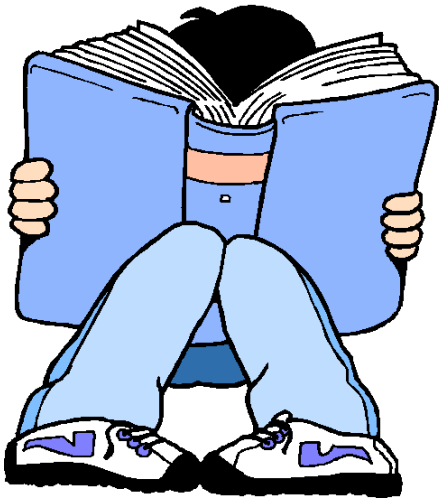


OBJECT ORIENTED PROGRAMMING

OBJECT ORIENTED PROGRAMMING(R 403)

Module 1

Introduction to OOP - Evolution of object oriented languages - Need of Objects - Definition of Object-Oriented Language – Classes and Objects – Creating and Using Classes and objects – Member functions and variables – Constructors and Destructors.

Module 2

Inheritance and Access Control - Member access control in classes – Friend functions and classes – Extending classes - Public Private and Protected Inheritance – Classification of Inheritance – Single – Multiple – Multilevel – Hierarchical – Hybrid.

Module 3

Polymorphism – Runtime and compile time polymorphism – overloading functions and operators – selecting friend member function for operator overloading - Virtual methods – pure virtual methods – Abstract classes - Defining and using of virtual methods, pure virtual methods and abstract classes – applications of abstract classes.

Module 4

Advanced Concepts- Virtual Destructors – Virtual Base Classes - Template classes – Creating and using templates – Namespaces

Module 5

Dynamic Objects - Dynamic object allocation - Inline functions.

Other Object oriented languages – Java – Object oriented features in Java – Comparison with C++

References

1. Object Oriented Programming in C ++ - Robert Lafore, Galgotia Pub.
2. Object Oriented Programming in C++ - Nabajyoti Barkakati, PHI
3. Structured and Object Oriented Problem Solving using C++ - Andrew C Staugaard Jr., PHI
4. Object oriented Programming with C++ - E. Balaguruswamy, TMH
5. Java 2 Complete Reference - Herbert, Schildt, TMH
6. The Java Programming Language 3rd Edition - Arnold, Gosling, Holmes, Pearson Education Asia
7. Object-oriented programming using C++ - Ira Pohl, Pearson Education Asia
8. C++ How to program - Dietel & Dietel, Pearson Education Asia
9. An Introduction to Object-oriented programming – Timothy Budd
10. Problem Solving with C++ - Walter Savitch, Pearson Education Asia
11. C++ Primer - Stanley B Lippman, Josee Zajoie, Pearson Education Asia

MODULE 1

INTRODUCTION TO OOP

Object Oriented Programming Paradigm

Oops is a better way of solving problems in computers compared to the procedural language programming such as in C. oops is designed around the data being operated upon as opposed to the operations, these operations are designed to fit data.

A type of programming in which programmers define not only the data type of a data structure, but also the types of operations 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.

One of the principal advantages of object-oriented programming techniques over procedural programming techniques is that they enable programmers to create modules that do not need to be changed when a new type of object is added. A programmer can simply create a new object that inherits many of its features from existing objects. This makes object-oriented programs easier to modify.

To perform object-oriented programming, one needs an object-oriented programming language such as Java C++ etc.

The C++ programming language provides a model of memory and computation that closely matches that of most computers. In addition, it provides powerful and flexible mechanisms for abstraction; that is, language constructs that allow the programmer to introduce and use new types of objects that match the concepts of an application.

Thus, C++ supports styles of programming that rely on fairly direct manipulation of hardware resources to deliver a high degree of efficiency plus higher-level styles of programming that rely on user-defined types to provide a model of data and computation that is closer to a human's view of the task being performed by a computer. These higher-level styles of programming are often called data abstraction, object-oriented programming, and generic programming.

Features of OOPs are the following:

- Encapsulation
- Data abstraction

- Inheritance
- Polymorphism
- Message passing
- Extensibility
- Persistence
- Delegation
- Genericity
- Multiple Inheritance

Elements of Object Oriented Programming

- Object-Oriented Programming is centered around the new concepts such as classes, polymorphism, inheritance, etc. it is a well suited paradigm for the following:
- Modeling the real world problem as close as possible to the users perspective.
- Interacting easily with computational environment using familiar metaphors
- Constructing reusable software components and easily extendable libraries.
- Easily modifying and extending implementations of components without having to recode everything from scratch.

Definition of OOP:

OOP uses objects as its fundamental building blocks. Each object is an instance of some class. Classes allow the mechanism of data abstraction for creating new data types. Inheritance allows building of new classes from existing classes. Hence if any of these elements are missing in a program we cannot consider that program as objected oriented program.

Object oriented programming is a programming methodology that associates data structures with a set of operators which act upon it. In OOP's terminology an instance of such an entity is known as an object. It gives importance to relationships between objects rather than implementation details. Hiding the implementation details within an object results in the user being more concerned with an objects relationship to the rest of the system, than the implementation of the object's behavior.

Extensibility

C++ allows the extension of the functionality of the existing software components. In C++ this is achieved through abstract classes and inheritance.

Persistence

The phenomenon where the object (data) outlives the program execution time and exists between executions of a program is known as persistence. All data base systems support persistence. In c++ it is not supported. However the user can build it explicitly using file streams in a program.

Delegation

Delegation is a way of making object composition as powerful as inheritance. In delegation two objects are involved in handling a request a receiving object delegates operations to its delegate. This is analogous to child class sending requests to the parent class.

Genericity

It is technique for defining software components that have more than one interpretation depending on the data type of parameters. Thus it allows the declaration of data items without specifying their exact data type.

Multiple Inheritance

The mechanism by which a class is derived from more than one base class is known as multiple inheritance. Instances of classes with multiple inheritance have instance variables for each of the inherited base classes. C++ supports multiple inheritance.

Basic Concepts of OOPS

In this tutorial you will learn about Objects, Classes, Inheritance, Data Abstraction, Data Encapsulation, Polymorphism, Overloading, Reusability.

Before starting to learn C++ it is essential that one must have a basic knowledge of the concepts of Object oriented programming. Some of the important object oriented features are namely:

- Objects
- Classes
- Inheritance
- Data Abstraction
- Data Encapsulation
- Polymorphism
- Overloading
- Reusability
- Dynamic Binding
- Message Passing

In order to understand the basic concepts in C++, the programmer must have a command of the basic terminology in object-oriented programming. Below is a brief outline of the concepts of Object-oriented programming languages:

Objects:

Object is the basic unit of object-oriented programming. Objects are identified by its unique name. An object represents a particular instance of a class. There can be more than one instance of an object. Each instance of an object can hold its own relevant data.

An Object is a collection of data members and associated member functions also known as methods.

Classes:

Classes are data types based on which objects are created. Objects with similar properties and methods are grouped together to form a Class. Thus a Class represent a set of individual objects. Characteristics of an object are represented in a class as **Properties**. The actions that can be performed by objects becomes functions of the class and is referred to as **Methods**.

For example consider we have a Class of *Cars* under which *Santro Xing*, *Alto* and *WaganR* represents individual Objects. In this context each *Car* Object will have its own, Model, Year of Manufacture, Colour, Top Speed, Engine Power etc., which form **Properties** of the *Car* class and the associated actions i.e., object functions like Start, Move, Stop form the **Methods** of *Car* Class.

No memory is allocated when a class is created. Memory is allocated only when an object is created, i.e., when an instance of a class is created.

Inheritance:

Inheritance is the process of forming a new class from an existing class or *base class*. The base class is also known as *parent class* or *super class*, The new class that is formed is called *derived class*. Derived class is also known as a *child class* or *sub class.* Inheritance helps in reducing the overall code size of the program, which is an important concept in object-oriented programming.

Inheritance is a means of specifying hierarchical relationships between types C++ classes can inherit both data and function members from other (parent) classes. Terminology: "the child (or derived) class inherits (or is derived from) the parent (or base) class

Data Abstraction:

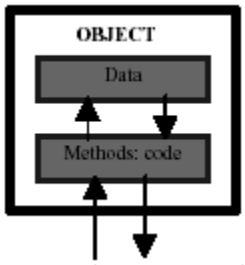
Data Abstraction increases the power of programming language by creating user defined data types. Data Abstraction also represents the needed information in the program without presenting the details.

A data abstraction is a simplified view of an object that includes only features one is interested in while hides away the unnecessary details. In programming languages, a data abstraction becomes an abstract data type or a user-defined type. In OOP, it is implemented as a class.

Data Encapsulation:

Data Encapsulation combines data and functions into a single unit called Class. When using Data Encapsulation, data is not accessed directly; it is only accessible through the functions present inside the class. Data Encapsulation enables the important concept of data hiding possible.

Encapsulation is the mechanism that binds together code and the data it manipulates, and keeps them both safe from outside. In an object-oriented language, code and data can be combined in such a way that a self-contained 'black box' is created. When code and data are link together in this fashion , an object is created: Within an object, code, data, or both may be *private* to that object or *public*. Private code or data is known to and accessible only by another part of the object (i.e. cannot be accessed by a piece of the program that exists outside the object. Public code or data can be accessed by other parts of the program even though it is defined within an object. Public parts of an object are used to provide a controlled interface to the private elements of the object.



An object is a variable of a user-defined type. Each time you define a new type of object, you are creating a new data type. Each specific instance of this data type is a compound variable.

Polymorphism:

Polymorphism allows routines to use variables of different types at different times. An operator or function can be given different meanings or functions. Polymorphism refers to a single function or multi-functioning operator performing in different ways.

Polymorphism is in short the ability to call different functions by just using one type of function call. It is a lot useful since it can group classes and their functions together. Polymorphism means that the same thing can exist in two forms. This is an important characteristic of true object oriented design - which means that one could develop good OO design with data abstraction and inheritance, but the real power of object oriented design seems to surface when polymorphism is used.

In C++, polymorphism means that if the same message is sent to different objects, the object's behavior depends on the nature of the object itself. This is sort of obvious for completely different objects, but the concept starts making sense when combined with inheritance.

In C++ it is possible to use one function name for many different purposes. This type of polymorphism is called function overloading. Polymorphism can also be applied to operators. In that case it is called operator overloading.

More generally the concept of polymorphism is characterised by the idea 'one interface, multiple methods'. The key point to remember about polymorphism is that it allows you to handle greater complexity by allowing the creation of standard interfaces to related activities.

Overloading:

Overloading is one type of Polymorphism. It allows an object to have different meanings, depending on its context. When an exiting operator or function begins to operate on new data type, or class, it is understood to be overloaded.

Reusability:

This term refers to the ability for multiple programmers to use the same written and debugged existing class of data. This is a time saving device and adds code efficiency to the language. Additionally, the programmer can incorporate new features to the existing class, further developing the application and allowing users to achieve increased performance. This time saving feature optimizes code, helps in gaining secured applications and facilitates easier maintenance on the application.

Dynamic Binding

Binding refers to the act of associating an object or a class with its member. If we can call a method fn() on an object o of a class c, we say that the object o is binded with the method fn(). This happens at compile time and is known as static or compile - time binding.If it is done at the time of execution it is known as runtime polymorphism.

Message passing

It is the process of invoking an operation on an object. In response to a message the corresponding method is executed in the object.

C++ and Object oriented programming

C++ was developed by Bjarne Stroustrup of AT&T Bell Laboratories in the early 1980's, and is based on the C language. The name is a pun - "++" is a syntactic construct used in C (to increment a variable), and C++ is intended as an incremental improvement of C. Most of C is a subset of C++, so that most C programs can be compiled (i.e. converted into a series of low-level instructions that the computer can execute directly) using a C++ compiler.

Standard C++ is the version of C++ created by the ANSI/ISO2 standardisation committee. The Standard C++ contains several enhancements not found in the traditional C++. Thus, Standard C++ is a superset of traditional C++. Standard C++ is the one that is currently accepted by all major compilers. Therefore, you can be confident that what you learn here will also apply in the future. However, if you are using an older compiler it might not support one or more of the features that are specific to Standard C++. This is important because two recent additions to the C++ language affect every program you will write. If you are using an older compiler that does not accept these new features, don't worry. There is an easy workaround, as you will in a later paragraph. Since C++ was

invented to support object-oriented programming. OOP concepts will be reminded. As you will see, many features of C++ are related to OOP in a way or another. In fact the theory of OOP permeates C++. However, it is important to understand that C++ can be used to write programs that are and are not object oriented. How you use C++ is completely up to you. A few comments about the nature and form of C++ are in order. For most part C++ programs look like C programs. Like a C program, a C++ program begins execution at main( ). To include command-line arguments, C++ uses the same argc, argv convention that C uses. Although C++ defines its own, object oriented library. It also supports all the functions in the C standard library. C++ uses the same control structures as C. C++ includes all the build-in data types defined by C programming.

A Brief History of C++

The C++ Programming Language is basically an extension of the C Programming Language. The C Programming language was developed from 1969-1973 at Bell labs, at the same time the UNIX operating system was being developed there. C was a direct descendant of the language B, which was developed by Ken Thompson as a systems programming language for the fledgling UNIX operating system. B, in turn, descended from the language BCPL which was designed in the 1960s by Martin Richards while at MIT.

In 1971 Dennis Ritchie at Bell Labs extended the B language (by adding types) into what he called NB, for "New B". Ritchie credits some of his changes to language constructs found in Algol68, although he states "although it [the type scheme], perhaps, did not emerge in a form that Algol's adherents would approve of." After restructuring the language and rewriting the compiler for B, Ritchie gave his new language a name: "C".

In 1983, with various versions of C floating around the computer world, ANSI established a committee that eventually published a standard for C in 1989.

In 1983 Bjarne Stroustrup at Bell Labs created C++. C++ was designed for the UNIX system environment, it represents an enhancement of the C programming language and enables programmers to improve the quality of code produced, thus making reusable code easier to write.

C++ Programming Basics

C++ uses a convenient abstraction called *streams* to perform input and output operations in sequential media such as the screen or the keyboard. A stream is an object where a program can either insert or extract characters to/from it. We do not really need to care about many specifications about the physical media associated with the stream - we only need to know it will accept or provide characters sequentialy. The standard C++ library includes the header file iostream, where the standard input and output stream objects are declared.

Standard Output (cout)

By default, the standard output of a program is the screen, and the C++ stream object defined to access it is cout. cout is used in conjunction with the *insertion operator*, which is written as << (two "less than" signs).

```
cout << "Output sentence";
cout << 120;
cout << x;
```

The << operator inserts the data that follows it into the stream preceding it. In the examples above it inserted the constant string Output sentence, the numerical constant 120 and variable x into the standard output stream cout. Notice that the sentence in the first instruction is enclosed between double quotes (") because it is a constant string of characters. Whenever we want to use constant strings of characters we must enclose them between double quotes (") so that they can be clearly distinguished from variable names. For example, these two sentences have very different results:

```
cout << "Hello";
cout << Hello;
```

The insertion operator (<<) may be used more than once in a single statement:

cout << “Hello” << “I am “ << “a C++statement” ;

This last statement would print the message Hello, I am a C++ statement on the screen.

It is important to notice that cout does not add a line break after its output unless we explicitly indicate it, therefore, the following statements:

```
cout << "This is a sentence.";
cout << "This is another sentence." ;
```

will be shown on the screen one following the other without any line break between them:

This is a sentence.This is another sentence.

Additionally, to add a new-line, you may also use the endl manipulator. For example:

```
cout << "First sentence." << endl;
cout << "Second sentence." << endl;
```

The endl manipulator produces a newline character, exactly as the insertion of '\n' does, but it also has an additional behavior when it is used with buffered streams: the buffer is flushed. Anyway, cout will be an unbuffered stream in most cases, so you can generally use both the \n escape character and the endl manipulator in order to specify a new line without any difference in its behavior.

Standard Input (cin)

The standard input device is usually the keyboard. Handling the standard input in C++ is done by applying the overloaded operator of extraction (>>) on the cin stream. The operator must be followed by the variable that will store the data that is going to be extracted from the stream. For example:

```
int age;
cin >> age;
```

The first statement declares a variable of type int called age, and the second one waits for an input from cin (the keyboard) in order to store it in this integer variable.

cin can only process the input from the keyboard once the RETURN key has been pressed. Therefore, even if you request a single character, the extraction from cin will not process the input until the user presses RETURN after the character has been introduced.

You must always consider the type of the variable that you are using as a container with cin extractions. If you request an integer you will get an integer, if you request a character you will get a character and if you request a string of characters you will get a string of characters.


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

int main ()
{
  int i;
  cout << "Please enter an integer value: ";
  cin >> i;
  cout << "The value you entered is " << i;
  cout << " and its double is " << i*2 << ".\n";
  return 0;
}
```

Output : Please enter an integer value: 702
The value you entered is 702 and its double is 1404.

The user of a program may be one of the factors that generate errors even in the simplest programs that use cin (like the one we have just seen). Since if you request an integer value and the user introduces a name (which generally is a string of characters), the result may cause your program to misoperate since it is not what we were expecting from the user. So when you use the data input provided by cin extractions you will have to trust that the user of your program will be cooperative and that he/she will not introduce his/her name or something similar when an integer value is

requested. A little ahead, when we see the stringstream class we will see a possible solution for the errors that can be caused by this type of user input.

You can also use cin to request more than one datum input from the user:

cin >> a >> b;

is equivalent to:

```
cin >> a;
cin >> b;
```

In both cases the user must give two data, one for variable a and another one for variable b that may be separated by any valid blank separator: a space, a tab character or a newline.

C++ program

Our first C++ program will print the message "Hello World" on the screen. Open a text editor and start by typing the following line:

```
#include<iostream>
```

The above line includes the header file called iostream which will allow us to use the command to print words on the screen. Next you must type:


```
using namespace std;
```

This will let us use certain commands without having to type out their full name. Now we will type the main function.

```
int main()
{
}
```

The main function is where a program always starts. Every program must have a main function. The word int in front of main is to say what the return value is. The curly brackets belong to the main function and show where it begins and where it ends. Now we will type the command that prints "Hello World" on the screen between the curly brackets.

```
cout << "Hello World\n";
```

The cout command is used to print things on the screen. The << means that the text must be output. The words that you want printed must be surrounded by quotes. The \n means that the cursor must go the beginning of the next line. Lastly we must return 0 to the operating system to tell it that there were no errors while running the program.

```
return 0;
```

The full program should look like this:

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

int main()
{
  cout << "Hello World\n";
  return 0;
}
```

Save the file as hello.cpp. You now need to compile the program. You need to open a command prompt and type the command name of your C++ compiler and the name of the C++ file you have just created. Here is an example of how to do it with Borland C++:

```
bcc32 hello.cpp
```

If you are given error messages then you have made mistakes which means you should go through this lesson again and fix them. If you don't get any errors then you should end up with an executable file which in my case is called hello.exe. Enter hello to run the program and you should see "Hello World" printed on the screen. Congratulations! You have just made your first C++ program.

Overview of the Basic Structure of C++ Programming

- The // in first line is used for representing comment in the program.
- The second line of the program has a # symbol which represents the preprocessor directive followed by header file to be included placed between < >.
- The next structure present in the program is the class definition. This starts with the keyword class followed by class name *employee*. Within the class are data and functions. The data defined in the class are generally private and functions are public. These explanations we will be detailed in later sections. The class declaration ends with a semicolon.
- main() function is present in all C++ programs.
- An object e1 is created in employee class. Using this e1 the functions present in the employee class are accessed and there by data are accessed.
- The input namely ename and eno is got using the input statement namely cin and the values are outputted using the output statement namely cout.

Comments

Comments explain what a program does and are ignored by the compiler. A single line comment goes after a // and a multiple line comment goes between a /* and a */. Here is an example of how to comment the program we have just made:

```
/* This program will print the words
Hello World on the screen. */
#include<iostream>
using namespace std;

int main()
{
  cout << "Hello World\n"; // Print Hello World on the screen
  return 0; // Return 0 to the OS
}
```

Variable, Constants and Data types in C++

Variables

A variable is the storage location in memory that is stored by its value. A variable is identified or denoted by a variable name. The variable name is a sequence of one or more letters, digits or underscore, for example: character _

Rules for defining variable name:

- A variable name can have one or more letters or digits or underscore for example character _.
- White space, punctuation symbols or other characters are not permitted to denote variable name. .
- A variable name must begin with a letter.
- Variable names cannot be keywords or any reserved words of the C++ programming language.
- C++ is a case-sensitive language. Variable names written in capital letters differ from variable names with the same name but written in small letters. For example, the variable name EXFORSYS differs from the variable name exforsys.

As previously explained, a variable is the storage location in memory that is stored by variable value. The amount of memory allocated or occupied by each variable differs as per the data stored. The amount of memory used to store a single character is different from that of storing a single integer. A variable must be declared for the specific data type.

Data Types

Below is a list of the most commonly used *Data Types* in C++ programming language:

- short int
- int
- long int
- float
- double
- long double
- char
- bool

short int : This data type is used to represent short integer.

int: This data type is used to represent integer.

long int: This data type is used to represent long integer.

float: This data type is used to represent floating point number.

double: This data type is used to represent double precision floating point number.

long double: This data type is used to represent double precision floating point number.

char: This data type is used to represent a single character.

bool: This data type is used to represent boolean value. It can take one of two values: True or False.

Using variable names and data type, we shall now learn how to declare variables.

Declaring Variables:

In order for a variable to be used in C++ programming language, the variable must first be declared. The syntax for declaring variable names is

data type variable name;

The date type can be int or float or any of the data types listed above. A variable name is given based on the rules for defining variable name (refer above rules).

Example:

int a;

This declares a variable name a of type int. If there exists more than one variable of the same type, such variables can be represented by separating variable names using comma.

For instance

int x,y,z

This declares 3 variables x, y and z all of data type int. The data type using integers (int, short int, long int) are further assigned a value of signed or unsigned. Signed integers signify positive and negative number value. Unsigned integers signify only positive numbers or zero.

For example it is declared as

unsigned short int a;
signed int z;

By default, unspecified integers signify a signed integer.

For example:

int a;

is declared a signed integer

It is possible to initialize values to variables:

data type variable name = value;

Example:

int b=5;

Constants

Constants have fixed value. Constants, like variables, contain data type. Integer constants are represented as decimal notation, octal notation, and hexadecimal notation. Decimal notation is represented with a number. Octal notation is represented with the number preceded by a zero character. A hexadecimal number is preceded with the characters 0x.

Example

80 represent decimal
0115 represent octal
0x167 represent hexadecimal

By default, the integer constant is represented with a number.

The unsigned integer constant is represented with an appended character u. The long integer constant is represented with character l.

Example:

78 represent int
85u present unsigned int
78l represent long

Floating point constants are numbers with decimal point and/or exponent.

Example

2.1567
4.02e24

These examples are valid floating point constants.

Floating point constants can be represented with f for floating and l for double precision floating point numbers.

Character constants have single character presented between single quotes.

Example

'c'
'a'

are all character constants.

Strings are sequences of characters signifying string constants. These sequence of characters are represented between double quotes.

Example:

"Exforsys Training"

is an example of string constant.

Loops and Decisions

Decision statements / Conditional structure

A program is usually not limited to a linear sequence of instructions. During its process it may bifurcate, repeat code or take decisions. For that purpose, C++ provides control structures that serve to specify what has to be done by our program, when and under which circumstances.

With the introduction of control structures we are going to have to introduce a new concept: the *compound-statement* or *block*. A block is a group of statements which are separated by semicolons (;) like all C++ statements, but grouped together in a block enclosed in braces: { }:

```
{ statement1; statement2; statement3; }
```

Most of the control structures that we will see in this section require a generic statement as part of its syntax. A statement can be either a simple statement (a simple instruction ending with a semicolon) or a compund statement (several instructions grouped in a block), like the one just described. In the case that we want the statement to be a simple statement, we do not need to enclose it in braces ({}). But in the case that we want the statement to be a compund statement it must be enclosed between braces ({}), forming a block.

if and else

The if keyword is used to execute a statement or block only if a condition is fulfilled. Its form is:

```
if (condition) statement
```

Where condition is the expression that is being evaluated. If this condition is true, statement is executed. If it is false, statement is ignored (not executed) and the program continues right after this conditional structure.
For example, the following code fragment prints x is 100 only if the value stored in the x variable is indeed 100:

```
if (x == 100)
```

If we want more than a single statement to be executed in case that the condition is true we can specify a block using braces { }:

```
if (x == 100)
{
  cout << "x is ";
  cout << x;

}
```

We can additionally specify what we want to happen if the condition is not fulfilled by using the keyword else. Its form used in conjunction with if is:

if (condition) statement1 else statement2

For example:

```
if (x == 100)
  cout << "x is 100";
else
  cout << "x is not 100";
```

prints on the screen x is 100 if indeed x has a value of 100, but if it has not -and only if not- it prints out x is not 100.

The if + else structures can be concatenated with the intention of verifying a range of values. The following example shows its use telling if the value currently stored in x is positive, negative or none of them (i.e. zero):

```
if (x > 0)
  cout << "x is positive";
else if (x < 0)
  cout << "x is negative";
else
  cout << "x is 0";
```


Remember that in case that we want more than a single statement to be executed, we must group them in a block by enclosing them in braces { }.

Iteration structures / loops

Loops have as purpose to repeat a statement a certain number of times or while a condition is fulfilled.

The while loop

Its format is:

```
while (expression) statement
```

and its functionality is simply to repeat statement while the condition set in expression is true.For example, we are going to make a program to countdown using a while-loop:

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

```
int main ()
{
  int n;
  cout << "Enter the starting number > ";
  cin >> n;

  while (n>0) {
    cout << n << ", ";
    --n;
  }

  cout << "FIRE!\n";
  return 0;
}
```

Output : Enter the starting number > 8
8, 7, 6, 5, 4, 3, 2, 1, FIRE!

When the program starts the user is prompted to insert a starting number for the countdown. Then the while loop begins, if the value entered by the user fulfills the condition n>0 (that n is greater than zero) the block that follows the condition will be executed and repeated while the condition (n>0) remains being true.

The whole process of the previous program can be interpreted according to the following script (beginning in main):

1. User assigns a value to n
2. The while condition is checked (n>0). At this point there are two posibilities:
   * condition is true: statement is executed (to step 3)
   * condition is false: ignore statement and continue after it (to step 5)
3. Execute statement:
   cout << n << ", ";
   --n;
   (prints the value of n on the screen and decreases n by 1)
4. End of block. Return automatically to step 2
5. Continue the program right after the block: print FIRE! and end program.

When creating a while-loop, we must always consider that it has to end at some point, therefore we must provide within the block some method to force the condition to become false at some point, otherwise the loop will continue looping forever. In this case we have included --n; that decreases the value of the variable that is being evaluated in the condition (n) by one - this will eventually make the condition (n>0) to become false after a certain number of loop iterations: to be more specific, when n becomes 0, that is where our while-loop and our countdown end.

The do-while loop

Its format is:

```
do statement while (condition);
```

Its functionality is exactly the same as the while loop, except that condition in the do-while loop is evaluated after the execution of statement instead of before, granting at least one execution of statement even if condition is never fulfilled. For example, the following example program echoes any number you enter until you enter 0.

```
include <iostream>
using namespace std;

int main ()
{
  unsigned long n;
  do {
    cout << "Enter number (0 to end): ";
    cin >> n;
    cout << "You entered: " << n << "\n";
  } while (n != 0);
  return 0;
}
```


Output :

```
Enter number (0 to end): 12345
You entered: 12345
Enter number (0 to end): 160277
You entered: 160277
Enter number (0 to end): 0
You entered: 0
```

The do-while loop is usually used when the condition that has to determine the end of the loop is determined within the loop statement itself, like in the previous case, where the user input within the block is what is used to determine if the loop has to end.

The for loop

Its format is:

```
for (initialization; condition; increase) statement;
```

and its main function is to repeat statement while condition remains true, like the while loop. But in addition, the for loop provides specific locations to contain an initialization statement and an increase statement. So this loop is specially designed to perform a repetitive action with a counter which is initialized and increased on each iteration.

It works in the following way:

1. initialization is executed. Generally it is an initial value setting for a counter variable. This is executed only once.
2. condition is checked. If it is true the loop continues, otherwise the loop ends and statement is skipped (not executed).
3. statement is executed. As usual, it can be either a single statement or a block enclosed in braces { }.
4. finally, whatever is specified in the increase field is executed and the loop gets back to step 2.

Here is an example of countdown using a for loop:

```
#include <iostream>
int main ()
{
  for (int n=10; n>0; n--) {
    cout << n << " , ";
  }
  cout << " FIRE!\n";
  return  0;

}
```


The initialization and increase fields are optional. They can remain empty, but in all cases the semicolon signs between them must be written. For example we could write: for (;n<10;) if we wanted to specify no initialization and no increase; or for (;n<10;n++) if we wanted to include an increase field but no initialization. Optionally, using the comma operator (,) we can specify more than one expression in any of the fields included in a for loop, like in initialization, for example. The comma operator (,) is an expression separator, it serves to separate more than one expression where only one is generally expected. For example, suppose that we wanted to initialize more than one variable in our loop:

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

This loop will execute for 50 times if neither n or i are modified within the loop.

Jump statements

The break statement

Using break we can leave a loop even if the condition for its end is not fulfilled. It can be used to end an infinite loop, or to force it to end before its natural end. For example, we are going to stop the count down before its natural end .

```
#include <iostream>
```

```
using namespace std;

int main ()
{
  int n;
  for (n=10; n>0; n--)
  {
    cout << n << ", ";
    if (n==3)
    {
      cout << "countdown aborted!";
      break;
    }
  }
  return 0;
}
```

Output : 10, 9, 8, 7, 6, 5, 4, 3, countdown aborted!

The continue statement

The continue statement causes the program to skip the rest of the loop in the current iteration as if the end of the statement block had been reached, causing it to jump to the start of the following iteration. For example, we are going to skip the number 5 in our countdown:


```
// continue loop example
#include <iostream>
using namespace std;

int main ()
{
  for (int n=10; n>0; n--) {
    if (n==5) continue;
    cout << n << ", ";
  }
  cout << "FIRE!\n";
  return 0;
}
```

Output : 10, 9, 8, 7, 6, 4, 3, 2, 1, FIRE!

The goto statement

Goto allows to make an absolute jump to another point in the program. You should use this feature with caution since its execution causes an unconditional jump ignoring any type of nesting limitations.The destination point is identified by a label, which is then used as an argument for the goto statement. A label is made of a valid identifier followed by a colon (:).

