MINISTRY OF EDUCATION

DIPLOMA IN INFORMATION COMMUNICATION TECHNOLOGY

KENYA INSTITUTE OF CURRICULUM DEVELOPMENT
STUDY NOTES

OBJECT ORIENTED PROGRAMMING

MODULE 2: SUBJECT NO 3

Contents

CHAPTER 1: INTRODUCTION TO OBJECT ORIENTED PROGRAMMING

Define Object Oriented Programming

A type of programming in which programmers define not only the data type of a data structure, but also the types of operations (functions) that can be applied to the data structure. In this way, the data structure becomes an *object* that includes both data and functions. In addition, programmers can create relationships between one object and another. For example, objects can *inherit* characteristics from other objects.

Object-oriented programming (**OOP**) is a programming paradigm based on the concept of "objects", which are data structures that contain data, in the form of fields, often known as *attributes;* and code, in the form of procedures, often known as *methods.* A distinguishing feature of objects is that an object's procedures can access and often modify the data fields of the object with which they are associated (objects have a notion of "this"). In object-oriented programming, computer programs are designed by making them out of objects that interact with one another. There is significant diversity in object-oriented programming, but most popular languages are class-based, meaning that objects are instances of classes, which typically also determines their type.

Evolution of Object Oriented Programming

The object-oriented paradigm took its shape from the initial concept of a new programming approach, while the interest in design and analysis methods came much later.

- The first object–oriented language was Simula (Simulation of real systems) that was developed in 1960 by researchers at the Norwegian Computing Center.
- In 1970, Alan Kay and his research group at Xerox PARK created a personal computer named Dynabook and the first pure object-oriented programming language (OOPL) - Smalltalk, for programming the Dynabook.
- In the 1980s, Grady Booch published a paper titled Object Oriented Design that mainly presented a design for the programming language, Ada. In the ensuing editions, he extended his ideas to a complete object–oriented design method.
- In the 1990s, Coad incorporated behavioral ideas to object-oriented methods.

The other significant innovations were Object Modelling Techniques (OMT) by James Rumbaugh and Object-Oriented Software Engineering (OOSE) by Ivar Jacobson.

Programming paradigms

A **programming paradigm** is a fundamental style of computer **programming**, a way of building the structure and elements of computer programs.

The following are considered the main programming paradigms. There is inevitably some overlap in these paradigms but the main features or identifiable differences are summarized below:

- ✓ **Imperative programming** – defines computation as statements that change a program state
- ✓ **Procedural programming**, structured programming – specifies the steps the program must take to reach the desired state.
- ✓ **Structured programming** (sometimes known as *modular programming*) is a subset of procedural programming that enforces a logical structure on the program being written to make it more efficient and easier to understand and modify. Modular programming is the process of subdividing a computer program into separate sub-programs.
- ✓ **Declarative programming** – defines computation logic without defining its control flow.
- ✓ **Functional programming** – treats computation as the evaluation of mathematical functions and avoids state and mutable data
- ✓ **Object-oriented programming** (OOP) – organizes programs as *objects*: data structures consisting of datafields and methods together with their interactions.
- ✓ **Event-driven programming** – the flow of the program is determined by events, such as sensor outputs or user actions (mouse clicks, key presses) or messages from other programs or threads.
- ✓ **Automata-based programming** – a program, or part, is treated as a model of a finite state machine or any other formal automata.

Merits and demerits of OOP

Object-Oriented Programming has the following advantages over conventional approaches:

- ✓ OOP provides a clear modular structure for programs which makes it good for defining abstract datatypes where implementation details are hidden and the unit has a clearly defined interface.

- ✓ OOP makes it easy to maintain and modify existing code as new objects can be created with small differences to existing ones.

- ✓ OOP provides a good framework for code libraries where supplied software components can be easily adapted and modified by the programmer. This is particularly useful for developing graphical user interfaces.

Examples of object oriented languages

Many of the most widely used programming languages are multi-paradigm programming languages that support object-oriented programming to a greater or lesser degree, typically in combination with imperative, procedural programming. Significant object-oriented languages include C++, Objective-C, Smalltalk, Delphi, Java, C#, Perl, Python, Ruby and PHP.

Object Oriented Databases (OODBs)

An object-oriented database management system (OODBMS) is a database management system that supports the creation and modeling of data as objects. OODBMS also includes support for classes of objects and the inheritance of class properties, and incorporates methods, subclasses and their objects. Most of the object databases also offer some kind of query language, permitting objects to be found through a declarative programming approach.

Terms:
A hybrid database - is usually an object-oriented framework created to act as an interface between an "impure" object-oriented language like C++ and a relational database manager. The hybrid manager allows the language to access the database as though it were truly object-oriented, while leaving the database itself unchanged. The hybrid design allows the object-oriented programmer to use nearly any OOP feature that they want (much like an OODBMS), while keeping the database itself relational which allows the use of commercially available and supported products, allowing the "best of both worlds" at the cost of the run-time overhead to support the hybrid framework.

Persistent object oriented databases - data manipulated by the application is transient and data in the database is persisted (Stored on a permanent storage device). In object databases, the application can manipulate both transient and persisted data. Persistence is often defined as objects (and their classes in the case of OODBs) that outlive the programs that create them.

Pure object oriented databases is based solely on the object-oriented data model

CHAPTER 2: OBJECT ORIENTED PROGRAMMING CONCEPTS

Concepts associated with OOP

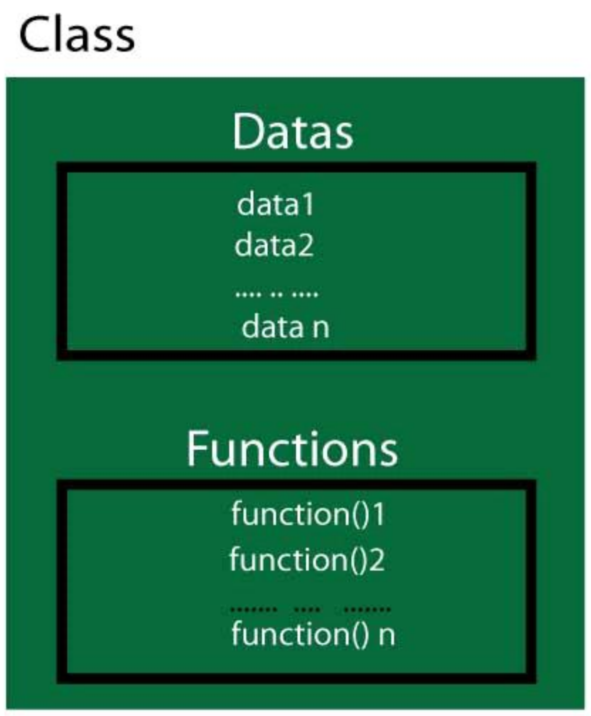
Concepts of OOP:

- ✓ Objects
- ✓ Classes
- ✓ Data Abstraction and Encapsulation
- ✓ Inheritance
- ✓ Polymorphism
- ✓ Overloading
- ✓ Reusability

Objects and Classes

Class is a collection of data member and member function. **Class** is a user **define** data type. **Object** is a **class** type variable. **Objects** are also called instance of the **class**. Each **object** contains all members (variables and functions) declared in the **class**.

A **class** is the collection of related data and function under a single name. A C++ program can have any number of classes. When related data and functions are kept under a class, it helps to visualize the complex problem efficiently and effectively.



A Class is a blueprint for objects
When a class is defined, no memory is allocated. You can imagine like a datatype.
int var;

The above code specifies *var* is a variable of type integer; int is used for specifying variable *var* is of integer type. Similarly, class are also just the specification for objects and object bears the property of that class.

Defining the Class
Class is defined in C++ programming using keyword class followed by identifier(name of class). Body of class is defined inside curly brackets an terminated by semicolon at the end in similar way as structure.

```
class class_name
  {
  // some data
  // some functions
  };
```

Example of Class in C++

```
class temp
   {
      private:
         int data1;
         float data2;
      public:
         void func1()
           {   data1=2;  }
        float func2(){
              data2=3.5;
              retrun data;
           }
   };
```

Explanation
As mentioned, definition of class starts with keyword class followed by name of class(*temp*) in this case. The body of that class is inside the curly brackets and terminated by semicolon at the end. There are two keywords: private and public mentioned inside the body of class.

Keywords: private and public
Keyword private makes data and functions private and keyword public makes data and functions public. Private data and functions are accessible inside that class only whereas, public data and functions are accessible both inside and outside the class. This feature in OOP is known as data hiding. If programmer mistakenly tries to access private data outside the class, compiler shows error which prevents the misuse of data. Generally, data are private and functions are public.

Objects
When class is defined, only specification for the object is defined. Object has same

relationship to class as variable has with the data type. Objects can be defined in similary way as structure is defined.

Syntax to Define Object in C++

class_name variable name;

For the above defined class *temp*, objects for that class can be defined as:

temp obj1,obj2;

Here, two objects(*obj1* and *obj2*) of *temp* class are defined.

Data member and Member functions

The data within the class is known as data member. The function defined within the class is known as member function. These two technical terms are frequently used in explaining OOP. In the above class *temp*, *data1* and *data2* are data members and *func1()* and *func2()* are member functions.

Accessing Data Members and Member functions

Data members and member functions can be accessed in similar way the member of structure is accessed using member operator(.). For the class and object defined above, *func1()* for object *obj2* can be called using code:
obj2.func1();

Similary, the data member can be accessed as:

object_name.data_memeber;

Note: You cannot access the data member of the above class *temp* because both data members are private so it cannot be accessed outside that class.

Example to Explain Working of Object and Class in C++ Programming

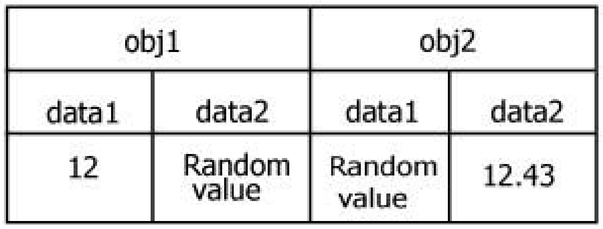
```
/* Program to illustrate working of Objects and Class in C++
Programming */
#include <iostream>
using namespace std;
class temp
{
    private:
        int data1;
        float data2;
    public:
       void int_data(int d){
          data1=d;
          cout<<"Number: "<<data1;
         }
       float float_data(){
           cout<<"\nEnter data: ";
           cin>>data2;
           return data2;
         }
};
 int main(){
      temp obj1, obj2;
      obj1.int_data(12);
      cout<<"You entered "<<obj2.float_data();
      return 0;
 }
```

Output:

```
Number: 12
Enter data: 12.43
You entered: 12.43
```

Explanation of Program
In this program, two data members *data1* and *data2* and two member function int_data() and float_data() are defined under *temp* class. Two objects *obj1* and *obj2* of that class are declared. Function int_data() for the obj1 is executed using code obj1.int_data(12);, which sets 12 to the *data1* of object *obj1*. Then, function float_data() for the object *obj2* is executed which takes data from user; stores it in *data2* of *obj2* and returns it to the calling function.

Note: In this program, *data2* for object *obj1* and *data1* for object *obj2* is not used and contains garbage value.



obj1		obj2	
data1	data2	data1	data2
12	Random value	Random value	12.43

Defining Member Function Outside the Class
A large program may contain many member functions. For the clarity of the code, member functions can be defined outside the class. To do so, member function should be declared inside the class(function prototype should be inside the class). Then, the function definition can be defined using scope resolution operator **::**. Learn more about defining member function outside the class.

Abstraction and Encapsulation

Abstraction allows us to represent complex real world in simplest manner. It is process of identifying the relevant qualities and behaviors an object should possess, in other word represent the necessary feature without representing the back ground details.

Encapsulation It is a process of hiding all the internal details of an object from the outside real world. The word Encapsulation, like Enclosing into the capsule. It restrict client from seeing its internal view where behavior of the abstraction is implemented

Inheritance and polymorphism

Inheritance is when an object or class is based on another object or class, using the same implementation (inheriting from a class) specifying implementation to maintain the same behavior (realizing an interface; inheriting behavior).

Polymorphism is a *concept* wherein a name may denote instances of many different classes as long as they are related by some common superclass.

Comparison between structured and OOP

Keywords and identifiers

Keywords:

Keywords are the reserved words used in programming. Each keywords has fixed meaning and that cannot be changed by user. For example:

int *money*;

Here, int is a keyword that indicates, *'money'* is of type integer.

As, C++ programming is case sensitive, all keywords must be written in lowercase. Here is the list of all keywords predefined by C++.

Keywords in C++ Language			
auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
continue	for	signed	void
do	if	static	while
default	goto	sizeof	volatile
const	float	short	unsigned

Identifiers

In C++ programming, identifiers are names given to C++ entities, such as variables, functions, structures etc. Identifier are created to give unique name to C++ entities to identify it during the execution of program. For example:

```
int money;
int mango_tree;
```

Here, *money* is a identifier which denotes a variable of type integer. Similarly, *mango_tree* is another identifier, which denotes another variable of type integer.

Rules for writing identifier

1) An identifier can be composed of letters (both uppercase and lowercase letters), digits and underscore '_' only.

2) The first letter of identifier should be either a letter or an underscore. But, it is discouraged to start an identifier name with an underscore though it is legal. It is because, identifier that starts with underscore can conflict with system names. In such cases, compiler will complain about it. Some system names that start with underscore are *_fileno*, *_iob*, *_wfopen* etc.

3) There is no rule for the length of an identifier. However, the first 31 characters of an identifier are discriminated by the compiler. So, the first 31 letters of two identifiers in a program should be different.

Literals and constants

A literal is some data that's presented directly in the code, rather than indirectly through a variable or function call.

Here are some examples, one per line:

```
42
128
3.1415
'a'
"hello world"
```

A value written exactly as it's meant to be interpreted. In contrast, a variable is a name that can represent different values during the execution of the **program**. And a constant is a name that represents the same value throughout a **program**. But a **literal** is not a name -- it is the value itself

A constant is an identifier with an associated value which cannot be altered by the program during normal execution – the value is **constant**. This is contrasted with a variable, which is an identifier with a value that can be changed during normal execution – the value is variable.

Comments and Punctuators

Comments are portions of the code ignored by the compiler which allow the user to make simple notes in the relevant areas of the source code. Comments come either in block form or as single lines.

- **Single-line comments** (informally, *C++ style*), start with // and continue until the end of the line. If the last character in a comment line is a \ the comment will continue in the next line.

- **Multi-line comments** (informally, *C style*), start with /* and end with */.

Punctuators: Punctuators are used to group or separate the part of code. These helps to demarcate the program structure
In the above code, {, } and ; are the punctuators.

- Braces are used to group the multiple statements into a separate block.
- Semicolon is used to terminate the statement.

Operators: It combines the expressions or transforms them.
Ex: a + b.
'+' is called as an operator which combines a and b and performs addition.
In the above example code, ., (), + and = are the operators.
There are several kinds of operators are there in **C++** which does different operations based on the operands(literals).

Reasons for embracing OOP

- Code Reuse and Recycling: Objects created for Object Oriented Programs can easily be reused in other programs.
- Encapsulation (part 1): Once an Object is created, knowledge of its implementation is not necessary for its use.

CHAPTER 3: LANGUAGE STRUCTURES OF OOP

Basic Structure of C++ Program

As C++ is a programming language so it follows a predefined structure. The program is divided into many sections, it is important to know the need of every section. The easiest way to understand the basic structure of c++ program is by writing a program. The basic C++ program is as follows:

```
//simple c++ program

#include<iostream> // header file included

 using namespace std;

 int main()

{

int a=10,b=34;

cout<<"simple c++ program \n";  // c++ statement

cout<<"hello world";

cout<<a<<b;

return 0;   // returning no errors

 }
```

The basic structure of c++ program mentioned above can be divided into following sections:

- **Documentation Section :** This section comprises of comments. As the name suggests, this section is used to improve the readability and understanding of the program.// (Double Slash) represents comments in C++ program. Comments can be of single line or multiple lines. Double Slash comments are used to represent single line comments. For multiple line comment, you can begin with /* and end with */. For example :

/* Text line number 1

Text line number 2

Text line number 3 */

In the above C++ program **//simple c++ program** represents single line comment.

