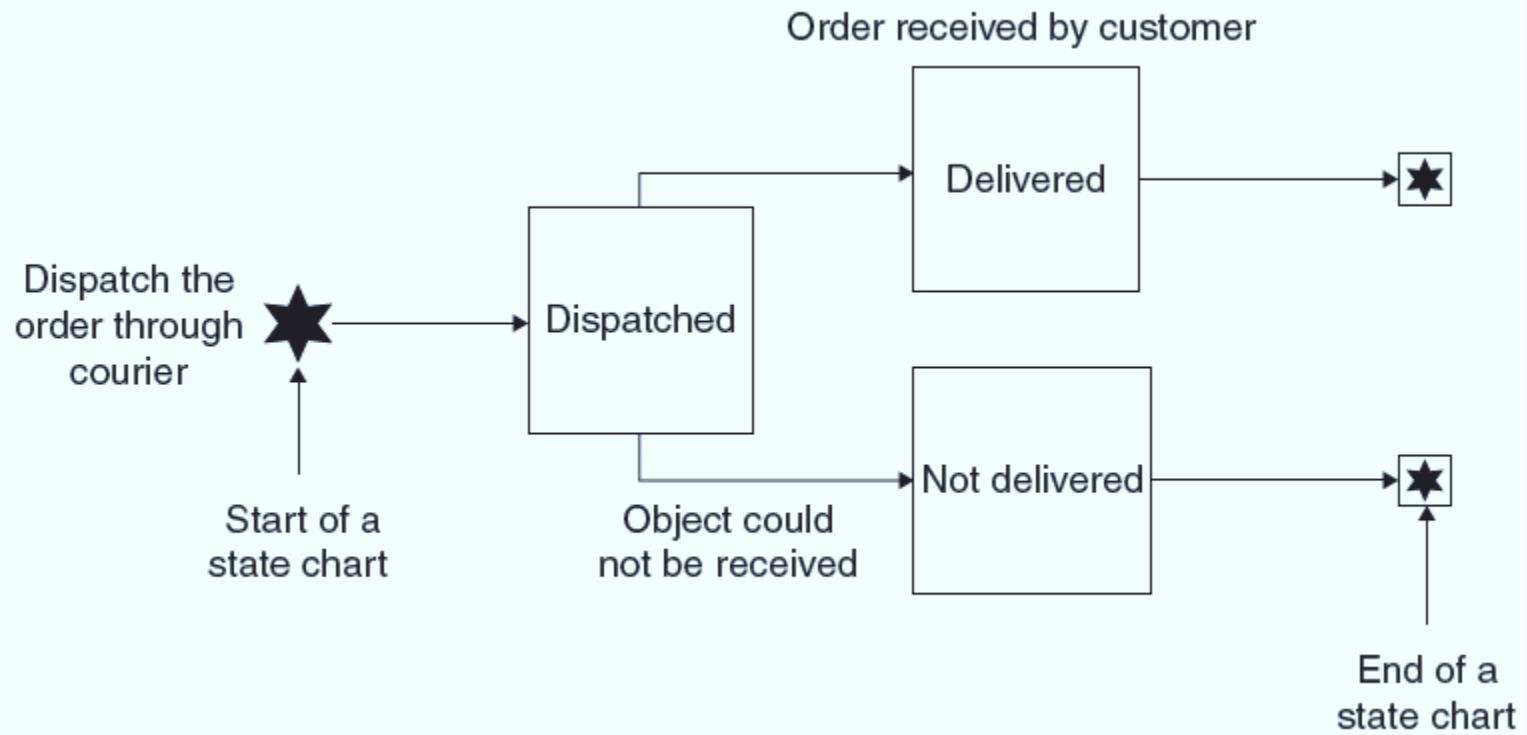
- It is a dynamic diagram that is used to describe some type of interactions among the different elements in the model.
- For this, it emphasizes on time sequence of messages that are exchanged between system elements.
- They are used to describe the message flow in the system and describe interaction among objects



# State Chart Diagram

- It is used to describe different states of a component or object in a system.
- These states that are controlled by external or internal events vary from one component or object to another and are clearly depicted by the state chart diagram.
- State chart diagrams are useful to model reactive systems (that respond to external or internal events) to describe the flow of control is defined as a condition in which an object exists and it changes when some event is triggered.
- Therefore, the objective of a state chart diagram is to model life time of an object from creation to its termination from one state to another state.

# State Chart Diagram



# Activity Diagram

- Activity diagram represents the dynamic aspects of the system.
- It is basically a flow chart to represent the flow from one activity to another activity where an activity means an operation of the system.
- The flow can be sequential, branched, or concurrent.
- Since activities denote the functions the system is supposed to perform, a number of activity diagrams are prepared to capture the entire flow in a system.
- Activity diagrams are used to get an idea of how the system will work when executed.

# Activity Diagram