Generally speaking, this instruction has no concrete use in structured or object oriented programming aside from those that low-level programming fans may find for it. For example, here is our countdown loop using goto:

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

int main ()
{
  int n=10;
  loop:
  cout << n << ", ";
  n--;
  if (n>0) goto loop;
  cout << "FIRE!\n";
  return 0;
}
```

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

The exit function

exit is a function defined in the stdlib library. The purpose of exit is to terminate the current program with a specific exit code. Its prototype is:


```
 void exit (int exitcode);
```

The exitcode is used by some operating systems and may be used by calling programs. By convention, an exit code of 0 means that the program finished normally and any other value means that some error or unexpected results happened.

The selective structure: switch

The syntax of the switch statement is a bit peculiar. Its objective is to check several possible constant values for an expression. Something similar to what we did at the beginning of this section with the concatenation of several if and else if instructions. Its form is the following:

```
switch (expression)
{
  case constant1:
     group of statements 1;
     break;
  case constant2:
     group of statements 2;
     break;
  .
```

```
   .
  default:
    default group of statements
}
```

It works in the following way: switch evaluates expression and checks if it is equivalent to constant1, if it is, it executes group of statements 1 until it finds the break statement. When it finds this break statement the program jumps to the end of the switch selective structure.

If expression was not equal to constant1 it will be checked against constant2. If it is equal to this, it will execute group of statements 2 until a break keyword is found, and then will jump to the end of the switch selective structure.

Finally, if the value of expression did not match any of the previously specified constants, the program will execute the statements included after the default: label, if it exists.

Both of the following code fragments have the same behavior:

switch example	if-else equivalent

switch (x) {
  case 1:
  cout << "x is 1";
  break;
  case 2:
  cout << "x is 2";
  break;
  default:
  cout << "value of x unknown";
  }
	if (x == 1) {
	  cout << "x is 1";
	  }
	else if (x == 2) {
	  cout << "x is 2";
	  }
	else {
	  cout << "value of x unknown";
	  }

The switch statement is a bit peculiar within the C++ language because it uses labels instead of blocks. This forces us to put break statements after the group of statements that we want to be executed for a specific condition. Otherwise the remainder statements -including those corresponding to other labels- will also be executed until the end of the switch selective block or a break statement is reached.

For example, if we did not include a break statement after the first group for case one, the program will not automatically jump to the end of the switch selective block and it would continue executing the rest of statements until it reaches either a break instruction or the end of the switch selective block. This makes unnecessary to include braces { } surrounding the statements for each of the cases, and it can also be useful to execute the same block of instructions for different possible values for the expression being evaluated. For example:

```
switch(x) {
```

```
case 1:
case 2:
case 3:
  cout << "x is 1,2 or 3" ;
  break;
default:
  cout << "x is not 1,2 or 3" ;
}
```

Notice that switch can only be used to compare an expression against constants. Therefore we cannot put variables as labels or ranges because they are not valid C++ constants.

Type Conversions in C++

What is Type Conversion

It is the process of converting one type into another. In other words converting an expression of a given type into another is called type casting.

How to achieve this

There are two ways of achieving the type conversion namely:

Automatic Conversion otherwise called as **Implicit Conversion**
Type casting otherwise called as **Explicit Conversion**

Let us see each of these in detail:

Automatic Conversion otherwise called as Implicit Conversion

This is not done by any conversions or operators. In other words value gets automatically converted to the specific type in which it is assigned.

Let us see this with an example:

```
#include <iostream.h>
void main()
{
short x=6000;
int y;
y=a;
}
```

In the above example the data type short namely variable x is converted to int and is assigned to the integer variable y.

So as above it is possible to convert short to int, int to float and so on.

Type casting otherwise called as Explicit Conversion

Explicit conversion can be done using type cast operator and the general syntax for doing this is

datatype (expression);

Here in the above datatype is the type which the programmer wants the expression to gets changed as

In C++ the type casting can be done in either of the two ways mentioned below namely:

C-style casting
C++-style casting

The C-style casting takes the synatx as

(type) expression

This can also be used in C++.

Apart from the above the other form of type casting that can be used specifically in C++ programming language namely C++-style casting is as below namely:

type (expression)

This approach was adopted since it provided more clarity to the C++ programmers rather than the C-style casting.Say for instance the as per C-style casting

(type) firstVariable * secondVariable

is not clear but when a programmer uses the C++ style casting it is much more clearer as below

type (firstVariable) * secondVariable

Let us see the concept of type casting in C++ with a small example:

```
#include <iostream.h>
void main()
{
```

```
int a;
float b,c;
cout<< "Enter the value of a:";
cin>>a;
cout<< "\n Enter the value of b:";
cin>>b;
c = float(a)+b;
cout<<"\n The value of c is:"<<c;
}
```

The output of the above program is

Enter the value of a: 10
Enter the value of b: 12.5
The value of c is: 22.5

In the above program a is declared as integer and b and c are declared as float. In the type conversion statement namely

c = float(a)+b;

The variable a of type integer is converted into float type and so the value 10 is converted as 10.0 and then is added with the float variable b with value 12.5 giving a resultant float variable c with value as 22.5

Function

A function is a structure that has a number of program statements grouped as a unit with a name given to the unit. Function can be invoked from any part of the C++ program.

Features of Function:

To understand why the program structure is written separately and given a name, the programmer must have a clear idea of the features and benefits of function. This will encourage better understanding of function usage:

- Use of Functions gives a Structured Programming Approach
- Reduces Program Size:

The piece of code that needs to be executed, or the piece of code that is repeated in different parts of the program, can be written separately as a function and stored in a place in memory. Whenever and

wherever needed, the programmer can invoke the function and use the code to be executed. Thus, the program size is reduced.

Having known about the function and its features let us see how to declare, define and call a function in a C++ program.

Declaring a Function:

It has been discussed that, in order for a variable to be used, it must be declared. Just like variables, it follows that function definitions must be declared. The general method of declaring a function is to declare the function in the beginning of the program.

The general format for declaring a function is

return_datatype function name(arguments);

Suppose we have a function named as exforsys which return nothing from the function it is declared as

void exforsys( );

This declared would inform the compiler that the presence of the function exforsys is there in the program. In the above the return type void tells the compiler that the function has no return value and also the empty braces ( ) indicate that the function takes no arguments.

Defining a function:

The definition is the place where the actual function program statements or in other words the program code is placed. The general format of the function definition is as follows:

```
return_datatype functionname(arguments) //Declarator
{
program statements //Function Body
……..
}
```

In the above the declarator must match the function declaration already made. That is the function name, the return type and the number of arguments all must be same as the function declaration made. In fact if arguments are declared and defined. The order of arguments defined here must match the order declared in the function declaration.

After the declarator the braces starts and the function body is placed. The function body has the set of program statements which are to be executed by the function. Then the function definition ends with } ending braces.

For example let us see how to define a function exforsys declared as void that prints first five integers.

```
void exforsys( )
{
int i;
for (i=1;i<=5;i++)
cout<< i;
}
```

The output of the function when it is called would be

12345

Calling the function:

The function must be called for it to get executed. This process is performed by calling the function wherever required.

The general format for making the function call would be as follows:

```
functionname();
```


When the function is called the control, transfers to the function and all the statements present in the function definition gets executed and after which the control, returns back to the statement following the function call.

In the above example when the programmer executes the function exforsys, he can call the function in main as follows:

```
exforsys();
```

Let us see a complete program in C++ to help the programmer to understand the function concepts described above:

```
#include
void main( )
{
void exforsys( ); //Function Declaration
exforsys( ); //Function Called
cout<<"\n End of Program";
}
```

```
void exforsys( ) //Function Definition
{
int i;
for(i=1;i<=5;i++)
cout<< i;
}
```

The output of the above program is

12345

End of Program

Argument

An argument is the value that is passed from the program to the function call. This can also be considered as input to the function from the program from which it is called.

How to declare a function passed with argument

Declaring a function:

The general format for declaring the function remains the same as before except the data type passed as arguments in functions are in the same order in which it is defined in function.

The format for declaring a function with arguments is:

return_datatype functionname(datatype1,datatype2,..);

In this example, the data types are the types of data passed in the function definition as arguments to the function. Care must be taken to mention the number of arguments and the order of arguments in the same way as in function definition.

For example, suppose a function named exforsys takes two integer values as arguments in its functions definition and returns an integer value. The function declaration of exforsys would be written:

int exforsys(int,int);

Function Definition:

The function definition has the same syntax as the function definition previously defined, but with added arguments. The general format for defining a function with arguments is written as:

```
return_datatype functionname(datatype1 variable1,datatype2 variable2,..)
{
```

```
    ……
    Program statements
    ……
    return( variable);
}
```

In the above example, the return data type defines the data type of the value returned by the function. The arguments are passed inside the function name after parentheses with the data type and the variable of each argument. Care must be taken to mention the number of arguments and the order of arguments in the same way as in function declaration.

For example, if the function exforsys takes two integer values x and y and adds them and returns the value z the function definition would be defined as follows:

```
int exforsys(int x,int y)
{
    int z;
    z=x+y;
    return(z);
}
```

In the above program segment, a return statement takes the general format

return(variable) ;

This value specified in the return as argument would be returned to the calling program. In this example, the value returned is z, which is an integer value, the data type returned by the function exforsys is mentioned as int.

Calling the function:

The calling of function takes the same syntax as the name of the function but with value for the arguments passed. The function call is made in the calling program and this is where the value of arguments or the input to the function definition is given.

The general format for calling the function with arguments is

functionname(value1,value2,…);

In the above exforsys function suppose integer value 5 and 6 are passed, the function call would be as follows:

exforsys(5,6);

As soon as the function call exforsys is made the control, transfers to the function definition and the assignment of 5 to x and 6 to y are made as below.

int exforsys(int x,int y)

exforsys(5,6);

It is possible to store the return value of the function in a variable. In the above example, the assignment of the function call to integer variable b can be produced as follows:

int b;
b = exforsys(5,6);

The above statement assigns the returned value of the function exforsys. The value z is then added to the value of x and y to the variable b. So, variable b takes the value 11.

Let us see the whole program to understand in brief the concept of function with arguments

The output of the above program would be

```
#include
int exforsys(int,int);
void main()
{
    int b;
    int s=5,u=6;
    b=exforsys(s,u);
    cout&lt;&lt;"\n The Output is:"&lt;
}

int exforsys(int x,int y)
{
    int z;
    z=x+y;
    return(z);
}
```

The Output is:11

Reference Variable

C++ introduces a new kind of variable known as reference variable.A reference variable provides an alias (alternative name) for a previously defined variable.A reference variable is creaed as follows:

Datatype & reference-name=variable-name

Eg:float total=100;
Float &sum=total;
Then the sumand total can be used interchangeably to represent that variable.

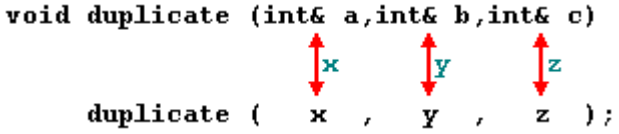
// passing parameters by reference

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

void duplicate (int& a, int& b, int& c)
{
  a*=2;
  b*=2;
  c*=2;
}

int main ()
{
  int x=1, y=3, z=7;
  duplicate (x, y, z);
  cout << "x=" << x << ", y=" << y << ", z=" << z;
  return 0;
}
```

Output: x=2, y=6, z=14

The first thing that should call your attention is that in the declaration of duplicate the type of each parameter was followed by an ampersand sign (&). This ampersand is what specifies that their corresponding arguments are to be passed *by reference* instead of *by value*.

When a variable is passed by reference we are not passing a copy of its value, but we are somehow passing the variable itself to the function and any modification that we do to the local variables will have an effect in their counterpart variables passed as arguments in the call to the function.



To explain it in another way, we associate a, b and c with the arguments passed on the function call (x, y and z) and any change that we do on a within the function will affect the value of x outside it. Any change that we do on b will affect y, and the same with c and z. That is why our program's output, that shows the values stored in x, y and z after the call to duplicate, shows the values of all the three variables of main doubled.

If when declaring the following function: void duplicate (int& a, int& b, int& c)

we had declared it this way: duplicate (int a, int b, int c)

without the ampersand signs (&), we would have not passed the variables by reference, but a copy of their values instead, and therefore, the output on screen of our program would have been the values of x, y and z without having been modified. Passing by reference is also an effective way to allow a

function to return more than one value. For example, here is a function that returns the previous and next numbers of the first parameter passed.

```
// more than one returning value
#include <iostream>
using namespace std;

void prevnext (int x, int& prev, int& next)
{
  prev = x-1;
  next = x+1;
}

int main ()
{
  int x=100, y, z;
  prevnext (x, y, z);
  cout << "Previous=" << y << ", Next=" << z;
  return 0;
}
```

Output: Previous=99, Next=101

Call by value

Call-by-value evaluation is the most common evaluation strategy, used in languages as far-ranging as C and Scheme. In call-by-value, the argument expression is evaluated, and the resulting value is bound to the corresponding variable in the function (usually by capture-avoiding substitution or by copying the value into a new memory region). If the function or procedure is able to assign values to its parameters, only the local copy is assigned -- that is, anything passed into a function call is unchanged in the caller's scope when the function returns.

Call-by-value is not a single evaluation strategy, but rather the family of evaluation strategies in which a function's argument is evaluated before being passed to the function. While many programming languages that use call-by-value evaluate function arguments left-to-right, some (such as O'Caml) evaluate functions and their arguments right-to-left, and others (such as Scheme and C) leave the order unspecified (though they generally guarantee sequential consistency).

Call by reference

In *call-by-reference* evaluation, a function is passed an implicit reference to its argument rather than the argument value itself. If the function is able to modify such a parameter, then any changes it makes will be visible to the caller as well. If the argument expression is an L-value, its address is used. Otherwise, a temporary object is constructed by the caller and a reference to this object is passed; the object is then discarded when the function returns.

Some languages contain a notion of references as first-class values. ML, for example, has the "ref" constructor; references in C++ may also be created explicitly. In these languages, "call-by-reference" may be used to mean passing a reference value as an argument to a function.

In languages (such as C) that contain unrestricted pointers instead of or in addition to references, *call-by-address* is a variant of call-by-reference where the reference is an unrestricted pointer

Default Arguments

When declaring a function we can specify a default value for each parameter. This value will be used if the corresponding argument is left blank when calling to the function. To do that, we simply have to use the assignment operator and a value for the arguments in the function declaration. If a value for that parameter is not passed when the function is called, the default value is used, but if a value is specified this default value is ignored and the passed value is used instead. For example:


```
// default values in functions
#include <iostream>
using namespace std;

int divide (int a, int b=2)
{
  int r;
  r=a/b;
  return (r);
}

int main ()
{
  cout << divide (12);
  cout << endl;
  cout << divide (20,4);
  return 0;
}
```

Output: 6,5

As we can see in the body of the program there are two calls to function divide. In the first one:

divide (12)

we have only specified one argument, but the function divide allows up to two. So the function divide has assumed that the second parameter is 2 since that is what we have specified to happen if this parameter was not passed (notice the function declaration, which finishes with int b=2, not just int b). Therefore the result of this function call is 6 (12/2).

In the second call: divide (20,4)

there are two parameters, so the default value for b (int b=2) is ignored and b takes the value passed as argument, that is 4, making the result returned equal to 5 (20/4).

Operators available in C++

The operators available in C++ programming language are:

- Assignment Operator denoted by =
- Arithmetic operators denoted by +, -, *, /, %
- Compound assignment Operators denoted by +=, -=, *=, /=, %=, >>=, <<=, &=, ^=,|=
- Increment and Decrement operator denoted by ++, --
- Relational and equality operators denoted by ==, !=, >, <, >=, <=
- Logical operators denoted by !, &&, ||
- Conditional operator denoted by ?
- Comma operator denoted by ,
- Bitwise Operators denoted by &, |, ^, ~, <<, >>
- Explicit type casting operator
- sizeof()

Assignment Operator

This is denoted by symbol =. This operator is used for assigning a value to a variable. The left of the assignation operator is known as the lvalue (left value), which must be a variable. The right of the assignation operator is known as the rvalue (right value). The rvalue can be a constant, a variable, the result of an operation or any combination of these.

For example:

x = 5;

By following the right to left rule the value 5 is assigned to the variable x in the above assignment statement.

Arithmetic operators

The operators used for arithmetic operation sin C++ are:

- + For addition

- - For subtraction
- * For multiplication
- / For division
- % For modulo

Compound assignment Operators

This operator is used when a programmer wants to update a current value by performing operation on the current value of the variable.

For example:

Old += new is equal to
Old = old + new

Compound assignment operators function in a similar way the other operators +=, -=, *=, /=, %=, >>=, <<=, &=, ^=, |= function.

Increment and Decrement Operator

The increment operator is denoted by ++ and the decrement operator by --. The function of the increment operator is to increase the value and the decrement operator is to decrease the value. These operators may be used as either prefix or postfix. A Prefix operator is written before the variable as ++a or –a. A Postfix operator is written after the variable as a++ or a--.

The Functionality of Prefix and Postfix Operators

In the case that the increment or decrement operator is used as a prefix (++a or –a), then the value is respectively increased or decreased before the result of the expression is evaluated. Therefore, the increased or decreased value, respectively, is considered in the outer expression. In the case that the increment or decrement operator is used as a postfix (a++ or a--), then the value stored in a is respectively increased or decreased after being evaluated. Therefore, the value stored before the increase or decrease operation is evaluated in the outer expression.

For Example:

```
y=3;
x=++y;          //Prefix : Here Value of x becomes 4

But for the postfix operator namely as below:

y=3                     //Postfix : Here Value of x is 3 and Value of y is  4
x=y++;
```

Relational and Equality Operators

These operators are used for evaluating a comparison between two expressions. The value returned by the relational operation is a Boolean value (true or false value).
The operators used for this purpose in C++ are:

- == Equal to
- != Not equal to
- > Greater than
- < Less than
- >= Greater than or equal to
- <= Less than or equal to

Logical Operators

The logical operators used are : !, &&, ||
The operator ! is called *NOT* operator. This has a single operand which reverses its value.

For example:

!true gives the value of false
!false gives the value of true

The operator && corresponds with Boolean logical operation *AND*. This operator returns the value of true if both its operands are true or if it returns false. The following table reflects the value of && operator:

&& Operator

x	y	x && y
true	true	true
true	false	false
false	true	false
false	false	false

The operator || corresponds with Boolean logical operation *OR*. The operator produces a true value if either one of its two operands are true and produces a false value only when both operands are false. The following table reflects the

value of || operator:

|| Operator

x	y	x || y
true	true	true
true	false	true
false	true	true
false	false	false

Conditional Operator

The conditional operator evaluates an expression returning a value if that expression is true and a different value if the expression is evaluated as false.

The syntax is:

```
condition ? value1 : value2
```

For example:

In

```
7>5 ? x : y
```

Since 7 is greater than 5, true is returned and hence the value x is returned.

Comma Operator

This is denoted by, and it is used to separate two or more expressions.

For example:

```
exfor = (x=5, x+3);
```

Here value of 5 is assigned to x and then the value of x+3 is assigned to the variable exfor. Hence, value of the variable exfor is 8.

Bitwise Operators

The following are the bitwise operators available in C++:

- & AND Bitwise AND
- | OR Bitwise Inclusive OR
- ^ XOR Bitwise Exclusive OR
- ~ NOT Unary complement (bit inversion)
- << SHL Shift Left
- >> SHR Shift Right
- Explicit type casting operator

sizeof() Operator

This operator accepts a single parameter and this can be a variable or data type. This operator returns the size in bytes of the variable or data type.

For example:

```
x = sizeof (char);
```

This returns the size of char in bytes. In this example, the size is 1 byte which is assigned to variable x.

Manipulators

What is a Manipulator?

Manipulators are operators used in C++ for formatting output. The data is manipulated by the programmer's choice of display.

There are numerous manipulators available in C++. Some of the more commonly used manipulators are provided here below:

endl Manipulator:

This manipulator has the same functionality as the '\n' newline character.

For example:

```
cout << "Exforsys" << endl;
cout << "Training";
```

produces the output:


```
Exforsys
Training
```

setw Manipulator:

This manipulator sets the minimum field width on output. The syntax is:

setw(x)

Here setw causes the number or string that follows it to be printed within a field of x characters wide and x is the argument set in setw manipulator. The header file that must be included while using setw manipulator is <iomanip.h>

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



void main( )
{
int x1=12345,x2= 23456, x3=7892;
cout << setw(8) << ”Exforsys” << setw(20) << ”Values” << endl
<< setw(8) << “E1234567” << setw(20)<< x1 << end
<< setw(8) << “S1234567” << setw(20)<< x2 << end
<< setw(8) << “A1234567” << setw(20)<< x3 << end;
}
```

The output of the above example is:

```
setw(8)   setw(20)
Exforsys Values
E1234567 12345
S1234567 23456
A1234567 7892
```


setfill Manipulator:

This is used after setw manipulator. If a value does not entirely fill a field, then the character specified in the setfill argument of the manipulator is used for filling the fields.

```
#include <iostream.h>
#include <iomanip.h>
void main( )
{
cout << setw(10) << setfill('$') << 50 << 33 << endl;
}
```

The output of the above program is

$$$$$$$$5033

This is because the setw sets 10 width for the field and the number 50 has only 2 positions in it. So the remaining 8 positions are filled with $ symbol which is specified in the setfill argument.

setprecision Manipulator:

The setprecision Manipulator is used with floating point numbers. It is used to set the number of digits printed to the right of the decimal point. This may be used in two forms:

- fixed
- scientific

These two forms are used when the keywords fixed or scientific are appropriately used before the setprecision manipulator. The keyword fixed before the setprecision manipulator prints the floating point number in fixed notation. The keyword scientific before the setprecision manipulator prints the floating point number in scientific notation.

```
#include <iostream.h>
#include <iomanip.h>
void main( )
{
float x = 0.1;
cout << fixed << setprecision(3) << x << endl;
cout << sceintific << x << endl;
}
```


The above gives ouput as:

```
0.100
1.000000e-001
```

The first cout statement contains fixed notation and the setprecision contains argument 3. This means that three digits after the decimal point and in fixed notation will output the first cout statement as 0.100. The second cout produces the output in scientific notation. The default value is used since no setprecision value is provided.

Pointers

A pointer is a variable that holds a memory address. It is called a pointer because it points to the value at the address that it stores.

Using pointers

If you want to declare a pointer variable you must first choose what data type it will point to such as an int or a char. You then declare it as if you were declaring a variable in the normal way and then

put a * in front of its name to show that it is a pointer. Here is an example of how to declare a pointer to an integer.

```
int *pi;
```

You can store the address of another variable in a pointer using the & operator. Here is an example of how to store the address of variable called i in the pointer called pi.

```
int i;
int *pi;
pi = &i;
```

You must dereference a pointer to get the value at the memory location that the pointer points to. You use the * operator to dereference a pointer. Here is an example of how we first set the value of i to 5 and then set its value to 7 by dereferencing the pointer.

```
int i = 5;
int *pi;
pi = &i;
*pi = 7;
cout << i;
```

New And Delete

The new operator is used to allocate memory that is the size of a certain data type. It returns a pointer to the address of the newly allocated memory. Here is an example of how to allocate memory for an integer and then set its value to 5.

```
int *pi;
pi = new int;
*pi = 5;
```

The delete operator deallocates memory. You need to deallocate the memory for all the memory that you have previously allocated before exiting the program or else you will have memory leaks.

```
int *pi;
pi = new int;
*pi = 5;
delete pi;
```

Typed and untyped pointers

We have been using typed pointers so far because they point to a specific data type. An untyped pointer can point to anything. You declare an untyped pointer as the data type void.

```
void *up;
```

Malloc And Free

The malloc command allocates a certain number of bytes and returns a pointer to the first byte. You must use the free command to deallocate the memory that was allocated with malloc. To be able to use malloc and free you must include the malloc header file. Here is an example that allocates 100 bytes of memory and stores the address of it in the pointer called up and then deallocates the memory.

```
#include<malloc>
...
void *up;
up = malloc(100);
free(up);
```

Arrays

Concepts of Arrays in C++

In this C++ Tutorial you will learn about Concepts of Arrays in C++, What is an array?, How to access an array element, Declaration of Array and How to Access Array Elements.

What is an array?

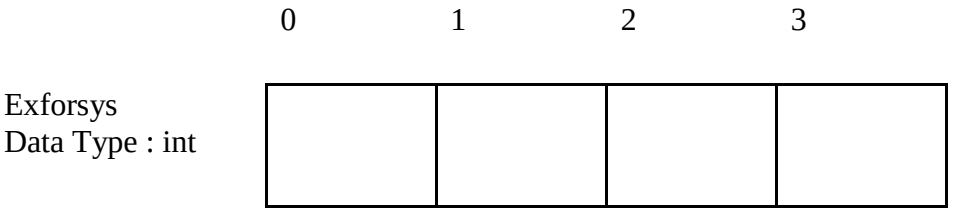
An array is a group of elements of the same type that are placed in contiguous memory locations.

How to access an array element?

You can access an element of an array by adding an index to a unique identifier.

For example

Suppose Exforsys is an array that has 4 integer values in it that is of int data type, then Exforsys is internally represented as:



Declaration of Array:

Just like variables, We have to declare arrays before using them. The general syntax for declaring an array is

data type array name[number of elements];

In the above example the declaration of array Exforsys would be written as:

int Exforsys[4];

One of the key points when declaring arrays is the number of elements defined in the array argument must be a constant value. The size of the array must be known before its execution. This relates to the concept of dynamic memory allocation in C++ which will be covered later in the tutorial.

After declaration the next step is the initialization of array values.

Initialization of array:

If a programmer wants to initialize an array elements it can be done by specifying the values enclosed within braces { }.

For example, using the array Exforsys, if a programmer wants to initialize integer values 10, 20, 30, 40 respectively to each of the array positions it can be written.

int Exforsys[4] = {10,20,30,40};

So the above array would take values as below

	0	1	2	3
Exforsys Data Type : int	10	20	30	40

One interesting fact is that the size of the array element can also be left blank, in which case the array takes the size of the array as the number of elements initialized within { }.

For example if the array takes the form as

int Exforsys = {10,20,30,40,50};

Then the size of the array Exforsys is 5 which is the number of elements initialized within { }.

Now the next step is to know how to access the array elements.

How to Access Array Elements:

Just like variable sit is possible to access any element of the array for reading or modifying.

The general format for accessing arrays:

array name[index]

For example in the above example of array initialized with

int Exforsys[4] = {10,20,30,40};

When a programmer wants to access the 30 this can be written:

Exforsys[2]

Note that the array starts with index 0 and hence to access third element namely 30 one has to write array name with index 2.

example:

```
#include &lt;iostream.h&gt;
int Exforsys[ ] = { 10,20,30,40,50};
int i, outp=0;
void main()
{
  for(i=0;i&lt;5;i++)
  {
     outp=outp+ Exfosys[i];
  }
  cout&lt;&lt; outp;
}
```


The output of the above program is:
150

What is a Multidimensional Array?

A Multidimensional array is an array of arrays.

How are Multidimensional Arrays represented in C++

Suppose a programmer wants to represent the two-dimensional array Exforsys as an array with three rows and four columns all having integer elements. This would be represented in C++ as:

int Exforsys[3][4];

It is represented internally as:

Exforsys
Data Type: int

	0	1	2	3
0				
1				
2				

How to access the elements in the Multidimensional Array

Exforsys
Data Type: int

	0	1	2	3
0				
1				
2				

Highlighted cell represent Exforsys[1][2]

Based on the above two-dimensional arrays, it is possible to handle multidimensional arrays of any number of rows and columns in C++ programming language. This is all occupied in memory. Better utilization of memory must also be made.

Multidimensional Array Example:

```
#include <iostream.h>
const int ROW=4;
const int COLUMN =3;
void main()
{
  int i,j;
  int Exforsys[ROW][COLUMN];
  for(i=0;i<ROWS;i++)
```

```
  for(j=0;j<COLUMN;J++)
  {
  cout << "Enter value of Row "<<i+1;
  cout<<",Column "<<j+1<<":";
  cin>>Exforsys[i][j];
  }
  cout<<"\n\n\n";
  cout<< " COLUMN\n";
  cout<< " 1 2 3";
  for(i=0;i<ROW;i++)
  {
  cout<<"\nROW "<<i+1;
  for(j=0;j<COLUMN;J++)
  cout<<Exforsys[i][j];
}
```

The output of the above program is

```
Enter value of Row 1, Column 1:10
Enter value of Row 1, Column 2:20
Enter value of Row 1, Column 3:30
Enter value of Row 2, Column 1:40
Enter value of Row 2, Column 2:50
Enter value of Row 2, Column 3:60
Enter value of Row 3, Column 1:70
Enter value of Row 3, Column 2:80
Enter value of Row 3, Column 3:90
Enter value of Row 4, Column 1:100
Enter value of Row 4, Column 2:110
Enter value of Row 4, Column 3:120
```



```
        COLUMN
       1   2   3
ROW 1  10  20  30
ROW 2  40  50  60
ROW 3  70  80  90
ROW 4 100 110 120
```

In the above example, the keyword const (specifying constant) precedes the data type that specifies the variable ROW and COLUMN to remain unchanged in value throughout the program. This is used for defining the array Exforsys ROW size and COLUMN size, respectively

Structures

What is a Structure?

Structure is a collection of variables under a single name. Variables can be of any type: int, float, char etc. The main difference between structure and array is that arrays are collections of the same data type and structure is a collection of variables under a single name.

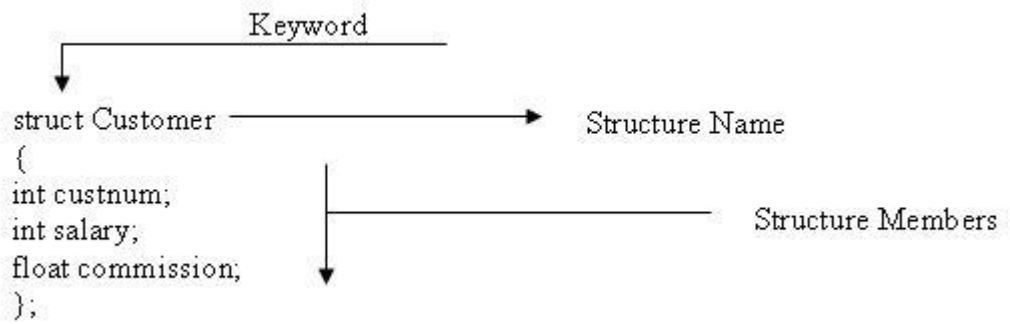
How to declare and create a Structure

Declaring a Structure:

The structure is declared by using the keyword struct followed by structure name, also called a tag. Then the structure members (variables) are defined with their type and variable names inside the open and close braces { and }. Finally, the closed braces end with a semicolon denoted as ; following the statement. The above structure declaration is also called a Structure Specifier.

Example:

Three variables: *custnum* of type int, *salary* of type int, *commission* of type float are structure members and the structure name is Customer. This structure is declared as follows:



In the above example, it is seen that variables of different types such as int and float are grouped in a single structure name Customer.

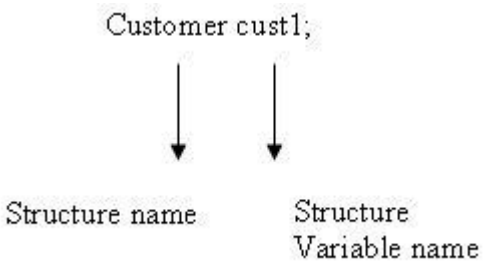
Arrays behave in the same way, declaring structures does not mean that memory is allocated. Structure declaration gives a skeleton or template for the structure.

After declaring the structure, the next step is to define a structure variable.

How to declare Structure Variable?

This is similar to variable declaration. For variable declaration, data type is defined followed by variable name. For structure variable declaration, the data type is the name of the structure followed by the structure variable name.

In the above example, structure variable cust1 is defined as:



What happens when this is defined? When structure is defined, it allocates or reserves space in memory. The memory space allocated will be cumulative of all defined structure members. In the above example, there are 3 structure members: custnum, salary and commission. Of these, two are of type in and one is of type float. If integer space allocated by a system is 2 bytes and float four bytes the above would allo9acter 2bytes for custnum, 2 bytes for salary and 4 bytes for commission.

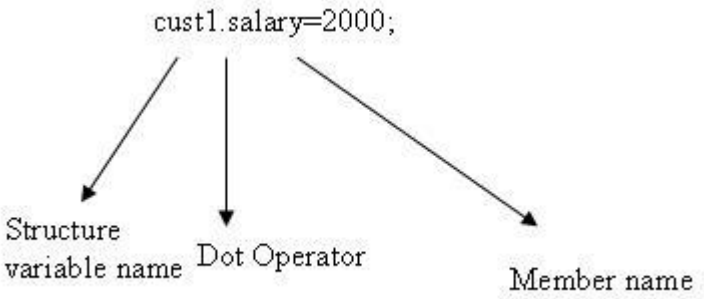
How to access structure members in C++?

To access structure members, the operator used is the dot operator denoted by (.). The dot operator for accessing structure members is used thusly:

structure variable name.member name

For example:

A programmer wants to assign 2000 for the structure member *salary* in the above example of structure *Customer* with structure variable *cust1* this is written as:



C++ String Representation and Handling

How is string represented?

Strings, or sequences of characters, are represented as arrays of char elements. For example, when a programmer wants to represent a char array Exforsys having 30 elements it is declared:

char Exforsys[30];

This means the character array has 30 storage location allocated where it can store up to a maximum of 30 characters.

End String Notation

The character array can store the specified maximum of array limit. This means it can store less than the allocated array limit up to the maximum. It is essential that a notation exist to mark the end of character array or string representation. This end of character array is denoted using backslash zero '\0'.

The Differences Between Character Constants and String Literal:

Initializing Char array

Suppose the programmer wants to initialize the array char Exforsys[30] with value Training. This initialization of char array can be performed using two methods:

- Character representation
- String Literals

The character representation is notated as follows:

char Exforsys[30]= {'T','r','a','i','n','i','n','g,'\0'};

The string literal representation is notated as follows:

char Exforsys[30]= "Training";

Character representation uses single quotes to represent each character while string literal representation uses double quotes for the entire string literal.

Another main difference with character representation is the end of character denoted by '\0'. String literal representation uses double quotes, there is no need for representing the end of character. The end of character task is automatically performed by C++ programming language.

It is also possible to initialize array as follows:

char Exforsys[ ]=”Company”;

In the above case, the char array Exforsys is not mentioned with the size and is initialized with the string literal Company. The character array Exforsys takes the size of the initialized string literal.

An Example: Input string defined as character array with the same output on the display screen:

```
#include <iostream.h>
void main( )
{
char Exforsys[80];
cout <<”Input the String: ”;
cin >> Exforsys;
cout <<“The string entered is: ”<< Exforsys
}
```

The output of the above program is

Input the String: Sample
The string entered is: Sample

Detailed example to understand the concept of character array and accessing of character array:

The below example uses two input strings as two character arrays, accessing one character array character by character, then copies it into the other character array.