- **Linking and Directives Section :** The program written above begins with #include<iostream>. <iostream> represents header file which includes the functionalities of predefined functions. In linking section, the compiler in-built functions such as cout<<, cin>> etc are linked with INCLUDE subdirectory's header file <iostream>. The '#' symbols tells about "address to" or "link to". Iostream is input/output stream which includes declarations of standard input-output library in c++.
- **main() Section :** This is the section in which the program coding is written. Basically, it acts as a container for c++ program. The execution of the c++ program begins with main() function and it is independent of the location of main() function in the program. main() is a function as represented by parenthesis "()". This is because it is a function declaration. The body of the main() function can be found right after these parenthesis, the body is enclosed in braces "{ }".
- **Body of main() Section :** The body of the main() function begins with "{".
  - <u>Local Variable Declaration :</u> In this the variables which are used in the body o f the main() functions are declared. These are called the local variables as their scope is limited within the main() function only, unless they are declared globally outside the main() function. " **int a=10, b=34;**" in the above program represents local variables
  - <u>Statements to Execute :</u> This section includes statements for reading, writing and executing data using I/O functions, library functions, formulas, conditional statements etc. Above written program has many executable statements like **cout<<"simple c++ program \n";**
  - **return 0;** in the above program causes the function to finish and **0** represents that function has been executed with zero errors. This is considered as most usual way to end a C++ program.
  - Finally the body of the main() function ends with "}".
- **Global Declaration Section :** There are certain programs which requires variables that can be used in more than one function, so then the variables can be declared outside the main() function or respective functions. Then those variables become accessible in any of the functions, Hence named as Global Variables as their scope becomes global to the program.
- **User Defined Functions :** There are certain functions that are called by calling statements from the main() function. Every function includes local variable declaration section and executable statement section similar to main program.

One more example which explains the basic structure of c++ program is as follows :

```
/* basic example
   which
   explains
   the structure of c++ program */
 #include <iostream> // header file include
using namespace std;
 float f=10.2,j=4.5;
 int main()
{
 int a=10,b=34;
cout<<"simple c++ program \n";  // c++ statement
cout<<"hello world";
cout<<a<<b;
 return 0;   // returning no errors
}
```

- Multiple line comments :

  /* basic example

  which

  explains

  the structure of c++ program */ .

- **float f=10.2, j=4.5;** are global variables which are declared outside the main() function.

The statements written in the above mentioned programs can be written in a single line for example :

```
int main(){int a=10,b=34; cout<<"simple c++ program \n";
cout<<"hello world"; cout<<a<<b; return 0;}
```

- The separation between statements is specified with a semicolon (;). The statements are written in separate lines just to improve the readability of the program.

Features of the Object Oriented programming

- Emphasis on data rather than procedure
- Programs are divided into entities known as objects
- Data Structures are designed such that they characterize objects
- Functions that operate on data of an object are tied together in data structures
- Data is hidden and cannot be accessed by external functions
- Objects communicate with each other through functions
- New data and functions can be easily added whenever necessary
- Follows bottom up design in program design

Header and Source Files and extensions

In this lesson we will talk about some relatively new concepts that I've postponed over and over, until this point, where we can actually use them for a better understanding. We will talk about separating your code into separate **header** and **source** files.

You will finally see what **headers** look like and how we can use them to separate different parts of our code into separate **source files** and **header files**. In this way we can keep our code more organized, by separating different concepts, which will make it much easier to find what we are looking for when trying to modify our program.

There are also much more upsides of doing this, and we will go through each one of them, but let's first take a look at what **headers** really are and look like.

Header files
You've been using the help of a **header** file even from the first program that we wrote in our lessons. If you remember, I've explained, in a very general way, what **headers** are, when we've first encountered the **#include <iostream>** in our programs.

Headers usually have the **.h** extension and contain declarations that we will use in our **source files**. Let's take as an example the **iostream** header file that we include when working with **input-output** in our program. We could have never used **cout** to print to the screen without including the **iostream** header file because we have never declared and defined that identifier anywhere in our program. That is why we are telling the compiler to **include** the **iostream** header file, which actually means that, the compiler will locate and read all the declarations from that header file when it reaches the preprocessor directive, **#include**.

Usually **header files** only contain declarations and do not provide the actual definitions. When we only declare a **function** and do not provide the definition for it, when we are calling that **function** in our program, the **linker** will complain about "**Unresolved Symbols**". So, how does the compiler know where to get the definition for **cout** then?

Well, the **cout** is actually defined in the **standard runtime library** which is automatically **linked** in the **link** process.

What are libraries

Well, simply put, libraries are packages, containers of useful code(**functions**, **objects**) with the purpose of being reused in programs. Now, when you are writing a **library**, you are also writing a **header file** which contains the declarations of the reusable code that exists in that library and you wish to provide to others to include in their own programs.

When you, or others, wish to use the functionality provided by any **library**, you actually do not need the entire source code to be included in your projects. You only need the compiled **library**, (.a, .so, .lib, .dll etc. – depending on the platform you are using) and a **header** file that you will include in your sources when using any of the functionality it provides. Think of **header files** just like you think of a table of contents. It is just a very simple container of declarations, so the compiler will know the minimum it needs about the functionality you are using, when it compiles your code.

When using **libraries**, you do not have to compile the code again, which would be a waste of time, because libraries are always provided as they are, without the need of modifying them. When using a **library** that is not from the **standard runtime**, you will also need to let the compiler know which libraries you wish to include, so it knows where to get the symbols from and link them, once they are used in your program.

For now, we will not use separate **libraries**, but we will get to learn how all of these new concepts work by using separate **source files** and **header files** in our project.

Creating your first separate header and source files

For the purpose of this lesson, we shall create separate header and source files that will contain a couple of basic **functions** that we will use in our **main.cpp**. Let's call that source file **mathPrimer.cpp** and the header file, **mathPrimer.h**. Let's create the file **mathPrimer.h** first, which will contain the following declarations:

mathPrimer.h

```
//  mathPrimer.h
//  ChapterII.HeaderSourceFiles
//
//  Created by Vlad Isan on 20/04/2013.
//  Copyright (c) 2013 INNERBYTE SOLUTIONS LTD. All rights
reserved.
//

#ifndef mathPrimer_h
#define mathPrimer_h

int add(int a, int b);
int subtract(int a, int b);

#endif
```

First, let’s take a look a little at the preprocessor directives in this file. You already know what **#define** is, as we have covered it a little when talking about variables and constants. Let’s talk a little about the **conditional compilation**, **#ifdef**, **#ifndef** and **#endif**.
The **conditional compilation** preprocessor directives, tell your compiler what should be compiled or not and under what conditions.
The **#ifdef** preprocessor directive basically checks if something was previously defined using the **#define** preprocessor directive. If this condition is met, then the code between the **#ifdef** and the corresponding **#endif** is compiled, otherwise it is ignored by the compiler.
The same goes for **#ifndef**, and, as you can already imagine, it is the complete opposite of **#ifdef**, and it allows the compiler to check whether a name has not been defined with **#define**.
So, why are we using these preprocessor directives in our **header file**? These preprocessor directives in the **header files** are called **header guards**, and helps us to avoid including the same declaration in multiple times.
This works by skipping the entire contents of the **header file** if it was already included in some other place. In our example, when you first include our header file, **mathPrimer.h**, the **#ifndef** condition is met, because we haven’t defined the name **mathPrime_h** by using the **#define** preprocessor directive until now. So, the condition is met and everything inside the **header file** will be compiled, and most importantly, the **mathPrimer_h** will be defined as well, by using the **#define mathPrimer_h** right after the **#ifndef**. Now, when you include the **header file** a second time, the **#ifndef mathPrimer_h** condition will not be met, because we have already defined that name when we included the **header file** in some other place.
By using these **header guards** we are avoiding the complaint we could get from the compiler when declaring the **header** contents twice, or even multiple times.
OK, now let’s look at what is declared inside our **mathPrimer.h** header file. We have two function declarations, **add** and **subtract**, both having two integer parameters and an integer return type.
We have to also define our **functions**. Let’s create another file called **mathPrimer.cpp**. This file will contain the **function** definitions:

mathPrimer.cpp

```
//
//  mathPrimer.cpp
//  ChapterII.HeaderSourceFiles
//
//  Created by Vlad Isan on 20/04/2013.
//  Copyright (c) 2013 INNERBYTE SOLUTIONS LTD. All rights
reserved.
```

```
#include "mathPrimer.h"

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

int subtract(int a, int b) {
    return (a - b);
}
```

The first thing we do in this **source file** is to include our **header file**, by using the preprocessor directive **#include "mathPrimer.h"**. Now, when including **header files** from the standard library, you've probably noticed we used angled brackets. The reason is that when including **header files** that come with the compiler, such as standard library **header files**, we use angled brackets, but when including **header files** that we are supplying, we use double quotes, **" "**, which tell the compiler to look for the **header file**, by first searching for it in the current directory where the **source files** are contained in.

Please note that when we only have declarations in the **header files**, we do not need to include the **header file** when defining the **functions**, as we do in our case. We have to only include it when we are calling the **functions** declared in it. But, as we already are using **header guards**, it is a good practice to include it in here as well, because **header files** can also contain **constants** which could be used in here as well. As an example, we can declare a **constant** in our **header file** to hold the value of **PI**, and use that **constant** when defining the other **functions**, in our **mathPrimer.cpp**.

As you can notice, we are also defining the functions **add** and **subtract** which we are going to use in our **main** function.

Now, let's take a look at our **main.cpp:**

```
//  main.cpp
//  ChapterII.HeaderSourceFiles
//
//  Created by Vlad Isan on 20/04/2013.
//  Copyright (c) 2013 INNERBYTE SOLUTIONS LTD. All rights
reserved.
//

#include <iostream>
#include "mathPrimer.h" /* including our header file */

using namespace std;

int main()
{

    /* calling our functions, add and subtract */
```

```
        cout << add(10, 2) << endl;
        cout << subtract(10, 2) << endl;

        return 0;
    }
```

As you can see we are including our **header file** in here as well, so we can use the **functions** declared in it, **add** and **subtract**. If you were to remove the **#include "mathPrimer.h"** from here, you will see that you cannot compile the code, because the compiler would not know anything about those functions, and it will complain about "**Undeclared identifiers**".

An important thing to note about **header files** is to never include **variables** in them, unless they are constants. **Header files** should only be used for declarations. Also, you should never include the definition for a **function** in the **header file** because it will change the whole scope of having a **header file**, and it will make it hard to read.

Also, you should always split the parts of your code into separate **header** and **source files** grouped by a certain criteria or **functionality**, because when only needing a part of it, you do not need to include all of the declarations that reside in your program. As in our example, we have called our **header file mathPrime.h**, as it will only contain basic math **functions**. If we want to make some other helper **functions**, for example, for printing to the screen and retrieving user input, we should make another pair of **header/source files**, that we should only include when needed.

Data Types in OOP

Data type

In C++ programming, we store the variables in our computer's memory, but the computer has to know what kind of data we want to store in them. The amount of memory required to store a single number is not the same as required by a single letter or a large number. Further, interpretation of different data is different inside computer's memory.

The memory in computer system is organized in bits and bytes. A byte is the minimum amount of memory that we can manage in C++. A byte can store a relatively small amount of data: one single character or a small integer. In addition, the computer can manipulate more complex data types that come from grouping several bytes, such as long numbers or non-integer numbers.
Data types in C++ is used to define the type of data that identifiers accepts in programming and operators are used to perform a special task such as addition, multiplication, subtraction and division etc of two or more operands during programming. C++ supports a large number of data types. Data types can be categorized into three types which are described below;

Built-in/Simple Data Types

There are four types of built-in data types and let us discuss each of these and the range of values accepted by them one by one.

- **Integer data type (int):**
  An integer is an integral whole number without a decimal point. These numbers are used for counting. For example 46, 167, -223 are valid integers. Normally an integer can hold numbers from -32768 to 32767. The int data type can further categorized into short, long and unsigned int.
  The short int data type is used to store integer with a range of -32768 to 32767, However, if the need be a long integer (long int) can also be used to hold integers from -2,147,483,648 to 2,147,483,648. The unsigned int can have only positive integers and its range lies up to 65536.

- **Floating point data type (float):**
  A floating point number has a decimal point. Even if it has an integral value, it must include a decimal point at the end. These numbers are used for measuring quantities. Examples of valid floating point numbers are: 35.5, -66.3, and 49.07.
  A float type data can be used to hold numbers from 3.4*10-38 to 3.4*10+38 with six or seven digits of precision. However, for more precision a double precision type (double) can be used to hold numbers from 1.7*10-308 to 1.7*10+308 with about 15 digits of precision.

- **Void data type:**
  It is used for following purposes;

  - It specifies the return type of a function when the function is not returning any value.
  - It indicates an empty parameter list on a function when no arguments are passed.
  - A void pointer can be assigned a pointer value of any basic data type.

- **Char data type:**
  It is used to store character values in the identifier. Its size and range of values is given in table below;

Name	Description	Size*	Range
Char	Character or small integer	1 byte	signed: -128 to 127 unsigned: 0 to 255
short int (short)	Short Integer.	2 bytes	signed: -32768 to 32767 unsigned: 0 to 65535
int	Integer	4 bytes	signed: -2147483648 to 2147483648 unsigned: 0 to 4294967295
long int (long)	Long Integer	4 bytes	signed: -2147483648 to 2147483648 unsigned: 0 to 4294967295

Bool	Boolean value. It can take one of two values: true or false	1 byte	true or false
float	Floating point number.	4 bytes	+/- 3.4e +/-38 (~7 digits)
double	Double precision floating point number.	8 bytes	+/- 1.7e +/-308 (~15 digits)
long double	Long double precision floating point number	8 bytes	+/- 1.7e +/-308 (~15 digits)
wchar_t	Wide character	2 or 4 bytes	1 wide character

Derive Data Types

C++ also permits four types of derived data types. As the name suggests, derived data types are basically derived from the built-in data types. There are four derived data types. These are:

- Array
- Function
- Pointer and
- Reference

Array An array is a set of elements of the same data type that are referred to by the same name. All the elements in an array are stored at contiguous (one after another) memory locations and each element is accessed by a unique index or subscript value. The subscript value indicates the position of an element in an array.

Function A function is a self-contained program segment that carries out a specific well-defined task. In C++, every program contains one or more functions which can be invoked from other parts of a program, if required.

Reference A reference is an alternative name for a variable. That is, a reference is an alias for a variable in a program. A variable and its reference can be used interchangeably in a program as both refer to the same memory location. Hence, changes made to any of them (say, a variable) are reflected in the other (on a reference).

Pointer A pointer is a variable that can store the memory address of another variable. Pointers allow to use the memory dynamically. That is, with the help of pointers, memory can be allocated or de-allocated to the variables at run-time, thus, making a program more efficient

User Defined Data Types

C++ also permits four types of user defined data types. As the name suggests, user defined data types are defined are defined by the programmers during the coding of software development. There are four user defined data types. These are:

- Structure
- Union
- Class, and

- Enumerator

A structure allows for the storage, in contiguous areas of memory, of associated data items. A structure is a template for a new data type whose format is defined by the programmer. A structure is one of a group of language constructs that allow the programmer to compose his/her own data types.

```
struct [tag]
{
     [variable];
     members;
}
Example
struct Person
{
     char ssn[12]
          ,last_name[20]
          ,first_name[16]
          ,street[20]
          ,city[20]
          ,state[3]
          ,zip_code[11]
          ;
     int age;
     float height
          ,weight
          ;
     double salary;
};
```

Where **tag** is optional and only needs to be present if no **variable** is present. The **members** are variables declared as any C supported data type or composed data type. A structure is a set of values that can be referenced collectively through a variable name. The components of a structure are referred to as members of a structure. A structure differs from an array in that members can be of different data types. A structure is defined by creating a template. A structure template does not occupy any memory space and does not have an address, it is simply a description of a new data type. The name of the structure is the **tag**. The **tag** is optional when a **variable** is present and the **variable** is optional when the **tag** is present. Each member-declaration has the form

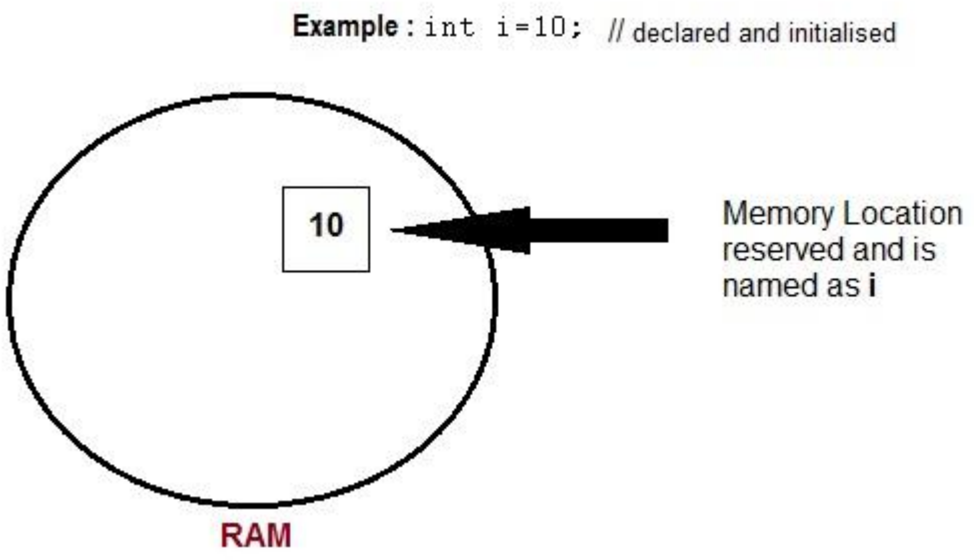
A Class specified by the keyword *class*, is a user defined type that contains both data members and member functions...

A union is a user-defined data or class type that, at any given time, contains only one object from its list of members (although that object can be an array or a class type).

An enumeration, specified by the keyword *enum*, is a set of integer constants associated by identifiers--called enumerators. Enumerations provide a manner to implement names (or identifiers), in place of **integer constants**. Enumerator values begin at zero (0), if a value for the initial enumerator was not provided. Enumerators may be used wherever an int value is utilized. If no user specified value is assigned, compilers will assign the following integer value after the integer value assigned to the preceding enumerator.

Variable and variable declaration

Variables are named memory storage reserved for our programs to use (store, manipulate).
Variable are used in C++, where we need storage for any value, which will change in program. Variable can be declared in multiple ways each with different memory requirements and functioning. Variable is the name of memory location allocated by the compiler depending upon the data type of the variable.


Example : int i=10; // declared and initialised
10
Memory Location reserved and is named as i
RAM

Declaration and Initialization

Variable must be declared before they are used. Usually it is preferred to declare them at the starting of the program, but in C++ they can be declared in the middle of program too, but must be done before using them.

Example :

```
int i;      // declared but not initialized
char c;
int i, j, k;  // Multiple declaration
```

Initialization means assigning value to an already declared variable,

```
int i;   // declaration
i = 10;  // initialization
```

Initialization and declaration can be done in one single step also,

```
int i=10;         //initialization and declaration in same step
int i=10, j=11;
```

If a variable is declared and not initialized by default it will hold a garbage value. Also, if a variable is once declared and if try to declare it again, we will get a compile time error.

```
int i,j;
i=10;
j=20;
int j=i+j;   //compile time error, cannot redeclare a variable in same
scope
```

Program Example

Let's look at an example of how to declare an integer variable in the C++ language and use it.

a) **Below is an example C++ program where we declare an integer variable and assign value in it:**

```
#include <iostream>

int main()
{
   int age;

   age = 10;
   cout<<"The data in variable age is %d yrs.\n"<< age;

   return 0;
}
```

This C program would print “The `data in variable age is` 10 yrs."

b) **Below is an example C++ program where we declare an integer variable and assign value in it throng an input statement “Cin”:**

```
#include <iostream>

int main()
{
   String name;
   int age;
   cout<<"Enter your name plz:\n";
   cin>>name;
   cout<<"Enter your age plz:\n";
   cin>>age;

   cout<<"Your Name is”<<name<<”and your age is”<<age<<”yrs"<< endl;

   return 0;
}
```

This C program would print `"Your Name is KIM and your age is 10 yrs"`.

Scope of Variables

All the variables have their area of functioning, and out of that boundary they don't hold their value, this boundary is called scope of the variable. For most of the cases its between the curly braces,in which variable is declared that a variable exists, not outside it. We will study the storage classes later, but as of now, we can broadly divide variables into two main types,

- Global Variables
- Local variables

Global variables

Global variables are those, which ar once declared and can be used throughout the lifetime of the program by any class or any function. They must be declared outside the `main()` function. If only declared, they can be assigned different values at different time in program lifetime. But even if they are declared and initialized at the same time outside the main() function, then also they can be assigned any value at any point in the program.

Example : Only declared, not initialized

```
include <iostream>
using namespace std;
int x;                // Global variable declared
int main()
{
 x=10;                 // Initialized once
 cout <<"first value of x = "<< x;
 x=20;                 // Initialized again
 cout <<"Initialized again with value = "<< x;
}
```

Local Variables

Local variables are the variables which exist only between the curly braces, in which its declared. Outside that they are unavailable and leads to compile time error.

Example :

```
include <iostream>
using namespace std;
int main()
{
 int i=10;
 if(i<20)        // if condition scope starts
  {
    int n=100;   // Local variable declared and initialized
  }              // if condition scope ends
 cout << n;      // Compile time error, n not available here
}
```

Some special types of variable

There are also some special keywords, to impart unique characteristics to the variables in the program. Following two are mostly used, we will discuss them in details later.

1. **Final** - Once initialized, its value cant be changed.
2. **Static** - These variables holds their value between function calls.

Example :

```
include <iostream>
using namespace std;
int main()
{
 final int i=10;
 static int y=20;
{
```

Type Conversion and Type Casting

In computer science, **type conversion** or **typecasting** refers to changing an entity of one datatype into another. There are two types of conversion: implicit and explicit. The term for implicit type conversion is **coercion**. Explicit type conversion in some specific way is known as **casting**. Explicit type conversion can also be achieved with separately defined conversion routines such as an overloaded object constructor.

Both Type conversion and Type casting in C++ are used to convert one predefined type to another type.
Type Conversion is the process of converting one predefined type into another type.
and type Casting is the converting one predefined type into another type forcefully.

Need of Type Conversion and Type Casting in C++

An Expression is composed of one or more operations and operands. Operands consists of constants and variables. Constants and expressions of different types are mixed together in an expression. so they are converted to same type or says that a conversion is necessary to convert different types into same type.

Types of Type Conversions

C++ facilitates type conversion into 2 forms :

- Implicit Type Conversion
- Explicit Type Conversion

Implicit Type Conversions :

Implicit Type Conversion is the conversion performed by the compiler without programmer's intervention.
It is applied, whenever, different data types are intermixed in an expression, so as not to loose information.

The C++ compiler converts all operands upto the type of the largest operand, which is called **type promotion.**

Usual Arithmetic Conversions are summarized in the following table –

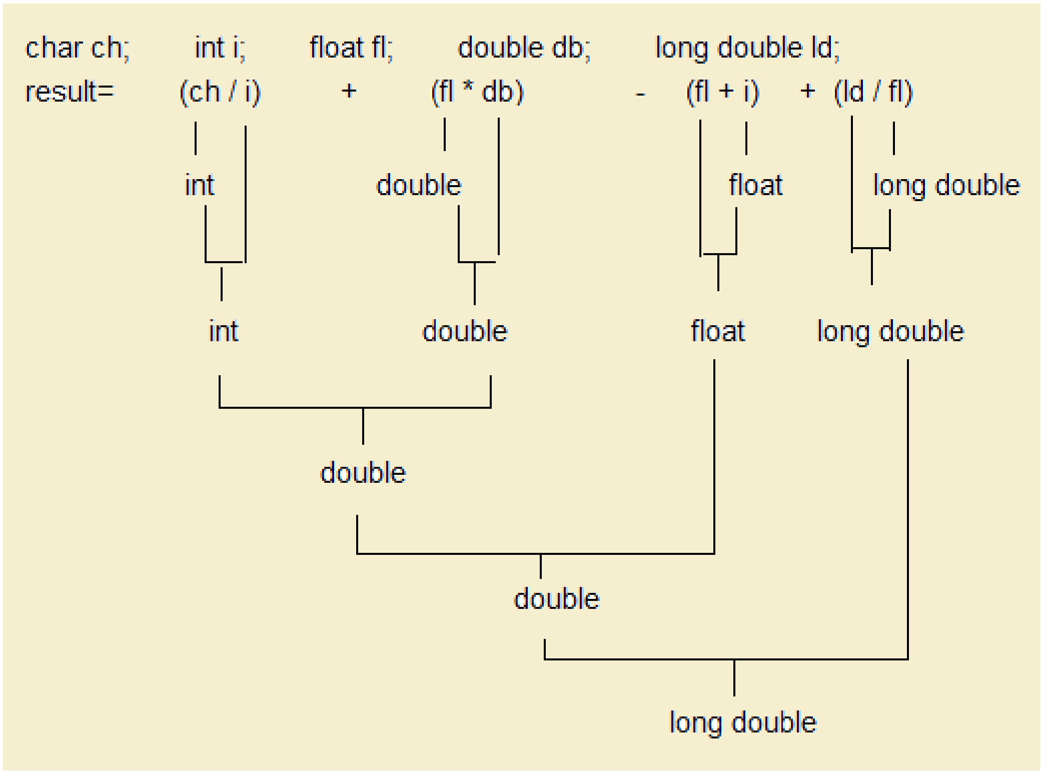
StepNo.	If either'stype of	Then resultant type of other operand	Otherwise

1	long double	long double	Step 2
2	double	double	Step 3
3	float	float	Step 4
4	—	integral promotion takes place followed by step 5	—
5	unsigned long	unsigned long	Step 6
6	long int ant the other is unsigned int	(i) long int (provided long int can represent all values of unsigned int)	Step 7
		(ii) unsigned long int(if all values of unsigned int can't be represented by long int)	Step 7
7	long	long	Step 8
8	unsigned	unsigned	Both operandsare int

The step 1 and 2 in the above table will be read as –
Step 1: If either operand is of type **long double**, the other is converted to **long double.**
Step2 : Otherwise, if either is of type **double**, the other is converted to **double.**

After applying above arithmetic conversions, each pair f operands is of same type and the result of each operation is the same as the type of both operands.

Example of Implicit Type Conversion :


char ch; int i; float fl; double db; long double ld;
result= (ch / i) + (fl * db) - (fl + i) + (ld / fl)
int
double
float
long double
int
double
float
long double
double
double
long double

Explicit Type Conversion :

Explicit Type conversion is also called type casting. It is the conversion of one operand to a specific type. An explicit conversion is a user defined that forces an expression to be of specific type.

Syntax : (type) expression

Example : float(a+b/5) ; This expression evaluates to type float.

Problem in Explicit Type Conversion :

Assigning a value of smaller data type to a larger data type, may not pose any problem. But, assigning a value of larger data type to smaller type, may poses problems. The problem is that assigning to a smaller data type may loose information, or result in losing some precision.

Conversion Problems –

S.no	**Conversion**	**Potential Problems**
1	Double to float	Loss of precision(significant figures)
2	Float to int	Loss of fractional part
3	Long to int/short	Loss of Information as original valuemay be out of range for target type

Type Compatibility

In an assignment statement, the types of right types and left side of an assignment should be compatible, so that conversion can take place. For example,

ch=x; (where ch is of char data type and x is of integer data type)

How and Why Information is loose ?

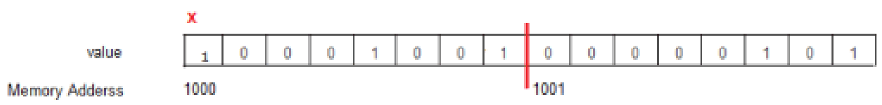
what is Big Endian ?? ⇒ refer to the link Click here

since the memory representation in ***Big-Endian,*** Let

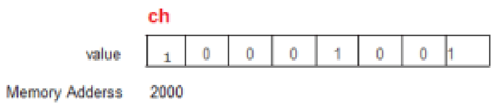
int x=1417;
ch=x;

now, x will be 00000101 10001001 in binary.

x will store in memory like this, so,



when assignment statement ch=x; will be execute, then ch will have the lower order bits ..i.e 10001001



So, the information i.e. the value stores at address 1001, will be lost after assigning an integer variable value to a character variable

Example of C++ Program for Type Casting –

```
#include <stdio.h>
int main ()
{
    float x;
    x = (float) 7/5;
    cout<<"x="<<x;
}
```

CHAPTER4 : ESSENCE OF OBJECTS AND CLASSES

Objects and classes in OOP

A class is a template definition of the method s and variable s in a particular kind of object. Thus, **an object** is a specific instance of a class; it contains real values instead of variables.

Importance of objects and classes in OOP

Classes and objects provide a way to modularize programming code and encapsulating functionality.
One of my favorite dimensions to OOP is the habit of "Black Boxing" programming functionality. By making methods and properties private, programmers are able to bundle programming code up into a module that other programmers don't have to know anything about. They simply have to know what services the object offers and how to access them.

Another important dimension of OOP is "loose coupling." Well-written classes make it so that it is possible to easily replace portions of programming code without impacting the rest of the system. In a well-written OOP architecture, you can switch databases without having to touch the code in your control or view layers.

Implementation of objects and classes

✓ **Initialization**

Initialization is the assignment of an initial value for a data **object** or **variable**. The manner in which **initialization** is performed depends on programming language, as well as type, storage class, etc., of an **object** to be **initialized**.

The variable's *name* is what you ***declare*** it to be. The *value* is what you ***assign*** to it.

Variables are initialized

All variables are always given an *initial* value at the point the variable is *declared*. Thus all variables are ***initialized***.

For *value* types, like `int` the compiler will give them a valid value if you do not do so explicitly. `int`'s ***initialize*** to zero by default, `DateTime`'s ***initialize*** to `DateTime.MinValue` by default.

Reference type variables ***initialize*** to the object you give it. The compiler will not ***assign*** an object (i.e. a valid value) if you don't. In this case the value is `null` - nothing. So we say that the reference is ***initialized*** to null.

Objects are Instantiated

Humans are born. Objects are instantiated. A baby is an *instance* of a Human, an object is an *instance* of some Class.

The act of creating an *instance* of a Class is called ***instantiation***

- ✓ **Free store**

  Static Storage vs Heap vs Stack

  1. Static vs Dynamic

  Static: Storage can be made by compiler looking only at the text of the program. One reason for statically allocating as many data objects as possible is that the addresses of these objects can be compiled into target code.

  Dynamic: Storage can be made by looking at what the program does when the program is running.

  2. Static

  Global constants and other data generated by the compiler (e.g. info to support garbage collection) are allocated static storage. Static variables are bound to memory cells before execution begins and remains bound to the same memory cell throughout execution. E.g., C static variables.

  Advantage: efficiency (direct addressing), history-sensitive subprogram support

  Disadvantage: lack of flexibility, no recursion if this is the *only* kind of variable, as was the case in FORTRAN

  3. Heap/free-store

  Data that may outlive the call to the procedure that created it is usually allocated on a heap. E.g. **new** to create objects that may be passed from procedure to procedure. The size of heap cannot be determined at compile time. Referenced only through pointers or references, e.g., dynamic objects in C++, all objects in Java

  Advantage: provides for dynamic storage management

  Disadvantage: inefficient and unreliable

  4. Stack

  Names local to a procedure are allocated space on a stack. The size of stack can not be determined at compile time.

  Advantages: allows recursion conserves storage

  Disadvantages: Overhead of allocation and deallocation Subprograms cannot be history sensitive
  Inefficient references (indirect addressing)

- ✓ **Static objects**

  Static Objects: Global objects, static data members, file scope objects and local variables declared *static* are variables with static storage duration. A strategy for initialization of objects with static storage duration is needed to avoid the risk of accessing uninitialized objects.

Objects with static storage duration should only be declared within the scope of a class, function or anonymous namespace.

Static objects make it possible to access an object inside a function without having to pass along a pointer or reference to it. Many objects can use the same object without each object storing a pointer to the object, which can save space and sometimes make the code less complex.

Function local static object

```
int randomValue(int seed)
{
   static int oldValue = seed;
   // calculate new value
   return oldValue;
}
```

✓ **Implicit pointer**

In object-oriented programming, type conversion allows programs to treat objects of one type as one of their ancestor types to simplify interacting with them.

In most languages, the word ***coercion*** is used to denote an ***implicit* conversion**, either during compilation or during run time. A typical example would be an expression mixing integer and floating point numbers (like 5 + 0.1), where the integers are normally converted into the latter. **Explicit type** conversions can either be performed via built-in routines (or a special syntax) or via separately defined conversion routines such as an overloaded object constructor.