```
#include <iostream.h>
#include <string.h>
void main( )
{
char examp1[] = “Exforsys”;
char examp2[80];
for(int i=0; i<strlen(exampl1);i++)
examp2[i]=examp1[i];
examp2[i]=’\0’;
cout<< “The copied string is: “<< examp2;
}
```

The output of the above program is

The copied string is: Exforsys

In the above program, the string function *strlen* is used. This string function is used to find the length of the string given in the argument of the function. In this case, it finds the length of the string *Exforsys*. This predefined special string function *strlen* is present in the header file *string.h*. Therefore, this is also included in the program.

In this example, the first string initializes with string literal Exforsys and the second literal declares as an empty string. The for loop uses the first character from the first character array *examp1[]* and copies it into the first position in the second character array *examp2[]*. The loop continues until the end of string value is reached. The second character array is terminated by end of character and then printed.

The above example provides the programmer with insight on how to declare character arrays, how to initialize character arrays and how to access character arrays.

The above example can also be performed using a predefined string function called *strcpy*.

The string function *strcpy* takes two strings as argument.

The general syntax is as follows:

strcpy(string1,string2);

The string function *strcpy* copies the *string2* (the second argument) into *string1* (the first argument).

How to represent array of Strings:

It is possible to declare two-dimensional arrays in character arrays or array of strings.

The general syntax is as follows:

char arrayname[numb1][numb2];

In this numb1 denotes the number of string literals defined in the array and numb2 defines the maximum length of characters of the strings.

For example, if the programmer wants to represent a char array Exforsys with string literals as:

Training
Forums
Program

Then numb1 takes the value 3 and numb2 takes the value 8 because Training has the maximum character 8. The representation is as follows:

char Exforsys[3][8];

The access of the character array can be performed by using for loop.

EVOLUTION OF OBJECT ORIENTED LANGUAGES

Simula

Simula is a name for two programming languages, Simula I and Simula 67, developed in the 1960s at the Norwegian Computing Center in Oslo, by Ole-Johan Dahl and Kristen Nygaard. Syntactically, it is a fairly faithful superset of Algol 60.

Simula 67 introduced objects, classes, subclasses, virtual methods, coroutines, discrete event simulation, and features garbage collection.

Simula is considered the first object-oriented programming language. As its name implies, Simula was designed for doing simulations, and the needs of that domain provided the framework for many of the features of object-oriented languages today.

Simula has been used in a wide range of applications such as simulating VLSI designs, processes, protocols, algorithms, and other applications such as typesetting, computer graphics, and education. Since Simula-type objects are reimplemented in C++, Java and C# the influence of Simula is often understated. The creator of C++, Bjarne Stroustrup, has acknowledged that Simula 67 was the greatest influence on him to develop C++, to bring the kind of productivity enhancements offered by Simula to the raw computational speed offered by lower level languages like BCPL.

History

Kristen Nygaard started writing computer simulation programs in 1957. Nygaard saw a need for a better way of describing the heterogeneity and the operation of a system. To go further with his ideas on a formal computer language for describing a system, Nygaard realized that he needed someone with more programming skills than he had. Ole-Johan Dahl joined him on his work January 1962. The decision of linking the language up to Algol 60 was made shortly after. By May 1962 the main concepts for a simulation language were set. "SIMULA I" was born, a special purpose programming language for simulating discrete event systems.

Kristen Nygaard was invited to Univac late May 1962 in connection with the marketing of their new UNIVAC 1107 computer. At that visit Nygaard presented the ideas of Simula to Robert Bemer, the director of systems programming at Univac. Bemer was a sworn ALGOL fan and found the Simula project compelling. Bemer was also chairing a session at the second international conference on information processing hosted by IFIP. He invited Nygaard, who presented the paper "SIMULA -- An Extension of ALGOL to the Description of Discrete-Event Networks".

Norwegian Computing Center got a UNIVAC 1107 August 1963 at a considerable discount, on which Dahl implemented the SIMULA I under contract with Univac. The implementation was based on the UNIVAC Algol 60 compiler. SIMULA I was fully operational on UNIVAC 1107 January 1965. In the following couple of years Dahl and Nygaard spent a lot of time teaching Simula. Simula spread to several countries around the world and SIMULA I was later implemented on Burroughs B5500 computers and the Russian URAL-16 computer.

In 1966 C. A. R. Hoare introduced the concept of record class construct, which Dahl and Nygaard extended with the concept of prefixing and other features to meet their requirements for a generalized process concept. Dahl and Nygaard presented their paper on Class and Subclass Declarations at the IFIP Working Conference on simulation languages in Oslo, May 1967. This paper became the first formal definition of Simula 67. In June 1967 a conference was held to standardize the language and initiate a number of implementations. Dahl proposed to unify the Type and the Class concept. This led to serious discussions, and the proposal was rejected by the board. SIMULA 67 was formally standardized on the first meeting of the SIMULA Standards Group (SSG) in February 1968.

Simula was influential in the development of Smalltalk and later object-oriented programming languages. It also helped inspire the Actor model of concurrent computation although Simula only supports co-routines and not true concurrency.

In the late sixties and the early seventies there were four main implementations of Simula:

- UNIVAC 1100 by NCC
- System/360 and System/370 by Swedish Research Institute for National Defence (FOA)
- CDC 3000 by University of Oslo's Joint Computer Installation at Kjeller
- TOPS-10 by ENEA AB

These implementations were ported to a wide range of platforms. The TOPS-10 implemented the concept of public, protected, and private member variables and methods, that later was integrated into Simula 87. Simula 87 is the latest standard and is ported to a wide range of platforms.

There are mainly three implementations:

- Simula AS
- Lund Simula
- GNU Cim - Download available from the GNU ftp site

In November 2001 Dahl and Nygaard were awarded the IEEE John von Neumann Medal by the Institute of Electrical and Electronic Engineers "For the introduction of the concepts underlying object-oriented programming through the design and implementation of SIMULA 67". In February 2002 they received the 2001 A. M. Turing Award by the Association for Computing Machinery (ACM), with the citation: "For ideas fundamental to the emergence of object oriented programming, through their design of the programming languages Simula I and Simula 67." Unfortunately neither

Dahl, nor Nygaard could make it to the ACM Turing Award Lecture, scheduled to be delivered at the OOPSLA 2002 conference in Seattle, as they both passed away within two months of each other in June and August, respectively.

Simula Research Laboratory is a research institute named after the Simula language, and Nygaard held a part time position there from the opening in 2001.

Sample Code

The empty computer file is the minimal program in Simula, measured by the size of the source code. It consists of one thing only; a dummy statement.However, the minimal program is more conveniently represented as an empty block:

```
Begin
End;
```

It begins executing and immediately terminates. The language does not have any return value from the program itself.

Classic Hello world

Note that Simula is case-insensitive. An example of a Hello world program in Simula:


```
Begin
  OutText ("Hello World!");
  Outimage;
End;
```

Smalltalk

Smalltalk is an object-oriented, dynamically typed, reflective programming language. Smalltalk was created as the language to underpin the "new world" of computing exemplified by "human-computer symbiosis."[1] It was designed and created in part for educational use, more so for constructionist learning, at Xerox PARC by Alan Kay, Dan Ingalls, Adele Goldberg, Ted Kaehler, Scott Wallace, and others during the 1970s, influenced by Lisp, Logo, Sketchpad and Simula.

The language was first generally released as Smalltalk-80 and has been widely used since. Smalltalk-like languages are in continuing active development, and have gathered loyal communities of users around them. ANSI Smalltalk was ratified in 1998 and represents the standard version of Smalltalk.

Smalltalk has influenced the wider world of computer programming in four main areas. It inspired the syntax and semantics of other computer programming languages. Secondly, it was a prototype for a model of computation known as message passing. Thirdly, its WIMP GUI inspired the windowing environments of personal computers in the late twentieth and early twenty-first centuries,

so much so that the windows of the first Macintosh desktop look almost identical to the MVC windows of Smalltalk-80. Finally, the integrated development environment was the model for a generation of visual programming tools that look like Smalltalk's code browsers and debuggers.

Smalltalk laid the groundwork and proved all the principles that led to the development and commercial success of Java.

Python and Ruby have reimplemented some Smalltalk ideas with more C/Java-like syntax. The Smalltalk "metamodel" also serves as the inspiration for the object model design for Perl 6.

There is also a modular Smalltalk-like implementation designed for scripting called S# (S-Sharp). S# uses just-in-time compilation technology and supports an extended Smalltalk-like language written by David Simmons of Smallscript Corp.[6][7]

Object-oriented programming

As in other object-oriented languages, the central concept in Smalltalk-80 (but not in Smalltalk-72) is that of an *object*. An object is always an *instance* of a *class*. Classes are "blueprints" that describe the properties and behavior of their instances. For example, a Window class would declare that windows have properties such as the label, the position and whether the window is visible or not. The class would also declare that instances support operations such as opening, closing, moving and hiding. Each particular Window object would have its own values of those properties, and each of them would be able to perform operations defined by its class.

A Smalltalk object can do exactly three things:

1. Hold state (references to other objects).
2. Receive a message from itself or another object.
3. In the course of processing a message, send messages to itself or another object.

The state an object holds is always private to that object. Other objects can query or change that state only by sending requests (messages) to the object to do so. Any message can be sent to any object: when a message is received, the receiver determines whether that message is appropriate. (Alan Kay has commented that despite the attention given to objects, messaging is the most important concept in Smalltalk.)

Smalltalk is a 'pure' OO language, meaning that, unlike Java and C++, there is no difference between values which are objects and values which are primitive types. In Smalltalk, primitive values such as integers, booleans and characters are also objects, in the sense that they are instances of corresponding classes, and operations on them are invoked by sending messages. A programmer can change the classes that implement primitive values, so that new behavior can be defined for their instances--for example, to implement new control structures--or even so that their existing behavior will be changed. This fact is summarised in the commonly heard phrase "In Smalltalk everything is an object" (which would more accurately be expressed as "all values are objects", as variables aren't).

Since all values are objects, classes themselves are also objects. Each class is an instance of the *metaclass* of that class. Metaclasses in turn are also objects, and are all instances of a class called Metaclass. Code blocks are also objects.

Syntax

Smalltalk-80 syntax is rather minimalist, based on only a handful of declarations and reserved words. In fact, only six keywords are reserved in Smalltalk: **true**, **false**, **nil**, **self**, **super** and **thisContext**. (These are not actually keywords, but *pseudo-variables*. The true, false, and nil pseudo-variables are singleton instances. Smalltalk does not really define any keywords.) The only built-in language constructs are message sends, assignment, method return and literal syntax for some objects. From its origins as a teaching language, standard Smalltalk syntax uses punctuation in a manner more like English than mainstream coding languages. The remainder of the language, including control structures for conditional evaluation and iteration, is implemented on top of the built-in constructs by the standard Smalltalk class library. (For performance reasons implementations may recognize and treat as special some of those messages; however, this is only an optimization, not hardwired into the language syntax).

Need Of Objects:

- Modeling the real world problem as close as possible to the users perspective.
- Interacting easily with computational environment using familiar metaphors
- Constructing reusable software components and easily extendable libraries.
- Easily modifying and extending implementations of components without having to recode every thing from scratch.

Definition of OOP:

OOP uses objects as its fundamental building blocks. Each object is an instance of some class. Classes allow the mechanism of data abstraction for creating new data types. Inheritance allows building of new classes from existing classes. Hence if any of these elements are missing in a program we cannot consider that program as objected oriented program.

Object oriented programming is a programming methodology that associates data structures with a set of operators which act upon it. In OOP's terminology an instance of such an entity is known as an object. It gives importance to relationships between objects rather than implementation details. Hiding the implementation details within an object results in the user being more concerned with an objects relationship to the rest of the system, than the implementation of the object's behavior.

Object oriented programming is an approach that provides a way f modularizing programs by creating partitioned memory area for both data and functions that can be used as template for creating copies of such modules on demand.

Definition of Object Oriented Languages

A language that is specially designed to support the OOP concepts makes it easier to implement them. Such languages are known as object oriented Languages. Depending upon the features they support, they can be classified into the following categories.

1. **Object-based Programming Languages**
   It is the style of programming that primarily supports encapsulation and object identity.
   Major features that are required for object-based programming are
   - Data encapsulation
   - Data hiding and access mechanisms
   - Automatic initialization and clear-up of objects
   - Operator Overloading
     Eg: Ada
2. **Object –oriented programming**
   Object –oriented programming incorporates all of object-based programming features along with two additional features, namely inheritance and dynamic binding.
   Object-based features + inheritance + dynamic biding
   Eg: C++, Smalltalk, Java

C++ Objects and Classes

An Overview about Objects and Classes

In object-oriented programming language C++, the data and functions (procedures to manipulate the data) are bundled together as a self-contained unit called an *object*. A *class* is an extended concept similar to that of *structure* in C programming language, this class describes the data properties alone. In C++ programming language, *class* describes both the properties (data) and behaviors (functions) of objects. *Classes* are not *objects*, but they are used to instantiate *objects*.

Features of Class:

Classes contain data known as members and member functions. As a unit, the collection of members and member functions is an object. Therefore, this unit of objects make up a class.

How to write a Class:

In Structure in C programming language, a structure is specified with a name. The C++ programming language extends this concept. A class is specified with a name after the keyword class.

The starting flower brace symbol, {is placed at the beginning of the code. Following the flower brace symbol, the body of the class is defined with the member functions data. Then the class is closed with a flower brace symbol} and concluded with a semicolon;.

```
class exforsys
{
  data;
  member functions;
  ……
};
```

There are different access specifiers for defining the data and functions present inside a class.

Declaration and syntax

In C++, a class is declared using the **class** keyword. The syntax of a classdeclaration is similar to that of a structure. Its general form is,

```
class class-name
{
   // private functions and variables
public:
   // public functions and variables
} object-list;
```


In a class declaration the *object-list* is optional.The *class-name* is technically optional. From a practical point of view it is virtually always needed. The reason is that the *class-name* becomes a new type name that is used to declare objects of the class.Functions and variables declared inside the class declaration are said to be*members* of the class.By default, all member functions and variables are *private* to that class. This means that they are accessible by other members of that class. To declare *public* class members, the **public** keyword is used, followed by a colon. All functions and variables declared after the **public** specifier are accessible both by other members of the class and by any part of the program that contains the class.

Access specifiers:

Access specifiers are used to identify access rights for the data and member functions of the class. There are three main types of access specifiers in C++ programming language:

- private
- public
- protected

- A *private* member within a class denotes that only members of the same class have accessibility. The *private* member is inaccessible from outside the class.
- *Public* members are accessible from outside the class.
- A protected access specifier is a stage between *private* and *public* access. If member functions defined in a class are *protected*, they cannot be accessed from outside the class but can be accessed from the derived class.

When defining access specifiers, the programmer must use the keywords: *private*, *public* or *protected* when needed, followed by a semicolon and then define the data and member functions under it.

```
class exforsys
{
  private:
  int x,y;
  public:
  void sum()
  {
    ……
    ……
  }
};
```


In the code above, the member *x* and *y* are defined as private access specifiers. The member function sum is defined as a public access specifier.

General Template of a class:

General structure for defining a class is:

```
class classname
{
  acess specifier:
  data member;
  member functions;

  acess specifier:
  data member;
  member functions;
};
```

Generally, in class, all members (data) would be declared as private and the member functions would be declared as public. Private is the default access level for specifiers. If no access specifiers are identified for members of a class, the members are defaulted to private access.

```
class exforsys
{
  int x,y;
  public:
  void sum()
  {
    ……
    ……
  }
};
```

In this example, for members x and y of the class *exforsys* there are no access specifiers identified. *exforsys* would have the default access specifier as private.

Creating and Using Classes and Objects

Creation of Objects:

Once the class is created, one or more objects can be created from the class as objects are instance of the class.

Juts as we declare a variable of data type *int* as:

int x;

Objects are also declared as:

class name followed by object name;

exforsys e1;

This declares *e1* to be an object of class exforsys.

For example a complete class and object declaration is given below:

```
class exforsys
{
  private:
```

```
  int x,y;
  public:
  void sum()
  {
     ……
     ……
  }
};



main()
{
  exforsys e1;
  ……
  ……
}
```

The object can also be declared immediately after the class definition. In other words the object name can also be placed immediately before the closing flower brace symbol } of the class declaration.

For example

```
class exforsys
{
  private:
  int x,y;
  public:
  void sum()
  {
     ……
     ……
  }
}e1 ;
```

The above code also declares an object *e1* of class exforsys.

It is important to understand that in object-oriented programming language, when a class is created no memory is allocated. It is only when an object is created is memory then allocated.

Member Functions and Variables

The variables declared inside the class are known as data members and the functions are known as member functions. Only the member functions can have access to the private data members and member functions.

Eg:#include < iostream >
using namespace std;
// *class declaration*
class myclass {
// *private members to myclass*
int a;
public:
// *public members to myclass*
void set_a(int num);
// *member functions definition inside the class*

int get_a( )
{
return a;
}
};

This class has one private variable,called a,and two public functions set_a( )and get_a( ). Notice that the functions are declared within a class using their prototype forms. The functions that are declared to be part of a class are called *member functions*. Since a is private it is not accessible by any code outside myclass. However since set_a( ) and get_a( ) are member of myclass, they have access to a and as they are declared as public member of myclass, they can be called by any part of the program that contains myclass.The member functions need to be defined. It can be done by preceding the function name with the class name followed by two colons (:: *are called scope resolution operator*). For example, after the class declaration, you can declare the member function as

// *member functions definition outside the class*
void myclass::set_a(int num) {
a=num;
}

In general to declare a member function, you use this form:
return-type class-name::*func-name*(*parameter- list*)
{
// *body of function*
}
Here the *class-name* is the name of the class to which the function belongs. The declaration of a class does not define any objects of the type myclass**.** It only defines the type of object that will be

created when one is actually declared. To create an object, use the class name as type specifier. For example,

```
// from previous examples
void main( ) {
myclass ob1, ob2;//these are object of type myclass
// ... program code
}
```

An object declaration creates a physical entity of that type. That is, an object occupies memory space, but a type definition does not. Once an object of a class has been created, your program can reference its public members by using the dot operator in much the same way that structure members are accessed. Assuming the preceding object declaration, here some examples,

```
ob1.set_a(10); // set ob1's version of a to 10
ob2.set_a(99); // set ob2's version of a to 99
cout << ob1.get_a( ); << "\n";
cout << ob2.get_a( ); << "\n";
ob1.a=20; // error cannot access private member
ob2.a=80; // by non-member functions.
```

There can be public variables, for example


```
#include < iostream >
using namespace std;
// class declaration
class myclass {
public:
int a; //a is now public
// and there is no need for set_a( ), get_a( )
};
int main( ) {
myclass ob1, ob2;
// here a is accessed directly
ob1.a = 10;
ob2.a = 99;
cout << ob1.a << "\n";
cout << ob1.a << "\n";
return 0;
}
```

It is important to remember that although all objects of a class share their functions, each object creates and maintains *its own data*.

How to Access C++ Class Members

It is possible to access the class members after a class is defined and objects are created.

General syntax to access class member:

Object_name.function_name (arguments);

The dot ('.') used above is called the *dot operator* or *class member access operator*. The dot operator is used to connect the object and the member function. This concept is similar to that of accessing structure members in C programming language. The private data of a class can be accessed only through the member function of that class.

For example,

A class and object are defined below:

```
class exforsys
{
  int a, b;
  public:
  void sum(int,int);
} e1;
```

Then the member access is written as:


```
e1.sum(5,6);
```

Where e1 is the object of class exforsys and sum() is the member function of the class.

The programmer now understands declaration of a *class*, creation of an object and accessibility of members of a *class*.

It is also possible to declare more than one object within a *class*:

```
class exforsys
{
  private:
  int a;
  public:
  void sum(int)
  {
    ……
    ……
  }
};

main()
```

```
{
  exforsys e1,e2;
  ……
  ……
}
```

In these two objects e1 and e2 are declared of class exforsys.

By default, the access specifier for members of a class is private. The default access specifier for a structure is public. This is an important difference to recognize and understand in object-oriented C++ programming language.

```
class exforsys
{
    int x;          //Here access specifier is private by default
};
```

whereas

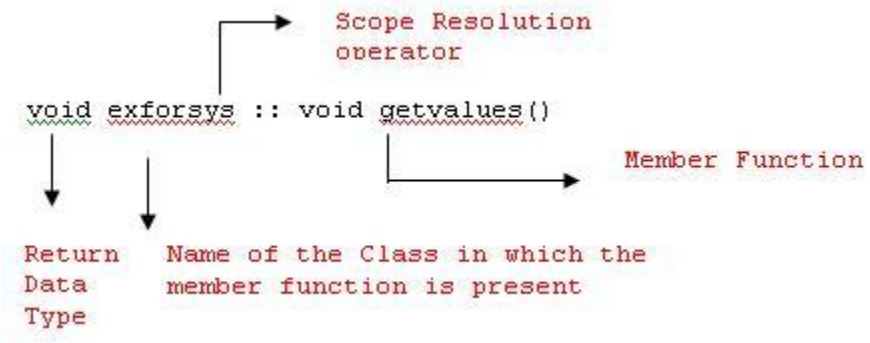
```
struct exforsys
{
  int x;   //Here access specifier is public by default
};
```

It is not always the case that the member function declaration and definition takes place within the class itself. Sometimes, declaration of member function alone can occur within the class. The programmer may choose to place the definition outside the class. In a situation such as this, it is important to understand the identifying member function of a particular class. This is performed by using the operator :: this is called *scope resolution operator*.

```
class exforsys
{
  private:
  int a;
  public:
  void getvalues()              // Only Member Function declaration is done
};



void exforsys :: void getvalues() // Here Member Function is defined
{
  ……
  ……
```

```
}



main()
{
  exforsys e1,e2;
  ……….
}
```

So the usage of scope resolution operator is



// Program to illustrate Nested classes


```
#include<iostream.h>
class first
{
    private:
        int a;
        int b;
    public:
    void assign(int p,int q)
    {
        a=p;
        b=q;
    }

    int multiply()
    {
        return a*b;
    }
};



class second
{
    private:
        int a ;
        int b;
        first fff;
    public:
        void assign1(int p,int q, int r,int s)
```

```
        {
                a=p;
                b=q;
                fff.assign(r,s);
        }

        int multiply()
        {
            return a*b*fff.multiply();
        }
};


void main()
{

  second obj;
  obj.assign1(3,2,5,6);
  int z=obj.multiply();
  cout<<"\n The result is :"<<z;
}
```

Test data

Output
The result is : 180

Class Constructors and destructors in C++

Constructors:

What is the use of Constructor

The main use of constructors is to initialize objects. The function of initialization is automatically carried out by the use of a special member function called a constructor.

General Syntax of Constructor

Constructor is a special member function that takes the same name as the class name. The syntax generally is as given below:

<class name> { arguments};

The default constructor for a class X has the form

X::X()

In the above example the arguments is optional.

The constructor is automatically named when an object is created. A constructor is named whenever an object is defined or dynamically allocated using the "new" operator.

A constructor is a member function that is executed automatically whenever an object is created. There are two restrictions regarding it. First, the name of the constructor has to be name of the class. Second, it cannot have any return type, not even void. When such a constructor is present, it is automatically invoked when an object is declared.

For example,

```
class employee
{
        public :
                employee ( ) ;
                employee ( long ) ;
                employee ( long,  char * ) ;
} ;
```



```
employee  A ;   // calls  first  constructor

employee  B ( 15362 ) ;   // calls  second  constructor

employee  B ( 15362,  "Naik" ) ;   // calls  third  constructor
```

There are three constructors in the class employee. The first, which takes no arguments, is used to create objects which are not initialized. The second, which takes one argument, is used to create objects and initialize them. The third constructor, which takes two arguments, is also used to create objects and initialize them to specific values.

C++ compiler has an implicit constructor (default constructor) which creates objects, even though it was not defined in the class.

```
employee ( ) { }
```

It contains the empty body and does not do anything.

This works fine as long as we do not use any other constructors in the class. However, once we define a constructor, we must also define the default constructor. This constructor will not do anything and is defined just to satisfy the compiler.

Constructors with Default Arguments

It is possible to define constructors with default arguments. It means that if the constructor is defined with n parameters, we can invoke it with less than n parameters specified in the call.

For example,

Date (int d, int m, y = 2007) ;

This constructor has three parameters. Parameter number three is initialized with a value 2007. Now we can call the constructor in two possible ways.

Date (15, 3, 2006) ;
Date (15, 3) ;

There are several forms in which a constructor can take its shape namely:

Default Constructor:

This constructor has no arguments in it. Default Constructor is also called as *no argument constructor*.

For example:

```
class Exforsys
{
   private:
      int a,b;
   public:
      Exforsys();
      ...
};
Exforsys :: Exforsys()
{
   a=0;
   b=0;
}
```

Parameterised constructors

It is possible to pass one or more arguments to a constructor function. Simply add the appropriate parameters to the constructor function's declaration and definition. Then, when you declare an object, specify the arguments.

```
#include < iostream >
using namespace std;
// class declaration
class myclass {
int a;
public:
myclass(int x); //constructor
void show( );
};
myclass::myclass(int x) {
cout << "In constructor\n";
a=x;
}
void myclass::show( ) {
cout << a <<"\n";
}
int main( ) {
myclass ob(4);
ob.show( );
return 0;
}
```


Copy Constructor

A copy constructor is a constructor that executes when you initialize a new object of the class with an existing object of the same class. You don't need to create a constructor for this; one is already built into all classes. It's called default copy constructor. It's a one argument constructor whose argument is an object of the same class.

For example,

String s1 (" hi ") ;

String s2 (s1) ;

String s3 = s1 ;

The object s2 is initialized in the statement String s2 (s1) ;

This causes the default copy constructor for the String class to perform a member-by-member copy of s1 into s2. Surprisingly, a different format has exactly the same effect, causing s1 to be copied member-by-member into s3:

String s3 = s1 ;

Although this looks like an assignment statement, it is not. Both formats invoke the default copy constructor, and can be used interchangeably

If we are not satisfied with the default copy constructor we can define a copy constructor on our own. It takes the form of

class_name (const class_name & object_name) ;

For example,

String (const String & tmp) ;

Difference between initialization & assignment

The difference between initialization of an object with another object, and assignment of one object to another is this: Assignment assigns the value of an existing object to another existing object; initialization creates a new object and initializes it with the contents of the existing object. The compiler can distinguish between the two by using your overloaded assignment operator for assignments and your copy constructor for initializers.

For example, the statement

String s2 = s1 ;

would define the object s2 and at the same time initialize it to the values of s1.

Remember the statement

s2 = s1 ;

will not invoke the copy constructor. However, if s1 and s2 are objects, this statement is legal and simply assigns the values of s1 to s2, member-by-member. This is the task of the overloaded assignment operator (=).

Copy constructor:

This constructor takes one argument. Also called one argument constructor. The main use of copy constructor is to initialize the objects while in creation, also used to copy an object. The copy constructor allows the programmer to create a new object from an existing one by initialization.

For example to invoke a copy constructor the programmer writes:

Exforsys e3(e2);
or
Exforsys e3=e2;

Both the above formats can be sued to invoke a copy constructor.

For Example:

```
#include <iostream.h>
class Exforsys()
{
   private:
      int a;
   public:
      Exforsys()
      { }
      Exforsys(int w)
   {
      a=w;
   }
   Exforsys(Exforsys& e)
   {
      a=e.a;
      cout<<" Example of Copy Constructor";
   }
   void result()
   {
      cout<< a;
   }
};



void main()
{
   Exforsys e1(50);
   Exforsys e3(e1);
   cout<< "\ne3=";e3.result();
}
```

In the above the copy constructor takes one argument an object of type Exforsys which is passed by reference. The output of the above program is

Example of Copy Constructor
e3=50

Some important points about constructors:

- A constructor takes the same name as the class name.
- The programmer cannot declare a constructor as virtual or static, nor can the programmer declare a constructor as const, volatile, or const volatile.
- No return type is specified for a constructor.
- The constructor must be defined in the public. The constructor must be a public member.
- Overloading of constructors is possible. This will be explained in later sections of this tutorial.

Private Constructors

If a class constructor is private or protected, the object cannot be created in a scope where its constructor is not visible. In the following example, Distance :: Distance ( ) is declared private.


```
class  Distance
  {
                friend  class  CC;  //  a  friend  class
                friend  int  Display ( ) ;  //  a  friend  function

        public :
                Distance ( float ) ;

                // . . . .
        private :

                Distance ( ) ;
                // . . . .
  } ;

  class  CC
  {
        public :
                void  Compute ( )
                {
                        Distance  d1 ;    //  ok
```

```
                    }
};

int Display ( )
{
            Distance d2 ; // ok
}

int main ( )
{
        Distance d3 ; // error
        // . . . .
}
```

Within the program, only the Distance member functions, friend class CC and friend function Display ( ) may declare Distance objects that take no argument. There is no restriction on the declaration of Distance objects that take an argument of type float.

Constructors in the Base and Derived Classes

It is important to note that the constructors are not inherited in the inheritance process, unlike other methods that are inherited. This fact puts the burden on the programmer to provide a compulsorily special constructor in the derived class.

When you declare an object of a derived class, the compiler executes the constructor function of the base class followed by the constructor function of the derived class.

The parameter list for the derived class's constructor function could be different from that of the base class's constructor function. Therefore, the constructor function for the derived class must tell the compiler what values to use as arguments to the constructor function for the base class.The derived class's constructor function specifies the arguments to the base class's constructor function.

```
D ( int a1, int a2, float b1, float b2, int d1 ) : A ( a1, a2 ), B ( b1, b2 )
{
      d = d1 ;
}
```

The colon (:) operator after the derived constructor's parameter list specifies in this case that an argument list for a base class's constructor follows. The argument list is in parentheses and follows the name of the base class.

When a base class has more than one constructor function, the compiler decides which one to call based on the types of the arguments in the base constructor argument list as specified by the derived class constructor function.

One important thing to note here is that, as long as no base class constructor takes any arguments, the derived class need not have a constructor function. However, if any base class contains a constructor with one or more arguments, then it is mandatory for the derived class to have a constructor and pass the arguments to the base class constructors.

When both the derived and base classes contain constructors, the base constructor is executed first and then the constructor in the derived class is executed.

In case of multiple inheritance, the base classes are constructed in the order in which they appear in the declaration of the derived class.

Destructors

What is the use of Destructors

Destructors are also special member functions used in C++ programming language. Destructors have the opposite function of a constructor. The main use of destructors is to release dynamic allocated memory. Destructors are used to free memory, release resources and to perform other clean up. Destructors are automatically named when an object is destroyed. Like constructors, destructors also take the same name as that of the class name.

General Syntax of Destructors

~ classname();

The above is the general syntax of a destructor. In the above, the symbol tilda ~ represents a destructor which precedes the name of the class.

Some important points about destructors:

- Destructors take the same name as the class name.
- Like the constructor, the destructor must also be defined in the public. The destructor must be a public member.
- The Destructor does not take any argument which means that destructors cannot be overloaded.
- No return type is specified for destructors.

For example:

```
class Exforsys
{
   private:
      ……….
   public:
      Exforsys()
      { }
      ~ Exforsys()
      { }
}
```

Arrays as Class Members

We can use arrays to hold the data of a class, but the size of these arrays will not be determined until run-time/object-instantiation-time. Below is an example of how to do it .

```
class MyClass{
public:
 MyClass(int size){
   myvararray = new double[size];
 }

private:
 double *myvararray;

}
```


Here the size of the array myvararray is determined when an object of MyClass is instantiated, using the argument of the constructer.

Arrays of objects

Array of objects is a linear sequence of objects in memoryie it is a collection of objects of the same class.The declaration syntax is as follows .
Class *p = new Class[sz];

where *p is a pointer to the class and sz is the required size of the object array ie the maximum number of objects that should be included in the array.Here is an example.