A **pointer** is a programming language object, whose value refers directly to (or "**points** to") another value stored elsewhere in the computer memory using its address. *implicit* conversion help result of data pointed to by a pointer be converted to respective type of type of the pointer

Example, in C++

```
int *money;
char *bags;
```

money would be an integer pointer and bags would be a char pointer. The following would yield a compiler warning of "assignment from incompatible pointer type"

```
bags = money;
```

Because money and bags were declared with different types. To suppress the compiler warning, it must be made implicit that you do indeed wish to make the assignment by typecasting it

```
bags = (*char) money;   //implicit conversion
bags = (char*) money;    // implicit conversion
```

Typed pointers and casting

In many languages, pointers have the additional restriction that the object they point to has a specific type. For example, a pointer may be declared to point to an integer; the language will then attempt to prevent the programmer from pointing it to objects which are not integers, such as floating-point numbers, eliminating some errors.

✓ **In-line function**

Inline Functions

An inline function is a function whose statements are embedded in the caller as a substitute for the function call. This avoids the overhead a context switch which must be performed whenever a function is called and whenever a function returns to the caller. The only purpose for inlining is to increase the execution speed of a program.

Inline functions should be small and simple. Since a function may be called from many different places in a program, inlining large functions can result in an undesirable increase in the size of a program. Inline functions should not contain any function calls or I/O statements. As a general rule, if a function has more than five statements it should not be inlined.

When you specify that a function is an inline function, it is merely a request of the compiler. The compiler may or may not actually inline the function.

Inlining Global Functions

To inline a global function:

- Put the keyword inline before the return type in the function header.
- Make sure that the function definition (not just the prototype) appears before any calls to the function.

Example:

```
  inline void swap(int &a, int &b)
{
   int c = a;
   a = b;
   b = c;
}

int main()
{
   int x = 5;
   int y = 10;
   swap(x,y);
   cout << x << "  " y << endl;
}
```

Inlining Member Functions

There are two ways to inline a member function: implicit and explicit.
Implicit
In implicit inlining, the inline member function is defined within the class definition. The keyword inline is not used.

Example:

```
  class Date
   {
   public:
      Date(int mm,int dd,int yy) {month = mm; day = dd; year
= yy;}
      void setdate(int mm,int dd,int yy)
                          {month = mm; day = dd; year = yy;}
      void showdate();  // not inlined

      // additional member functions

   private:
      int month;
      int day;
      int year;
   };
```

Explicit
In explicit inlining, the prototype in the class definition is preceded by the keyword inline. The member function defintion is defined outside the class definition, and is also preceded by the keyword inline.

Example:

```
     class Date
   {
   public:
      inline Date(int,int,int);
      inline void setdate(int,int,int);
      void showdate();  // not inlined

      // additional member functions

   private:
      int month;
      int day;
      int year;
   };

   inline Date::Date(int mm,int dd,int yy)
   {
      month = mm;
      day = dd;
      year = yy;
   }

   inline void Date::setdate(int mm,int dd,int yy)
   {
      month = mm;
      day = dd;
      year = yy;
   }

   void showdate()
   {
      cout << month << '/' << day << '/' << year;
   }
```

✓ **Friend of class**

A **friend class** in C++ can access the "private" and "protected" members of the class in which it is declared as a friend

Example

```
class B {
    friend class A; // A is a friend of B

private:
    int i;
};

class A {
public:
    A(B b) {
        b.i = 0; // legal access due to friendship
    }
};
```

Features

- **Friendships are not symmetric** – If class `A` is a friend of class `B`, class `B` is not automatically a friend of class `A`.
- **Friendships are not transitive** – If class `A` is a friend of class `B`, and class `B` is a friend of class `C`, class `A` is not automatically a friend of class `C`.
- **Friendships are not inherited** – A friend of class `Base` is not automatically a friend of class `Derived` and vice versa; equally if `Base` is a friend of another class, `Derived` is not automatically a friend and vice versa.
- **Access due to friendship *is* inherited** – A friend of `Derived` can access the

restricted members of Derived that were inherited from Base. Note though that a friend of Derived only has access to members inherited from Base to which Derived has access itself, e.g. if Derived inherits publicly from Base, Derived only has access to the protected (and public) members inherited from Base, not the private members, so neither does a friend.

✓ **Static class members**

Static Class Members: Class Variables and Functions

The member variables declared in a class declaration are normally *instance variables* – a separate copy of them is created for each class object. Sometimes, however, it is convenient to have *class variables* that are associated with the class itself rather than with any class object. Every instance of a class accesses the same class variables, although each has its own private copy of any instance variables.

In C++, class variables are declared as `static`. Thus in

```
 class c {

     int i;

     int j;

     static int m;

     static int n;

public:

     void zap();

     static void clear();

};


void c::zap() {

     i = 0; j = 0; m = 0; n = 0;

}


void c::clear() {

      m = 0; n = 0;

}
```

`i` and `j` are instance variables and `m` and `n` are class variables. Every object of class `c` will have its own private `i` and `j`, which can have different values for different objects; however, all objects will access the same `m` and `n`, which will, of course, have the same values for all objects.

Static variables are like non-inline member functions in that they are declared in a class declaration and defined in the corresponding source file. To define static variables `m` and `n`, the source file for class `c` must contain the following declarations (which are also definitions):

```
    int c::m;

    int c::n;
```

These definitions can also be used to assign initial values to the static member variables. For example, the following definitions give `m` the initial value 5 and `n` the initial value 6:

```
    int c::m = 5;

    int c::n = 6;
```

The accessibility rules for class variables are the same as for instance variables. Thus private and protected class variables can be accessed by member and friend functions, but not by functions that are not associated with the class. Likewise, a derived class with a public base class inherits access to the protected and public class variables of the base class.

In addition, we can define *static member functions* that can manipulate only static member variables – they cannot access the instance variables of class objects. Like static member variables, static member functions are associated with a class rather than with any class object. A static member function is invoked via the name of the class rather than via the name of a class object.

The preceding example defines an ordinary member function `zap()` and a static member function `clear()`. To invoke the ordinary member function `zap()`, we must declare a class object `cc` and apply `zap()` to it with the dot operator:

```
 c cc;       // declare cc as c object

 cc.zap();   // apply zap() to class object cc
```

From the definition of zap(), we note that it can manipulate the instance variables i and j of cc as well as the class variables m and n.

In contrast, the static member function clear() is invoked using the name of the class and the :: operator:

```
  c::clear();
```

We can call clear() even if no class objects have been created. From the definition of clear(), we note that it manipulates only the class variables m and n; clear() does not have access to the instance variables i and j of any class object.

✓ **Specifies – const, enum, typedef**

A constant is an identifier with an associated value which cannot be altered by the **program** during normal execution – the value is **constant**. This is contrasted with a variable, which is an identifier with a value that can be changed during normal execution – the value is variable.

Examples:

```
const float PI = 3.1415927;   // maximal single float precision
const unsigned int MTU = 1500;  // Ethernet v2, RFC 894
const unsigned int COLUMNS = 80;
```

An enumerated type (also called **enum**) is a data type consisting of a set of named values called elements, members or enumerators of the type.

```
Syntax: enum type_name{ value1, value2,...,valueN };
```

Example:

```
1) enum Color {Red, Green, Blue};

2) enum week{ sunday, monday, tuesday, wednesday, thursday,
   friday, saturday};
```

typedef keyword allows the programmer to create new names for types such as int or it literally stands for "type definition". Typedefs can be used both to provide more clarity to your code and to make it easier to make changes to the underlying data types that you use.

Note:

- Typedef is a keyword that is used to give a new symbolic name for the existing name in a C program. This is same like defining alias for the commands.
- Consider the below structure.

```
struct student
{
         int mark [2];
         char name [10];
         float average;
}
```

- Variable for the above structure can be declared in two ways.

```
1st way  :
```

```
struct student record;        /* for normal variable */
struct student *record;      /* for pointer variable */
```

2nd way :

```
typedef struct student status;
```

- When we use “typedef” keyword before struct <tag_name> like above, after that we can simply use type definition “status” in the C program to declare structure variable.
- Now, structure variable declaration will be, “status record”.
- This is equal to “struct student record”. Type definition for “struct student” is status. i.e. status = “struct student”

<u>An alternative way for structure declaration using typedef in C:</u>

```
typedef struct student
{
        int mark [2];
        char name [10];
        float average;
} status;
```

- To declare structure variable, we can use the below statements.

```
status record1;          /* record 1 is structure variable */
status record2;          /* record 2 is structure variable */
```

✓ **Enumerated constant**

An *enumeration* is a distinct type whose value is restricted to one of several explicitly **named constants** ("***enumerators***"). The values of the constants are values of an integral type known as the *underlying type* of the enumeration.

✓ **Pointer to members**

Pointer-to-Member Operators: .* and ->*

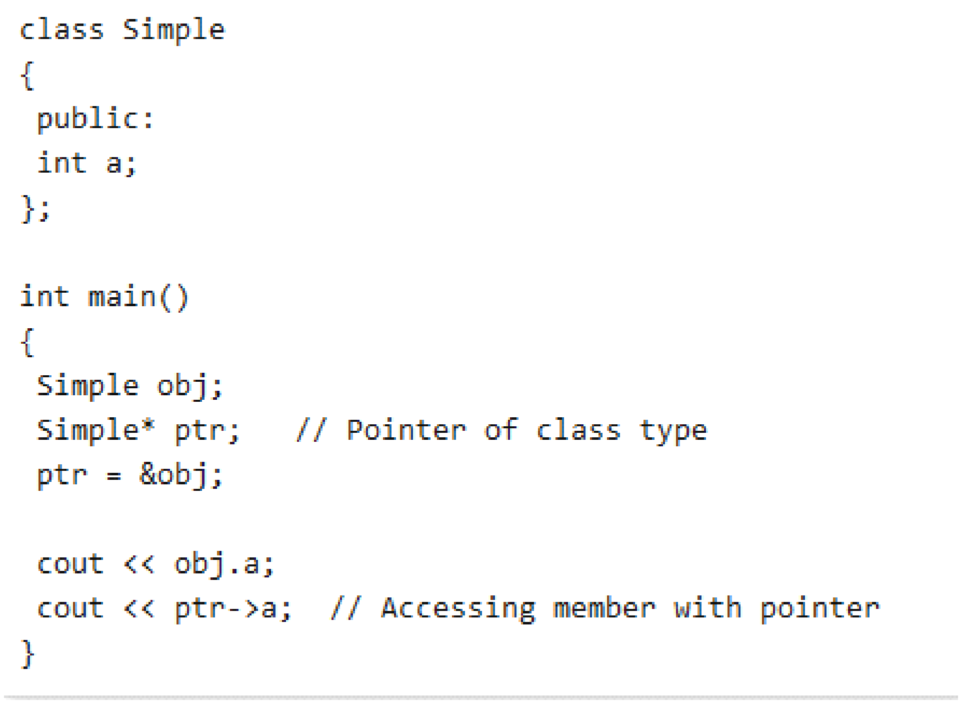
<u>Syntax</u>

```
expression .* expression
expression ->* expression
```

There are two pointer to member operators: `.*` and `->*`.

C++ is object-oriented, so classes can have methods. Non-static member functions (instance methods) have an implicit parameter (the ***this* pointer**) which is the pointer to the object it is operating on, so the type of the object must be included as part of the type of the function pointer. The method is then used on an object of that class by using one of the "pointer-to-member" operators: .* or ->* (for an object or a pointer to object, respectively).

Simple Example
We can define pointer of class type which can be used to point to class objects



```
class Simple
{
 public:
 int a;
};

int main()
{
 Simple obj;
 Simple* ptr;   // Pointer of class type
 ptr = &obj;

 cout << obj.a;
 cout << ptr->a;  // Accessing member with pointer
}
```

Here you see that we have declared a ppointer of class type which points to class's object. We can access data members and member fucntions using pointer name with arrow ->symbol.

✓ **nested classes**
Nested class is a class defined inside a class, that can be used within the scope of the class in which it is defined.
Example:

```
class Nest
{
public:
class Display
{
private:
int s;
public:
void sum( int a, int b)
{ s =a+b; }
void show( )
{ cout << "\nSum of a and b is:: " << s;}
};
};
void main()
{
Nest::Display x;
x.sum(12, 10);
x.show();
}
```

Result:

```
Sum of a and b is::22
```

- ✓ **container class libraries**

A container is a holder object that stores a collection of other objects (its elements). They are implemented as class a template, which allows a great flexibility in the types supported as elements.

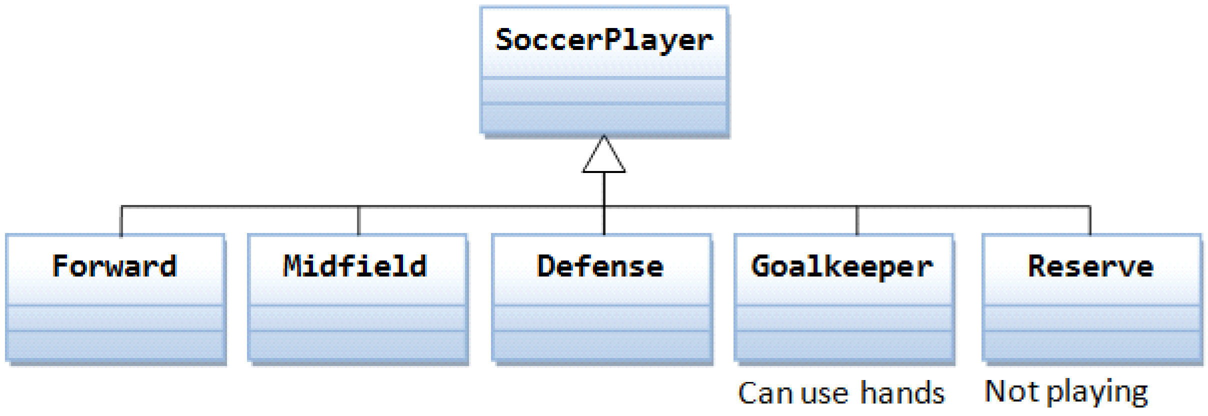
The container manages the storage space for its elements and provides member functions to access them, either directly or through iterators (reference objects with similar properties to pointers).

Containers replicate structures very commonly used in programming: dynamic arrays (vector), queues (queue), stacks (stack), heaps (priority_queue), linked lists (list), trees (set), associative arrays (map)...

CHAPTER 5: INHERITANCE

Introduction and Rules in inheritance

In OOP, we often organize classes in *hierarchy* to *avoid duplication and reduce redundancy*. The classes in the lower hierarchy inherit all the variables (static attributes) and methods (dynamic behaviors) from the higher hierarchies. A class in the lower hierarchy is called a *subclass* (or *derived*, *child*, *extended class*). A class in the upper hierarchy is called a *superclass* (or *base*, *parent class*). By pulling out all the common variables and methods into the superclasses, and leave the specialized variables and methods in the subclasses, *redundancy* can be greatly reduced or eliminated as these common variables and methods do not need to be repeated in all the subclasses. For example,



Example

The following C++ code establishes an explicit inheritance relationship between classes **B** and **A**, where **B** is both a subclass and a subtype of **A**, and can be used as an **A** wherever a **B** is specified (via a reference, a pointer or the object itself).

```
class A
{ public:
   void DoSomethingALike() const {}
};

class B : public A
{ public:
   void DoSomethingBLike() const {}
};

void UseAnA(A const& some_A)
{
   some_A.DoSomethingALike();
}

void SomeFunc()
{
   B b;
   UseAnA(b); // b can be substituted for an A.
}
```

Types of inheritance

There are various types of inheritance, depending on paradigm and specific language.

- **Single Inheritance** : In the single inheritance, subclasses inherits the features of a single super class. A class acquire the property of another class.
- **Multiple Inheritance** : Multiple Inheritance allows a class to have more than one super class and to inherit features from all parent class.
- **Multilevel Inheritance** : In multilevel inheritance a subclass is inherited from another subclass.
- **Hierarchical Inheritance** : In hierarchical inheritance a single class serves as a superclass (base class) for more than one sub class.
- **Hybrid Inheritance** : It is a mixture of all the above types of inheritance.

Importance of Inheritance

Inheritance is a good choice when:-

- Your inheritance hierarchy represents an "is-a" relationship and not a "has-a" relationship.
- You can reuse code from the base classes.
- You need to apply the same class and methods to different data types.
- The class hierarchy is reasonably shallow, and other developers are not likely to add many more levels.
- You want to make global changes to derived classes by changing a base class.

Inheritance Advantages and Disadvantages

Advantages:-

1. One of the key benefits of inheritance is to minimize the amount of duplicate code in an application by sharing common code amongst several subclasses. Where equivalent code exists in two related classes, the hierarchy can usually be refactored to move the common code up to a mutual superclass. This also tends to result in a better organization of code and smaller, simpler compilation units.
2. Inheritance can also make application code more flexible to change because classes that inherit from a common superclass can be used interchangeably. If the return type

of a method is superclass

3. Reusability -- facility to use public methods of base class without rewriting the same

4. Extensibility -- extending the base class logic as per business logic of the derived class

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

6. Overriding--With inheritance, we will be able to override the methods of the base class so that meaningful implementation of the base class method can be designed in the derived class.

Disadvantages:-

1. One of the main disadvantages of inheritance in Java (the same in other object-oriented languages) is the increased time/effort it takes the program to jump through all the levels of overloaded classes. If a given class has ten levels of abstraction above it, then it will essentially take ten jumps to run through a function defined in each of those classes

2. Main disadvantage of using inheritance is that the two classes (base and inherited class) get tightly coupled.
This means one cannot be used independent of each other.

3. Also with time, during maintenance adding new features both base as well as derived classes are required to be changed. If a method signature is changed then we will be affected in both cases (inheritance & composition)

4. If a method is deleted in the "super class" or aggregate, then we will have to re-factor in case of using that method. Here things can get a bit complicated in case of inheritance because our programs will still compile, but the methods of the subclass will no longer be overriding superclass methods. These methods will become independent methods in their own right.

Implementation of Inheritance

Following are attributes observed in the implementation of inheritance

Base & Derived Classes:

A class can be derived from more than one classes, which means it can inherit data and functions from multiple base classes. To define a derived class, we use a class derivation list to specify the base class(es). A class derivation list names one or more base classes and has the form:

```
class derived-class: access-specifier base-class
```

Where access-specifier is one of **public, protected,** or **private**, and base-class is the name of a previously defined class. If the access-specifier is not used, then it is private by default.

Consider a base class **Shape** and its derived class **Rectangle** as follows:

```
#include <iostream>

using namespace std;

// Base class
class Shape
{
   public:
      void setWidth(int w)
      {
         width = w;
      }
      void setHeight(int h)
      {
         height = h;
      }
   protected:
      int width;
      int height;
};

// Derived class
class Rectangle: public Shape
{
   public:
      int getArea()
      {
         return (width * height);
      }
};

int main(void)
{
   Rectangle Rect;

   Rect.setWidth(5);
   Rect.setHeight(7);

   // Print the area of the object.
   cout << "Total area: " << Rect.getArea() << endl;

   return 0;
}
```

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

```
Total area: 35
```

Access Control and Inheritance:

A derived class can access all the non-private members of its base class. Thus base-class members that should not be accessible to the member functions of derived classes should be declared private in the base class.

We can summarize the different access types according to who can access them in the following way:

Access	public	protected	private
Same class	yes	yes	yes
Derived classes	yes	yes	no
Outside classes	yes	no	no

A derived class inherits all base class methods with the following exceptions:

- Constructors, destructors and copy constructors of the base class.
- Overloaded operators of the base class.
- The friend functions of the base class.

When deriving a class from a base class, the base class may be inherited through **public, protected** or **private** inheritance. The type of inheritance is specified by the access-specifier as explained above.

We hardly use **protected** or **private** inheritance, but **public** inheritance is commonly used. While using different type of inheritance, following rules are applied:

- **Public Inheritance:** When deriving a class from a **public** base class, **public** members of the base class become **public** members of the derived class and **protected** members of the base class become **protected** members of the derived class. A base class's **private** members are never accessible directly from a derived class, but can be accessed through calls to the **public** and **protected** members of the base class.
- **Protected Inheritance:** When deriving from a **protected** base class, **public** and **protected** members of the base class become **protected** members of the derived class.
- **Private Inheritance:** When deriving from a **private** base class, **public** and **protected** members of the base class become **private** members of the derived class.

Concepts in inheritance

Single Inheritance is method in which a derived class has only one base class.

Example:

```
#include <iostream.h> class Value
{
protected:
int val;
public:
void set_values (int a)
{ val=a;}
}; class Cube: public Value
{
public:
int cube()
{ return (val*val*val); }
}; int main ()
{
Cube cub;
cub.set_values (5);
cout << "The Cube of 5 is::" << cub.cube() << endl;
return 0;
}
```

Result:

```
The Cube of 5 is:: 125
```

Multiple inheritance is achieved whenever more than one class acts as base classes for other classes. This makes the members of the base classes accessible in the derived class, resulting in better integration and broader re-usability.

example:

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

class Cpolygon
{
	protected:
		int width, height;
	public:
		void input_values (int one, int two)
		{
			width=one;
			height=two;
		}
};

class Cprint
{
	public:
		void printing (int output);
};

void Cprint::printing (int output)
{
	cout << output << endl;
}
```

```
class Crectangle: public Cpolygon, public Cprint
{
        public:
                int area ()
                {
                        return (width * height);
                }
};

class Ctriangle: public Cpolygon, public Cprint
{
        public:
                int area ()
                {
                        return (width * height / 2);
                }
};

int main ()
{
        Crectangle rectangle;
        Ctriangle triangle;
        rectangle.input_values (2,2);
        triangle.input_values (2,2);
        rectangle.printing (rectangle.area());
        triangle.printing (triangle.area());
        return 0;
}
```

Note:the two public statements in the Crectangle class and Ctriangle class.

Inheritance and friends

Friend functions

In principle, private and protected members of a class cannot be accessed from outside the same class in which they are declared. However, this rule does not apply to *"friends"*.

Friends are functions or classes declared with the `friend` keyword.

A non-member function can access the private and protected members of a class if it is declared a *friend* of that class. That is done by including a declaration of this external function within the class, and preceding it with the keyword `friend`:

```
// friend functions
#include <iostream>
using namespace std;

class Rectangle {
    int width, height;
  public:
    Rectangle() {}
    Rectangle (int x, int y) : width(x), height(y) {}
    int area() {return width * height;}
    friend Rectangle duplicate (const Rectangle&);
};

Rectangle duplicate (const Rectangle& param)
{
  Rectangle res;
  res.width = param.width*2;
  res.height = param.height*2;
  return res;
}

int main () {
  Rectangle foo;
  Rectangle bar (2,3);
  foo = duplicate (bar);
  cout << foo.area() << '\n';
  return 0;
}
```

The `duplicate` function is a *friend* of class `Rectangle`. Therefore, function `duplicate` is able to access the members `width` and `height` (which are private) of different objects of type `Rectangle`.

Friend classes

Similar to friend functions, a friend class is a class whose members have access to the private or protected members of another class:

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

class Square;

class Rectangle {
    int width, height;
  public:
    int area ()
      {return (width * height);}
    void convert (Square a);
};

class Square {
  friend class Rectangle;
  private:
    int side;
  public:
    Square (int a) : side(a) {}
};

void Rectangle::convert (Square a) {
  width = a.side;
  height = a.side;
}

int main () {
  Rectangle rect;
  Square sqr (4);
  rect.convert(sqr);
  cout << rect.area();
  return 0;
}
```

In this example, class `Rectangle` is a friend of class `Square` allowing `Rectangle`'s member functions to access private and protected members of `Square`. More concretely, `Rectangle` accesses the member variable `Square::side`, which describes the side of the square.

Pointers to objects

A variable that holds an address value is called a pointer variable or simply pointer.
Pointer can point to objects as well as to simple data types and arrays. sometimes we don't know, at the time that we write the program , how many objects we want to creat. when this is the case we can use new to create objects while the program is running. new returns a pointer to an unnamed objects. Let's see the example of student that will clear your idea about this topic

Pointer to Members in C++ Classes

Just like pointers to normal variables and functions, we can have pointers to class member functions and member variables.

Defining a pointer of class type

We can define pointer of class type, which can be used to point to class objects.

```
class Simple
{
 public:
 int a;
};

int main()
{
 Simple obj;
 Simple* ptr;   // Pointer of class type
 ptr = &obj;

 cout << obj.a;
 cout << ptr->a;  // Accessing member with pointer
}
```

Here you can see that we have declared a pointer of class type which points to class's object. We can access data members and member functions using pointer name with arrowsymbol.

Pointer to Data Members of class

We can use pointer to point to class's data members (Member variables).

Syntax for Declaration :

```
datatype class_name :: *pointer_name ;
```

Syntax for Assignment :

```
pointer_name = &class_name :: datamember_name ;
```

Both declaration and assignment can be done in a single statement too.

```
datatype class_name::*pointer_name =
&class_name::datamember_name ;
```

Using with Objects

For accessing normal data members we use the dot "." operator with object and "->" with pointer to object. But when we have a pointer to data member, we have to dereference that pointer to get what its pointing to, hence it becomes,

```
Object.*pointerToMember
```

and with pointer to object, it can be accessed by writing,

```
ObjectPointer->*pointerToMember
```

Let's take an example, to understand the complete concept.

```
class Data
{
 public:
 int a;
 void print() { cout << "a is="<< a; }
};

int main()
{
 Data d, *dp;
 dp = &d;     // pointer to object

 int Data::*ptr=&Data::a;   // pointer to data member 'a'

 d.*ptr=10;
 d.print();

 dp->*ptr=20;
 dp->print();
}


Output : a is=10 a is=20
```

The syntax is very tough, hence they are only used under special circumstances.

```
Pointer to Member Functions
```

Pointers can be used to point to class's Member functions.

```
Syntax :
```

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

Below is an example to show how we use pointer to member functions.

```
class Data
{ public:
  int f (float) { return 1; }
};

int (Data::*fp1) (float) = &Data::f;   // Declaration and
assignment
int (Data::*fp2) (float);        // Only Declaration

int main(0
{
 fp2 = &Data::f;   // Assignment inside main()
}
```

```
Some Points to remember
```

1. You can change the value and behaviour of these pointers on runtime. That means, you can point it to other member function or member variable.

2. To have pointer to data member and member functions you need to make them public.

Inheritance and constructors

Constructors and destructors are special member functions of classes that are used to construct and destroy class objects. Construction may involve memory allocation and initialization for objects. Destruction may involve cleanup and deallocation of memory for objects.

Derived classes do not inherit constructors or destructors from their base classes, but they do call the constructor and destructor of base classes. Destructors can be declared with the keyword **virtual**.

Constructors are also called when local or temporary class objects are created, and destructors are called when local or temporary objects go out of scope.

Constructor and Destructor invoking sequence with inheritance

Example Program

```
class Base
{
  public:
```

```
  Base ( )
  {
    cout << "Inside Base constructor" << endl;
  }


  ~Base ( )
  {
    cout << "Inside Base destructor" << endl;
  }

};

class Derived : public Base
{

  public:

  Derived  ( )
  {
    cout << "Inside Derived constructor" << endl;
  }

  ~Derived ( )
  {
    cout << "Inside Derived destructor" << endl;
  }

};

void main( )
{
  Derived x;
}
```

So, here is what the output of the code above would look like:

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

<u>Base class constructors and derived class destructors are called first</u>

In the code above, when the object "x" is created, first the Base class constructor is called, and after that the Derived class constructor is called. Because the Derived class inherits from the Base class, both the Base class and Derived class constructors will be called when a Derived class object is created.

When the main function is finished running, the object x's destructor will get called first, and after that the Base class destructor will be called.

Base class conversions

Upcasting is converting a derived-class reference or pointer to a base-class. In other words, upcasting allows us to treat a derived type as though it were its base type. It is always allowed for **public** inheritance, without an explicit type cast. This is a result of the **is-a** relationship between the base and derived classes.

Here is the code dealing with shapes. We created **Shape** class, and derived **Circle**, **Square**, and **Triangle** classes from the **Shape** class. Then, we made a member function that talks to the base class:

```
void play(Shape& s)
{
   s.draw();
   s.move();
   s.shrink();
   ....
}
```

The function speaks to any **Shape**, so it is independent of the specific type of object that it's drawing, moving, and shrinking. If in some other part of the program we use the **play( )** function like below:

```
Circle c;
Triangle t;
Square sq;
play(c);
play(t);
play(sq);
```

Let's check what's happening here. A **Triangle** is being passed into a function that is expecting a **Shape**. Since a **Triangle** is a **Shape**, it can be treated as one by **play()**. That is, any message that **play()** can send to a **Shape** a **Triangle** can accept.

Upcasting allows us to treat a derived type as though it were its base type. That's how we decouple ourselves from knowing about the exact type we are dealing with.

Note that it doesn't say "If you're a **Triangle**, do this, if you're a **Circle**, do that, and so on." If we write that kind of code, which checks for all the possible types of a **Shape**, it will soon become a messy code, and we need to change it every time we add a new kind of **Shape**. Here, however, we just say "You're a **Shape**, I know you can move(), draw(), and shrink( ) yourself, do it, and take care of the details correctly."

The compiler and runtime linker handle the details. If a member function is virtual, then when we send a message to an object, the object will do the right thing, even when upcasting is involved.
Note that the most important aspect of inheritance is not that it provides member functions for the new class, however. It's the **relationship** expressed between the new class and the base class. This relationship can be summarized by saying, **"The new class is a type of the existing class."**

```
class Parent {
public:
  void sleep() {}
};

class Child: public Parent {
public:
  void gotoSchool(){}
};

int main( )
{
  Parent parent;
  Child child;

  // upcast - implicit type cast allowed
  Parent *pParent = &child;

  // downcast - explicit type case required
  Child *pChild =  (Child *) &parent;

  pParent -> sleep();
  pChild -> gotoSchool();

  return 0;
}
```

A **Child** object is a **Parent** object in that it inherits all the data members and member functions of a **Parent** object. So, anything that we can do to a **Parent** object, we can do to a **Child** object. Therefore, a function designed to handle a **Parent** pointer (reference) can perform the same acts on a **Child** object without any problems. The same idea applies if we pass a pointer to an object as a function argument. Upcasting is **transitive**: if we derive a **Child** class from **Parent**, then **Parent** pointer (reference) can refer to a **Parent** or a **Child** object.

Upcasting can cause object slicing when a derived class object is passed by value as a base class object, as in **foo(Base derived_obj)**.

Downcasting

The opposite process, converting a base-class pointer (reference) to a derived-class pointer (reference) is called **downcasting**. Downcasting is not allowed without an explicit type cast. The reason for this restriction is that the **is-a** relationship is not, in most of the cases, symmetric. A derived class could add new data members, and the class member functions that used these data members wouldn't apply to the base class.

As in the example, we derived **Child** class from a **Parent** class, adding a member function, **gotoSchool()**. It wouldn't make sense to apply the **gotoSchool()** method to a **Parent** object. However, if implicit downcasting were allowed, we could accidentally assign the address of a **Parent** object to a pointer-to-**Child**

```
Child *pChild =  &parent; // actually this won't compile
        // error: cannot convert from 'Parent *' to 'Child *'
```

and use the pointer to invoke the **gotoSchool()** method as in the following line.

```
pChild -> gotoSchool();
```

Because a **Parent** isn't a **Child** (a **Parent** need not have a **gotoSchool()** method), the downcasting in the above line can lead to an **unsafe** operation.

C++ provides a special explicit cast called **dynamic_cast** that performs this conversion. Downcasting is the opposite of the basic object-oriented rule, which states objects of a derived class, can always be assigned to variables of a base class.

One more thing about the upcasting:
Because **implicit upcasting** makes it possible for a base-class pointer (reference) to refer to a base-class object or a derived-class object, there is the need for **dynamic binding**. That's why we have **virtual** member functions.

- **Pointer (Reference)** type: known at **compile** time.
- **Object** type: not known until **run** time.

Dynamic Casting

The **dynamic_cast** operator answers the question of whether we can **safely** assign the address of an object to a pointer of a particular type.

Here is a similar example to the previous one.

```
#include <string>

class Parent {
public:
  void sleep() {
  }
};

class Child: public Parent {
private:
  std::string classes[10];
public:
  void gotoSchool(){}
};

int main( )
{
  Parent *pParent = new Parent;
  Parent *pChild = new Child;

  Child *p1 = (Child *) pParent;  // #1
  Parent *p2 = (Child *) pChild;  // #2
  return 0;
}
```

Let look at the lines where we do type cast.

```
Child *p1 = (Child *) pParent;  // #1
Parent *p2 = (Child *) pChild;  // #2
```

Which of the type cast is safe?
The only one guaranteed to be safe is the ones in which the pointer is the same type as the object or else a base type for the object.

Type cast #1 is not safe because it assigns the address of a base-class object (**Parent**) to a derived class (**Child**) pointer. So, the code would expect the base-class object to have derived class properties such as **gotoSchool()** method, and that is false. Also, **Child** object, for example, has a member **classes** that a **Parent** object is lacking.