```
#include <iostream>
class MyClass {
  int i;
public:
  void setInt(int j) {
     i=j;
```

```
  }
  int getInt() {
     return i;
  }
};
int main()
{
  MyClass myObject[3];
  int i;
  for(i=0; i<3; i++)
     myObject[i].setInt(i+1);
  for(i=0; i<3; i++)
     cout << myObject[i].getInt() << "\n";
  return 0;
}
```

MODULE 2

Inheritance

Inheritance is the process by which new classes called *derived* classes are created from existing classes called *base* classes. The derived classes have all the features of the base class and the programmer can choose to add new features specific to the newly created derived class.

For example, a programmer can create a *base* class named fruit and define *derived* classes as mango, orange, banana, etc. Each of these derived classes, (mango, orange, banana, etc.) has all the features of the *base* class (fruit) with additional attributes or features specific to these newly created derived classes. Mango would have its own defined features, orange would have its own defined features, banana would have its own defined features, etc.

This concept of *Inheritance* leads to the concept of *polymorphism*.

Features or Advantages of Inheritance:

Reusability:

Inheritance helps the code to be reused in many situations. The base class is defined and once it is compiled, it need not be reworked. Using the concept of inheritance, the programmer can create as many derived classes from the base class as needed while adding specific features to each derived class as needed.

Saves Time and Effort:

The above concept of reusability achieved by inheritance saves the programmer time and effort. Since the main code written can be reused in various situations as needed.

Increases Program Structure which results in greater reliability.

Polymorphism

General Format for implementing the concept of Inheritance:

class derived_classname: access specifier baseclassname

For example, if the *base* class is *exforsys* and the derived class is sample it is specified as:

```
class sample: public exforsys
```

The above makes sample have access to both *public* and *protected* variables of base class *exforsys*. Reminder about public, private and protected access specifiers:

- If a member or variables defined in a class is private, then they are accessible by members of the same class only and cannot be accessed from outside the class.
- Public members and variables are accessible from outside the class.
- Protected access specifier is a stage between private and public. If a member functions or variables defined in a class are protected, then they cannot be accessed from outside the class but can be accessed from the derived class.

Inheritance Example:

```
class exforsys
{
public:
exforsys(void) { x=0; }
void f(int n1)
{
x= n1*5;
}

void output(void) { cout<<x; }

private:
int x;
};

class sample: public exforsys
{
public:
sample(void) { s1=0; }

void f1(int n1)
{
s1=n1*10;
}

void output(void)
{
exforsys::output();
cout << s1;
```

```
}

private:
int s1;
};

int main(void)
{
sample s;
s.f(10);
s.output();
s.f1(20);
s.output();
}
```

The output of the above program is

```
50
200
```


In the above example, the derived class is sample and the base class is *exforsys*. The *derived* class defined above has access to all *public* and *private* variables. *Derived* classes cannot have access to base class *constructors* and *destructors*. The derived class would be able to add new member functions, or variables, or new constructors or new destructors. In the above example, the derived class sample has new member function f1( ) added in it. The line:

```
sample s;
```

creates a derived class object named as s. When this is created, space is allocated for the data members inherited from the base class *exforsys* and space is additionally allocated for the data members defined in the derived class *sample*.

The *base* class constructor *exforsys* is used to initialize the base class data members and the *derived* class *constructor* sample is used to initialize the data members defined in *derived* class.

The access specifier specified in the line:

```
class sample: public exforsys
```

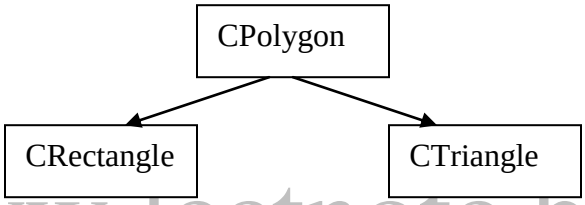
Public indicates that the *public* data members which are inherited from the *base* class by the derived class sample remains *public* in the *derived* class.

Inheritance And Access Control

Inheritance

A key feature of C++ classes is inheritance. Inheritance allows to create classes which are derived from other classes, so that they automatically include some of its "parent's" members, plus its own. For example, we are going to suppose that we want to declare a series of classes that describe polygons like our CRectangle, or like CTriangle. They have certain common properties, such as both can be described by means of only two sides: height and base.

This could be represented in the world of classes with a class CPolygon from which we would derive the two other ones: CRectangle and CTriangle.



The class CPolygon would contain members that are common for both types of polygon. In our case: width and height. And CRectangle and CTriangle would be its derived classes, with specific features that are different from one type of polygon to the other.

Classes that are derived from others inherit all the accessible members of the base class. That means that if a base class includes a member A and we derive it to another class with another member called B, the derived class will contain both members A and B.

In order to derive a class from another, we use a colon (:) in the declaration of the derived class using the following format:

```
class derived_class_name: public base_class_name
{ /*...*/ };
```

Where derived_class_name is the name of the derived class and base_class_name is the name of the class on which it is based. The public access specifier may be replaced by any one of the other access specifiers protected and private. This access specifier describes the minimum access level for the members that are inherited from the base class.

Member access control in classes

We can use the following chart for seeing the accessibilities of the members in the Base class (first class) and derived class (second class).

		Inheritance Mode		
		public	protected	private
Members in Base Class	public	public	protected	private
	protected	protected	protected	private
	private	X	X	X
		Members in derived class		

Here X indicates that the members are not inherited, i.e. they are not accessible in the derived class.

Friend Functions and Classes

A **friend** function is a function that is not a member of a class but has access to the class's private and protected members. Friend functions are not considered class members; they are normal external functions that are given special access privileges. Friends are not in the class's scope, and they are not called using the member-selection operators (. and –>) unless they are members of another class. A **friend** function is declared by the class that is granting access. The **friend** declaration can be placed anywhere in the class declaration. It is not affected by the access control keywords.

Need for Friend Function:

when a data is declared as private inside a class, then it is not accessible from outside the class. A function that is not a member or an external class will not be able to access the private data. A programmer may have a situation where he or she would need to access private data from non-member functions and external classes. For handling such cases, the concept of Friend functions is a useful tool.

What is a Friend Function?

A friend function is used for accessing the non-public members of a class. A class can allow non-member functions and other classes to access its own private data, by making them friends. Thus, a friend function is an ordinary function or a member of another class.

How to define and use Friend Function in C++:

The friend function is written as any other normal function, except the function declaration of these functions is preceded with the keyword friend. The friend function must have the class to which it is declared as friend passed to it in argument.

Some important points to note while using friend functions in C++:

- The keyword friend is placed only in the function declaration of the friend function and not in the function definition.
  .
- It is possible to declare a function as friend in any number of classes.
  .
- When a class is declared as a friend, the friend class has access to the private data of the class that made this a friend.
  .
- A friend function, even though it is not a member function, would have the rights to access the private members of the class.
  .
- It is possible to declare the friend function as either private or public.
  .
- The function can be invoked without the use of an object. The friend function has its argument as objects, seen in example below.

Example to understand the friend function:


```
#include<iostream.h>
class exforsys
{
private:
int a,b;
public:
void test()
{
a=100;
b=200;
}
friend int compute(exforsys e1)

//Friend Function Declaration with keyword friend and with the object of class
exforsys to which it is friend passed to it
};


int compute(exforsys e1)
{
//Friend Function Definition which has access to private data
return int(e1.a+e2.b)-5;
```

```
}

main()
{
exforsys e;
e.test();
cout<<"The result is:"<
//Calling of Friend Function with object as argument.
}
```

The output of the above program is

The result is:295

The function compute() is a non-member function of the class exforsys. In order to make this function have access to the private data a and b of class exforsys , it is created as a friend function for the class exforsys. As a first step, the function compute() is declared as friend in the class exforsys as:

friend int compute (exforsys e1)

Friend Class

A **friend** class is a class all of whose member functions are friend functions of a class, that is, whose member functions have access to the other class's private and protected members. Suppose the **friend** declaration in class B had been:

Copy Code

```
friend class A;
```

In that case, all member functions in class A would have been granted friend access to class B. The following code is an example of a friend class:

Copy Code

```
// classes_as_friends2.cpp
// compile with: /EHsc
#include <iostream>

using namespace std;
class YourClass {
friend class YourOtherClass;  // Declare a friend class
public:
```

```
  YourClass() : topSecret(0){}
  void printMember() { cout << topSecret << endl; }
private:
  int topSecret;
};

class YourOtherClass {
public:
  void change( YourClass& yc, int x ){yc.topSecret = x;}
};

int main() {
  YourClass yc1;
  YourOtherClass yoc1;
  yc1.printMember();
  yoc1.change( yc1, 5 );
  yc1.printMember();
}
```

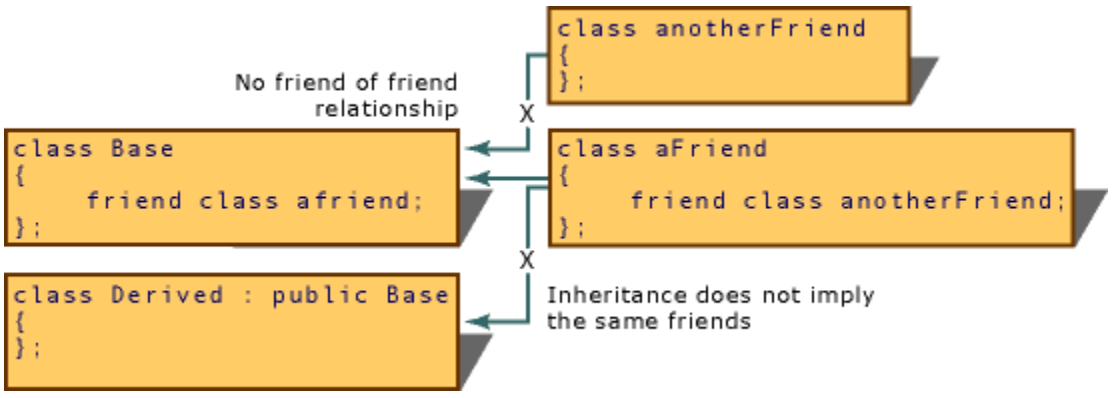
Friendship is not mutual unless explicitly specified as such. In the above example, member functions of YourClass cannot access the private members of YourOtherClass.

A managed type cannot have any friend functions, friend classes, or friend interfaces.

Friendship is not inherited, meaning that classes derived from YourOtherClass cannot access YourClass's private members. Friendship is not transitive, so classes that are friends of YourOtherClass cannot access YourClass's private members.

The following figure shows four class declarations: Base, Derived, aFriend, and anotherFriend. Only class aFriend has direct access to the private members of Base (and to any members Base might have inherited).

Implications of friend Relationship



Extending Classes

Eg:

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

class CPolygon {
  protected:
    int width, height;
  public:
    void set_values (int a, int b)
      { width=a; height=b;}
  };

class CRectangle: public CPolygon {
  public:
    int area ()
      { return (width * height); }
  };
```



```
class CTriangle: public CPolygon {
  public:
    int area ()
      { return (width * height / 2); }
  };

int main () {
  CRectangle rect;
  CTriangle trgl;
  rect.set_values (4,5);
  trgl.set_values (4,5);
  cout << rect.area() << endl;
  cout << trgl.area() << endl;
  return 0;
}
```

Output : 20
10

The objects of the classes CRectangle and CTriangle each contain members inherited from CPolygon. These are: width, height and set_values().

The protected access specifier is similar to private. Its only difference occurs in fact with inheritance. When a class inherits from another one, the members of the derived class can access the protected members inherited from the base class, but not its private members.

Since we wanted width and height to be accessible from members of the derived classes CRectangle and CTriangle and not only by members of CPolygon, we have used protected access instead of private.

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

Access	public	protected	private
members of the same class	yes	yes	yes
members of derived classes	yes	yes	no
not members	yes	no	no

Where "not members" represent any access from outside the class, such as from main(), from another class or from a function. In our example, the members inherited by CRectangle and CTriangle have the same access permissions as they had in their base class CPolygon:

```
CPolygon::width          // protected access
CRectangle::width        // protected access
CPolygon::set_values()   // public access
CRectangle::set_values() // public access
```


This is because we have used the public keyword to define the inheritance relationship on each of the derived classes:

```
Class CRectangle: public CPolygon { ... }
```

This public keyword after the colon (:) denotes the minimum access level for all the members inherited from the class that follows it (in this case CPolygon). Since public is the most accessible level, by specifying this keyword the derived class will inherit all the members with the same levels they had in the base class.

If we specify a more restrictive access level like protected, all public members of the base class are inherited as protected in the derived class. Whereas if we specify the most restricting of all access levels: private, all the base class members are inherited as private.

For example, if daughter was a class derived from mother that we defined as:

class daughter: protected mother;

This would set protected as the maximum access level for the members of daughter that it inherited from mother. That is, all members that were public in mother would become protected in daughter. Of course, this would not restrict daughter to declare its own public members. That maximum access level is only set for the members inherited from mother. If we do not explicitly specify any access level for the inheritance, the compiler assumes private for classes declared with class keyword and public for those declared with struct.

In principle, a derived class inherits every member of a base class except:

- its constructor and its destructor
- its operator=() members
- its friends

Although the constructors and destructors of the base class are not inherited themselves, its default constructor (i.e., its constructor with no parameters) and its destructor are always called when a new object of a derived class is created or destroyed.

If the base class has no default constructor or you want that an overloaded constructor is called when a new derived object is created, you can specify it in each constructor definition of the derived class:

derived_constructor_name (parameters) : base_constructor_name (parameters) {...}

// Program to illustrate constructor and destructor in inheritance


```
#include<iostream.h>
class base
{
    public:
    base()
    {
        cout<<"\n Base class constructor";
    }

    ~base()
    {
        cout<<"\n Base class destructor";
    }
};


class derived : public base
{
    public:
    derived()
    {
        cout<<"\n Derived class constructor";
    }

    ~derived()
    {
```

```
        cout<<"\n Derived class destructor";
    }
};


void main()
{
    derived obj;
}
```

Test data

```
Base class constructor
Derived class constructor
Derived class destructor
Base class destructor
```

Public ,Private and Protected Inheritance

Inheritance is probably the most powerful feature of object oriented programming. Inheritance is the technique that is used to build new classes from existing ones. In inheritance, you derive a new class from an existing class. The class from which you derive is called the base class, and the new class is called the derived class.

Public Derivation

In a public derivation, the derived class inherits public members as public, and protected members as protected. They can be accessed by a new member function of the derived class. However, instances of the derived classes may access only the public members.

Protected Derivation

In a protected derivation, the derived class inherits public and protected members as protected. They can be accessed by a new member function of the derived class. However, instances of the derived classes may access only the public members.

Private Derivation

In a private derivation, the derived class inherits public and protected members as private. They can be accessed by a new member function of the derived class. However, instances of the derived classes may not access them. Also, public and protected members of the base class are not available for subsequent derivations.

Regardless of the type of derivation, private members (of base class) are not accessible in

the derived class. If these members are to be accessed, that can only be done by the methods of the base class.

What is inherited?

When inheritance is done, various links and tables (index, virtual etc) are created which are used to provide the accessibilities of the members of the base class in derived class and in other class hierarchy. This means saying **"public members are inherited"** is better to say as **"public members become accessible"**.

A derived class inherits every member of a base class except:

- its constructor and its destructor
- its friends
- its operator=() members

Classification of Inheritance

1.Single inheritance : one base class one derived class
2.Multilevel inheritance : deriving from another derived class
3. Hierarchical inheritance : one base, many derived classes
4.Multiple inheritance : more than one base class
5.Hybrid inheritance : combination of all the types

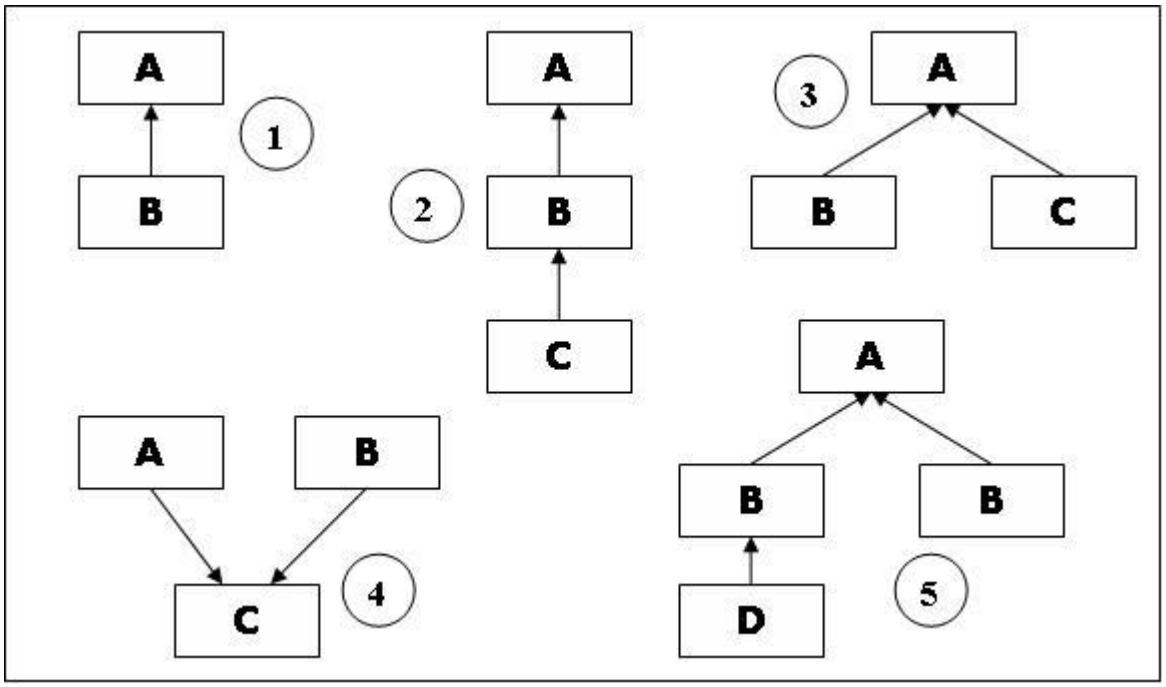


Fig 2.1: Types of Inheritance

Single Inheritance

For example:

```
#include <iostream>

class mother {
  public:
    mother ()
      { cout << "mother: no parameters\n"; }
    mother (int a)
      { cout << "mother: int parameter\n"; }
};

class daughter : public mother {
  public:
    daughter (int a)
      { cout << "daughter: int parameter\n\n"; }
};

class son : public mother {
  public:
    son (int a) : mother (a)
```

```
    { cout << "son: int parameter\n\n"; }
};

int main () {
  daughter cynthia (0);
  son daniel(0);
    return 0;
}
```

Output : mother: no parameters
daughter: int parameter
mother: int parameter
son: int parameter

Multiple Inheritance

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 reusability.

For 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.

.Eg2:
if we had a specific class to print on screen (COutput) and we wanted our classes CRectangle and CTriangle to also inherit its members in addition to those of CPolygon we could write:

```
class CRectangle: public CPolygon, public COutput;
class CTriangle: public CPolygon, public COutput;
```

Here is the complete example:

```
#include <iostream>
```

```
class CPolygon {
  protected:
    int width, height;
  public:
    void set_values (int a, int b)
      { width=a; height=b;}
  };

class COutput {
  public:
    void output (int i);
  };

void COutput::output (int i) {
  cout << i << endl;
  }

class CRectangle: public CPolygon, public COutput {
  public:
    int area ()
      { return (width * height); }
  };
```



```
class CTriangle: public CPolygon, public COutput {
  public:
    int area ()
      { return (width * height / 2); }
  };

int main () {
  CRectangle rect;
  CTriangle trgl;
  rect.set_values (4,5);
  trgl.set_values (4,5);
  rect.output (rect.area());
  trgl.output (trgl.area());
  return 0;
}
```

Output : 20
10

Multilevel Inheritance

/******* Implementation Of Multilevel Inheritance *********/**

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

class student // Base Class
{
protected:
int rollno;
char *name;
public:
void getdata(int b,char *n)
{
rollno = b;
name = n;
}
void putdata(void)
{
cout< < " The Name Of Student \t: "< < name< < endl;
cout< < " The Roll No. Is \t: "< < rollno< < endl;
}
};
```



```
class test:public student // Derieved Class 1
{
protected:
float m1,m2;
public:
void gettest(float b,float c)
{
m1 = b;
m2 = c;
}
void puttest(void)
{
cout< < " Marks In CP Is \t: "< < m1< < endl;
cout< < " Marks In Drawing Is \t: "< < m2< < endl;
}
};

class result:public test // Derieved Class 2
{
protected:
float total;
```

```
public:
void displayresult(void)
{
total = m1 + m2;
putdata();
puttest();
cout< < " Total Of The Two \t: "< < total< < endl;
}
};
void main()
{
clrscr();
int x;
float y,z;
char n[20];
cout< < "Enter Your Name:";
cin>>n;
cout< < "Enter The Roll Number:";
cin>>x;
result r1;
r1.getdata(x,n);
cout< < "ENTER COMPUTER PROGRAMMING MARKS:";
cin>>y;
cout< < "ENTER DRAWING MARKS:";
cin>>z;
r1.gettest(y,z);
cout< < endl< < endl< < "********** RESULT **********"< < endl;
r1.displayresult();
cout< < "**********************************"< < endl;
getch();

}
/********** OUTPUT **********
Enter Your Name:Lionel
Enter The Roll Number:44
ENTER COMPUTER PROGRAMMING MARKS:95
ENTER DRAWING MARKS:90


********** RESULT **********
The Name Of Student : Lionel
The Roll No. Is : 44
Marks In CP Is : 95
Marks In Drawing Is : 90
Total Of The Two : 185
```

****

Hierarchical Inheritance

When from one base class more than one classes are derived that is called hierarchical inheritance.With the help of hierarchical inheritance we can distribute the property of one class into many classes.

Example:

```
#include<iostream.h>
#include<conio.h>
Class B
{
Protected:
int x,int y;
public:
      void assign()
{
X=10;y=20;
}
Class D1:public B
{
int s;
public:
void add()
{
s=x+y;
cout<<"x+y="<<s<<endl;
}

};
Class D2:public B
{
int t;
public:
void sub()
{
t=x-y;
cout<<"x-y="<<t<<endl;
}
};
Class D3:public B
{
int m;
public:
void mul()
{
m=x*y;
```

```
cout<<"x*y="<<m<<endl;
}

};
};
Void main()
{
D1 d1;
D2 d2;
D3 d3;
Clrscr();
d1,assign();
d1.add();
d2,assign();
d2.sub();
d3,assign();
d3.mul();
getch();

}
```

The Output of above program is

```
x+y=30
x-y=-10
x*y=200
```


Hybrid Inheritance

If we apply more than one type of inheritance to design a problem then that is known as hybrid inheritance.

Example:

```
#include<iostream>
using namespace std;
class student
{
protected:
int roll_number;
public:
void getnumber(int a)
{
roll_number=a;
}

void putnumber()
{
cout<<"Roll No:"<<roll_number<<"\n";
```

```
}
};
class test:public student
{
protected:
float  part1,part2;
public:
void get_marks(float x,float y)
{
part1=x;
part2=y;
}
void put_marks()
{
cout<<"Marks obtained:"<<"\n"<<"Part1="<<part1<<"\n"
<<"Part2="<<part2<<"\n";
}
};
class sports
{
protected:
float score;
public:
void get_score(float s)
{
score=s;
}
void put_score()
{
cout<<"Sports wt:"<<score<<"\n\n";

}
};
class result:public test,public sports
{
float total;
public:
void display()
{
total=part1+part2+score;
putnumber();
put_marks();
put_score();
cout<<"Total score :"<<total<<"\n"
}
};
```

```
int main()
{
result s1;
s1.getnumber(1234)
s1.get_marks(27.5,33.0);
s1.get_score(6.0);
s1.display();
return 0;
}
```

Output
Roll No :1234
Marks obtained:
Part1=27.5
Part2=33
Sports wt:6
Total Score:66.5

Constructors in the Base and Derived Classes

It is important to note that the constructors are not inherited in the inheritance process, unlike other methods that are inherited. This fact puts the burden on the programmer to provide a compulsorily special constructor in the derived class.

When you declare an object of a derived class, the compiler executes the constructor function of the base class followed by the constructor function of the derived class.

The parameter list for the derived class's constructor function could be different from that of the base class's constructor function. Therefore, the constructor function for the derived class must tell the compiler what values to use as arguments to the constructor function for the base class.The derived class's constructor function specifies the arguments to the base class's constructor function.

```
D ( int a1, int a2, float b1, float b2, int d1 ) : A ( a1, a2 ), B ( b1, b2 )
{
        d = d1 ;
}
```

The colon (:) operator after the derived constructor's parameter list specifies in this case that an argument list for a base class's constructor follows. The argument list is in parentheses and follows the name of the base class.

When a base class has more than one constructor function, the compiler decides which one to call based on the types of the arguments in the base constructor argument list as specified by the derived class constructor function.

One important thing to note here is that, as long as no base class constructor takes any arguments, the derived class need not have a constructor function. However, if any base class contains a constructor with one or more arguments, then it is mandatory for the derived class to have a constructor and pass the arguments to the base class constructors.

When both the derived and base classes contain constructors, the base constructor is executed first and then the constructor in the derived class is executed.

In case of multiple inheritance, the base classes are constructed in the order in which they appear in the declaration of the derived class.

MODULE 3

Polymorphism

Introduction

Polymorphism is the ability to use an operator or function in different ways. Polymorphism gives different meanings for operators or functions. Poly, referring to many, signifies the many uses of these operators and functions. A single function usage or an operator functioning in many ways can be called polymorphism. Polymorphism refers to codes, operations or objects that behave differently in different contexts.

Below is a simple example of the above concept of polymorphism:

```
6 + 10
```

The above refers to integer addition.

The same + operator can be used with different meanings with strings:

```
"Exforsys" + "Training"
```


The same + operator can also be used for floating point addition:

```
7.15 + 3.78
```

Polymorphism is a powerful feature of the object oriented programming language C++. A single operator + behaves differently in different contexts such as integer, float or strings referring the concept of *polymorphism*. The above concept leads to operator *overloading*. The concept of overloading is also a branch of *polymorphism*. When the exiting operator or function operates on new data type it is *overloaded*. This feature of polymorphism leads to the concept of *virtual methods*.

Polymorphism refers to the ability to call different functions by using only one type of function call. Suppose a programmer wants to code vehicles of different shapes such as circles, squares, rectangles, etc. One way to define each of these classes is to have a member function for each that makes vehicles of each shape. Another convenient approach the programmer can take is to define a base class named Shape and then create an instance of that class. The programmer can have array that hold pointers to all different objects of the vehicle followed by a simple loop structure to make the vehicle, as per the shape desired, by inserting pointers into the defined array. This approach leads to different functions executed by the same function call. Polymorphism is used to give different meanings to the same concept. This is the basis for *Virtual function* implementation.

In polymorphism, a single function or an operator functioning in many ways depends upon the usage to function properly. In order for this to occur, the following conditions must apply:

- All different classes must be derived from a single base class. In the above example, the shapes of vehicles (circle, triangle, rectangle) are from the single base class called Shape.
- The member function must be declared virtual in the base class. In the above example, the member function for making the vehicle should be made as virtual to the base class.

Features and Advantages of the concept of Polymorphism:

Applications are Easily Extendable:

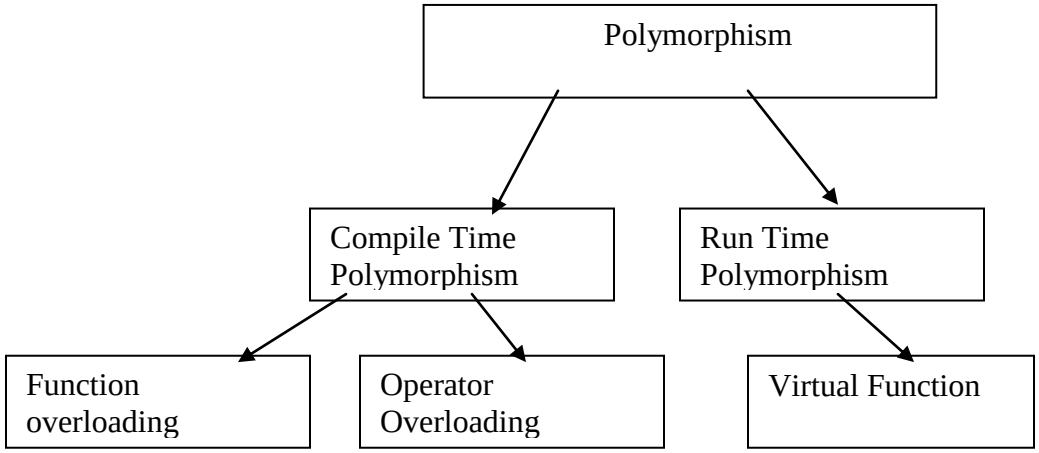
Once an application is written using the concept of polymorphism, it can easily be extended, providing new objects that conform to the original interface. It is unnecessary to recompile original programs by adding new types. Only re-linking is necessary to exhibit the new changes along with the old application. This is the greatest achievement of C++ object-oriented programming. In programming language, there has always been a need for adding and customizing. By utilizing the concept of polymorphism, time and work effort is reduced in addition to making future maintenance easier.

- Helps in reusability of code.
- Provides easier maintenance of applications.
- Helps in achieving robustness in applications.

Types of Polymorphism:

C++ provides three different types of polymorphism.

- Virtual functions
- Function name overloading
- Operator overloading



In addition to the above three types of polymorphism, there exist other kinds of polymorphism:

- run-time
- compile-time
- ad-hoc polymorphism
- parametric polymorphism

Other types of polymorphism defined:

Run-time:

The *run-time* polymorphism is implemented with inheritance and virtual functions. It is also known as late binding or dynamic binding. Late binding refers function calls that are not resolved until run time. Virtual functions are used to achieve late binding. Objects and functions are not linked until run time. The main advantage to late binding is flexibility.

Compile-time:

The *compile-time* polymorphism is implemented with templates. It is also known as early binding or static binding or static linking. Function overloading and operator overloading are compile time polymorphism. It occurs at compile time. All information needed to call a function is known at compile time. Efficiency and fastness is its main advantage.

Ad-hoc polymorphism:

If the range of actual types that can be used is finite and the combinations must be individually specified prior to use, this is called *ad-hoc* polymorphism.

Parametric polymorphism:

If all code is written without mention of any specific type and thus can be used transparently with any number of new types it is called *parametric* polymorphism.

In general, there are two main categories of *Polymorphism* namely

- Ad Hoc Polymorphism
- Pure Polymorphism

Overloading concepts fall under the category of *Ad Hoc Polymorphism* and *Virtual methods*. Templates or parametric classes fall under the category of *Pure Polymorphism*.

Polymorphism is in short the ability to call different functions by just using one type of function call.It is a lot useful since it can group classes and their functions together.Polymorphism is the most important part of Object-Oriented Programming. Some people feel that if they have an idea of what classes are they have stepped in the object-oriented world.But this is not true. Polymorphism is the core of object-oriented programming and if anybody stops here he's missing out the best part of Object Oriented Programming(OOP).

Let us now try to understand polymorphism with the help of an example.Suppose we want to draw a picture consisting of circles,squares,lines and triangles.So we can make a class Shape and create an instance of it like this:


```
Shape *s[100];
```

Now all the addresses of the objects of the other classes(line,circle etc.) are stored in the Shape Array.And then to draw the Picture all we have to do is this:

```
for(int i=0;i<=100;i++)
    s[i]->draw();
```

Now as the loop runs different draw functions of each class is called.

Function Overloading

Function overloading means that we can use the same function name to create functions that perform a variety of different tasks by changing the number or type of parameters but not on the return type. This is known as function polymorphism.

A function call first matches the prototype having the same number and type of arguments and then calls the appropriate function for execution. A best match must be unique. The function selection involves the following steps.

1.The compiler first tries to find an exact match in which the types of actual arguments are same, and use that function.'

2.If an exact match is not found, the compiler uses the integral promotiopns to the act actul srgument, such as

Char to int
Float to double.
3.When either of them fails, the compiler tries to use the built_in conversion (implicit assignment conversion)to the actual arguments and then uses the function whose match is unique. If the conversion is possible to have multiple matches, then the compiler will generate an error message.
Eg:
Long square(long n);
Double square(double x);
A function call such as square(10) will cause an error because int argument can be converted to either long or double, thereby creating an ambiguous situation as to which version of square() should be used.
4. If all of the steps fail, then the compiler will try the user-defined conversions to find a unique match. User-defined conversions are often used in handling class objects.

// Program to illustrate function overloading

```
#include<iostream.h>
float max(float,float,float);
int max(int,int,int);
void main()
{
    int choice;
    float x,y,z;
    int a,b,c;
    cout<<"\n Enter 1 to find maximum number among floats ";
    cout<<"\n Enter 2 to find maximum number among integers \n";
    cin>>choice;
    switch(choice)
    {
        case 1: cout<<"\n Enter 3 float values \n";
        cin>>x>>y>>z;
        cout<<"\n Maximum number is :"<<max(x,y,z);
        break;
        case 2: cout<<"\n Enter 3 integer values \n";
        cin>>a>>b>>c;
        cout<<"\n Maximum number is :"<<max(a,b,c);
        break;

        default:
        cout<<"\n invalid choice try again";
    }
}
float max(float p,float q, float r)
{
    if(p>q && p>r)
    return p;
    else if(q>r)
    return q;
    else
    return r;
}


int max(int p, int q, int r)
```

```
{
    if(p>q && p>r)
    return p;
    else if(q>r)
    return q;
    else
    return r;
}
```

Test data

```
Enter 1 to find maximum number among floats
Enter 2 to find maximum number among integers
2
Enter 3 integer values
10  6  13
```

output

```
Maximum number is : 13
```

Operator Overloading

Operator overloading is a very important feature of Object Oriented Programming. It is because by using this facility programmer would be able to create new definitions to existing operators. In fact in other words a single operator can take up several functions as desired by programmers depending on the argument taken by the operator by using the operator overloading facility.

Rules for Overloading Operators

There are important things to consider in operator overloading with C++ programming language. Operator overloading adds new functionality to its existing operators. The programmer must add proper comments concerning the new functionality of the overloaded operator. The program will be efficient and readable only if operator overloading is used only when necessary.

1.Only existing operators can be overloaded. New Operators can not be created

2.The overloaded operator must have at least one operand that is of user-defined .

3.We can not change the basic meaning of an operator.we can not redefine the plus(+) operator to subtract one value from the other.

4.Overloaded operators follow the syntax rules of original operators.

5. Some operators cannot be overloaded:
scope resolution operator denoted by ::
member access operator or the dot operator denoted by .
The conditional operator denoted by ?:
and pointer to member operator denoted by .*

Sizeof operator

6. We can not use friend functions to overload certain operators.eg:=(Assignment operator),()Function call,[]subscripting operator,->class member access operator.

7. If in this case of unary operator overloading if the function is a member function then the number of arguments taken by the operator member function is zero, and in case if the function defined for the operator overloading is a friend function then it takes one reference argument.

8. If in this case of binary operator overloading if the function is a member function then the number of arguments taken by the operator member function is one, and in case if the function defined for the operator overloading is a friend function then it takes two arguments.

9.When using binary operators overloaded through a member function, the left hand operand must be an object of the relevant class

10.Binary arithmetic operators such as +,-,* and / must explicitly return a value. They must not attempt to change their own arguments.

Broadly classifying operators are of two types namely:

- Unary Operators
- Binary Operators

Unary Operators:

As the name implies takes operate on only one operand. Some unary operators are namely ++ called as Increment operator, -- called as Decrement Operator, ! , ~, unary minus.

Binary Operators:

The arithmetic operators, comparison operators, and arithmetic assignment operators all this which we have seen in previous section of operators come under this category. Both the above classification of operators can be overloaded.

Overloading Unary Operators (member functions)

As said before operator overloading helps the programmer to define a new functionality for the existing operator. This is done by using the keyword operator.

The general syntax for defining an operator overloading is as follows:

```
return_type classname :: operator operator symbol(argument)
{
……..
statements;
}
```

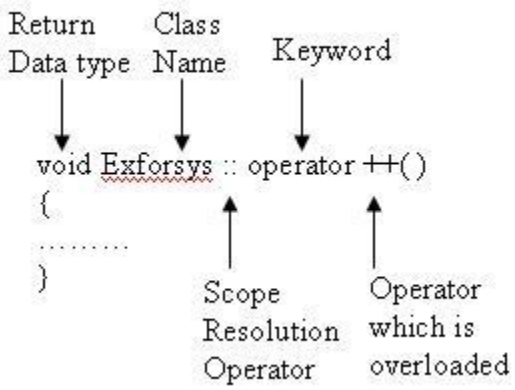
Thus the above clearly specifies that operator overloading is defined as a member function by making use of the keyword operator.

In the above:

- return_type – is the data type returned by the function
- class name - is the name of the class
- operator – is the keyword
- operator symbol – is the symbol of the operator which is being overloaded or
- defined for new functionality
- :: - is the scope resolution operator which is used to use the function definition outside the class. The usage of this is clearly defined in our earlier section of How to define class members.

For example

Suppose we have a class say Exforsys and if the programmer wants to define a operator overloading for unary operator say ++, the function is defined as



Inside the class Exforsys the data type that is returned by the overloaded operator is defined as

```
class Exforsys
{
   private:
   …….
   public:
```

```
   void operator ++( );
   ……
};
```

So the important steps involved in defining an operator overloading in case of unary operators are namely:

Inside the class the operator overloaded member function is defined with the return data type as member function or a friend function. The concept of friend function we will define in later sections. If in this case of unary operator overloading if the function is a member function then the number of arguments taken by the operator member function is none as seen in the below example. In case if the function defined for the operator overloading is a friend function which we will discuss in later section then it takes one argument. The operator overloading is defined as member function outside the class using the scope resolution operator with the keyword operator as explained above

Now let us see how to use this overloaded operator member function in the program

```
#include <iostream.h>
class Exforsys
{
   private:
   int x;
   public:
   Exforsys( ) { x=0; }      //Constructor
   void display();
   void operator++( );
};

void Exforsys :: display()
{
   cout<<"\nValue of x is: " << x;
}

void Exforsys :: operator ++( )  //Operator Overloading for operator ++ defined
{
   ++x;
}

void main( )
{
   Exforsys e1,e2;         //Object e1 and e2 created
   cout<<"Before Increment"
   cout <<"\nObject e1: "<<e1.display();
   cout <<"\nObject e2: "<<e2.display();
   ++e1;  //Operator overloading applied
   ++e2;
```

```
   cout<<"\n After Increment"
   cout <<"\nObject e1: "<<e1.display();
   cout <<"\nObject e2: "<<e2.display();
}
```

The output of the above program is:

```
Before Increment
Object e1:
Value of x is: 0
Object e1:
Value of x is: 0
Before Increment
Object e1:
Value of x is: 1
Object e1:
Value of x is: 1
```

In the above example we have created 2 objects e1 and e2 f class Exforsys. The operator ++ is overloaded and the function is defined outside the class Exforsys.

When the program starts the constructor Exforsys of the class Exforsys initialize the values as zero and so when the values are displayed for the objects e1 and e2 it is displayed as zero. When the object ++e1 and ++e2 is called the operator overloading function gets applied and thus value of x gets incremented for each object separately. So now when the values are displayed for objects e1 and e2 it is incremented once each and gets printed as one for each object e1 and e2.

This is how unary operators get overloaded

Operator overloading is one of the most exciting features of object oriented programming. It is an important technique that has enhanced the power of extensibility of C++.

For Example,

```
a3 . add ( a1,  a2 ) ;

        or

a3 = a1 . add ( a2 ) ;
```

can be changed to the much more readable

```
a3 = a1 + a2 ;
```

C++ tries to make the user-defined data types behave in much the same way as the built-in types. For instance, C++ permits us to add two variables of user-defined data types with the same syntax that is applied to the basic types.

When the operator is used in the context of ordinary variables, ordinary meaning is assigned. When an operator is assigned to an object of the class in which the operator has been overloaded the control goes to the overloading function (calling it implicitly) and a new meaning is given.

For Example,

If k is an integer, then ++k has the normal meaning of incrementing k by 1. Now suppose we have an object pt1 of class Point**,** which has two attributes x and y coordinates. The increment operator is overloaded and has to be suitably defined a method in the class Point**.** Most natural possible definition is to increment the x and y coordinates by 1.

An overloaded operator function is defined in the same way as a normal function except that its name consists of the keyword *operator* followed by one of the predefined permissible C++ operators.

For Example,


```
class string
{
      public :
            string & operator = ( const string & ) ;
            // . . .
 } ;

int main ( )
{
     string s1 ( "helo world" ) ;
     string s2;
     s2 = s1 ;
     return 0 ;
}
```

The statement

```
s2 = s1 ;
```

invokes the function string : : operator = ( ). Here s2 becomes the invoking instance and s1 becomes the explicit argument. The same result can be obtained if the assignment statement is written using functional notation:

s2 . operator = (s1) ;

It should return a reference to a string object to allow the function cascading, such as

string s2, s3 ;

s1 = s2 = s3 ;

which is equivalent to:

s1 . operator = (s2 . operator = (s3)) ;

Operator functions must be either member functions (non static) or friend functions. A basic difference between them is that a friend function will have only one argument for unary operators and two for binary operators, while a member function has no arguments for unary operators and only one for binary operators. This is because the object used to invoke the member function is passed implicitly and therefore is available for the member function. This is not the case with friend functions.

Let us consider the unary minus operator. A minus operator, when used as a unary, takes just one operand. We will see here how to overload this operator so that it can be applied to an object in much the same way as is applied to an **int** or **float** variable. The unary minus when applied to an object should change the sign of each of its data items.

For Example,

```
class X
{
        int a ;
        int b ;

    public :

        void get ( int m, int n ) ;
        void display ( void ) ;
        void operator - ( ) ;  // overloaded unary minus
} ;
void X : : get ( int m, int n )
{

    a = m ;
```

```
            b = n ;
}
void  X : : display ( void )
{

        cout  <<  a  << " " ;
        cout  <<  b   << " " ;
}
void  X : : operator  -  ( )
{

            a = -a ;
            b = -b ;

}

int  main ( )
{
            X  x ;
            x . get ( 10,  20 ) ;
            - x ;          //  calls  operator - ( )
            x . display ( ) ;
            return  0 ;
}
```


This program produces the output -10 -20.

Note that the function operator - ( ) takes no argument. It changes the sign of data members of the object **x.** Since this function is a member function of the same class, it can directly access the members of the object which activated it.

Overloading Binary Operators (Member Functions)

Binary operators, when overloaded, are given new functionality. The function defined for binary operator overloading, as with unary operator overloading, can be member function or friend function.

The difference is in the number of arguments used by the function. In the case of binary operator overloading, when the function is a member function then the number of arguments used by the operator member function is one (see below example). When the function defined for the binary operator overloading is a friend function, then it uses two arguments.

Binary operator overloading, as in unary operator overloading, is performed using a keyword operator.

Binary operator overloading example:

```
#include <iostream.h>
class Exforsys
{
private:
int x;
int y;

public:
Exforsys()                  //Constructor
{ x=0; y=0; }

void getvalue( )            //Member Function for Inputting Values
{
cout << "\n Enter value for x: ";
cin >> x;
cout << "\n Enter value for y: ";
cin>> y;
}

void displayvalue( )        //Member Function for Outputting Values
{
cout <<"value of x is: " << x <<"; value of y is: "<<y
}

Exforsys operator +(Exforsys);
};

Exforsys Exforsys :: operator + (Exforsys e2)
//Binary operator overloading for + operator defined
{
int x1 = x+ e2.x;
int y1 = y+e2.y;
return Exforsys(x1,y1);
}

void main( )
{
Exforsys e1,e2,e3;          //Objects e1, e2, e3 created
cout<<\n"Enter value for Object e1:";
e1.getvalue( );
```

```
cout<<\n"Enter value for Object e2:";
e2.getvalue( );
e3= e1+ e2;                 //Binary Overloaded operator used
cout<< "\nValue of e1 is:"<<e1.displayvalue();
cout<< "\nValue of e2 is:"<<e2.displayvalue();
cout<< "\nValue of e3 is:"<<e3.displayvalue();
}
```

The output of the above program is:

```
Enter value for Object e1:
Enter value for x: 10
Enter value for y: 20
Enter value for Object e2:
Enter value for x: 30
Enter value for y: 40
Value of e1 is: value of x is: 10; value of y is: 20
Value of e2 is: value of x is: 30; value of y is: 40
Value of e3 is: value of x is: 40; value of y is: 60
```


In the above example, the class Exforsys has created three objects e1, e2, e3. The values are entered for objects e1 and e2. The binary operator overloading for the operator '+' is declared as a member function inside the class Exforsys. The definition is performed outside the class Exforsys by using the scope resolution operator and the keyword operator.

The important aspect is the statement:

e3= e1 + e2;

The binary overloaded operator '+' is used. In this statement, the argument on the left side of the operator '+', e1, is the object of the class Exforsys in which the binary overloaded operator '+' is a member function. The right side of the operator '+' is e2. This is passed as an argument to the operator '+' . Since the object e2 is passed as argument to the operator'+' inside the function defined for binary operator overloading, the values are accessed as e2.x and e2.y. This is added with e1.x and e1.y, which are accessed directly as x and y. The return value is of type class Exforsys as defined by the above example.

Let us implement binary operator overloading taking matrix operation as example. Here we are overloading an operator which has two operands, also known as binary operator. All operations are carried out just like arithmetic operations involving basic types.

A statement like

C = add (A, B) ;
was used to add two matrices.

The functional notation can be replaced by a natural looking expression

C = A + B ;

by overloading the + operator using an **operator + ( )** function.

// Program to illustrate binary operator overloading

```
#include<iostream.h>
class matrix
{

private:
    int element[10][10];
    int  rows,cols;
public:
    matrix(int,int);
    void input();
    void display();
    matrix operator + (matrix);
} ;


matrix::matrix(int r,int c)
{
    rows=r;
    cols=c;
}
void matrix::input()
{
    for(int i=0;i<rows;i++)
    for(int j=0;j<cols;j++)
    cin>>element[i][j];
}

void matrix:: display()
{
    for(int i=0;i<rows;i++)
    {
        for(int j=0;j<cols;j++)
        cout<<element[i][j]<<"\t";
        cout<<endl;
    }
}

matrix matrix::operator + (matrix m)
{
```

```
    matrix temp(rows,cols);
    for(int i=0;i<rows;i++)
    for(int j=0;j<cols;j++)
    temp.element[i][j]=element[i][j] + m.element[i][j];
    return temp;
}

void main()
{
    matrix A(2,2),B(2,2),C(2,2);

    A.input();
    B.input();
    C=A+B;
    C.display();
}
```

Test data

```
1  2
3  4



3  4
5  6
```

output

```
4  6
8  10
```


Selecting friend member function for operator overloading

Overloading Unary Operators (Friend functions)

It is possible to overload a unary minus operator using a friend function.

```
friend void operator - ( X & obj ) ;   // declaration

void operator - ( X & obj )
{
        obj . a = - obj . a ;
        obj . b = - obj . b ;

}
```

Note that the argument is passed by reference. It will not work if we pass argument by value because only a copy of the object that activated the call is passed to operator - ( ) . Therefore, the changes made inside the operator function will not reflect in the called

object.

// Program to illustrate operator overloading

```
#include<iostream.h>
class sample
{
    private:
        int value1,value2;
    public:
        sample(int x,int y);
        void operator ++ ();
        void display();
};
sample::sample(int x,int y)
{
    value1=x;
    value2=y;
}
void sample::operator ++ ()
{
    ++value1;
    ++value2;
}

void sample:: display()
{
    cout<<"\n Value1 = "<<value1;
    cout<<"\n Value2 = "<<value2;
}

void main()
{
    sample obj(10,15);
    cout<<"\n Before overloading  ";
    obj.display();
    ++obj;
    cout<<"\n After overloading ";
    obj.display();
}
```

Test data

Before overloading
Value1 = 10
Value2 = 15

After overloading
Value1 = 11
Value2 = 16

Overloading Binary Operators (Friend functions)

It is possible to overload a binary + operator using a friend function.

```
friend matrix operator + ( matrix a, matrix b ) ;   // declaration

matrix operator + ( matrix a, matrix b )
        {
            matrix temp ( rows, cols ) ;
            for ( int i = 0 ; i < rows ; i++ )
            for ( int j = 0 ; j < cols ; j++ )
            temp . element [i] [j] = a . element [i] [j] + b . element [i] [j] ;
            return temp ;
        }
```


We have three distinct ways (styles) of overloading.

1) Object + Object

For Example,

```
Complex operator + ( Complex b ) ;
```

It will imply that you are adding two objects of class Complex and the result is also of class Complex.

```
C = A + B ;  // A object invokes operator + ( ) function
```

2) Object + basic data type (say **int**)

For Example,

```
class A operator + ( int m ) ;
```

This poses no problem.

C = A + 15 ; // A object invokes operator + ( ) function

3) Basic data type + Object

This poses problem as basic data type is not a class.

C = 15 + A ; // 15 cannot invoke operator + ( ) function

The solution is to use a friend function.

A declaration will take the form:

friend classA operator + (int m, classA a) ;

C = 15 + A ; // ok friend function takes all arguments explicitly

// Program to illustrate binary operator overloading (friend functions)

```
#include<iostream.h>
class matrix
{
    private:
    int element[10][10];
    int rows,cols;
    public:
    matrix(int,int);
    void input();
    void display();
    friend matrix operator + (matrix,matrix);
} ;

matrix::matrix(int r,int c)
{
    rows=r;
    cols=c;
}

void matrix::input()
{
    for(int i=0;i<rows;i++)
    for(int j=0;j<cols;j++)
    cin>>element[i][j];
}

void matrix:: display()
{
    for(int i=0;i<rows;i++)
    {
        for(int j=0;j<cols;j++)
        cout<<element[i][j]<<"\t";
        cout<<endl;
    }
}
```

```
matrix operator + (matrix m,matrix n)
{

    matrix temp(m.rows,m.cols);
    for(int i=0;i<m.rows;i++)
    for(int j=0;j<m.cols;j++)
    temp.element[i][j]=m.element[i][j] + n.element[i][j];
    return temp;
}

void main()
{

    matrix A(2,2),B(2,2),C(2,2);
    A.input();
    B.input();
    C=A+B;
    C.display();
}
```

Test data

```
1  2
3  4

3  4
5  6
```


Output

```
4  6
8  10
```

Overloading input ,output operators

Overloading Output Operator/insertion operator(<<)

It is quite simple to create an overload insertion operator(inserter)for a class.All inserter function have the general form :
It must be a friend function.

```
ostream& operator<<(ostream &stream,class_type obj)
{
//body of inserter
return stream;
}
```

ostream means output stream

Overloading Input Operator/extraction operator(>>)

It is quite simple to create an overload extraction operator(extractor)for a class.All extractor function have the general form :
It must be a friend function.

```
istream& operator>>(istream &stream,class_type &obj)
{
//body of extractor
return stream;
}
```

istream means input stream

Eg:

```
#include<iostream.h>
#include<conio.h>
class complex
{
int real,imag;
public:
friend istream& operator>>(istream &in,complex &ob);
friend ostream& operator<<(ostream &out,complex ob);

};
istream& operator>>(istream &in,complex &ob)
{
cout<<"Enter real &imag";
in>>ob.real>>ob.imag;
return in;
}
ostream& operator<<(ostream &out,complex ob)
{
out<<"The complex no is"<<ob.real<<"+j"<<ob.imag;
return out;
}
int main()
{
complex s;
cin>>s;
cout<<s;
getch();
return 0;
}
/* Output
Enter real &imag
2 4
The complex no is 2+j4
*/
```

Virtual Methods (Virtual Function)

Virtual, as the name implies, is something that exists in effect but not in reality. The concept of virtual function is the same as a function, but it does not really exist although it appears in needed places in a program. The object-oriented programming language C++ implements the concept of virtual function as a simple member function, like all member functions of the class.

The functionality of virtual functions can be over-ridden in its derived classes. The programmer must pay attention not to confuse this concept with function overloading. Function overloading is a different concept and will be explained in later sections of this tutorial. Virtual function is a mechanism to implement the concept of polymorphism (the ability to give different meanings to one function).

Need for Virtual Function:

The vital reason for having a virtual function is to implement a different functionality in the derived class.

For example: a Make function in a class Vehicle may have to make a Vehicle with red colour. A class called FourWheeler, derived or inherited from Vehicle, may have to use a blue background and 4 tires as wheels. For this scenario, the Make function for FourWheeler should now have a different functionality from the one at the class called Vehicle. This concept is called Virtual Function.

Properties of Virtual Functions:

- Dynamic Binding Property:

Virtual Functions are resolved during run-time or dynamic binding. Virtual functions are also simple member functions. The main difference between a non-virtual C++ member function and a virtual member function is in the way they are both resolved. A non-virtual C++ member function is resolved during compile time or static binding. Virtual Functions are resolved during run-time or dynamic binding

- Virtual functions are member functions of a class.
- Virtual functions are declared with the keyword virtual, detailed in an example below.
- Virtual function takes a different functionality in the derived class.

Declaration of Virtual Function:

Virtual functions are member functions declared with the keyword virtual.

For example, the general syntax to declare a Virtual Function uses:

```
class  classname //This denotes the base class of C++ virtual function
{
public:
virtual void memberfunctionname() //This denotes the C++ virtual function
{
.....
.....
}
};
```

Referring back to the Vehicle example, the declaration of Virtual function would take the shape below:

```
class Vehicle //This denotes the base class of C++ virtual function
{
public:
virtual void Make() //This denotes the C++ virtual function
{
cout <<"Member function of Base Class Vehicle Accessed"<<endl;
}
};
```


After the virtual function is declared, the derived class is defined. In this derived class, the new definition of the virtual function takes place.

When the class FourWheeler is derived or inherited from Vehicle and defined by the virtual function in the class FourWheeler, it is written as:

```
class Vehicle   //This denotes the base class of C++ virtual function
{
public:
virtual void Make()   //This denotes the C++ virtual function
{
cout <<"Member function of Base Class Vehicle Accessed"<<endl;
```

```
}
};



class FourWheeler : public Vehicle
{
public:
void Make()
{
cout<<"Virtual Member function of Derived class FourWheeler Accessed"<<endl;
}
};



void main()
{
Vehicle *a, *b;
a = new Vehicle();
a->Make();
b = new FourWheeler();
b->Make();
}
```

In the above example, it is evidenced that after declaring the member functions Make() as virtual inside the base class Vehicle, class FourWheeler is derived from the base class Vehicle. In this derived class, the new implementation for virtual function Make() is placed.

The programmer might be surprised to see the function call differs and the output is then printed as above. If the member function has not been declared as virtual, the base class member function is always called because linking takes place during compile time and is therefore static.

In this example, the member function is declared virtual and the address is bounded only during run time, making it dynamic binding and thus the derived class member function is called.

To achieve the concept of dynamic binding in C++, the compiler creates a v-table each time a virtual function is declared. This v-table contains classes and pointers to the functions from each of the objects of the derived class. This is used by the compiler whenever a virtual function is needed.

The Literal Meaning of Virtual means to appear like something while in reality it is something else ie. when virtual functions are used, a program appears to call a function of one class but actually it may be calling a function from another class. In the previous example draw() is a virtual function since it

calls different draw functions from different classes by using the same function
calldraw();

Now how do we know which version of draw() would be called during execution? Which draw() function would get used depends on the contents of s[i].But for this polymorphic approach to work we must satisfy the following conditions:
->The Base class must contain a draw() function which is declared virtual.
->All other classes(line,circle etc.) should be derived from the base class.

Well, all this may be hard to understand in just one go so we'll start using programs that'll help us understand better. Here's the First One.

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

class base                      //Base Class
{
public:
   void func()
   {
     cout<<"In base::func()\n";
   }
};
```



```
class d1:public base                    // Derived Class 1
{
public:
   void func()
   {
    cout<<"In d1::func()\n";
   }
};

class d2:public base     // Derived Class 2
{
public:
   void func()
   {
    cout<<"In d2::func()\n";
   }
};

int main()
{
  d1 d;
  base *b=&d;
```

```
  b->func();
  d2 e;
  b=&e;
  b->func();
  return 0;
}
```

Run this program and you would see that the output would be:
In base::func()
In base::func()

Shouldn't this statement give an error? (b=&e;) No. Since the compiler allows a pointer of a base class to accept addresses of derived class objects. This is known as UPCASTING. Here the Compiler looks at the type of pointer b and since it belongs to the base class it calls the base class function.

But now, let's make a slight modification in our program. Precede the declaration of func() in the base class with the keyword virtual so that it looks like this:

```
virtual void func()
{
  cout<<"In base::func()\n";
}
```


Now Compile and Run the Program. Now the Output is:
In d1::func()
In d2::func()

This time the Compiler looks at the contents of the pointer instead of it's type. Hence since addresses of objects of d1 and d2 classes are stored in *b the respective func() is called. But this way how does the compiler know which function to compile when it doesn't know which object's address 'b' might contain? Which version does the compiler call?

Actually even the compiler does not know which function to call at compile-time. Hence it decides which function to call at run-time with the help of a table called VTABLE. Using this table the compiler finds what object is pointed by the pointer b and then calls the appropriate function. VTABLE is explained later.

The method by which the compiler decides which function to call at run-time is known as late-binding or dynamic-binding. It slows down the program but makes it a lot more flexible.

// Program to illustrate virtual functions

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

```
#include<conio.h>
class base
{
    public:
        virtual void show()
        {
            cout<<"\n From base class";
            cout<<endl;
        }
};

class derived:public base
{
    public:
        void show()
        {
            cout<<"\n From derived class ";
            cout<<endl;
        }
};

void main()
{
    clrscr();
    base bbb;
    derived ddd;
    base *ptr;
    ptr=&bbb;
    ptr->show();
    ptr=&ddd;
    ptr->show();
}
```


Test data

Output
From base class
From derived class

Pure Virtual Functions

What is Pure Virtual Function:

Pure Virtual Function is a Virtual function with no body.

Declaration of Pure Virtual Function:

Since pure virtual function has no body, the programmer must add the notation =0 for declaration of the pure virtual function in the base class.

General Syntax of Pure Virtual Function takes the form:

```
class classname //This denotes the base class of C++ virtual function
{
public:
virtual void virtualfunctioname() = 0 //This denotes the pure virtual function in C++
};
```

The other concept of pure virtual function remains the same as described in the previous section of virtual function.

To understand the declaration and usage of Pure Virtual Function, refer to this example:

```
class Exforsys
{
public:
virtual void example()=0; //Denotes pure virtual Function Definition
};

class Exf1:public Exforsys
{
public:
void example()
{
cout<<"Welcome";
}
};

class Exf2:public Exforsys
{
public:
void example()
```

```
{
cout<<"To Training";
}
};

void main()
{
Exforsys* arra[2];
Exf1 e1;
Exf2 e2;
arra[0]=&e1;
arra[1]=&e2;
arra[0]->example();
arra[1]->example();
}
```

Since the above example has no body, the pure virtual function example() is declared with notation =0 in the base class Exforsys. The two derived class named Exf1 and Exf2 are derived from the base class Exforsys. The pure virtual function example() takes up new definition. In the main function, a list of pointers is defined to the base class.

Two objects named e1 and e2 are defined for derived classes Exf1 and Exf2. The address of the objects e1 and e2 are stored in the array pointers which are then used for accessing the pure virtual function example() belonging to both the derived class EXf1 and EXf2 and thus, the output is as in the above example.

The programmer must clearly understand the concept of pure virtual functions having no body in the base class and the notation =0 is independent of value assignment. The notation =0 simply indicates the Virtual function is a pure virtual function as it has no body.

Some programmers might want to remove this pure virtual function from the base class as it has no body but this would result in an error. Without the declaration of the pure virtual function in the base class, accessing statements of the pure virtual function such as, arra[0]->example() and arra[1]->example() would result in an error. The pointers should point to the base class Exforsys. Special care must be taken not to remove the statement of declaration of the pure virtual function in the base class.

Abstract Classes and Pure Virtual Functions

Abstract class is a class that serves only as a base class from which classes are derived. No objects of an abstract base class are created. A base class that contains pure virtual functions

is an abstract base class.

Pure virtual function can be defined as

```
virtual void show ( ) = 0 ; // pure virtual function
```

The equal sign here has nothing to do with assignment; the value 0 is not assigned to anything. The =0 syntax is simply how we tell the compiler that a function will be pure virtual function.

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

// Program to illustrate pure virtual function


```
#include<iostream.h>
#include<conio.h>
class base
{
    public:
    virtual void show() =0;
};


class derived1:public base
{
    public:
    void show()
    {
        cout<<"\nFrom derived1 class";
        cout<<endl;
    }
};


class derived2:public base
{
```

```
    public:
    void show()
    {
        cout<<"\nFrom derived2 class";
        cout<<endl;
    }
};


void main()
{
    base *ptr;
    derived1 d1;
    derived2 d2;
    clrscr();
    ptr=&d1;
    ptr->show();
    ptr=&d2;
    ptr->show();
}
```


Test data

Output
From derived1 class
From derived2 class

Defining and Using of Virtual Methods

Using Virtual Functions is just one part of polymorphism and knowing how they work completes the other half. When the keyword 'virtual' is inserted in the declaration of the function the compiler inserts all mechanisms in the program to use Virtual Functions. Each Class has a VTABLE that stores the functions that it can access and each class contains a VPTR which can access the VTABLE. Look at this program and the table below it and you will understand the VTABLE and the VPTR.

```
#include <stdio.h>
class item
{
 public:
```

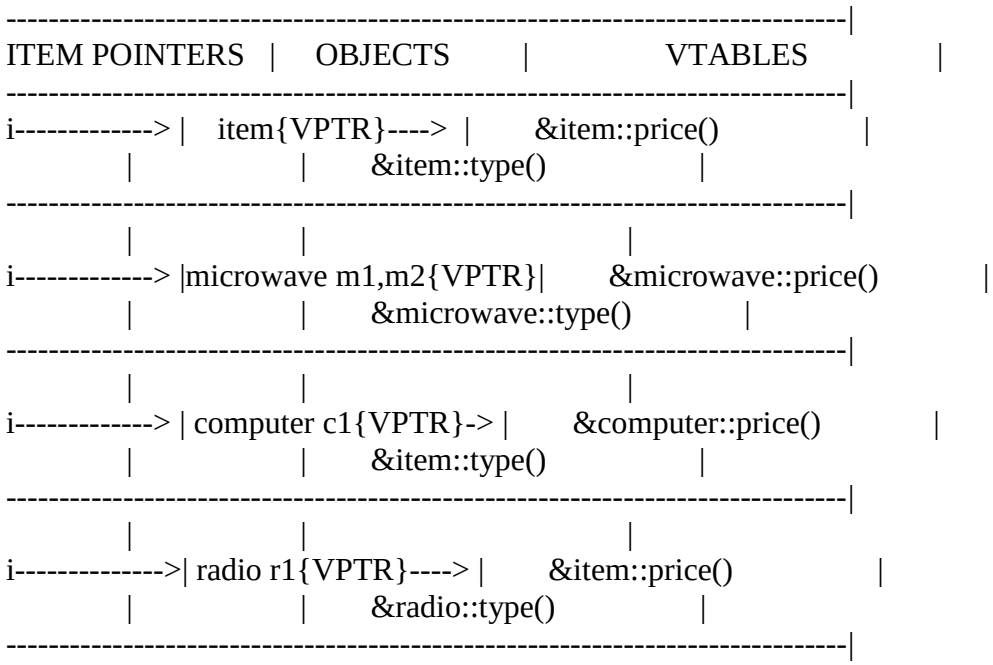
```
    virtual void price()
    {
      printf("In item::price()\n");
    }
    virtual void type()
    {
      printf("In item::type()\n");
    }
    void display();
};
void item::display(){printf("In item::display()\n");}

class microwave:public item
{
public:
   void price()
   {
     printf("Microwave::Price()\n");
   }

   void type()
   {
     printf("Microwave::type()\n");
   }
};

class computer:public item
{
 public:
    void price()
    {
       printf("Computer::Price()\n");
    }
};

class radio:public item
{
public:
   void type()
   {
     printf("radio::type()\n");
   }
};

int main()
{
```

```
 microwave m1;
 computer c1;
 radio r1;

 item *i=&m1;
 i->price();
 i->type();
 printf("\n");

 i=&c1;
 i->price();
 i->type();
 i->display();
 printf("\n");

 i=&r1;
 i->price();
 i->type();
 printf("\n");

 microwave m2;
 i=&m2;
 i->price();
 i->type();
 i->display();
 printf("\n");
  return 0;
}
```


The Output of this Program would be:

```
Microwave::Price()
Microwave::type()

Computer::Price()
In item::type()
In item::display()

In item::price()
radio::type()

Microwave::Price()
Microwave::type()
In item::display()
```

Now here is how the VTABLE of each class Looks Like:



First the VPTR Pointer is initialised to it's proper VTABLE by the constructor which is automatically done by the compiler. When a Virtual Function is being called the VPTR looks up the VTABLE and calls the virtual function. If the function is not present in the VTABLE [like here display()] then the function of the base class is called. So everywhere where the display function is called item::display() is called everytime. No matter how many objects of a class are created they all point to the same VTABLE of the class.