Type case #2, however, is safe because it assigns the address of a derived-class object to a base-class pointer. In other words, public derivation promises that a **Child** object is also a **Parent** object.

The question of whether a type conversion is safe is more useful than the question of what kind of object is pointed to. The usual reason for wanting to know the type is so that we can know if it's safe to invoke a particular method.

Here is the syntax of **dynamic_cast**.

```
Child *p = dynamic_cast<Child *>(pParent)
```

This code is asking whether the pointer **pParent** can be type cast safely to the type **Child ***.

- It returns the address of the object, if it can.
- It returns **0**, otherwise.

How do we use the **dynamic_cast**?

```
void f(Parent* p) {
  Child *ptr = dynamic_cast<Child*>(p);
   if(ptr) {
     // we can safely use ptr
  }
}
```

In the code, if **(ptr)** is of the type **Child** or else derived directly or indirectly from the type **Child**, the **dynamic_cast** converts the pointer **p** to a pointer of type **Child**. Otherwise, the expression evaluates to **0**, the null pointer.

In other words, we want to check if we can use the passed in pointer **p** before we do some operation on a child class object even though it's a pointer to base class.

"The need for **dynamic_cast** generally arises because we want perform **derived class operation** on a **derived class object**, but we have only a pointer-or reference-to-**base**." -Scott Meyers

<u>Notes</u>

- *downcast* - A downcast is a cast from a base class to a class derived from that base class.

- *cross-cast* - A cross-cast is a cast between unrelated types (user-defined conversion)

Class Scope under Inheritance

Each class defines its own scope within which its members are defined. Under inheritance, the scope of a derived class is nested inside the scope of its base classes. If a name is unresolved within the scope of the derived class, the enclosing base-class scopes are searched for a definition of that name.

The fact that the scope of a derived class nests inside the scope of its base classes can be surprising. After all, the base and derived classes are defined in separate parts of our program's text. However, it is this hierarchical nesting of class scopes that allows the members ...

Overloading with Inheritance

Overloading doesn't work for derived class in C++ programming language. There is no overload resolution between Base and Derived. The compiler looks into the scope of Derived, finds the single function "double f(double)" and calls it. It never disturbs with the (enclosing) scope of Base. In C++, there is no overloading across scopes – derived class scopes are not an exception to this general rule.

Inheritance relationship

Subclasses and superclasses can be understood in terms of the **is a** relationship. A subclass **is a** more specific instance of a superclass. For example, an orange **is a** citrus fruit, which **is a** fruit. A shepherd **is a** dog, which **is an** animal.
If the **is a** relationship does not exist between a subclass and superclass, you should not use inheritance.

Note: inheritance only reuses implementation and establishes a syntactic relationship, not necessarily a semantic relationship (inheritance does not ensure **behavioral subtyping**). To distinguish these concepts, subtyping is also known as *interface inheritance,* while inheritance as defined here is known as **implementation inheritance**.

CHAPTER 6: POLYMORPHISM

Introduction to Polymorphism

Polymorphism is an object-oriented programming concept that refers to the ability of a variable, function or object to take on multiple forms. A language that features polymorphism allows developers to program in the general rather than program in the specific.

In a programming language that exhibits polymorphism, objects of classes belonging to the same hierarchical tree (i.e. inherited from a common base class) may possess functions bearing the same name, but each having different behaviors.

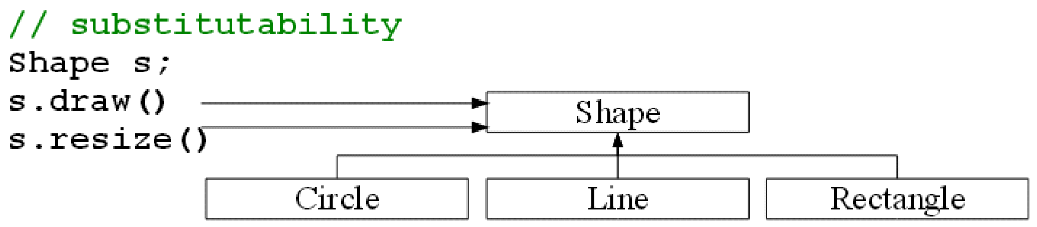
As an example, let us assume there is a base class named Animals from which the subclasses Horse, Fish and Bird are derived. Let us also assume that the Animals class has a function named Move, which is inherited by all subclasses mentioned. With polymorphism, each subclass may have its own way of implementing the function. So, for example, when the Move function is called in an object of the Horse class, the function might respond by displaying trotting on the screen. On the other hand, when the same function is called in an object of the Fish class, swimming might be displayed on the screen. In the case of a Bird object, it may be flying.

In effect, polymorphism trims down the work of the developer because he can now create a sort of general class with all the attributes and behaviors that he envisions for it. When the time comes for the developer to create more specific subclasses with certain unique attributes and behaviors, the developer can simply alter code in the specific portions where the behaviors will differ. All other portions of the code can be left as is.

Advantages/importance of Polymorphism

When an object has a reference to another, it can invoke methods on that object reference without knowing, or caring, what the implementation is.

Why Polymorphism?



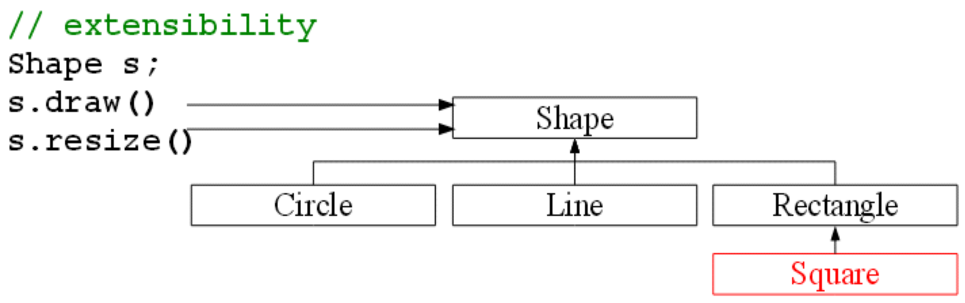
```
// substitutability
Shape s;
s.draw()
s.resize()
```

Note

•Substitutability means, the type of the variable does not have to match with the type of the value assigned to that variable.

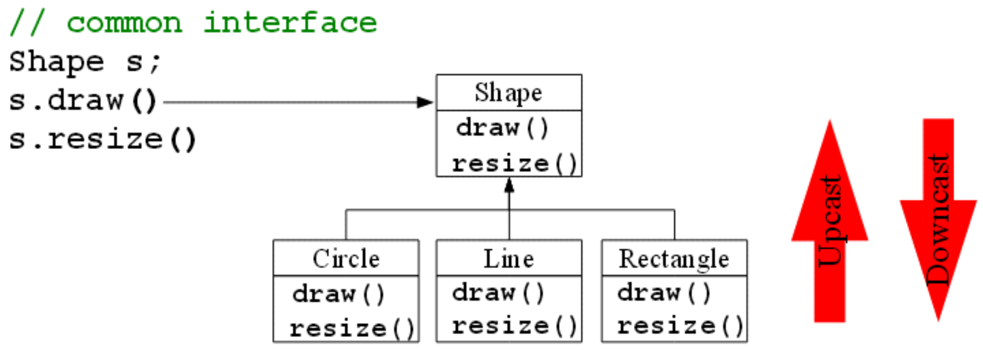
•Substitutability cannot be achieved in conventional languages in C, but can be achieved in Object Oriented languages like Java.

•We have already seen the concept of “Assigning a subclass object to superclass variable or reference”. This is called substitutability. Here I am substituting the superclass object with the object of subclass.



```
// extensibility
Shape s;
s.draw()
s.resize()
```

Extensibility is the ability of an object-oriented system to add new behaviors to an existing system without changing the application shell.



```
// common interface
Shape s;
s.draw()
s.resize()
```

```
// upcasting
Shape s = new Line();
s.draw()
s.resize()
```

Discussion

The word **polymorphism** means having many forms. Typically, polymorphism occurs when there is a hierarchy of classes and they are related by inheritance.

C++ polymorphism means that a call to a member function will cause a different function to be executed depending on the type of object that invokes the function.

Consider the following example where a base class has been derived by other two classes:

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

class Shape {
   protected:
      int width, height;
   public:
      Shape( int a=0, int b=0)
      {
         width = a;
         height = b;
      }
      int area()
      {
         cout << "Parent class area :" <<endl;
         return 0;
      }
};
class Rectangle: public Shape{
   public:
      Rectangle( int a=0, int b=0):Shape(a, b) { }
      int area ()
      {
         cout << "Rectangle class area :" <<endl;
         return (width * height);
      }
};
class Triangle: public Shape{
   public:
      Triangle( int a=0, int b=0):Shape(a, b) { }
      int area ()
      {
         cout << "Triangle class area :" <<endl;
         return (width * height / 2);
      }
};
// Main function for the program
int main( )
{
   Shape *shape;
   Rectangle rec(10,7);
   Triangle  tri(10,5);

   // store the address of Rectangle
   shape = &rec;
   // call rectangle area.
   shape->area();

   // store the address of Triangle
   shape = &tri;
   // call triangle area.
   shape->area();

   return 0;
}
```

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

```
Parent class area
Parent class area
```

The reason for the incorrect output is that the call of the function area() is being set once by the compiler as the version defined in the base class. This is called **static resolution** of the function call, or **static linkage** - the function call is fixed before the

program is executed. This is also sometimes called **early binding** because the area() function is set during the compilation of the program.
But now, let's make a slight modification in our program and precede the declaration of area() in the Shape class with the keyword **virtual** so that it looks like this:

```
class Shape {
   protected:
      int width, height;
   public:
      Shape( int a=0, int b=0)
      {
         width = a;
         height = b;
      }
      virtual int area()
      {
         cout << "Parent class area :" <<endl;
         return 0;
      }
};
```

After this slight modification, when the previous example code is compiled and executed, it produces the following result:

```
Rectangle class area
Triangle class area
```

This time, the compiler looks at the contents of the pointer instead of it's type. Hence, since addresses of objects of tri and rec classes are stored in *shape the respective area() function is called.

As you can see, each of the child classes has a separate implementation for the function area(). This is how **polymorphism** is generally used. You have different classes with a function of the same name, and even the same parameters, but with different implementations.

Encapsulation / Information hiding

Information hiding is one of the most important principles of OOP inspired from real life which says that all information should not be accessible to all persons. Private information should only be accessible to its owner.
By Information Hiding we mean "*Showing only those details to the outside world which are necessary for the outside world and hiding all other details from the outside world.*"

Real Life Examples of Information Hiding

✓ Your name and other personal information is stored in your brain we can't access this information directly. For getting this information we need to ask you about it and it will be up to you how much details you would like to share with us.

- ✓ An email server may have account information of millions of people but it will share only our account information with us if we request it to send anyone else accounts information our request will be refused.

- ✓ A phone SIM card may store several phone numbers but we can't read the numbers directly from the SIM card rather phone-set reads this information for us and if the owner of this phone has not allowed others to see the numbers saved in this phone we will not be able to see those phone numbers using phone.

In object oriented programming approach we have objects with their attributes and behaviors that are hidden from other classes, so we can say that object oriented programming follows the principle of information hiding.
In the perspective of Object Oriented Programming Information Hiding is,
"Hiding the object details (state and behavior) from the users"
Here by users we mean **"an object"** of another class that is calling functions of this class using the reference of this class object or it may be some other program in which we are using this class.
Information Hiding is achieved in Object Oriented Programming using the following principles,

- All information related to an object is stored within the object
- It is hidden from the outside world
- It can only be manipulated by the object itself

Advantages of Information Hiding
Following are two major advantages of information hiding. It simplifies our Object Oriented Model:
As we saw earlier that our object oriented model only had objects and their interactions hiding implementation details so it makes it easier for everyone to understand our object oriented model. It is a barrier against change propagation. As implementation of functions is limited to our class and we have only given the name of functions to user along with description of parameters so if we change implementation of function it doesn't affect the object oriented model.
We can achieve information hiding using Encapsulation and Abstraction, so we see these two concepts in detail now.

Encapsulation means *"we have enclosed all the characteristics of an object in the object itself".*
Encapsulation and information hiding are much related concepts (information hiding is achieved using Encapsulation). We have seen in previous lecture that object characteristics include data members and behavior of the object in the form of functions. So we can say that Data and Behavior are tightly coupled inside an object and both the information structure and implementation details of its operations are hidden from the outer world.

Examples of Encapsulation
Consider the same example of object Ali of previous lecture we described it as follows.

Ali
Characteristics (attributes) · Name · Age
Behavior (operations) · Walks · Eats

You can see that Ali stores his personal information in itself and its behavior is also implemented in it. Now it is up to object Ali whether he wants to share that information with outside world or not. Same thing stands for its behavior if some other object in real life wants to use his behavior of walking it can not use it without the permission of Ali. So we say that attributes and behavior of Ali are encapsulated in it. Any other object don't know about these things unless Ali share this information with that object through an interface. Same concept also applies to phone which has some data and behavior of showing that data to user we can only access the information stored in the phone if phone interface allow us to do so.

Advantages of Encapsulation
The following are the main advantages of Encapsulation,

1. **Simplicity and clarity**

As all data and functions are stored in the objects so there is no data or function around in program that is not part of any object and is this way it becomes very easy to understand the purpose of each data member and function in an object.

2. **Low complexity**

As data members and functions are hidden in objects and each object has a specific behavior so there is less complexity in code there will be no such situations that a functions is using some other function and that functions is using some other function.

3. **Better understanding**

Everyone will be able to understand whole scenario by simple looking into object diagrams without any issue as each object has specific role and specific relation with other objects.

Encapsulation / Information hiding subject properties

Virtual Function:

A **virtual** function is a function in a base class that is declared using the keyword **virtual**. Defining in a base class a virtual function, with another version in a derived class, signals to the compiler that we don't want static linkage for this function. What we do want is the selection of the function to be called at any given point in the program to be based on the kind of object for which it is called. This sort of operation is referred to as **dynamic linkage**, or **late binding**.

Pure Virtual Functions:

It's possible that you'd want to include a virtual function in a base class so that it may be redefined in a derived class to suit the objects of that class, but that there is no meaningful definition you could give for the function in the base class.

We can change the virtual function area() in the base class to the following:

```
class Shape {
   protected:
      int width, height;
   public:
      Shape( int a=0, int b=0)
      {
         width = a;
         height = b;
      }
      // pure virtual function
      virtual int area() = 0;
};
```

The = 0 tells the compiler that the function has no body and above virtual function will be called **pure virtual function**.

Data Abstraction

Data abstraction refers to, providing only essential information to the outside world and hiding their background details, i.e., to represent the needed information in program without presenting the details.

Data abstraction is a programming (and design) technique that relies on the separation of interface and implementation.

Let's take one real life example of a TV, which you can turn on and off, change the channel, adjust the volume, and add external components such as speakers, VCRs, and DVD players, BUT you do not know its internal details, that is, you do not know how it receives signals over the air or through a cable, how it translates them, and finally displays them on the screen.

Thus, we can say a television clearly separates its internal implementation from its external interface and you can play with its interfaces like the power button, channel changer, and volume control without having zero knowledge of its internals.

Now, if we talk in terms of C++ Programming, C++ classes provides great level of **data abstraction**. They provide sufficient public methods to the outside world to play with the functionality of the object and to manipulate object data, i.e., state without actually knowing how class has been implemented internally.
For example, your program can make a call to the **sort()** function without knowing what algorithm the function actually uses to sort the given values. In fact, the underlying implementation of the sorting functionality could change between releases of the library, and as long as the interface stays the same, your function call will still work.
In C++, we use **classes** to define our own abstract data types (ADT). You can use

the **cout** object of class **ostream** to stream data to standard output like this:

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

int main( )
.{
   cout << "Hello C++" <<endl;
   return 0;
}
```

Here, you don't need to understand how **cout** displays the text on the user's screen. You need to only know the public interface and the underlying implementation of cout is free to change.

Access Labels Enforce Abstraction:
In C++, we use access labels to define the abstract interface to the class. A class may contain zero or more access labels:

- Members defined with a public label are accessible to all parts of the program. The data-abstraction view of a type is defined by its public members.
- Members defined with a private label are not accessible to code that uses the class. The private sections hide the implementation from code that uses the type.

There are no restrictions on how often an access label may appear. Each access label specifies the access level of the succeeding member definitions. The specified access level remains in effect until the next access label is encountered or the closing right brace of the class body is seen.

Benefits of Data Abstraction:
Data abstraction provides two important advantages:

- Class internals are protected from inadvertent user-level errors, which might corrupt the state of the object.
- The class implementation may evolve over time in response to changing requirements or bug reports without requiring change in user-level code.