Applications of Abstract class

One of the most powerful and flexible ways to implement the "one interface, multiple methods" approach is to use virtual functions,abstract classes, and run-time polymorphism.A base class can be used to define the nature of the interface to a general class.We can create a class hierarchy that moves from general to specific(base to derived).

```
//Virtual Function practical example
#include<iostream>
Using namespace std;
Class convert
{
Protected:
Double val1;
Double val2;
```

```
Public:
Convert(double i)
{
Val1=I;

}
Double getconv(){return val2;}
Double getinit(){return val1;}
Virtual void compute()=0;

};

//Liters to gallons
Class l_to_g:public convert
{
Public:
L_to_g(double i):convert(i){}
Void compute()
{
Val2=val1/3.7854;
}
};
```



```
//Fahrenheit to Celsius
class f_to_c:public convert
{
public:
f_to_c(double i):convert(i){}
void compute()
{
val2=(val1-32)/1.8;
}
};
Int main()
{
Convert *p;
L_to_g lgob(4);
F_to_c fcob(70);
//use virtual function mechanism
P=&lgob;
Cout<<p->getinit()<<"liters is";
p->compute();
cout<<p->getconv()<<"gallons\n";
p=&fcob;
Cout<<p->getinit()<<"Fahrenheit is";
```

```
p->compute();
cout<<p->getconv()<<"Celsius\n";
return 0;
}
```

Advantages:

1.Handling a new case is a very easy matter.For example, we can add aconversion from feet to meters by including the following class in the above example.

```
//Feet to  meters
class f_to_m:public convert
{
public:
f_to_m(double i):convert(i){}
void compute()
{
val2=val1/3.28;
}
};
```

2.an important use of abatract class is in class libraries.We can create generic extensible class library that will be used by other programmers.By creating class libraries ,we are able to create and control the interface of a general class while still letting other programmers adapt it to their specific needs.

MODULE 4

Advanced Concepts

Virtual destructor

If the destructors are not virtual, then just the destructor corresponding to the pointer type is called. If the destructors are virtual , the destructor corresponding to the object type is called. Using virtual destructors ensures that the correct sequence of destructors is called. If you destroy an object through a pointer or reference to a base class, and the base-class destructor is not virtual, the derived-class destructors are not executed, and the destruction might not be complete.

Destructors are declared as virtual because if do not declare it as virtual the base class destructor will be called before the derived class destructor and that will lead to **memory leak** because derived class objects will not get freed. Destructors are declared virtual so as to bind objects to the methods at runtime so that appropriate destructor is called.

```
#include <iostream.h>
class Base
{
    public:
      Base(){ cout<<"Constructor: Base"<<endl;}
      ~Base(){ cout<<"Destructor : Base"<<endl;}
};
class Derived: public Base
{

    public:
       Derived(){ cout<<"Constructor: Derived"<<endl;}
       ~Derived(){ cout<<"Destructor : Derived"<<endl;}
> };
void main()
{
     Base *Var = new Derived();
     delete Var;
}
```


Try executing this code, the constructors are getting called in the proper order. But to the dread of a programmer of a large project, the destructor of the derived class was not called at all.

This is where the virtual mechanism comes into our rescue. By making the Base class Destructor virtual, both the destructors will be called in order. The following is the corrected sample.

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

```
{
      public:
        Base(){ cout<<"Constructor: Base"<<endl;}
        virtual ~Base(){ cout<<"Destructor : Base"<<endl;}
};
class Derived: public Base
{

      public:
         Derived(){ cout<<"Constructor: Derived"<<endl;}
         ~Derived(){ cout<<"Destructor : Derived"<<endl;}
};
void main()
{
       Base *Var = new Derived();
       delete Var;
}
```

A Pure Virtual Destructor

Unlike ordinary member functions, a virtual destructor is not overridden when redefined in a derived class. Rather, it is extended: the lower-most destructor first invokes the destructor of its base class and only then, it is executed. Consequently, when you try to declare a pure virtual destructor, you may encounter compilation errors, or worse: a runtime crash. However, there's no need to despair--you can enjoy both worlds by declaring a pure virtual destructor without a risk. The abstract class should contain a declaration (without a definition) of a pure virtual destructor:

Note:
There is one more point to be noted regarding virtual destructor. We can't declare pure virtual destructor. Even if a virtual destructor is declared as pure, it will have to implement an empty body (at least) for the destructor.

```
            //Interface.h file
class Interface {
public:
                        virtual ~Interface() = 0; //pure virtual destructor declaration
};
```

Somewhere outside the class declaration, the pure virtual destructor has to be defined like this:

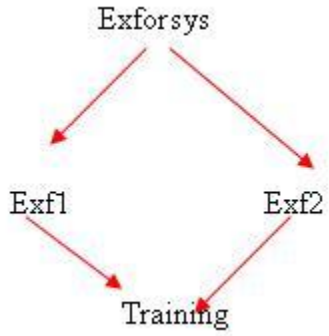
```
            //Interface.cpp file
Interface::~Interface()
{} //definition of a pure virtual destructor; should always be empty
```

Virtual constructor: Constructors cannot be virtual. Declaring a constructor as a virtual function is a syntax error

Virtual destructors are used in situations like:

1) A virtual destructor is used when one class needs to delete object of a derived class that are addressed by the base pointers and invoke a base class destructor to release resources allocated to it.
2) Destructor of a base class should be declared as virtual functions. It ensures that the memory is released effectively.

Virtual Base Class



In the above example, there are two derived classes Exf1 and Exf2 from the base class Exforsys. As shown in the above diagram, the Training class is derived from both of the derived classes Exf1 and Exf2. In this scenario, if a user has a member function in the class Training where the user wants to access the data or member functions of the class Exforsys it would result in error if it is performed like this:

```
class Exforsys
{
protected:
int x;
};

class Exf1:public Exforsys
{ };

class Exf2:public Exforsys
{ };

class Training:public Exf1,public Exf2
{
```

```
public:
int example()
{
return x;
}
};
```

The above program results in a compile time error as the member function example() of class Training tries to access member data x of class Exforsys. This results in an error because the derived classes Exf1 and Exf2 (derived from base class Exforsys) create copies of Exforsys called subobjects.

This means that each of the subobjects have Exforsys member data and member functions and each have one copy of member data x. When the member function of the class Training tries to access member data x, confusion arises as to which of the two copies it must access since it derived from both derived classes, resulting in a compile time error.

When this occurs, Virtual base class is used. Both of the derived classes Exf1 and Exf2 are created as virtual base classes, meaning they should share a common subobject in their base class.

For Example:


```
class Exforsys
{
protected:
int x;
;


class Exf1:virtual public Exforsys
{ };

class Exf2:virtual public Exforsys
{ };

class Training:public Exf1,public Exf2
{
public:
int example()
{
return x;
}
};
```

In the above example, both Exf1 and Exf2 are created as Virtual base classes by using the keyword virtual. This enables them to share a common subobject of their base class Exforsys. This results in only one copy that the member function example() of Class Training can access the member data x.

Now we realise that since the base class virtual function never gets called anyway we'd better keep it's body blank.But there's a better way to do this. We can change the virtual function func() in the base class to the following:

virtual void func()=0;

The =0 is not an assignment operator here but it is just a way of telling the compiler that the function has no body.But there is another side of this. An object of a class which contains a pure virtual function cannot be created.It seems logical enough ie. If you have classes triangle,square,circle derived from shape class we wouldn't want to make an object of the shape class.Hence the shape class should be provided with a pure virtual function.If you even try to create an object of a class containing a pure virtual function the compiler would report an error even pointing out which pure virtual function prevents you from creating an object.

Consider a situation when a 'base' class has two classes derived from it.For example derived1 and derived2.Suppose we create another class which derives itself from both the derivedclasses ie. derived3.Now suppose a member function of derived3 wants to access data or functions in a base class.Since derived1 and derived2 are are derived from base each inherits a copy of base.This copy is referred to as a subobject.Now when derived3 refers to the data in the base class, which of the two copies should it access? The compiler notices this ambiguous situation and reports an error. To get rid of this we should make derived1 and derived2 as virtual baseclasses.This is shown in the following program.


```
#include <iostream>
using namespace std;
class base
{
  protected:
  int data;

  public:
  base()
  {
   data=10;
  }
};

class derived1 : virtual public base
{};

class derived2 : virtual public base
{};
```

```
class derived3 : public derived1,public derived2
{
  public:
  int getdata()
  {
    return data;
  }
};

int main()
{
 derived3 d3;
 int val=d3.getdata();
 cout<<val<<endl;
 return 0;
}
```

Using the keyword virtual in the two classes derived1 and derived2 makes them share a single subobject of the base class hence eliminating all ambiguity since there is only one subobject for derived3 to access.Hence derived1 and derived2 are known as virtual base classes.

Virtual Base classes are related to multiple inheritance. To declare a virtual base class, add the C++ keyword virtual to the base class specifier. The keyword virtual prevents a base class from being duplicated within a derivation hierarchy.

Consider the situation with a base class, Parent; two derived classes, Child1 and Child2; and a fourth class, Grandchild, derived from both Child1 and Child2. In this arrangement, a problem can arise if a member function in the Grandchild class wants to access data or functions in the Parent class.

```
class  Parent
{
        protected :
                int  data ;
} ;


class  Child1 : public  Parent
{

} ;

class  Child2 : public  Parent
{
```

```
} ;


class  Grandchild : public  Child1,  public  Child2
{

        public :
                int  readdata ( )
                {
                        return  data ;
                }
} ;
```

A compiler error occurs when the readdata ( ) member function in Grandchild attempts to access data in Parent. When the Child1 and Child2 classes are derived from Parent, each inherits a copy of Parent. Each of the two child-classes contains its own copy of Parent's data. Now, when Grandchild refers to data, which of the two copies will it access? This situation is ambiguous, and that's what the compiler reports.

To eliminate the ambiguity, we make Child1 and Child2 into virtual base classes.


```
class  Parent
{
      protected :
              int  data ;
} ;


class  Child1 : virtual  public  Parent
{


} ;


class  Child2 : virtual  public  Parent
{


} ;


class  Grandchild : public  Child1,  public  Child2
{
        public :
```

```
                int  readdata ( )
                {
                    return  data ;
                }
} ;
```

Templates

Many C++ programs use common data structures like stacks, queues and lists. Re-inventing source code is not an intelligent approach in an object oriented environment which encourages re-usability. It seems to make more sense to implement a queue that can contain any arbitrary type rather than duplicating code. The solution is to use type parameterization, more commonly referred to as templates.

C++ templates allow one to implement a generic Queue<T> template that has a type parameter T. T can be replaced with actual types, for example, Queue<Customers>, and C++ will generate the class Queue<Customers>. Changing the implementation of the Queue becomes relatively simple. Once the changes are implemented in the template Queue<T>, they are immediately reflected in the classes Queue<Customers>, Queue<Messages>, and Queue<Orders>.

Templates are very useful when implementing generic constructs like vectors, stacks, lists, queues which can be used with any arbitrary type. C++ templates provide a way to re-use source code as opposed to inheritance and composition which provide a way to re-use object code.

C++ provides two kinds of templates: class templates and function templates. Use function templates to write generic functions that can be used with arbitrary types. For example, one can write searching and sorting routines which can be used with any arbitrary type. The Standard Template Library generic algorithms have been implemented as function templates, and the containers have been implemented as class templates.

Generic Function/Tempalte Function/Function Template

Function Templates

To perform identical operations for each type of data compactly and conveniently, use function templates. You can write a single function template definition. Based on the argument types provided in calls to the function, the compiler automatically instantiates separate object code functions to handle each type of call appropriately. The STL algorithms are implemented as function templates.

Implementing Template Functions

Function templates are implemented like regular functions, except they are prefixed with the keyword template. Here is a sample with a function template.

```
#include <iostream>
using namespace std ;
//max returns the maximum of the two elements
template <class T>
T max(T a, T b)
{
        return a > b ? a : b ;
}
```

Using Template Functions

Using function templates is very easy: just use them like regular functions. When the compiler sees an instantiation of the function template, for example: the call max(10, 15) in function main, the compiler generates a function max(int, int). Similarly the compiler generates definitions for max(char, char) and max(float, float) in this case.

```
#include <iostream>
using namespace std ;
//max returns the maximum of the two elements
template <class T>
T max(T a, T b)
{
        return a > b ? a : b ;
}
void main()
{

        cout << "max(10, 15) = " << max(10, 15) << endl ;
        cout << "max('k', 's') = " << max('k', 's') << endl ;
        cout << "max(10.1, 15.2) = " << max(10.1, 15.2) << endl ;
}
```


Output

```
max(10, 15) = 15
max('k', 's') = s
max(10.1, 15.2) = 15.2
```

Template Instantiation

When the compiler generates a class, function or static data members from a template, it is referred to as template instantiation.

- A class generated from a class template is called a generated class.
- A function generated from a function template is called a generated function.

- A static data member generated from a static data member template is called a generated static data member.

A generic function defines a general set of operations that will be applied to various types of data. The type of data that the function will operate up on is passed to it as a parameter.Through generic function, a single general procedure (same algorithm) can be applied to a wide range of data. By using generic function , define the nature of the algorithm independent of data.

Eg:Bubble sort for array of integers and floats.Here procedure is same , only data is different.So create a generic function which will work for all types of data.
The generic function is created by using the keyword template
General Form
Template<class T> return-type function-name(arglist)
{
//body of function
}
T is a placeholder for a datatype used by the function. It will automatically replace with an actual data type when it creates a specific version of the function.When the compiler creates a specific version of the function, it is said to have created a specialization.This is called generated function and this act of generating a function is referred to as instatiating it. Placeholder T is used within the function definition where need a general data type.
Eg1:
Write a generic function for the following

Int sum(int a, int b)
{
Return(a+b);
}
Generic Function is
1st Method :
Template<class T> T sum (T a,T b)
{
Return(a+b);
}
This template function will work for any type of data ,int, float, double. Compiler creates specific version of the function according to the type of data.
2nd Merthod:
Template Function as two lines:
Template<class T>
T sum(T a,T b)
{
Return(a+b);
}

A Function with two generic Types:

For define more than one generic data type in the template statement by using a comma separated list.

Eg: A Function with two generic Types

```
#include<iostream>
Using namespace std;
Template<class T1,class T2>//T1,T2 two generic type placeholders
Void myfun(T1 x,T2 y)
{
Cout<<x<<" "<<y<<"\n";
}
Int main()
{
myfun(10,"I like C++");
myfun(23.4,19L);
return 0;
}
```

The place holder types T1,T2 are replaced by the computer with the data types int and char *,double and long respectively, when the compiler generates the specific instances of myfun() with in main().

Template Function Specialization

In some cases it is possible to override the template-generated code by providing special definitions for specific types. This is called template specialization. The following example demonstrates a situation where overriding the template generated code would be necessary:

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

//max returns the maximum of the two elements of type T, where T is a
//class or data type for which operator> is defined.

template <class T>
T max(T a, T b)
{
   return a > b ? a : b ;
}

int main()
{
   cout << "max(10, 15) = " << max(10, 15) << endl ;
   cout << "max('k', 's') = " << max('k', 's') << endl ;
   cout << "max(10.1, 15.2) = " << max(10.1, 15.2) << endl ;
   cout << "max(\"Aladdin\", \"Jasmine\") = " << max("Aladdin", "Jasmine") << endl ;
   return 0 ;
```

```
}
```

Output

```
max(10, 15) = 15
max('k', 's') = s
max(10.1, 15.2) = 15.2
max("Aladdin", "Jasmine") = Aladdin
```

The function call max("Aladdin", "Jasmine") causes the compiler to generate code for max(char*, char*), which compares the addresses of the strings! To correct special cases like these or to provide more efficient implementations for certain types, one can use template specializations. The above example can be rewritten with specialization as follows:

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

//max returns the maximum of the two elements
template <class T>
T max(T a, T b)
{
   return a > b ? a : b ;
}
```



```
// Specialization of max for char*
template <>
char* max(char* a, char* b)
{
   return strcmp(a, b) > 0 ? a : b ;
}

int main()
{

   cout << "max(10, 15) = " << max(10, 15) << endl ;
   cout << "max('k', 's') = " << max('k', 's') << endl ;
   cout << "max(10.1, 15.2) = " << max(10.1, 15.2) << endl ;
   cout << "max(\"Aladdin\", \"Jasmine\") = " << max("Aladdin", "Jasmine") << endl ;
   return 0 ;
}
```

Output

```
max(10, 15) = 15
max('k', 's') = s
max(10.1, 15.2) = 15.2
max("Aladdin", "Jasmine") = Jasmine
```

Explicitly Overloading a Generic Function

A template function may be overloaded either by a template functions or ordinary functions of its name.This is called explicit specialization. If overload a generic function, that overloaded function overrides(hides) the generic function relative to that specific version.

The overloading resolution is accomplished as follows

(1)call an ordinary function that has an exact match.

(2)call a template function that could be created with an exact match.

(3)Try normal overloading resolution to ordinary functions and call the one that matches.

New style specialization syntax:

Template<>void swap<int>(int &a,int &b)

This is the specialization for int.

Generic Function Restrictions

Generic functions are more restrictive than overloaded functions.When functions are overloaded , different actions performed with in the body of each overloaded function.But a generic function must perform the same general action for all versions-only the type of data can differ that is single procedure, only change the type of data.

Applications of Generic Function

Advantage:

(1) make programming easy

(2) whenever a function defines a generalization algorithm.make it generic function.So it can be used for any type of data without having recode the function.

(3)Time saving

(4)Space Saving

(5)Debugging easy

Applications:

(1)Generic sort:Generic function for sorting

(2)Compacting an array

Class Templates

A class template definition looks like a regular class definition, except it is prefixed by the keyword template. For example, here is the definition of a class template for a Stack.

```
#include<iostream.h>
 #include<conio.h>
 const int size=3;
 template <class T>
 class stack
 {
 private:
 T a[size];
 int top;
 public:
 stack()
 {
 top=-1;
```

```
}
  void push(T item);
  T pop();
};
template <class T> void stack<T>:: push(T item)
{
if(top==size-1)
  cout<<"Stack Full\n";
else
  a[++top]=item;
}
template <class T> T stack<T>:: pop()
{
if(top==-1)
  {
   cout<<"Empty stack";
   return 0;
   }
else
return(a[top--]);
}
int main()
{
stack<int>s1;
stack<char>s2;
s1.push(2);
s1.push(5);
s1.push(7);
cout<<"Pop int stack::";
for(int i=0;i<3;i++)
  cout<<s1.pop()<<"\n";
  s2.push('a');
  s2.push('b');
  s2.push('c');
  cout<<"Pop char stack::";
for(i=0;i<3;i++)
  cout<<s2.pop()<<"\n";
getch();
return 0;

}
/*
Output

Pop int stack::
7
```

```
5
2
Pop char stack::
c
b
a

*/
```

T is a type parameter and it can be any type. For example, Stack<Token>, where Token is a user defined class. T does not have to be a class type as implied by the keyword class. For example, Stack<int> and Stack<float>are valid instantiations, even though int and float are not "classes".

Creating and Using Templates

Implementing class template member functions

Implementing template member functions is somewhat different compared to the regular class member functions. The declarations and definitions of the class template member functions should all be in the same header file. The declarations and definitions need to be in the same header file. Consider the following.


```
//B.H
template <class t>
class b
{
public:
        b() ;
        ~b() ;
} ;
```

```
// B.CPP
#include "B.H"
template <class t>
b<t>::b()
{
}
template <class t>
b<t>::~b()
{
}
```

```
//MAIN.CPP
#include "B.H"
void main()
{
        b<int> bi ;
        b <float> bf ;
}
```

When compiling B.cpp, the compiler has both the declarations and the definitions available. At this point the compiler does not need to generate any definitions for template classes, since there are no instantiations. When the compiler compiles main.cpp, there are two instantiations: template class B<int> and B<float>. At this point the compiler has the declarations but no definitions!

Using a class template

Using a class template is easy. Create the required classes by plugging in the actual type for the type parameters. This process is commonly known as "Instantiating a class".

A good programming practice is using typedef while instantiating template classes. Then throughout the program, one can use the typedef name. There are two advantages:

- **typedef's** are very useful when "templates of templates" come into usage. For example, when instantiating an STL vector of int's, you could use:
- typedef vector<int, allocator<int> > INTVECTOR ;
- If the template definition changes, simply change the typedef definition. For example, currently the definition of template class vector requires a second parameter.
- typedef vector<int, allocator<int> > INTVECTOR ;
- INTVECTOR vi1 ;

The compiler generates a class, function or static data members from a template when it sees an implicit instantiation or an explicit instantiation of the template.

1. Consider the following sample. This is an example of implicit instantiation of a class template .

```
template <class T>
class Z
{
  public:
          Z() {} ;
         ~Z() {} ;
         void f(){} ;
         void g(){} ;
} ;

int main()
{
      Z<int> zi ;   //implicit instantiation generates class Z<int>
      Z<float> zf ; //implicit instantiation generates class Z<float>
      return 0 ;
}
```


2.Consider the following sample. This sample uses the template class members Z<T>::f() and Z<T>::g().

```
template <class T>
class Z
{
  public:
Z() {} ;
```

```
~Z() {} ;
void f(){} ;
void g(){} ;
} ;

int main()
{
Z<int> zi ; //implicit instantiation generates class Z<int>
zi.f() ;    //and generates function Z<int>::f()
Z<float> zf ; //implicit instantiation generates class Z<float>
zf.g() ;      //and generates function Z<float>::g()
return 0 ;
}
```

This time in addition to the generating classes Z<int> and Z<float>, with constructors and destructors, the compiler also generates definitions for Z<int>::f() and Z<float>::g(). The compiler does not generate definitions for functions, nonvirtual member functions, class or member class that does not require instantiation. In this example, the compiler did not generate any definitions for Z<int>::g() and Z<float>::f(), since they were not required.

3.Consider the following sample. This is an example of explicit instantiation of a class template.


```
template <class T>
class Z
{
  public:
Z() {} ;
~Z() {} ;
void f(){} ;
void g(){} ;
} ;

int main()
{
template class Z<int> ; //explicit instantiation of class Z<int>
template class Z<float> ; //explicit instantiation of
                          //class Z<float>
return 0 ;
}
```

4.Consider the following sample. Will the compiler generate any classes in this case? The answer is NO.

```
template <class T>
class Z
{
```

```
  public:
Z() {} ;
~Z() {} ;
void f(){} ;
void g(){} ;
} ;

int main()
{
Z<int>* p_zi ; //instantiation of class Z<int> not required
Z<float>* p_zf ; //instantiation of class Z<float> not required
return 0 ;
}
```

This time the compiler does not generate any definitions! There is no need for any definitions. It is similar to declaring a pointer to an undefined class or struct.

5.Consider the following sample. This is an example of implicit instantiation of a function template.

```
//max returns the maximum of the two elements
template <class T>
T max(T a, T b)
{
  return a > b ? a : b ;
}
void main()
{
int I ;
I = max(10, 15) ; //implicit instantiation of max(int, int)
char c ;
c = max('k', 's') ; //implicit instantiation of max(char, char)
}
```


In this case the compiler generates functions max(int, int) and max(char, char). The compiler generates definitions using the template function max.

6.Consider the following sample. This is an example of explicit instantiation of a function template.

```
template <class T>
void Test(T r_t)
{
}

int main()
{
//explicit instantiation of Test(int)
```

```
        template void Test<int>(int) ;
        return 0 ;
}
```

In this case the compiler would generate function Test(int). The compiler generates the definition using the template function Test.

7.If an instantiation of a class template is required, and the template declared but not defined, the program is ill-formed. VC5.0 compiler generates error C2079.

```
template <class T> class X ;

int main()
{
        X<int> xi ; //error C2079: 'xi' uses undefined class 'X<int>'
        return 0 ;
}
```

8.Instantiating virtual member functions of a class template that does not require instantiation is implementation defined. For example, in the following sample, virtual function X<T>::Test() is not required, VC5.0 generates a definition for X<T>::Test.

```
template <class T>
class X
{
        public:
                virtual void Test() {}
};

int main()
{
        X<int> xi ; //implicit instantiation of X<int>
        return 0 ;
}
```


In this case the compiler generates a definition for X<int>::Test, even if it is not required.

Class Template Specialization

In some cases it is possible to override the template-generated code by providing special definitions for specific types. This is called template specialization. The following example defines a template class specialization for template class stream.

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

```
template <class T>
class stream
{
        public:
                void f() { cout << "stream<T>::f()"<< endl ;}
} ;

template <>
class stream<char>
{
        public:
                void f() { cout << "stream<char>::f()"<< endl ;}
} ;

int main()
{
        stream<int> si ;
        stream<char> sc ;

        si.f() ;
        sc.f() ;
        return 0 ;
}
```


Output

```
stream<T>::f()
stream<char>::f()
```

In the above example, stream<char> is used as the definition of streams of chars; other streams will be handled by the template class generated from the class template.

Template Class Partial Specialization

You may want to generate a specialization of the class for just one parameter, for example

```
//base template class
template<typename T1, typename T2>

class X
{
} ;

//partial specialization
template<typename T1>
class X<T1, int>
{
} ; //C2989 here
```

```
int main()
{
        // generates an instantiation from the base template
        X<char, char> xcc ;

        //generates an instantiation from the partial specialization
X<char, int> xii ;

        return 0 ;
}
```

A partial specialization matches a given actual template argument list if the template arguments of the partial specialization can be deduced from the actual template argument list.

Template Parameters

1.C++ templates allow one to implement a generic Queue<T> template that has a type parameter T. T can be replaced with actual types, for example, Queue<Customers>, and C++ will generate the class Queue<Customers>. For example,

```
template <class T>
class Stack
{
} ;
```


Here T is a template parameter, also referred to as type-parameter.

2.C++ allows you to specify a default template parameter, so the definition could now look like:

```
template <class T = float, int elements = 100> Stack { ....} ;
```

Then a declaration such as

```
Stack<> mostRecentSalesFigures ;
```

would instantiate (at compile time) a 100 element Stack template class named mostRecentSalesFigures of float values; this template class would be of type Stack<float, 100>.

C++ also allows non-type template parameters. In this case, template class Stack has an int as a non-type parameter.

If you specify a default template parameter for any formal parameter, the rules are the same as for functions and default parameters. Once a default parameter is declared all subsequent parameters must have defaults.

3.Default arguments cannot be specified in a declaration or a definition of a specialization. For example,

```
template <class T, int size>
class Stack
{
} ;

//defined as a non-template class
template <class T, int size = 10>
class Stack<int, 10>
{
} ;

int main()
{
        Stack<float,10> si ;
            return 0 ;
}
```

4.A type-parameter defines its identifier to be a type-name in the scope of the template declaration, and canot be re-declared within its scope (including nested scopes). For example,


```
template <class T, int size>
class Stack
{
        int T ; //error type-parameter re-defined.
        void f()
        {
                    char T ; //error type-parameter re-defined.
        }
} ;
class A {} ;
int main()
{
        Stack<A,10> si ;
      return 0 ;
}
```

5.The value of a non-type-parameter cannot be assigned to or have its value changed. For example,

```
template <class T, int size>
class Stack
{
        void f()
```

```
        {
                //error C2105: '++' needs l-value
           size++ ; //error change of template argument value
        }
} ;


int main()
{
        Stack<double,10> si ;
      return 0 ;
}
```

6.A template-parameter that could be interpreted as either a parameter-declaration or a type-parameter, is taken as a type-parameter. For example,

```
class T {} ;
int i ;

template <class T, T i>
void f(T t)
{
        T t1 = i ; //template arguments T and i
        ::T t2 = ::i ; //globals T and i
}

int main()
{
        f('s') ; //C2783 here
        return 0 ;
}
```



```
    class T {} ;
    int i ;

    template <class T, T i>
    void f(T t)
    {
      T t1 = i ; //template arguments T and i
      ::T t2 = ::i ; //globals T and i
    }

    int main()
    {
      f<char, 's'>('s') ; //workaround
      return 0 ;
```

```
    }
```

7.A non-type template parameter cannot be of floating type. For example,

```
template <double d> class X ; //error C2079: 'xd' uses
                    //undefined class 'X<1.e66>'
//template <double* pd> class X ; //ok
//template <double& rd> class X ; //ok

int main()
{
    X<1.0> xd ;
    return 0 ;
}
```

Namespaces

Defining namespaces

In order to uniquely identify a namespace, use the namespace keyword.

Namespace syntax


```
>>-namespace--+------------+--{--namespace_body--}-------------><
              '-identifier-'
```

The *identifier* in an original namespace definition is the name of the namespace. The identifier may not be previously defined in the declarative region in which the original namespace definition appears, except in the case of extending namespace. If an identifier is not used, the namespace is an unnamed namespace.

Declaring namespaces

The identifier used for a namespace name should be unique. It should not be used previously as a global identifier.

```
namespace Raymond {
  // namespace body here...
  }
```

In this example, Raymond is the identifier of the namespace. If you intend to access a namespace's elements, the namespace's identifier must be known in all translation units.

Creating a namespace alias

An alternate name can be used in order to refer to a specific namespace identifier.

```
namespace INTERNATIONAL_BUSINESS_MACHINES {
  void f();
}

namespace IBM = INTERNATIONAL_BUSINESS_MACHINES;
```

In this example, the IBM identifier is an alias for INTERNATIONAL_BUSINESS_MACHINES. This is useful for referring to long namespace identifiers.

If a namespace name or alias is declared as the name of any other entity in the same declarative region, a compiler error will result. Also, if a namespace name defined at global scope is declared as the name of any other entity in any global scope of the program, a compiler error will result.

Extending namespaces

Namespaces are extensible. You can add subsequent declarations to a previously defined namespace. Extensions may appear in files separate from or attached to the original namespace definition. For example:


```
namespace X { // namespace definition
  int a;
  int b;
  }

namespace X { // namespace extension
  int c;
  int d;
  }

namespace Y { // equivalent to namespace X
  int a;
  int b;
  int c;
  int d;
  }
```

In this example, namespace X is defined with a and b and later extended with c and d. namespace X now contains all four members. You may also declare all of the required members within one namespace. This method is represented by namespace Y. This namespace contains a, b, c, and d.

Unnamed namespaces

A namespace with no identifier before an opening brace produces an unnamed namespace. Each translation unit may contain its own unique unnamed namespace. The following example demonstrates how unnamed namespaces are useful.

```
#include <iostream>

using namespace std;

namespace {
  const int i = 4;
  int variable;
  }

int main()
{
  cout << i << endl;
  variable = 100;
  return 0;
}
```

In the previous example, the unnamed namespace permits access to i and variable without using a scope resolution operator.

The following example illustrates an improper use of unnamed namespaces.

```
#include <iostream>

using namespace std;

namespace {
  const int i = 4;
  }

int i = 2;

int main()
{
  cout << i << endl; // error
  return 0;
}
```

Inside main, i causes an error because the compiler cannot distinguish between the global name and the unnamed namespace member with the same name. In order for the previous example to work, the namespace must be uniquely identified with an identifier and i must specify the namespace it is using.

You can extend an unnamed namespace within the same translation unit. For example:

```
#include <iostream>
```

```
using namespace std;

namespace {
  int variable;
  void funct (int);
  }

namespace {
  void funct (int i) { cout << i << endl; }
  }

int main()
{
  funct(variable);
  return 0;
}
```

both the prototype and definition for funct are members of the same unnamed namespace.

Note:
Items defined in an unnamed namespace have internal linkage. Rather than using the keyword static to define items with internal linkage, define them in an unnamed namespace instead.

Namespace member definitions

A namespace can define its own members within itself or externally using explicit qualification. The following is an example of a namespace defining a member internally:

```
namespace A {
  void b() { /* definition */ }
}
```

Within namespace A member void b() is defined internally.

A namespace can also define its members externally using explicit qualification on the name being defined. The entity being defined must already be declared in the namespace and the definition must appear after the point of declaration in a namespace that encloses the declaration's namespace.

The following is an example of a namespace defining a member externally:

```
namespace A {
  namespace B {
    void f();
  }
  void B::f() { /* defined outside of B */ }
```

```
}
```

In this example, function f() is declared within namespace B and defined (outside B) in A.

The using directive

A using directive provides access to all namespace qualifiers and the scope operator. This is accomplished by applying the using keyword to a namespace identifier.

Using directive syntax

```
>>-using--namespace--name--;---------------------------------><
```

The *name* must be a previously defined namespace. The using directive may be applied at the global and local scope but not the class scope. Local scope takes precedence over global scope by hiding similar declarations.

If a scope contains a using directive that nominates a second namespace and that second namespace contains another using directive, the using directive from the second namespace will act as if it resides within the first scope.


```
namespace A {
  int i;
}
namespace B {
  int i;
  using namespace A;
}
void f()
{
  using namespace B;
  i = 7; // error
}
```

In this example, attempting to initialize i within function f() causes a compiler error, because function f() cannot know which i to call; i from namespace A, or i from namespace B.

The using declaration and namespaces

A using declaration provides access to a specific namespace member. This is accomplished by applying the using keyword to a namespace name with its corresponding namespace member.

Using declaration syntax

>>-using--namespace--::--member------------------------------><

In this syntax diagram, the qualifier name follows the using declaration and the *member* follows the qualifier name. For the declaration to work, the member must be declared inside the given namespace. For example:

```
namespace A {
  int i;
  int k;
  void f;
  void g;
  }

using A::k
```

In this example, the using declaration is followed by A, the name of namespace A, which is then followed by the scope operator (::), and k. This format allows k to be accessed outside of namespace A through a using declaration. After issuing a using declaration, any extension made to that specific namespace will not be known at the point at which the using declaration occurs.