By defining data members only in the private section of the class, the class author is free to make changes in the data. If the implementation changes, only the class code needs to be examined to see what affect the change may have. If data are public, then any function that directly accesses the data members of the old representation might be broken.

Data Abstraction Example:
Any C++ program where you implement a class with public and private members is an example of data abstraction. Consider the following example:

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

class Adder{
   public:
      // constructor
      Adder(int i = 0)
      {
        total = i;
      }
      // interface to outside world
      void addNum(int number)
      {
          total += number;
      }
      // interface to outside world
      int getTotal()
      {
          return total;
      };
   private:
      // hidden data from outside world
      int total;
};
int main( )
{
   Adder a;

   a.addNum(10);
   a.addNum(20);
   a.addNum(30);

   cout << "Total " << a.getTotal() <<endl;
   return 0;
}
```

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

```
Total 60
```

Above class adds numbers together, and returns the sum. The public members **addNum** and **getTotal**are the interfaces to the outside world and a user needs to know them to use the class. The private member **total** is something that the user doesn't need to know about, but is needed for the class to operate properly.

Designing Strategy:
Abstraction separates code into interface and implementation. So while designing your component, you must keep interface independent of the implementation so that if you change underlying implementation then interface would remain intact.

In this case whatever programs are using these interfaces, they would not be impacted and would just need a recompilation with the latest implementation.

Data Encapsulation

All C++ programs are composed of the following two fundamental elements:

- **Program statements (code):** This is the part of a program that performs actions and they are called functions.
- **Program data:** The data is the information of the program which affected by the program functions.

Encapsulation is an Object Oriented Programming concept that binds together the data and functions that manipulate the data, and that keeps both safe from outside interference and misuse. Data encapsulation led to the important OOP concept of **data**

hiding.

Data encapsulation is a mechanism of bundling the data, and the functions that use them and **data abstraction** is a mechanism of exposing only the interfaces and hiding the implementation details from the user.

C++ supports the properties of encapsulation and data hiding through the creation of user-defined types, called **classes**. We already have studied that a class can contain **private, protected** and **public**members. By default, all items defined in a class are private. For example:

```
class Box
{
   public:
      double getVolume(void)
      {
         return length * breadth * height;
      }
   private:
      double length;      // Length of a box
      double breadth;     // Breadth of a box
      double height;      // Height of a box
};
```

The variables length, breadth, and height are **private**. This means that they can be accessed only by other members of the Box class, and not by any other part of your program. This is one way encapsulation is achieved.

To make parts of a class **public** (i.e., accessible to other parts of your program), you must declare them after the **public** keyword. All variables or functions defined after the public specifier are accessible by all other functions in your program.

Making one class a friend of another exposes the implementation details and reduces encapsulation. The ideal is to keep as many of the details of each class hidden from all other classes as possible.

Data Encapsulation Example:

Any C++ program where you implement a class with public and private members is an example of data encapsulation and data abstraction. Consider the following example:

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

class Adder{
   public:
      // constructor
      Adder(int i = 0)
      {
        total = i;
      }
      // interface to outside world
      void addNum(int number)
      {
          total += number;
      }
      // interface to outside world
      int getTotal()
      {
          return total;
      };
   private:
      // hidden data from outside world
      int total;
};
int main( )
{
   Adder a;

   a.addNum(10);
   a.addNum(20);
   a.addNum(30);

   cout << "Total " << a.getTotal() <<endl;
   return 0;
}
```

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

```
Total 60
```

Above class adds numbers together, and returns the sum. The public members **addNum** and **getTotal**are the interfaces to the outside world and a user needs to know them to use the class. The private member **total** is something that is hidden from the outside world, but is needed for the class to operate properly.

Designing Strategy:
Most of us have learned through bitter experience to make class members private by default unless we really need to expose them. That's just good **encapsulation**.

This wisdom is applied most frequently to data members, but it applies equally to all members, including virtual functions.

Interfaces (Abstract Classes)

An interface describes the behavior or capabilities of a C++ class without committing to a particular implementation of that class.
The C++ interfaces are implemented using **abstract classes** and these abstract classes should not be confused with data abstraction which is a concept of keeping implementation details separate from associated data.
A class is made abstract by declaring at least one of its functions as **pure virtual** function. A pure virtual function is specified by placing "= 0" in its declaration as follows:

```
class Box
{
   public:
      // pure virtual function
      virtual double getVolume() = 0;
   private:
      double length;      // Length of a box
      double breadth;     // Breadth of a box
      double height;      // Height of a box
};
```

The purpose of an **abstract class** (often referred to as an ABC) is to provide an appropriate base class from which other classes can inherit. Abstract classes cannot be used to instantiate objects and serves only as an **interface**. Attempting to instantiate an object of an abstract class causes a compilation error.

Thus, if a subclass of an ABC needs to be instantiated, it has to implement each of the virtual functions, which means that it supports the interface declared by the ABC. Failure to override a pure virtual function in a derived class, then attempting to instantiate objects of that class, is a compilation error.

Classes that can be used to instantiate objects are called **concrete classes**.

Abstract Class Example:
Consider the following example where parent class provides an interface to the base class to implement a function called **getArea()**:

```
#include <iostream>

using namespace std;

// Base class
class Shape
{
public:
   // pure virtual function providing interface framework.
   virtual int getArea() = 0;
   void setWidth(int w)
   {
      width = w;
   }
   void setHeight(int h)
   {
      height = h;
   }
protected:
   int width;
   int height;
};

// Derived classes
class Rectangle: public Shape
{
public:
   int getArea()
   {
      return (width * height);
   }
};
class Triangle: public Shape
{
public:
   int getArea()
   {
      return (width * height)/2;
   }
};

int main(void)
{
   Rectangle Rect;
   Triangle  Tri;

   Rect.setWidth(5);
   Rect.setHeight(7);
   // Print the area of the object.
   cout << "Total Rectangle area: " << Rect.getArea() << endl;

   Tri.setWidth(5);
   Tri.setHeight(7);
   // Print the area of the object.
   cout << "Total Triangle area: " << Tri.getArea() << endl;

   return 0;
}
```

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

```
Total Rectangle area: 35
Total Triangle area: 17
```

You can see how an abstract class defined an interface in terms of getArea() and two other classes implemented same function but with different algorithm to calculate the area specific to the shape.

Designing Strategy:
An object-oriented system might use an abstract base class to provide a common and standardized interface appropriate for all the external applications. Then, through inheritance from that abstract base class, derived classes are formed that all operate similarly.

The capabilities (i.e., the public functions) offered by the external applications are provided as pure virtual functions in the abstract base class. The implementations of these pure virtual functions are provided in the derived classes that correspond to the specific types of the application.

This architecture also allows new applications to be added to a system easily, even after the system has been defined.

CHAPTER 7: CONSTRUCTORS AND DESTRUCTORS

Definition of Constructors

Constructors are special class functions which performs initialization of every object. The Compiler calls the Constructor whenever an object is created. Constructors iitialize values to object members after storage is allocated to the object

```
class A
{
    int x;
    public:
    A(); //Constructor
};
```

While defining a contructor you must remeber that the name of constructor will be same as the name of the class, and contructors never have return type.

Constructors can be defined either inside the class definition or outside class definition using class name and scope resolution"::" operator.

```
class A
{
    int i;
    public:
    A(); //Constructor declared
    };
    A::A() // Constructor definition
    {
    i=1;
}
```

Types of Constructors

Constructors are of three types :

1. Default Constructor
2. Parametrized Constructor
3. Copy Constructor

<u>Default Constructor</u>

Default constructor is the constructor which doesn't take any argument. It has no parameter.

Syntax :

class_name ()
{ Constructor Definition }

Example :
```
        class Cube
        {
              int side;
              public:
              Cube()
              {
                    side=10;
              }
        };
        int main()
        {
              Cube c;
              cout << c.side;
}
```

Output : 10

In this case, as soon as the object is created the constructor is called which initializes its data members.

A default constructor is so important for initialization of object members, that even if we do not define a constructor explicitly, the compiler will provide a default constructor implicitly.

```
      class Cube
         {
               int side;
         };
      int main()
      {
          Cube c;
          cout << c.side;
         }
```

Output : 0

In this case, default constructor provided by the compiler will be called which will initialize the object data members to default value, that will be 0 in this case.

Parameterized Constructor

These are the constructors with parameter. Using this Constructor you can provide different values to data members of different objects, by passing the appropriate values as argument.

Example :

```
    class Cube
    {
        int side;
        public:
        Cube(int x)
          {
              side=x;
          }
    };
    int main()
    {
         Cube c1(10);
         Cube c2(20);
         Cube c3(30);
         cout << c1.side;
         cout << c2.side;
         cout << c3.side;
}
```

OUTPUT : 10 20 30

By using parameterized construcor in above case, we have initialized 3 objects with user defined values. We can have any number of parameters in a constructor.

<u>Copy Constructor</u>
These are special type of Constructors which takes an object as argument, and is used to copy values of data members of one object into other object. We will study copy constructors in detail later.

Constructor Overloading

Just like other member functions, constructors can also be overloaded. Infact when you have both default and parameterized constructors defined in your class you are having Overloaded Constructors, one with no parameter and other with parameter.

You can have any number of Constructors in a class that differ in parameter list

```
        class Student
        {
           int rollno;
           string name;
           public:
           Student(int x)
           {
               rollno=x;
               name="None";
            }
           Student(int x, string str)
           {
               rollno=x ;
               name=str ;
           }
        };
        int main()
        {
           Student A(10);
           Student B(11,"Ram");
              }
```

In above case we have defined two constructors with different parameters, hence overloading the constructors.

One more important thing, if you define any constructor explicitly, then the compiler will not provide default constructor and you will have to define it yourself.

In the above case if we write   Student S;   in **main()**, it will lead to a compile time error, because we haven't defined default constructor, and compiler will not provide its default constructor because we have defined other parameterized constructors.

Destructors

Destructor is a special class function which destroys the object as soon as the scope of object ends. The destructor is called automatically by the compiler when the object goes out of scope.
The syntax for destructor is same as that for the constructor, the class name is used for the name of destructor, with a **tilde** ~ sign as prefix to it.

```
class A
{
   public:
   ~A();
};
```

Destructors will never have any arguments.

Example to see how Constructor and Destructor is called

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

Implementation of constructors and Destructors

Single Definition for both Default and Parameterized Constructor

In this example we will use **default argument** to have a single definition for both defualt and parameterized constructor.

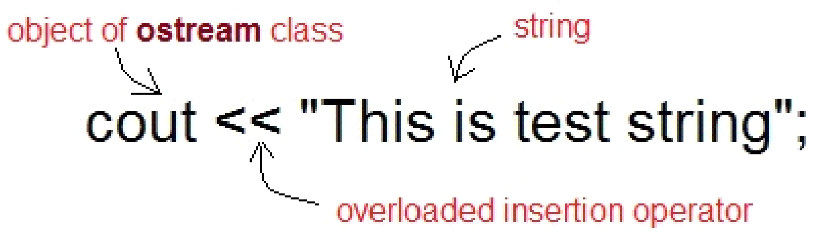
```
class Dual
{
    int a;
    public:
    Dual(int x=0)
      {
          a=x;
      }
};
int main()
{
    Dual obj1;
    Dual obj2(10);
}
```

Here, in this program, a single Constructor definition will take care for both these object initializations. We don't need separate default and parameterized constructors.

CHAPTER 8: OPERATOR OVERLOADING

Meaning and importance of operator overloading

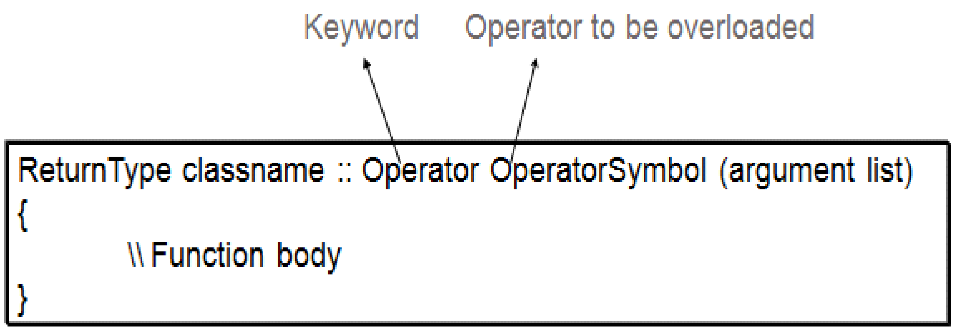
Operator overloading is an important concept in C++. It is a type of polymorphism in which an operator is overloaded to give user defined meaning to it. Overloaded operator is used to perform operation on user-defined data type. For example '+' operator can be overloaded to perform addition on various data types, like for Integer, String(concatenation) etc.


object of **ostream** class
string
cout << "This is test string";
overloaded insertion operator

Almost any operator can be overloaded in C++. However there are few operator which can not be overloaded. **Operator that are not overloaded** are follows

1. scope operator - **::**
2. sizeof
3. member selector - **.**
4. member pointer selector - *****
5. ternary operator - **?:**

Operator Overloading Syntax



```
ReturnType classname :: Operator OperatorSymbol (argument list)
{
        \\ Function body
}
```

Implementing Operator Overloading

Operator overloading can be done by implementing a function which can be :

1. Member Function
2. Non-Member Function
3. Friend Function

Operator overloading function can be a member function if the Left operand is an Object of that class, but if the Left operand is different, then Operator overloading

function must be a non-member function.
Operator overloading function can be made friend function if it needs access to the private and protected members of class.

Restrictions on Operator Overloading
Following are some restrictions to be kept in mind while implementing operator overloading.

1. Precedence and Associativity of an operator cannot be changed.
2. Arity (numbers of Operands) cannot be changed. Unary operator remains unary, binary remains binary etc.
3. No new operators can be created, only existing operators can be overloaded.

Cannot redefine the meaning of a procedure. You cannot change how integers are added.

Operator Overloading Examples
Almost all the operators can be overloaded in infinite different ways. Following are some examples to learn more about operator overloading. All the examples are closely connected.
Overloading Arithmetic Operator
Arithmetic operator are most commonly used operator in C++. Almost all arithmetic operator can be overloaded to perform arithmetic operation on user-defined data type. In the below example we have overridden the **+** operator, to add to Time (hh:mm:ss) objects.
Example: overloading '+' Operator to add two time object

```
#include< iostream.h>
#include< conio.h>
class time
{
    int h,m,s;
    public:
    time()
    {
        h=0, m=0; s=0;
    }
    void getTime();
    void show()
    {
        cout<< h<< ":"<< m<< ":"<< s;
    }
    time operator+(time); //overloading '+' operator
};
time time::operator+(time t1) //operator function
{
    time t;
    int a,b;
    a=s+t1.s;
```

```
        t.s=a%60;
        b=(a/60)+m+t1.m;
        t.m=b%60;
        t.h=(b/60)+h+t1.h;
        t.h=t.h%12;
        return t;
    }
    void time::getTime()
    {
        cout<<"\n Enter the hour(0-11) ";
        cin>>h;
        cout<<"\n Enter the minute(0-59) ";
        cin>>m;
        cout<<"\n Enter the second(0-59) ";
        cin>>s;
    }
    void main()
    {
        clrscr();
        time t1,t2,t3;
        cout<<"\n Enter the first time ";
        t1.getTime();
        cout<<"\n Enter the second time ";
        t2.getTime();
        t3=t1+t2; <font color="green">//adding of two time object using '+' operator
        cout<<"\n First time ";
        t1.show();
        cout<<"\n Second time ";
        t2.show();
        cout<<"\n Sum of times ";
        t3.show();
        getch();
    }
```

Overloading I/O operator

1. Overloaded to perform input/output for user defined datatypes.
2. Left Operand will be of types ostream& and istream&
3. Function overloading this operator must be a Non-Member function because left operand is not an Object of the class.
4. It must be a friend function to access private data members.

You have seen above that **<<** operator is overloaded with **ostream** class object **cout** to print primitive type value output to the screen. Similarly you can overload **<<** operator in your class to print user-defined type to screen. For example we will overload **<<** in **time** class to display time object using **cout** .

```
        time t1(3,15,48);
        cout << t1;
```

NOTE: When the operator does not modify its operands, the best way to overload the operator is via friend function.