Overloaded versions of a given function must be included in the namespace prior to that given function's declaration. A using declaration may appear at namespace, block and class scope.

Explicit access

To explicitly qualify a member of a namespace, use the namespace identifier with a :: scope resolution operator.

Explicit access qualification syntax

>>-namespace_name--::--member--------------------------------><

For example:

```
namespace VENDITTI {
  void j()
};

VENDITTI::j();
```

In this example, the scope resolution operator provides access to the function j held within namespace VENDITTI. The scope resolution operator :: is used to access identifiers in both global and local namespaces. Any identifier in an application can be accessed with sufficient qualification. Explicit access cannot be applied to an unnamed namespace.

MODULE 5

Dynamic Memory Allocation

Memory is utilized by your program in three different ways.

1) All global and static variables are found in a region known as static memory. These variables "live" throughout the running of your program, and are automatically initialized to zero.

2) The creation of automatic variables at function or block scope occurs in an area known as the stack. As each variable is created, it is said to be pushed onto the stack, and when it goes out of scope, it gets popped off the stack. In addition, whenever you call upon a function, the formal arguments and any auto variables that the function may need are pushed onto the stack, and when the function exits, these variables are popped from the stack.

3) You may allocate memory at execution time from an area known as the heap, or free memory. Such memory is always unnamed, and therefore is addressed by a pointer that contains the starting address. A data object created here will remain in existence until it is explicitly destroyed. That is, the lifetime of an object is directly under our control and is unrelated to the block structure of the program.

The C++ keyword new

The new operator can be used to allocate memory at execution time. It takes the following general form:

pointer-variable = new data-type ;

Here, pointer-variable is a pointer of type data-type. The new operator allocates sufficient memory to hold a data object of type data-type and returns the address of the object.

Memory for a single instance of a primitive type

```
int *ptr1 = new int ;
float *ptr2 = new float ;
char *ptr3 = new char ;
```

Primitive types allocated from the heap via **new** can be initialized with some user-specified value by enclosing the value within parentheses immediately after the type name.

For example,

```
int *ptr = new int ( 65 ) ;
```

Memory for an array of a primitive type

To get memory for an array of some primitive type using new, write the keyword new followed by the number of array elements enclosed within square brackets.

```
int *ptr = new int [ 5 ] ;
for ( int i = 0 ; i < 5 ; i++ )
cin >> ptr [ i ] ;
```

Note that it is not possible to initialize the individual elements of an array created when using new. The best you can do is assign into them after the creation has occurred.

The C++ keyword delete

When a data object is no longer needed, it is destroyed to release the memory space for reuse. The delete keyword in C++ is used to release the space that was reserved by new.

The general form of its use is:

delete pointer-variable ;

The pointer-variable is the pointer that points to a data object created with new.

For example,

delete ptr ;

How to delete a single instance from the heap

To delete a single instance from the heap, write the keyword delete followed by the name of the pointer that points to the heap space.

For example,

```
int * ptr = new int ;

delete ptr ;
```

How to delete an array of instances from the heap

To delete an array of instances from the heap, write the keyword delete followed by empty brackets and the name of the pointer variable.

For example,

```
int * ptr = new int [ 10 ] ;
```

delete [ ] ptr ;

Dynamic Memory Allocation for a Single Instance of a User-defined Type

Memory for a single instance of a user-defined class can be obtained from the heap in the same fashion as that of a built-in type. That is, you write the keyword new followed by the class type.

For example,

* ptr = new Item ;

allocate one object of type Item.

The pointer returned by

new Item ;

is of correct type (pointer to Item) without the need for explicit casting.

In this example, the first step is to allocate space for the instance of Item and the second step is to call the appropriate constructor of the class Item.

Item * ptr = Item (17) ;

The parentheses following the class name, if present, supply arguments to the class constructor. If the parentheses are not present, as in

Item * ptr = new Item ;

then the class must either define a constructor that does not require arguments or define no constructors at all.

Dynamic Memory Allocation for an array of Objects

Programmer may also ask for more than one instance, that is, an array of objects of a particular type. The array is allocated from the free store by following the type specifier with a bracket-enclosed dimension. The dimension can be any expression.

For example,

Item * = new Item [5] ;

allocate space for 5 objects of type Item and call default constructor for all objects.

In particular, if a class lacks a default constructor, there are restrictions on how you can use that class. It is not usually possible to create array of objects.

If a class lacks a default constructor then,

```
Item * = new Item [ 5 ] ;
```

results in an error, no way to call Item constructors

```
// rememb-o-matic
#include <iostream>
#include <new>
using namespace std;

int main ()
{
  int i,n;
  int * p;
  cout << "How many numbers would you like to
type? ";
  cin >> i;
  p= new (nothrow) int[i];
  if (p == 0)
    cout << "Error: memory could not be
allocated";
  else
  {
    for (n=0; n<i; n++)
    {
      cout << "Enter number: ";
      cin >> p[n];
    }
    cout << "You have entered: ";
    for (n=0; n<i; n++)
      cout << p[n] << ", ";
    delete[] p;
  }
  return 0;
}
```

```
How many numbers would you like to type? 5
Enter number : 75
Enter number : 436
Enter number : 1067
Enter number : 8
Enter number : 32
You have entered: 75, 436, 1067, 8, 32,
```

Notice how the value within brackets in the new statement is a variable value entered by the user (i), not a constant value:

```
p= new (nothrow) int[i];
```

But the user could have entered a value for i so big that our system could not handle it. For example, when I tried to give a value of 1 billion to the "How many numbers" question, my system could not allocate that much memory for the program and I got the text message we prepared for this case (Error: memory could not be allocated). Remember that in the case that we tried to allocate the memory without specifying the nothrow parameter in the new expression, an exception would be thrown, which if it's not handled terminates the program.

It is a good practice to always check if a dynamic memory block was successfully allocated. Therefore, if you use the nothrow method, you should always check the value of the pointer returned. Otherwise, use the exception method, even if you do not handle the exception. This way, the program will terminate at that point without causing the unexpected results of continuing executing a code that assumes a block of memory to have been allocated when in fact it has not.

Some of the important points the programmer must note while using memory management operators are described below:

- The programmer must take care not to free or delete a pointer variable that has already been deleted.
.
- Overloading of new and delete operator is possible (to be discussed in detail in later section on overloading).
.
- We know that sizeof operator is used for computing the size of the object. Using memory management operator, the size of the object is automatically computed.
.
- The programmer must take care not to free or delete pointer variables that have not been allocated using a new operator.
.
- Null pointer is returned by the new operator when there is insufficient memory available for allocation.

Example: to understand the concept of new and delete memory management operator in C++:

```
#include <iostream.h>
void main()
{
//Allocates using new operator memory space in memory for storing a integer
datatype
int *a= new a;
*a=100;
cout << " The Output is:a="<<a;
//Memory Released using delete operator
```

```
delete a;

}
```

The output of the above program is

The Output is:a=100

In the above program, the statement:

int *a= new a;

Holds memory space in memory for storing a integer datatype. The statement:

*a=100

This denotes that the value present in address location pointed by the pointer variable a is 100 and this value of a is printed in the output statement giving the output shown in the example above. The memory allocated by the new operator for storing the integer variable pointed by a is released using the delete operator as:

delete a;

Dynamic Objects Allocation

Objects, called dynamic objects, may be created whose lifetimes are not bounded by the existance of the scope in which they were created. Thus, objects can be constructed in a function that continue to exist after the function returns; or objects can be created in an if-then construct that exists after the if-then constructed has been executed. Dynamic object give the programmer greater flexibility in managing objects. However, the programmer also assumes greater responsibility for insuring that dynamic objects are managed properly. Two problems that can occur with dynamic objects are destructing the object "too soon" and failing to destruct the objects at all.

A dynamic object is created using a "new" operator that returns a pointer to the newly constructed object and is destructed by a "delete" operator. A pointer variable is used to hold the pointer to the object that is returned by the "new" operator. The delete operator takes a pointer variable as an operand and destructs the object pointed to by that variable. The following example illustrates how dynamic objects can be created and deleted.

```
Frame *window, *view;              // declaration of pointer variables

window = new Frame("First", 10, 20, 50, 50);          // create a new Frame object
view   = new Frame("Second",70, 20, 50, 50);          // create a new Frame object

Frame *edit = new Frame ("Editor", 50, 75, 100, 100); // combine declaration of pointer
                                                      // variable and object construction

Frame *default = new Frame;                // use default constructor values

delete window;                    // destruct window Frame
delete view;                              // destruct view   Frame
delete edit;                              // destruct edit   Frame
```

In this example, window and view are declared as variables that point to Frame objects. It is important to realize that this declaration **does not** create two frames; it merely creates to variables that will point to Frame objects that are created sometime in the future. The two Frame objects are then created and the variables "window" and "view" are set pointing to these objects. The declaration of the pointer variable and the construction of the object to which it points can be combined in a single statement as shown with the edit and default Finally, the three Frame objects are deleted.

Notice the syntax of creating dynamic objects. The name of the class appears immediately after the keyword "new". The constructor arguments, if any, are given in parenthesis after the name of the class. When there are no constructor arguments, the parenthesis are omitted as shown in the creation of the default object.

An operation is applied to a dynamic object using the "->" operator. This "arrow" operator is two adjacent characters, a minus sign and a greater than symbol. An exaple of operating on a dynamic object is the following:

```
Frame* display = new Frame ("A Display", 10, 20, 100, 200);

display->MoveTo(50, 50);
display->Resize(200, 200);
```


In this example the display object is moved and resized using the arrow operator to apply the corresponding methods.

Inline Functions

When a function is declared **inline**, the function is expanded at the calling block. The function is not treated as a separate unit like other normal functions.

But a compiler is free to decide, if a function qualifies to be an inline function. If the inline function is found to have larger chunk of code, it will not be treated as an inline function, but as like other normal functions.

Inline functions are treated like macro definitions by the C++ compiler. They are declared with the keyword

inline as follows.

```
  int add(int x,int y);
inline int add(int x,int y)
{
   return x+y;
}
```

In fact, the keyword inline is not necessary. If the function is defined with its body directly and the function has a smaller block of code, it will be automatically treated as inline by the compiler.

As implied, inline functions are meant to be used if there is a need to repetitively execute a small block of code, which is smaller. When such functions are treated inline, it might result in a significant performance difference.

What is Inline Function?

Inline functions are functions where the call is made to inline functions. The actual code then gets placed in the calling program.

Reason for the need of Inline Function:

Normally, a function call transfers the control from the calling program to the function and after the execution of the program returns the control back to the calling program after the function call. These concepts of function saved program space and memory space are used because the function is stored only in one place and is only executed when it is called. This concept of function execution may be time consuming since the registers and other processes must be saved before the function gets called.

The extra time needed and the process of saving is valid for larger functions. If the function is short, the programmer may wish to place the code of the function in the calling program in order for it to be executed. This type of function is best handled by the inline function. In this situation, the programmer may be wondering "why not write the short code repeatedly inside the program wherever needed instead of going for inline function?" Although this could accomplish the task, the problem lies in the loss of clarity of the program. If the programmer repeats the same code many times, there will be a loss of clarity in the program. The alternative approach is to allow inline functions to achieve the same purpose, with the concept of functions.

What happens when an inline function is written?

The inline function takes the format as a normal function but when it is compiled it is compiled as inline code. The function is placed separately as inline function, thus adding readability to the source program. When the program is compiled, the code present in function body is replaced in the place of function call.

General Format of inline Function:

The general format of inline function is as follows:

inline datatype function_name(arguments)

The keyword inline specified in the above example, designates the function as inline function. For example, if a programmer wishes to have a function named exforsys with return value as integer and with no arguments as inline it is written as follows:

inline int exforsys( )

Example:

The concept of inline functions:

```
#include <iostream.h>
int exforsys(int);
void main( )
{
int x;
cout << "\n Enter the Input Value: ";
cin>>x;
cout<<"\n The Output is: " << exforsys(x);
}

inline int exforsys(int x1)
{
return 5*x1;
}
```


The output of the above program is:

Enter the Input Value: 10
The Output is: 50

The output would be the same even when the inline function is written solely as a function. The concept, however, is different. When the program is compiled, the code present in the inline function exforsys( ) is replaced in the place of function call in the calling program. The concept of inline function is used in this example because the function is a small line of code.

The above example, when compiled, would have the structure as follows:

```
#include <iostream.h>
int exforsys(int);
void main( )
{
int x;
cout << "\n Enter the Input Value: ";
```

```
cin>>x;
//The exforsys(x) gets replaced with code return 5*x1;
cout<<"\n The Output is: " << exforsys(x);
}
```

When the above program is written as normal function the compiled code would look like below:

```
#include <iostream.h>
int exforsys(int);
void main( )
{
int x;
cout << "\n Enter the Input Value: ";
cin>>x;
//Call is made to the function exforsys
cout<<"\n The Output is: " << exforsys(x);
}

int exforsys(int x1)
{
return 5*x1;
}
```

A programmer must make wise choices when to use inline functions. Inline functions will save time and are useful if the function is very small. If the function is large, use of inline functions must be avoided.

OTHER OBJECT ORIENTED LANGUAGES-JAVA

INTRODUCTION TO JAVA

The term *Java* actual refers to more than just a particular language like C or Pascal. Java encompasses several parts, including :

- **A high level language** – the Java language is a high level one that at a glance looks very similar to C and C++ but offers many unique features of its own.
- **Java bytecode** - a compiler, such as Sun's javac, transforms the Java language source code to bytecode that runs in the JVM.

- **Java Virtual Machine (JVM)** – a program, such as Sun's java, that runs on a given platform and takes the bytecode programs as input and interprets them just as if it were a physical processor executing machine code.

Sun provides a set of programming tools such as javac, java and others in a bundle that it calls a Java *Software Development Kit* for each version of the language and for different platforms such as Windows, Linux, etc.. Sun also provides a runtime bundle with just the JVM when the programming tools are not needed.

1.1 History of Java

Around 1990 James Gosling , Bill Joy and others at Sun Microsystems began developing a language called *Oak*. The wanted it primarily to control microprocessors embedded in consumer items such as cable set-top boxes, VCR's, toasters, and also for personal data assistants (PDA).However, as of 1993, interactive TV and PDA markets had failed to take off. Then the Internet and Web explosion began, so Sun shifted the target market to Internet applications and changed the name of the project to Java.By 1994 Sun's *HotJava* browser appeared. Written in Java in only a few months, it illustrated the power of *applets*, programs that run within a browser, and also the capabilities of Java for speeding program development. Riding along with the explosion of interest and publicity in the Internet, Java quickly received widespread recognition and expectations grew for it to become the dominant software for browser and consumer applications.

OBJECT ORIENTED FEATURES IN JAVA

The basic features that make Java a powerful and popular programming language are:

- **Platform Independence**
  - The *Write-Once-Run-Anywhere* ideal has not been achieved (tuning for different platforms usually required), but closer than with other languages.
- **Object Oriented**
  - Object oriented throughout - no coding outside of class definitions, including main().

- An extensive class library available in the core language packages.

- **Compiler/Interpreter Combo**
  - Code is compiled to bytecodes that are interpreted by a Java virtual machines (JVM) .
  - This provides portability to any machine for which a virtual machine has been written.
  - The two steps of compilation and interpretation allow for extensive code checking and improved security.

- **Robust**
  - Exception handling built-in, strong type checking (that is, all data must be declared an explicit type), local variables must be initialized.

- **Several dangerous features of C & C++ eliminated:**
  - No memory pointers
  - No preprocessor
  - Array index limit checking

- **Automatic Memory Management**
  - Automatic garbage collection - memory management handled by JVM.

- **Security**
  - No memory pointers
  - Programs runs inside the virtual machine sandbox.
  - Array index limit checking
  - Code pathologies reduced by
    - *bytecode verifier* - checks classes after loading
    - *class loader* - confines objects to unique namespaces. Prevents loading a hacked "java.lang.SecurityManager" class, for example.
    - *security manager* - determines what resources a class can access such as reading and writing to the local disk.

- **Dynamic Binding**
  - The linking of data and methods to where they are located, is done at run-time.

- New classes can be loaded while a program is running. Linking is done *on the fly*.
- Even if libraries are recompiled, there is no need to recompile code that uses classes in those libraries.

  This differs from C++, which uses static binding. This can result in *fragile* classes for cases where linked code is changed and memory pointers then point to the wrong addresses.

- **Good Performance**
  - Interpretation of bytecodes slowed performance in early versions, but advanced virtual machines with adaptive and just-in-time compilation and other techniques now typically provide performance up to 50% to 100% the speed of C++ programs.

- **Threading**
  - *Lightweight* processes, called threads, can easily be spun off to perform multiprocessing.
  - Can take advantage of multiprocessors where available
  - Great for multimedia displays.

- **Built-in Networking**
  - Java was designed with networking in mind and comes with many classes to develop sophisticated Internet communications.

SIMPLE EXAMPLES

```
class welcome
{
public static void main(String a[])
{
System.out.println("Welcome to Java");
}}
```

* **class welcome** - declares a class called welcome .
class is a keyword and declares that a new class definition follows.

welcome is a java identifier that specifies the name of the class to be defined
Every class definition begin with { and ends with matching }

* **public static void main(String a[])**
public –keyword public specifies an access specifies that declared main method as unprotected and therefore making it accessible to all other classes.

static –this method belongs to the entire class and not part of any objects of the class
main must be always declared as static since the interpreter uses this method before any objects are created.

void –main method does not return any value
main method expects an argument list in the parentheses , which passes the string values
Here String a[] declares a parameter named a which contains an array of objects of the class type String

* System.out.println
Java.lang package consist of a useful class System.It is used to access system level behaviour.The System package provides three basic streams
System.in-standard input stream
System.out-Standard output stream
System.err - Standard error stream

2)Add 2 Numbers

```
class add
{
public static void main(String arg[])
{int a,b,c;
a=5;
b=10;
c=a+b;
System.out.println("a+b="+c);
}}
```

3)Command line arguments

```
class hello
{
public static void main(String arg[])
{
System.out.println("Hello");
System.out.println(arg[0]);
}
```

}…….
javac hello.java
java hello Smitha
o/p Hello Smitha

4)Command Line arguments –using length function

```
class hello1
{
public static void main(String arg[])
{
for(int i=0;i<arg.length;i++)
{
System.out.println(arg[i]);
}}}
```

……
Javac hello1.java
Java hello1 C C++ Java
o/p
C
C++
Java

5)Compute Square root of a Number

```
import java.lang.Math;
class sqroot
{
public static void main(String arg[])
{
double y,x=25;
y=Math.sqrt(x);
System.out.println("Sqroot of "+ x +"="+y);
}
}
```

…….
sqrt is a method of Math class.
import statement instruct the interpreter to load **'Math'** class from the package **'lang'**

6)Factorial

```
class fact
{
  public static void main(String args[])
   {
    int i,fact=1;
    i=Integer.parseInt(args[0]);
    for(int f=i;f>0;f--)
    {
     fact=fact*f;
     }
      System.out.println("Fact="+fact);
}
}
```

o/p javac fact.java
java fact 5
Fact=120

7)Generate Series

```
class series
{
public static void main(String args[])
{
int a,b;
for(a=1;a<=5;a++)
{
for(b=a;b<=5;b++)
{
System.out.print(b+" ");
}
```

```
System.out.println(" ");
}}}
```

o/p :javac series.java
java series
1 2 3 4 5
2 3 4 5
3 4 5
4 5
5

8)Interactive Input

```
//accept.java
import java.io.*;
class accept
{
public static void main(String arg[])throws
IOException
{
InputStreamReader ir=new
InputStreamReader(System.in);
BufferedReader br=new BufferedReader(ir);
System.out.println("Enter your name");
String name=br.readLine();
System.out.println("Hello "+ name +" Good
morning");
}}
```

o/p

javac accept.java
java accept
Enter your name
Smith
Hello Smith Good Morning

Java.lang package consist of a useful class **System**.It is used to access system level behaviour.The System package provides three basic streams **System.in is used for standard input stream System.in** provides a useful method **read()** which reads characters from standard input such as keyboard.If we use **read method** ,you must specify the **java.io.IOException** class **for handling input errors** By default **Standard input is line buffered.**so you must hit enter key to sent the character to your program.Return **type of read is integer**.we can cast the integer type to char type.

CREATING AND USING CLASSES IN JAVA

A class can be considered as a collection of related objects. W can declare data and associated methods in a class. Since java is strictly object oriented each and every code in a java program must be written with in a class. The java file is stored with the name of that class.

1.Class declaration

The class declaration declares the name of the class along with other attributes. One could make a class public, or private in the declaration. One could also extends another class (we'll leave this for another lab) or implements an interface (we'll leave this for later as well.) For our purposes, just a few keywords will suffice: public and private.

Here is an example of a class declaration.

```
public class FishingPole
{

}
```

Simply, this class is named FishingPole, and is public. We can also gather one more fact by looking at this declaration. Since this class is declared public, we know that this class HAS to be held in a file called "FishingPole.java". You should place this file into the same directory as the program you write to use the class. This class could not, as is, be included in the same file that has another public class. As a rule, only one public class is allowed per file.

2 Class Body

The class body comes after the class declaration and is placed within curly braces. The class body declares all instance variables and class variables (known collectively as member variables) for the class. In addition, the class body declares and implements all instance variables and class member functions for the class.

Here is an example.

```
class FishingPole
{
   private string manufacturer;
   public void setManufacturer(String man)
   {
      manufacturer = man;
   }
```

}The string named manufacturer is a variable, and setManufacturer() is a method. (Later we will see that manufacturer is actually called an instance variable and setManufacturer is called a class method. At this point, just know that they are variables and methods.) A real class would probably have many variables and methods defined.

The variable named manufacturer will contain the name of the fishing pole manufacturer. This is the private data within the class that is protected from direct use.

The setManufacturer() method has an actual parameter named man which is a string and will contain the name of the fishing pole manufacturer. All that setManufacturer() does is to put the string that is in man into manufacturer. (By the way, at this point it is easiest to give the two variables different names.

3 Instantiating an object

A possible main program that creates a FishingPole object could be as follows.

```
public static void main(String args[])
{
  FishingPole myPole = new FishingPole();
  myPole.setManufacturer("Spalding");
}
```

FishingPole myPole creates a variable that can hold a reference to a FishingPole object. new FishingPole creates an object of type FishingPole, allocating memory space to that object. The reference to this object is then assigned to the myPole variable by the assignment operator (=). Next we have a call to the method setManufacturer(). myPole.setManufacturer("Spalding"); says to invoke (call) the setManufacturer() method on the myPole object. The actual parameter is the string "Spaulding".

Now the program starts executing the setManufacturer() method defined in the FishingPole class. First the reference to the actual parameter ("Spaulding") is placed in the formal parameter, man. (Note that the types of the two parameters are the same. "Spaulding" is a string and man is of type String. The types of corresponding parameters are normally the same.) All that the setManufacturer() method does is store the name into the class variable manufacturer. Now the manufacturer associated with object myPole will have the value "Spaulding" until it is changed by another call to setManufacturer().

Creating classes- Simple Example

```
//Main.java

class rectangle
```

```
{      int length;
       int width;
  void getdata(int x,int y)
     { length=x;
       width=y;
     }
int rectarea()
{
   int area=length*width;
   return area;
}
}
class Main
{
  public static void main(String[] args)
    {
 int area1;
 rectangle rect=new rectangle();
 rect.getdata(20,10);
 area1=rect.rectarea();
 System.out.println("Area="+area1) ;
     }
}
```


4. Constructors

Constructors are used when creating classes. They are used to initialize member variables when a class is instantiated. Given the above FishingPole class, instead of creating an object of type FishingPole, then setting the manufacturer name via the setManufacturer() method, it may be easier to set that name when declaring and instantiating the FishingPole object in the first place.With a properly written constructor, we wouldn't need the call to method setManufacturer().Constructors are always named the same as the class they are in.

```
class FishingPole

{    private String Manufacturer ;

   // Constructor

   public FishingPole(String man)
```

```
    {       Manufacturer = man;

    }

    public void setManufacturer(String Man)

    {       Manufacturer = Man;

    }
}
```

Now we could modify our code above, and shorten our task.

```
public static void main(String[] args)

{    FishingPole myPole = new FishingPole("Spalding");

}
```

Keep in mind that this example only used one parameter. This is not a rule. Your constructors can have 0 or more parameters. Whatever suits your programming logic is fine. When using a constructor, set up the formal parameter(s) in the constructor definition and then supply the corresponding actual parameter(s) the constructor needs when you instantiate that class. If you have more than one parameter, the first (leftmost) actual parameter is put into the first (leftmost) formal parameter, the second into the second, etc.

As an example, the FishingPole class could also have a year of manufacture (private int year;), model name (private String modelName;), and a recommended retail price (private double retailPrice;). A method for changing the price is also shown.

```
class FishingPole
{
    private String Manufacturer;
    private int year;
   private String modelName;
   private double retailPrice;

    // Constructor
    public FishingPole(String man, int manYear, String model, double price)
    {
```

```
    Manufacturer = man;
    year = manYear;
    modelName = model;
    retailPrice = price;
  }

  public void changePrice(double price)
  {
    retailPrice = price;;
  }
}

public static void main(String[] args)
{
  FishingPole myPole = new FishingPole("Spalding", 2002, "BassOmatic", 155.95);
  // To change only the retail price use changePrice()
  myPole.changePrice(135.95);

}}
```

STATIC CLASS/METHODS

There are two types of methods.

- **Instance methods** are associated with an object and use the instance variables of that object. This is the default.
- **Static methods** use no instance variables of any object of the class they are defined in. If you define a method to be static, you will be given a rude message by the compiler if you try to access any instance variables. You can access static variables, but except for constants, this is unusual. Static methods typically take all they data from parameters and compute something from those parameters, with no reference to variables. This is typical of methods which do some kind of generic calculation. A good example of this are the many utility methods in the predefined Math class.

1 Qualifying a static call

From outside the defining class, an instance method is called by prefixing it with an *object*, which is then passed as an implicit parameter to the instance method, eg, inputTF.setText("");

A static method is called by prefixing it with a *class name*, eg, Math.max(i,j);. Curiously, it can also be qualified with an object, which will be ignored, but the class of the object will be used.

Here is a typical static method.

```
class MyUtils {
   . . .
   //=======mean
   public static double mean(int[] p) {
      int sum = 0;  // sum of all the elements
      for (int i=0; i<p.length; i++) {
         sum += p[i];
      }
      return ((double)sum) / p.length;
   }//endmethod mean
   . . .
}
```


The only data this method uses or changes is from parameters (or local variables of course).

2 Purpose of using static

The above mean() method would work just as well if it wasn't declared static, as long as it was called from within the same class. If called from outside the class and it wasn't declared static, it would have to be qualified (uselessly) with an object. Even when used within the class, there are good reasons to define a method as static when it could be.

- **Documentation**. Anyone seeing that a method is static will know how to call it (see below). Similarly, any programmer looking at the code will know that a static method can't interact with instance variables, which makes reading and debugging easier.
- **Efficiency**. A compiler will usually produce slightly more efficient code because no implicit object parameter has to be passed to the method.

3 Calling static methods

There are two cases.

Called from within the same class

Just write the static method name. Eg,

```
// Called from inside the MyUtils class
double avgAtt = mean(attendance);
```


Called from outside the class

If a method (static or instance) is called from another class, something must be given before the method name to specify the class where the method is defined. For instance methods, this is the object that the method will access. For static methods, the class name should be specified. Eg,

```
// Called from outside the MyUtils class.
double avgAtt = MyUtils.mean(attendance);
```

If an object is specified before it, the object value will be ignored and the the class of the object will be used.

4 Accessing static variables

Although a static method can't access instance variables, it can access static variables. A common use of static variables is to define "constants". Examples from the Java library are Math.PI or Color.RED. They are qualified with the class name, so you know they are static. Any method, static or not, can access static variables. Instance variables can be accessed only by instance methods.

INHERITANCE IN JAVA

Inheritance is a major component of object-oriented programming. Inheritance will allow you to define a very general class, and then later define more specialized classes by simply adding some new details to the older more general class definition. This saves work, because the more specialized class *inherits* all the properties of the general class and you, the programmer, need only program the new features.

For example, you might define a class for vehicles that has instance variables to record the vehicle's number of wheels and maximum number of occupants. You might then define a class for automobiles, and let the automobile class *inherit* all the instance variables and methods of the class for vehicles. The class for automobiles would have added instance variables for such things as the amount of fuel in the fuel tank and the license plate number, and would also have some added methods. (Some vehicles, such as a horse and wagon, have no fuel tank and normally no license plate, but an automobile is a vehicle that has these "added" items.) You would have to describe the added instance variables and added methods, but if you use Java's inheritance mechanism, you would get the instance variables and methods from the vehicle class automatically.

Before we construct an example of inheritance within Java, we first need to set the stage with the following Programming Example.

Programming Example: A Person Class

Display 1 contains a simple class called Person. This class is so simple that the only property it gives a person is a name. We will not have much use for the class Person by itself, but we will use the class Person in defining other classes. So, it is important to understand this class.

5.1 A Base Class

```
public class Person
{
  private String name;
  public Person()
  {
    name = "No name yet.";
  }
     public Person(String initialName)
  {
    name = initialName;
  }
     public void setName(String newName)
  {
    name = newName;
  }
  public String getName()
  {
    return name;
  }
  public void writeOutput()
  {
    System.out.println("Name: " + name);
  }
  public boolean sameName(Person otherPerson)
  {
    return (this.name.equalsIgnoreCase(otherPerson.name));
  }
}
```


Most of the methods for the class Person are straightforward. Notice that the method setName and the constructor with one String parameter do the same thing. We need these two methods, even though they do the same thing, because only the constructor can be used after new when we create a new object of the class Person, but we need a different method, such as setName, to make changes to an object after the object is created.

The method sameName is similar to the equals methods we've seen, but since it uses a few techniques that you may not have completely digested yet, let's go over that definition, which we reproduce in what follows:

```
public boolean sameName(Person otherPerson)
```

```
{
  return (this.name.equalsIgnoreCase(otherPerson.name));
}
```

Recall that when the method sameName is used, there will be a calling object of the class Person and an argument of the class Person. The sameName method will tell whether the two objects have the same name. For example, here is some sample code that might appear in a program:

```
Person p1 = new Person("Sam");
System.out.println("Enter the name of a person:");
String name = SavitchIn.readLine();
Person p2 = new Person(name);
if (p1.sameName(p2))
  System.out.println("They have the same name.");
else
  System.out.println("They have different names.");
```


Consider the call p1.sameName(p2). When the method sameName is called, the this parameter is replaced with p1, and the formal parameter otherPerson is replaced with p2, so that the value of true or false that is returned is

```
p1.name.equalsIgnoreCase(p2.name)
```

Thus, the two objects are considered to have the same name (sameName will return true) provided the two objects have the same value for their name instance variables (ignoring any differences between uppercase and lowercase letters).

So, if the user enters Sam in response to the prompt. Enter the name of a person: then the output will be : They have the same name.

If instead of Sam, the user enters Mary, then the output will be : They have different names.

5.2 Derived Classes

Suppose we are designing a college record-keeping program that has records for students, faculty, and (non teaching) staff. There is a natural hierarchy for grouping these record types. They are all records of people. Students are one subclass of people. Another subclass is employees, which includes both faculty and staff. Students divide into two smaller subclasses: undergraduate students and graduate students. These subclasses my further subdivide into still smaller subclasses.

Although your program may not need any class corresponding to people or employees, thinking in terms of such classes can be useful. For example, all people have names, and the methods of initializing, outputting, and changing a name will be the same for student, staff, and faculty records. In Java, you can define a class called Person that includes instance variables for the properties that belong to all subclasses of people, such as students, faculty, and staff. The class definition can also contain all the methods that manipulate the instance variables for the class Person. In fact, we have already defined such a Person class in Display 1.

Display 2 contains the definition of a class for students. A student is a person, so we define the class Student to be a *derived* class of the class Person. A *derived class* is a class defined by adding instance variables and methods to an existing class.