Example: overloading '<<' Operator to print time object

```
#include< iostream.h>
#include< conio.h>
class time
{
    int hr,min,sec;
    public:
    time()
        {
        hr=0, min=0; sec=0;
        }
    time(int h,int m, int s)
        {
        hr=h, min=m; sec=s;
        }
    friend ostream& operator << (ostream &out, time &tm); //overloading '<<'
    operator
};
ostream& operator<< (ostream &out, time &tm) //operator function
    {
    out << "Time is " << tm.hr << "hour : " << tm.min << "min : " << tm.sec << "sec";
    return out;
    }
    void main()
    {
    time tm(3,15,45);
    cout << tm;
}
```

Output

```
Time is 3 hour : 15 min : 45 sec
```

Overloading Relational operator

You can also overload Relational operator like **==** ,**!=** , **>=** , **<=** etc. to compare two user-defined object.

Example

```
        class time
```

```
    {
        int hr,min,sec;
        public:
        time()
          {
          hr=0, min=0; sec=0;
          }
        time(int h,int m, int s)
           {
           hr=h, min=m; sec=s;
           }
        friend bool operator==(time &t1, time &t2); //overloading '==' operator
           };
           bool operator== (time &t1, time &t2)   //operator function
           {
        return ( t1.hr == t2.hr &&
        t1.min == t2.min &&
        t1.sec == t2.sec );
    }
```

Copy constructor Vs. Assignment operator

Assignment operator is used to copy the values from one object to another **already existing object**. For example

time tm(3,15,45); //tm object created and initialized
time t1; //t1 object created
t1 = tm; //initializing t1 using tm

Copy constructor is a special constructor that initializes a **new object** from an existing object.

time tm(3,15,45); //tm object created and initialized
time t1(tm); //t1 object created and initialized using tm object

Note

✓ **Constructor/destructor**

Constructors are called automatically by the compiler when defining class objects. The destructor's are called when a class object goes out of scope.

Destructors do not take parameters/arguments. Passing Information to a Method or a Constructor involves declaration for a method or a constructor then declares the number and the type of the arguments for that method or constructor.

Default constructor is a constructor that can be called without having to provide any arguments, irrespective of whether the constructor is auto-generated or used-defined.

Like any other function, a constructor can also be overloaded with different versions taking different parameters: with a different number of parameters and/or parameters of different types. The compiler will automatically call the one whose parameters match the arguments

✓ **Operators are binary or unary.**

Unary Operators has only one operand. The operation takes place using a single operand. The Unary Operators are ++,--,&,*(indirection),-(positive),-(neg... etc.
Unary operators have precedence over binary operators.

Binary operators (”bi” as in “two”) have two operands. In “A*B” the * operator has two operands: A and B. In “!B” the “!” operator (meaning boolean NOT) has only one operand, and is therefore a unary operator. The “-” and “+” operators can be both binary and unary, in “-4” or “+4” it denotes a negative or a positive number, in “0-4” it acts as the subtraction operator.

✓ General Rules for Operator Overloading

1) Only built-in operators can be overloaded. New operators can not be created.
2) Arity of the operators (number of arguments or operands the function or operation accepts) cannot be changed.
3) Precedence and associativity of the operators cannot be changed.
4) Overloaded operators cannot have default arguments except the function call operator () which can have default arguments.
5) Operators cannot be overloaded for built in types only. At least one operand must be used defined type.
6) Assignment (=), subscript ([]), function call (“()”), and member selection (->) operators must be defined as member functions
7) Except the operators specified in point 6, all other operators can be either member functions or a non-member functions.
8) Some operators like (assignment)=, (address)& and comma (,) are by default overloaded.

✓ **Overloading new and delete Operators in C++**

Why overload new and delete?

1. To take charge or control over how to allocate memory
2. To aid in debugging; keep track of memory allocation and deallocation in the program
3. To do some other operation apart from allocating memory at the time of memory allocation/decallocation

Given below is a simple sample demonstrating overloaded `new` and `delete`:

```
void* operator new(size_t num)
{
    return malloc(num);
}

void operator delete(void *ptr)
{
    free(ptr);
}
```

Observe the following:

1. The overloaded `new` operator receives a parameter `num` of type `size_t`. This is the number of bytes of memory to be allocated. The compiler calculates and sends this to us!
2. The return type of the overloaded `new` must be `void*`. It is expected to return a pointer to the beginning of the block of memory allocated. Note that after our overloaded `new` returns, the compiler then automatically calls the constructor also as applicable.
3. The overloaded `delete` operator receives a parameter `ptr` of type `void*`. This is the pointer the user is trying to delete.
4. The overloaded `delete` operator should not return anything.
5. In this sample implementation, since the focus is only on showing how to overload, we have done the memory allocation and deallocation using `malloc()` and `free()` functions. In real life situations, we would prefer to do something more than this!

Deleting an array of objects is not the same as deleting an object, i.e. `delete[] ptr;` and `delete ptr;` are totally different and involve different operators! In case you are interested in overloading `delete` for an array, use the following:

```
void operator delete[](void *ptr)
{
    free(ptr);
}
```

✓ Reference and dereference operators

- & is the reference operator and can be read as "address of".
- * is the dereference operator and can be read as "value pointed by".

Examples

If the reference operator is used you will get the "address of" a variable.

ptr_p = &x;

Meaning: store the address of the variable x in the pointer ptr_p

If the dereference operator is used you will get the "value pointed by" a pointer.

```
x = 5;
ptr_p = &x;
cout << *ptr_p;
```

Meaning: print (or put into the stream) the value pointed by ptr_p. (It will print the contents of integer x.)

- ✓ **conversion operators** - It is not possible to redefine a pre-**defined conversion**. Thus, **conversion operators** are not allowed to **convert** from or to object because implicit and explicit conversions already exist between object and all other types.
- ✓ function call - A call/prompt that passes control to a subroutine/procedure/function; after the subroutine is executed control returns to the next instruction in main program

CHAPTER 9: FILE ORGANISATION

Introduction to File organization

Files contain computer records which can be documents or information which is stored in a certain way for later retrieval.

File organization refers primarily to the logical arrangement of data (which can itself be organized in a system of records with correlation between the fields/columns) in a file system. It should not be confused with the physical storage of the file in some types of storage media. There are certain basic types of computer file, which can include files stored as blocks of data and streams of data, where the information streams out of the file while it is being read until the end of the file is encountered.

file organization is a design decision, hence it must be done having in mind the achievement of good performance with respect to the most likely usage of the file. The criteria usually considered important are:

1. Fast access to single record or collection of related recors.
2. Easy record adding/update/removal, without disrupting .
3. Storage efficiency.
4. Redundance as a warranty against data corruption.

Description of File Stream

Stream is not a hardware it is linear queue which connect file to program and passes block of data in both direction .So it is independent of devices which we are using. We can also define stream as source of data. This source can be

(a) A file

(b) Hard disk or CD, DVD etc.

(c) I/O devices etc.

In c++ programming language there are two type of stream.

(a) Text streams

(b) Binary streams

Files and Streams properties

So far, we have been using the **iostream** standard library, which provides **cin** and **cout** methods for reading from standard input and writing to standard output respectively.

To read and write from a file, requires another standard C++ library called **fstream**, which defines three new data types:

Data Type	Description
ofstream	This data type represents the output file stream and is used to create files and to write information to files.
ifstream	This data type represents the input file stream and is used to read information from files.
fstream	This data type represents the file stream generally, and has the capabilities of both ofstream and ifstream which means it can create files, write information to files, and read information from files.

To perform file processing in C++, header files <iostream> and <fstream> must be included in your C++ source file.

Opening a File:

A file must be opened before you can read from it or write to it. Either the **ofstream** or **fstream** object may be used to open a file for writing and ifstream object is used to open a file for reading purpose only.

Following is the standard syntax for open() function, which is a member of fstream, ifstream, and ofstream objects.

```
void open(const char *filename, ios::openmode mode);
```

Here, the first argument specifies the name and location of the file to be opened and the second argument of the **open()** member function defines the mode in which the file should be opened.

Mode Flag	Description
ios::app	Append mode. All output to that file to be appended to the end.
ios::ate	Open a file for output and move the read/write control to the end of the file.
ios::in	Open a file for reading.
ios::out	Open a file for writing.
ios::trunc	If the file already exists, its contents will be truncated before opening the file.

You can combine two or more of these values by **OR**ing them together. For example if you want to open a file in write mode and want to truncate it in case it already exists, following will be the syntax:

```
ofstream outfile;
outfile.open("file.dat", ios::out | ios::trunc );
```

Similar way, you can open a file for reading and writing purpose as follows:

```
fstream  afile;
afile.open("file.dat", ios::out | ios::in );
```

Closing a File

When a C++ program terminates it automatically closes flushes all the streams, release all the allocated memory and close all the opened files. But it is always a good practice that a programmer should close all the opened files before program termination.

Following is the standard syntax for close() function, which is a member of fstream, ifstream, and ofstream objects.

```
void close();
```

Writing to a File:

While doing C++ programming, you write information to a file from your program using the stream insertion operator (<<) just as you use that operator to output information to the screen. The only difference is that you use an **ofstream** or **fstream** object instead of the **cout** object.

Reading from a File:

You read information from a file into your program using the stream extraction operator (>>) just as you use that operator to input information from the keyboard. The only difference is that you use an **ifstream** or **fstream** object instead of the **cin** object.

Read & Write Example:

Following is the C++ program which opens a file in reading and writing mode. After writing information inputted by the user to a file named afile.dat, the program reads information from the file and outputs it onto the screen:

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

int main ()
{

   char data[100];

   // open a file in write mode.
   ofstream outfile;
   outfile.open("afile.dat");

   cout << "Writing to the file" << endl;
   cout << "Enter your name: ";
   cin.getline(data, 100);

   // write inputted data into the file.
   outfile << data << endl;

   cout << "Enter your age: ";
   cin >> data;
   cin.ignore();
   // again write inputted data into the file.
   outfile << data << endl;

   // close the opened file.
   outfile.close();

   // open a file in read mode.
   ifstream infile;
   infile.open("afile.dat");

   cout << "Reading from the file" << endl;
   infile >> data;

   // write the data at the screen.
   cout << data << endl;

   // again read the data from the file and display it.
   infile >> data;
   cout << data << endl;

   // close the opened file.
   infile.close();

   return 0;
}
```

When the above code is compiled and executed, it produces the following sample input and output:

```
$./a.out
Writing to the file
Enter your name: Zara
Enter your age: 9
Reading from the file
Zara
9
```

Above examples make use of additional functions from cin object, like getline() function to read the line from outside and ignore() function to ignore the extra characters left by previous read statement.

File Position Pointers:

Both **istream** and **ostream** provide member functions for repositioning the file-position pointer. These member functions are **seekg** ("seek get") for istream and **seekp** ("seek put") for ostream.

The argument to seekg and seekp normally is a long integer. A second argument can be specified to indicate the seek direction. The seek direction can be **ios::beg** (the default) for positioning relative to the beginning of a stream, **ios::cur** for positioning relative to the current position in a stream or **ios::end** for positioning relative to the end of a stream.

The file-position pointer is an integer value that specifies the location in the file as a number of bytes from the file's starting location. Some examples of positioning the "get" file-position pointer are:

```
// position to the nth byte of fileObject (assumes ios::beg)
fileObject.seekg( n );

// position n bytes forward in fileObject
fileObject.seekg( n, ios::cur );

// position n bytes back from end of fileObject
fileObject.seekg( n, ios::end );

// position at end of fileObject
fileObject.seekg( 0, ios::end );
```

CHAPTER 10: EMERGING TRENDS IN OOP

The major goals of object-orientation are to produce *well-structured software* and to build more *extensible and reusable systems*. It seems that the second goal is more important because it is more difficult to achieve and there remains more work to be done. We believe that the following areas of OO will most prominently contribute to this goal in the future:

Frameworks
Frameworks have been recognized as one of the most promising OO technologies because they provide *large scale* reuse. The reuse of individual classes, while helpful, will not bring significant productivity leaps. Only the reuse of taylorable systems *as a whole* will lead to noticeable results. Although frameworks are widely used today, there remain many open problems to be tackled, for example:

- **How can a framework be designed systematically?**
  Current frameworks are developed in a rather ad-hoc manner by comparing existing systems and extracting their common features. This process is iterative and good frameworks are only obtained after many iterations.
  Why is framework design not more systematic? One reason is that current design methods hardly apply to frameworks. Their aim is to model *one specific* problem but not a set of *various similar* problems. Traditionally, the main question is "what are the objects in a given situation?" wheras for frameworks the main question should be "what is to be kept flexible so that it fits several situations?". We need special methods and guidelines for designing frameworks. Such methods could be based on design patterns that help putting together a framework from templates of proven OO knowledge.

- **How can a framework be documented so that it is easy to use?**
  The documentation of a framework has to be different from the documentation of traditional software because it must not only describe what the framework does but also what the user-provided parts are expected to do. It must show the "hot spots", i.e., the places where custom functionality can be hooked in. Providing the source code of the framework is a common solution but not a good one. Besides finding better graphical and/or formal notations, a promising approach seems to be the employment of more sophisticated development environments that use hypertext, visualization and other online assistance to understand and extend a framework.

- **What makes a framework good or bad?**
  Is it the number of hot spots, the reuse factor (i.e., the ratio between the framework code and the user code), or the simplicity with which a framework can be adapted? There are hardly any guidelines, let alone metrics, to make an objective judgement and thus to drive the framework development process. The work on frameworks has spawned off the idea of *componentware*. Instead of delivering a system as a prepackaged monolith containing any conceivable feature, modern systems consist of a light-weight kernel to which new features can be added (often dynamically) in the form of black box components. This helps keeping systems small and frees

users from carrying along unnecessary functionality. Pluggable components will also become an interesting issue for small software vendors and for creative programmers who want to distribute free software via the Net.

Design Patterns
OO software development is still in a state comparable to traditional programming in the sixties. There are sufficiently powerful languages but programs have to be assembled from low-level building blocks: in traditional languages these building blocks are integers, arrays, pointers; in OO languages they are objects.

In traditional programming, the invention of data structures such as lists, trees or queues solved common problems and abstracted from the building blocks involved. Similarly, design patterns can be viewed as *"object structures"* that capture common OO solutions and abstract from the underlying building blocks, namely the objects. In the same way as the traditional pattern of a "binary tree" is assembled from nodes and pointers, the object-oriented pattern of a "decorator", say, is assembled from objects of certain classes. Design patterns are data structures (and sometimes also the algorithms) in object-oriented software development. Research challenges of the future will be:

- Finding *new patterns* and unifying existing ones.
- Finding *better notations* to describe patterns, their structure, their dynamic behavior, and their integration into larger systems. Beside graphical notations, formal pattern languages might be a promising solution.
- *Integrating patterns with programming languages*. If a programming language could express and check the correct use of design patterns, programming could be raised to a higher level and source code could become better understandable.

Distributed Objects
Distributed objects that operate in a concurrent and active way have been a research topic in the OO community for a long time because the metaphor of communicating objects lends itself to distribution well. With the success of the Internet and appropriate programming languages such as Java this topic has gained additional relevance. How can intelligent and active objects improve the use of the Net? How can security problems be solved that result from such a distribution? How can objects in heterogeneous environments (different computers, languages, and operating systems) cooperate effectively?

Languages and Environments
If one looks through the proceedings of the recent OOPSLA and ECOOP conferences one will notice that languages and their implementations are still one of the most intensive research areas. Although I don't expect really spectacular innovations here in the near future I believe that there is still room for improvement in most widely used OO languages of today. These improvements do not have so much to do with new language features but with the more fundamental concepts of safety and dynamic extensibility.

- **Safety.**
  Object-orientation is intended for building large and complex systems. The larger a system the more important is safety and memory integrity. Any violation of semantic contracts must be detected, the earlier the better. This requires that contracts can be expressed in a language and that the compiler checks them. Memory integrity means that programming errors must not have disastrous effects on the data. For example, a garbage collector eliminates the frequent error of bad memory deallocation. Yet many languages do not rely on such mechanisms.

- **Dynamic extensibility.**
  Extensibility is one of the virtues of object-orientation. Yet many systems are restricted by static extensibility. They can only be extended by interrupting their use, relinking and reloading them. Relinking requires that the individual object files are available, which is usually not the case in commercial systems. In contrast, dynamic languages such as Smalltalk and Self allow extending a system at run time without disrupting its use. This should also become possible in compiled languages and systems (It is already the case in Java or Oberon, for example.)

Finally, an important challenge will be to bring all these techniques to the mainstream software industry. While many software products claim to be object-oriented, they often just use an OO language or at best classes as a structuring aid. The idea of (custom-) extensible systems is not yet common in practice.