A Derived Class

```
public class Student extends Person
{
  private int studentNumber;
   public Student()
  {
    super();//super is explained in a later section.
     studentNumber = 0;//Indicating no number yet
  }
   public Student(String initialName, int initialStudentNumber)
  {
    super(initialName);
    studentNumber = initialStudentNumber;
  }
  public void reset(String newName, int newStudentNumber)
  {
    setName(newName);
    studentNumber = newStudentNumber;
  }
  public int getStudentNumber()
```

```
  {
    return studentNumber;
  }
  public void setStudentNumber(int newStudentNumber)
  {
    studentNumber = newStudentNumber;
  }
  public void writeOutput()
  {
    System.out.println("Name: " + getName());
    System.out.println("Student Number : " + studentNumber);
  }
  public boolean equals(Student otherStudent)
  {
    return (this.sameName(otherStudent)
       && (this.studentNumber == otherStudent.studentNumber));
  }
}
```

The existing class that the derived class is built upon is called the *base class*. In our example, Person is the base class and Student is the derived class. The derived class has all the instance variables and methods of the base class, plus whatever added instance variables and methods you wish to add. If you look at Display 2, you will see that the way we indicate that Student is a derived class of Person is by including the phrase extends Person on the first line of the class definition, so that the class definition of Student begins.


```
public class Student extends Person;
```

When you define a derived class, you give only the added instance variables and the added methods. For example, the class Student has all the instance variables and all the methods of the class Person, but you do not mention them in the definition of Student. Every object of the class Student has an instance variable called name, but you do not specify the instance variable name in the definition of the class Student. The class Student (or any other derived class) is said to *inherit* the instance variables and methods of the base class that it extends.For example, suppose you create a new object of the class Student as follows:

```
Student s = new Student();
```

There is an instance variable s.name. Because name is a private instance variable, it is not legal to write s.name (outside of the definition of the class Person), but the instance variable is there, and it can be accessed and changed. Similarly, you can have the following method invocation: s.setName("Warren Peace");

The class Student inherits the method setName (and all the other methods of the class Person) from the base class Person.

A derived class, like Student, can also add some instance variables and/or methods to those it inherits from its base class. For example, Student adds the instance variable studentNumber and the methods reset, getStudentNumber, setStudentNumber, writeOutput, and equals, as well as some constructors. (But we will postpone the discussion of constructors until we finish explaining the other parts of these class definitions.)Display 3 contains a very small demonstration program to illustrate inheritance. Notice that the object s can invoke the method setName, even though this is a method of its base class Person. The class Student inherits setName from the class Person. The class Student also adds new methods. In the sample program, the object s of the class Student invokes the method setStudentNumber. The method setStudentNumber was not in the class Person.

1)Demonstrating Inheritance

```
public class InheritanceDemo
{  public static void main(String[] args)
  {
    Student s = new Student();
    s.setName("Warren Peace");
    //setName is inherited from the class Person.
    s.setStudentNumber(2001);
    s.writeOutput();
  }}
```

Screen Output

Name: Warren Peace

Student Number: 2001

2)Inheritance

```
 // in.java
class base
{
  int x,y;
   void viewxy(int a ,int b)
   {
     x=a;
     y=b;
    System.out.println("x="+x+"y="+y);
   }
   void baseadd()
     {
       System.out.println("x+y="+(x+y));
      }
     }
class derived extends base
{
 int z;
  void viewxyz(int a,int b,int c)
     {     x=a;
           y=b;
           z=c;
System.out.println("x="+x+"y="+y+"z="+z);
      }
   void derivedadd()
    {
     System.out.println("x+y+z="+(x+y+z));
    }
}
class in
{
public static void main(String a[])
{
//base b=new base();
//b.viewxy(2,2);
//b.baseadd();
derived d=new derived();
System.out.println("Base class values");
d.viewxy(2,2);                          //base class method
d.baseadd();
System.out.println("Derived class values");
d.viewxyz(4,5,6);                       //derived class method
d.derivedadd();
}}
```

3)using constructor and super keyword

```
class base
{
int x,y;
base(int a ,int b)                    //base class constructor
{
x=a;
y=b;
System.out.println("x="+a+"y="+b);
}
void baseadd()
{
System.out.println("x+y="+(x+y));
}}
class derived extends base                //inheriting base class
{
int z;
derived(int a,int b,int c)        //derived class constructor
{
super(a,b);                               //invoking base class constructor
z=c;
System.out.println("z="+z);
}
void derivedadd()
{
System.out.println("x+y+z="+(x+y+z));
}}
class in1
{public static void main(String a[])
{
derived d=new derived(5,6,7);             //creating derived class object
//base b=new base(3,3);
System.out.println("Base class values");
d.baseadd();                              //base class method
System.out.println("Derived class values");
d.derivedadd();                           //derived class method
}}
```


o/p

x=5 y=6 z=7

base class values

x+y=11

derived class values

x+y+z=18

The base class members can be accessed using the derived class object 'd'.Instead of initializing x,y,z using using viewxy() and viewxyz() method ,we can use constructor 'base()' and constructor 'derived() ' to assign the values.If we are using the constructor method we can assign the values of x'y'z while creating the object.

Ex) Instead of this line **d.viewxyz(4,5,6)** we can use the following line.

derived d=new derived(5,6,7);

The constructor in the derived class uses the **super** keyword to invoke the constructor method of the base class.The above line first calls the derived constructor method which in turn calls the base constructor method using **super** keyword.

The subclass constructor is used to construct instance variables of both the subclass and the superclass.The keyword super is used ,subject to the following conditions.

1)**super** may only be used within the derived class constructor method

2)The call to the base class constructor must appear as the first statement within the derived class constructor.

3)the parameters in the **super** call must match the order and type of the instance variable declared in the super class.

POLYMORPHISM IN JAVA

Overriding Method Definitions

In the definition of the class Student, we added a method called writeOutput that has no parameters. But, the class Person also has a method called writeOutput that has no parameters. If the class Student were to inherit the method writeOutput from the base class Person, then Student would contain two methods with the name writeOutput, both of which have no parameters. Java has a rule to avoid this problem. If a derived class defines a method with the same name as a method in the base class and that *also has the same number and types of parameters as in the base class,* then the definition in the derived class is said to *override* the definition in the base class, and the definition in the derived class is the one that is used for objects of the derived class.

For example, in Display 3, the following invocation of the method writeOutput for the object s of the class Student will use the definition of writeOutput in the class Student, not the definition of the method writeOutput in the class Person:

s.writeOutput();

Although you can change the body of an overridden method definition to anything you wish, you cannot make any changes in the heading of the overridden method. In particular, when overriding a method definition, you cannot change the return type of the method.

Overriding versus Overloading

Do not confuse method overriding with method overloading. When you override a method definition, the new method definition given in the derived class has the exact same number and types of parameters. On the other hand, if the method in the derived class were to have a different number of parameters or a parameter of a different type from the method in the base class, then the derived class would have both methods. That would be overloading. For example, suppose we added the following method to the definition of the class Student:


```
public String getName(String title)
{
  return (title + getName());}
```

In this case, the class Student would have two methods named getName: It would still inherit the method getName from the base class Person (refer to Display 1), and it would also have the method named getName that we just defined. This is because these two methods called getName have different numbers of parameters.

Overriding

```
class base
{
int x;
base(int a)
{
x=a;
}
void display()
{System.out.println("x="+x);
}}
class derived extends base
{
int y;
```

```
derived(int a,int b)
{
super(a);
y=b;
}
void display()     //method defined again
{
System.out.println("y="+y);
}
}
class over
{
public static void main(String a[])
{
derived d=new derived(100,200);
d.display();
}
}
```

O/P

Y=200

Overriding

```
class base
{
int x;
base(int a)
{
x=a;
}
void display()
{
System.out.println("x="+x);
}}
class derived extends base
{
int y;
derived(int a,int b)
{
super(a);
y=b;
}
void display()
{super.display()  //call to base class display()
System.out.println("y="+y);
}
}
class over
{
public static void main(String a[])
{
derived d=new derived(100,200);
d.display();
}}
```

o/p

x=100 y=200

Multilevel Inheritance

```
 class x
{
x()
{
System.out.println("i AM FIRST");
}
}
class y extends x
{
y()
{
  System.out.println("i AM second");
}
}
class z extends y
{
  z()
{
  System.out.println("i AM last");
}
}
class multi
{  public static void main(String a[])
     {
       z z1=new z();
      }
 }
```

o/p I am first
I am second
I am last

FINAL METHODS, VARIABLES AND CLASSES

In the Java programming language, the final keyword is used in several different contexts to define an entity which cannot later be changed.

A final class cannot be subclassed. This is done for reasons of security or efficiency. Accordingly, many of the Java standard library classes are final, for example java.lang.System and java.lang.String. All methods in a final class are implicitly final.

Example:

```
public final class MyFinalClass {...}
```

A final method cannot be overridden by subclasses. This is done for reasons of efficiency, since the method can then be placed inline wherever it is called.

Example:

```
public class MyClass {
   public final void myFinalMethod() {...}
}
```


A final variable is a constant. It can only be assigned a value once.

Example:

```
public class MyClass {
    public static final double PI = 3.141592653589793;
}
```

INTERFACES

An interface in the Java programming language is an abstract type which is used to specify an interface (in the generic sense of the term) that classes must implement. Interfaces are declared using the interface keyword, and may only contain method signatures and constant declarations (variable declarations which are declared to be both static and final).

As interfaces are abstract, they cannot be directly instantiated. Object references in Java may be specified to be of an interface type; in which case they must either be null, or be bound to an object which implements the interface.The keyword implements is used to declare that a given class implements an interface. A class which implements an interface must either implement all methods in the interface, or be an abstract class.

One benefit of using interfaces is that they simulate multiple inheritance. All classes in Java (other than java.lang.Object, the root class of the Java type system) must have exactly one base class; multiple inheritance of classes is not allowed. However, a Java class may implement any number of interfaces.

Uses of Interfaces

Interfaces are used to collect like similarities which classes of various types share, but do not necessarily constitute a class relationship. For instance, a human and a parrot can both whistle, however it would not make sense to represent Humans and Parrots as subclasses of a Whistler class, rather they would most likely be subclasses of an Animal class (likely with intermediate classes), but would both implement the Whistler interface.

Another use of interfaces is being able to use an object without knowing its type of class, but rather only that it implements a certain interface. For instance, if one were annoyed by a whistling noise, one may not know whether it is a human or a parrot, all that could be determined is that a whistler is whistling. In a more practical example, a sorting algorithm may expect an object of type Comparable. Thus, it knows that the object's type can somehow be sorted, but it is irrelevant what the type of the object is.

Defining an Interface

Interfaces must be defined using the following formula (compare to Java's class definition).

```
[visibility] interface InterfaceName [extends other interfaces] {
    constant declarations
    abstract method declarations
}
```

The body of the interface contains abstract methods, but since all methods in an interface are, by definition, abstract, the abstract keyword is not required. Since the interface specifies a set of exposed behaviours, all methods are implicitly public.

Thus, a simple interface may be

```
public interface Predator {
    boolean chasePrey(Prey p);
    void eatPrey(Prey p);
}
```

Implementing an Interface

The syntax for implementing an interface uses this formula:

... implements InterfaceName[, another interface, another, ...] ...

Classes may implement an interface. For example,

```
public class Cat implements Predator {
    public boolean chasePrey(Prey p) {
        // programming to chase prey p
    }
    public void eatPrey (Prey p) {
        // programming to eat prey p
    }}
```


If a class implements an interface and is not abstract, and does not implement all its methods, this will result in a compiler error. If a class is abstract, one of its subclasses is expected to implement its unimplemented methods.

Classes can implement multiple interfaces

```
public class Frog implements Predator, Prey { ... }
```

Interfaces are commonly used in the Java language for callbacks. Java does not allow the passing of methods (procedures) as arguments. Therefore, the practice is to define an interface and

use it as the argument and use the method signature knowing that the signature will be later implemented.

First a simple link List for the example

```
public class IntLinker {
    public int start;
    public IntLinker point;
        public IntLinker (int start, IntLinker point) {
        this.start = start; this.point = point;
    }
}
```

Second, the Interface

```
public interface IntMethod {
    int theMethod(int var);
}
```


Third, the final use of the IntMethod (not yet defined, but used)

```
public class IntFinal {
    IntLinker build (IntMethod var, IntLinker builder){
        return new IntLinker ( var.theMethod (builder.start), builder.point);
    }}
```

Then, the only code written at some later point defining theMethod by implementing IntMethod

Thus, a callback

```
public class ImplMethod implements IntMethod{
    public int theMethod(int positive) { return Math.abs(positive);}

    public static void main(String[] args) {
        IntFinal i = new IntFinal();
        IntLinker t = i.build(new ImplMethod(), new IntLinker(-11, new IntLinker(2, null)));
```

System.out.println(t.start); System.out.println(t.point.start);

}}

About I*nterface*

#Java does not support multiple inheritance

#classes in java cannot have more than one super class.

class A extends B extends C

{……

}is not permitted in Java

#Java provides an alternate approach known as interface to support the concept of multiple inheritanceAn interface is basically a kind of class.***interface contains Abstract methods and final fields***.interface does not provide any code to implement this methods and data fields contain constant values.It is the responsibility of the class that implements an interface to define the code for implantation of these methods

#Syntax

interface *Interfacename*

*{*variable decln;

method decln;

}

#variables are declared as

final type variablename=value; all variables are declared as constants

#method declaration contain only a list of methods without any body statements

return-type methodname(parameter-list)

Example

```
class test
{
int mark1,mark2;
void getmark(int m1,int m2)
{
mark1=m1;
mark2=m2;
}
```

```
void putmarks()
{
System.out.println("Marks");
System.out.println("Marks1="+mark1);
System.out.println("Marks2="+mark2);
}}
interface sports
{
final int sportwt=6;
void putwt();
}
class result extends test implements sports
{int total;
public void putwt()
{System.out.println("sportwt="+sportwt);
}
void display()
{total=mark1+mark2+sportwt;
putmarks();
putwt();
System.out.println("Total Score="+total);
}}
class inter4
{
public static void main(String a[])
{
result st=new result();
st.getmark(27,33);
st.display();
}}
```


O/P

mark1=27
mark2=33
spwt=6
total=66

Example I*nterface1*

```
interface Area
{
final static float pi=3.14F;
float compute(float x,float y);
}
```

```
class rectangle implements Area
{
public float compute(float x,float y)
{
return (x*y);
}
}
class circle implements Area
{
public float compute(float x,float y)
{
return (pi*x*x);
}
}
class inter
{
public static void main(String a[])
{
rectangle rect=new rectangle();
circle cir=new circle();
Area area;
area=rect;
System.out.println("Area of Rectangle="+area.compute(10,20));
area=cir;
System.out.println("Area of circle="+area.compute(10,0));
}
}
```


The interface approach enables you to inherit method descriptions not implementations *Interface are used to to define a standard behaviour that can be implemented by any class anywhere in the class hierarchy.*

For example the shape rectangle and circle.both of them require common function area().

If we use an interface for Area,we can implement that interface in both rect and circle class

NESTED CLASSES

1 Named Inner Classes

An inner class is a class defined inside another class.

An inner class object is scoped not to the outer class, but to the instantiating object. Even if the inner class object is removed from the outer class object that instantiated it, it retains the scoping to its creator. If an object has a reference to another object's ("parent object"'s) inner class object, the inner class object is dealt with by that object as a separate, autonomous object, but in fact, it still has access to the private variables and methods of its parent object.

An inner class instantiation always remembers who it's parent object is and if anything changes in the parent object (e.g. one of its property's value changes), that change is reflected in all its inner class instantiations.

Inner classes are very useful in factory pattern situations.

Syntax:

```
[normal class declaration]
{
    [scoping] class [inner class's name] [extends ....] [ implements ...]
    {
        // properties and methods of the inner class
    }     // rest of properties and methods of parent class
}
```


INNER CLASS

Inner classes let you define one class within another. They provide a type of scoping for your classes since you can make one class *a member of another class*. Just as classes

have member *variables* and *methods*, a class can also have member *classes*. You can define an inner class within the curly braces of the outer class, as follows:

```
class MyOuter {
class MyInner { }
}
```

And if you compile it,javac MyOuter.java

you'll end up with *two* class files:MyOuter.class and MyOuter$MyInner.class

The inner class is still, in the end, a separate class, so a class file is generated. But the inner class file isn't accessible to you in the usual way. You can't, for example, say

%java MyOuter$MyInner

in hopes of running the main method of the inner class, because a *regular* inner class can't have static declarations of any kind. *The only way you can access the inner class is through a live instance of the outer class!* In other words, only at runtime when there's already an instance of the outer class to tie the inner class instance to. You'll see all this in a moment. First, let's beef up the classes a little:


```
class MyOuter {
private int x = 7;
// inner class definition
class MyInner {
public void seeOuter() {
System.out.println("Outer x is " + x);
}
} // close inner class definition
} // close outer class
```

The preceding code is perfectly legal. Notice that the inner class is indeed accessing a private member of the outer class. That's fine, because the inner class is also a member of the outer class. So just as any member of the outer class

To instantiate an instance of an inner class, *you must have an instance of the outer*

class to tie to the inner class.

Creating an Inner Class Object from Outside the Outer Class

Instance Code

If we want to create an instance of the inner class, we must have an instance of the outer class.

it means you can't instantiate the inner class from a static method of the outer class (because, don't forget, in static code *there is no* this *reference*) or from any other code in any other class. Inner class instances are always handed an implicit reference to the outer class. The compiler takes care of it, so you'll never see anything but the end result—the ability of the inner class to access members of the outer class.

The code to make an instance from anywhere outside nonstatic code of the outer class is simple, but you must memorize this for the exam!

```
public static void main (String[] args) {
MyOuter mo = new MyOuter();
MyOuter.MyInner inner = mo.new MyInner();
inner.seeOuter();
}
```


Here's a quick summary of the differences between inner class instantiation code that's *within* the outer class (but not static), and inner class instantiation code that's *outside* the outer class:

■ **From inside the outer class instance code**, use the inner class name in the normal way:

```
MyInner mi = new MyInner();
```

■ **From outside the outer class instance code (including static method code within the outer class)**, the inner class name must now include the outer class name, MyOuter.MyInner

and to instantiate, you must use a reference to the outer class,

new MyOuter().new MyInner(); or outerObjRef.new MyInner();

if you already have an instance of the outer class.

 # program MyOut.java

```
class MyOut {
private int x = 7;
// inner class definition
class MyInner {
public void seeOuter() {
System.out.println("Outer x is " + x);
}
} // close inner class definition
public static void main (String[] args) {
MyOut mo = new MyOut();
MyOut.MyInner inner = mo.new MyInner();
inner.seeOuter();}}
```

……..

```
import java.awt.*;
import java.applet.*;
import java.awt.event.*;
```



```
public class event5 extends Applet
{
public void init()
{
addMouseListener(new myAdapter());
}
class myAdapter extends MouseAdapter
{
public void mousePressed(MouseEvent e)
{
showStatus("Mouse Pressed");
}
}}
/*
<applet code=event5.class width=250 height=200>
</applet>*/
```

2 Anonymous Inner Classes

Anonymous inner class is an inner class with no name.

Example: actionPerformed events utilize the Observer-Observable pattern -- An instance of the observer is added to the observable. JBuilder creates an anonymous inner class to give to the button as its ActionListener.

Anonymous inner classes are very similar to named inner classes: Anonymous inner classes can override methods of the superclass. Anonymous inner classes are scoped *inside* the private scoping of the outer class. They can access the internal (private) properties and methods of the outer class. References to an inner class can be passed to other objects. Note that it still retains its scope.

But there are some important differences:

Anonymous inner classes must use the no parameter constructor of the super class. Since an object made from the inner class has a "life" independent of its outer class object, there is a problem with accessing local variables, especially method input parameters.

Two ways to pass initializing parameters to an inner class: Initialize a property that the inner class then uses -- properties have the cross-method-call persistence that local variables lack. Make the local variable "final" -- compiler will automatically transfer the value to a more persistent portion of the computer memory. Disadvantage: the value cannot be changed. **Usages:** Very useful for controlled access to the innards of another class. Very useful when you only want one instance of a special class.

Syntax:

```
[variable_type_superclass =]  new superclass_name()  {
                                        // properties and methods
                                } [;]
```

Examples:

1 Timer object with anonymous ActionListener:

```
Timer timer = new Timer( 200, new ActionListener() {
   public void actionPerformed(ActionEvent e)
   {// do something
   }
});
```

2 Initialization using a final input parameter (each Object made will have the name that was used when makeObj() was called to make it.)

```
public MyClass makeObj(final String name)
{
    return new SubClassOfMyClass() {
        public String toString()
        {               return "My name is "+name;
        }    };
}
```


3 Initialization using final local variable

```
public Object makeObj(String name)
{
    final String objName = "My name is " + name;

    return new Object() {
        public String toString()
        {
            return objName;
        }    };}
```

4 Initialization using initialization block

```
public Object makeObj(String name)
{
      return new Object() {
        String objName;
        {
            objName = "My name is " + name;
         }
         public String toString()
        {
             return objName;
        }     };
}
```

An anonymous Inner class is one that is not assigned a name. Consider the applet shown below. It's goal is to display the string "Mouse Pressed" in the status Bar of the AppletViewer or browser when the mouse is pressed.The init() method calls the addMouseListener() method. Its argument is an expression that defines and instantiates an anonymous inner class.The syntax new MouseAdapter(){….} indicates to the compiler that the code between the braces defines an anonymous inner class. Furthermore the class extends the MouseAdapter.This new class is not named, but it is automatically instantiated when this expression is executed, because this anonymous inner class is defined within the scope of event6 class. Therefore it can call the showStatus() method directly.


```
import java.awt.*;
import java.applet.*;

import java.awt.event.*;
public class event6 extends Applet
{
public void init()
{
addMouseListener(new MouseAdapter(){
public void mousePressed(MouseEvent e)
{
showStatus("Mouse Pressed");
}});
}}
/*
```

```
<applet code=event6.class width=250 height=200>
</applet>*/
```

CREATING AND USING PACKAGES

Packages are nothing more than the way we organize files into different directories according to their functionality, usability as well as category they should belong to. An obvious example of packaging is the JDK package from SUN (java.xxx.yyy) as shown below:

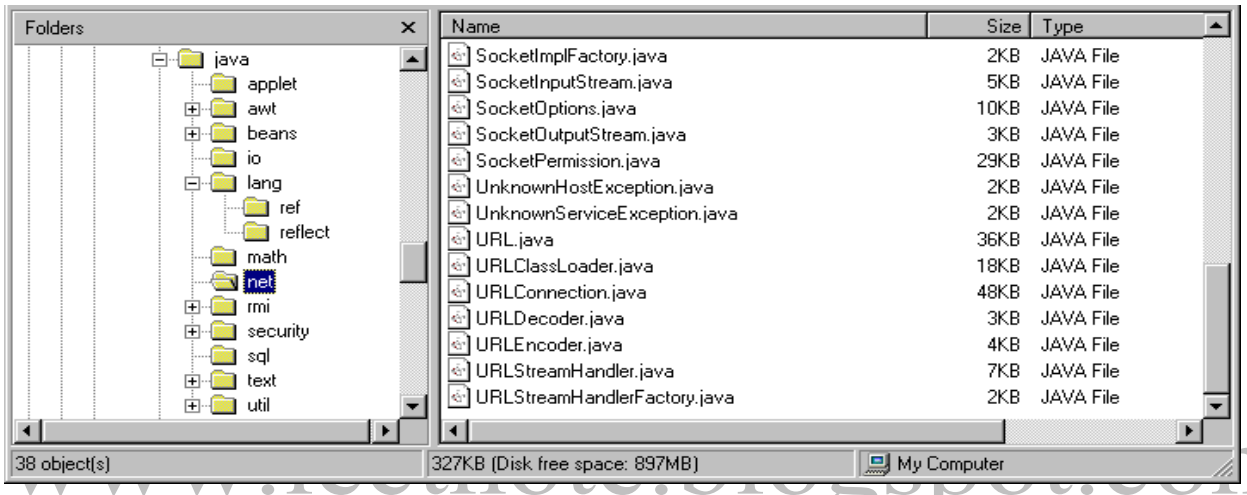


Figure 1. Basic structure of JDK package

Basically, files in one directory (or package) would have different functionality from those of another directory. For example, files in java.io package do something related to I/O, but files in java.net package give us the way to deal with the Network. In GUI applications, it's quite common for us to see a directory with a name "ui" (user interface), meaning that this directory keeps files related to the presentation part of the application. On the other hand, we would see a directory called "engine", which stores all files related to the core functionality of the application instead.

Packaging also help us to avoid class name collision when we use the same class name as that of others. For example, if we have a class name called "Vector", its name would crash with the Vector class from JDK. However, this never happens because JDK use java.util as a package name for the Vector class (java.util.Vector). So our Vector class can be named as "Vector" or we can put it into another package like com.mycompany.Vector without fighting with anyone. The benefits of using

package reflect the ease of maintenance, organization, and increase collaboration among developers. Understanding the concept of package will also help us manage and use files stored in jar files in more efficient ways.

11.1 Creating a package

Suppose we have a file called HelloWorld.java, and we want to put this file in a package **world**. First thing we have to do is to specify the keyword **package** with the name of the package we want to use (**world** in our case) on top of our source file, before the code that defines the real classes in the package, as shown in our HelloWorld class below:

```
// only comment can be here
package world;
public class HelloWorld {
  public static void main(String[] args) {
    System.out.println("Hello World");
  }}
```


One thing you must do after creating a package for the class is to create nested subdirectories to represent package hierachy of the class. In our case, we have the **world** package, which requires only one directory. So, we create a directory **world** and put our HelloWorld.java into it.

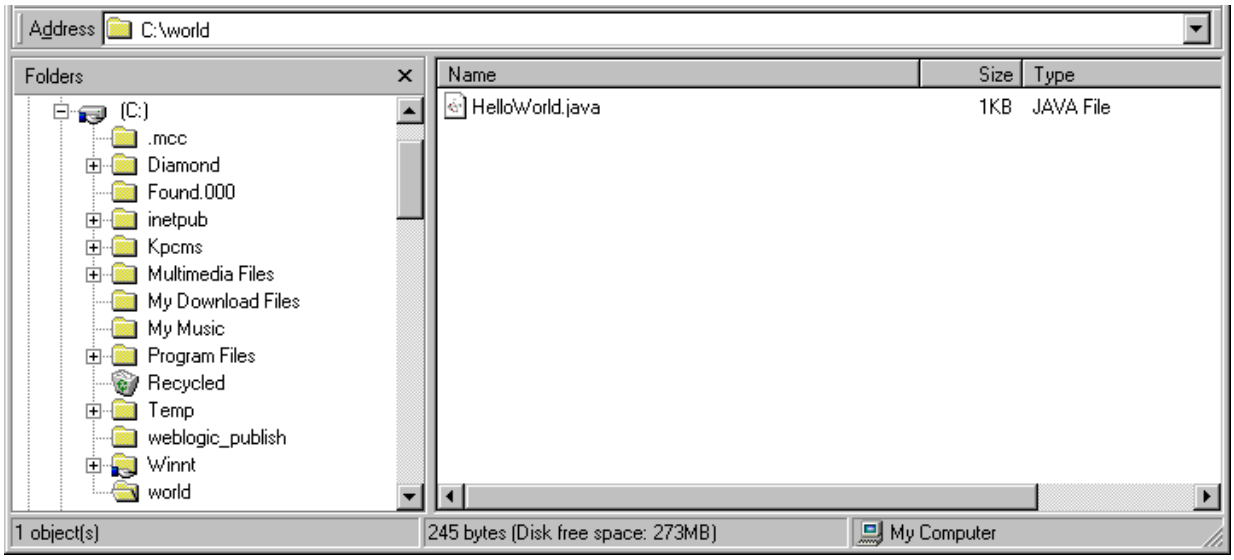


Figure 2. HelloWorld in world package (C:\world\HelloWorld.java)

That's it!!! Right now we have HelloWorld class inside world package. Next, we have to introduce the world package into our **CLASSPATH**.

11.1.1 Setting up the CLASSPATH

From figure 2 we put the package world under C:. So we just set our CLASSPATH as:

set CLASSPATH=.;C:\;

We set the CLASSPATH to point to 2 places, . (dot) and C:\ directory. Note: If you used to play around with DOS or UNIX, you may be familiar with . (dot) and .. (dot dot). We use . as an alias for the current directory and .. for the parent directory. In our CLASSPATH we include this . for convenient reason. Java will find our class file not only from C: directory but from the current directory as well. Also, we use ; (semicolon) to separate the directory location in case we keep class files in many places.

When compiling HelloWorld class, we just go to the world directory and type the command:

C:\world\javac HelloWorld.java

If you try to run this HelloWorld using **java HelloWorld**, you will get the following error:

```
C:\world>java HelloWorld
Exception in thread "main" java.lang.NoClassDefFoundError: HelloWorld (wrong name:
world/HelloWorld)
        at java.lang.ClassLoader.defineClass0(Native Method)
        at java.lang.ClassLoader.defineClass(ClassLoader.java:442)
        at java.security.SecureClassLoader.defineClass(SecureClassLoader.java:101)
        at java.net.URLClassLoader.defineClass(URLClassLoader.java:248)
        at java.net.URLClassLoader.access$1(URLClassLoader.java:216)
        at java.net.URLClassLoader$1.run(URLClassLoader.java:197)
        at java.security.AccessController.doPrivileged(Native Method)
        at java.net.URLClassLoader.findClass(URLClassLoader.java:191)
        at java.lang.ClassLoader.loadClass(ClassLoader.java:290)
        at sun.misc.Launcher$AppClassLoader.loadClass(Launcher.java:286)
        at java.lang.ClassLoader.loadClass(ClassLoader.java:247)
```

The reason is right now the HelloWorld class belongs to the package world. If we want to run it, we have to tell JVM about its **fully-qualified class name** (world.HelloWorld) instead of its plain class name (HelloWorld).


```
C:\world>java world.HelloWorld
C:\world>Hello World
```

Note: **fully-qualified class name** is the name of the java class that includes its package name

To make this example more understandable, let's put the HelloWorld class along with its package (world) be under **C:\myclasses** directory instead. The new location of our HelloWorld should be as shown in Figure 3:

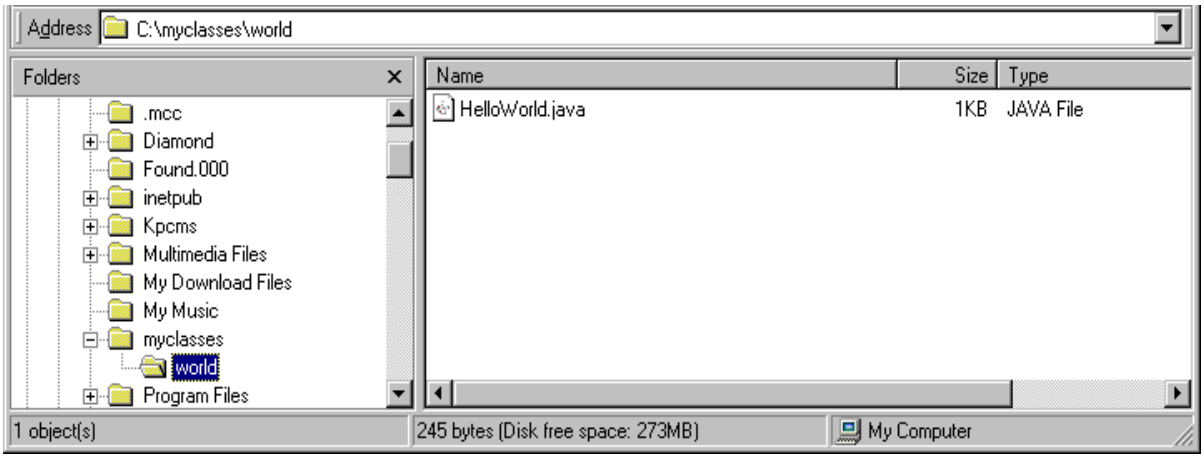


Figure 3. HelloWorld class (in world package) under **myclasses** directory

We just changed the location of the package from C:\world\HelloWorld.java to C:\myclasses\world\HelloWorld.java. Our CLASSPATH then needs to be changed to point to the new location of the package world accordingly.

set CLASSPATH=.;**C:\myclasses;**

Thus, Java will look for java classes from the current directory and **C:\myclasses** directory instead.

Someone may ask "Do we have to run the HelloWorld at the directory that we store its class file everytime?". The answer is NO. We can run the HelloWorld from anywhere as long as we still include the package world in the CLASSPATH. For example,

C:\>set CLASSPATH=.;C:\;

C:\>set CLASSPATH *// see what we have in CLSSPATH*

CLASSPATH=.;C:\;

C:\>cd world

C:\world>java world.HelloWorld

Hello World

C:\world>cd ..

C:\>java world.HelloWorld

Hello World

11.2 Subpackage (package inside another package)

Assume we have another file called **HelloMoon.java**. We want to store it in a subpackage **"moon"**, which stays inside package **world**. The HelloMoon class should look something like this:

```
package world.moon;

public class HelloMoon {
  private String holeName = "rabbit hole";
    public getHoleName() {
    return hole;
  }
    public setHole(String holeName) {
    this.holeName = holeName;
  }
}
```


If we store the package world under C: as before, the **HelloMoon.java** would be **c:\world\moon\HelloMoon.java** as shown in Figure 4 below:

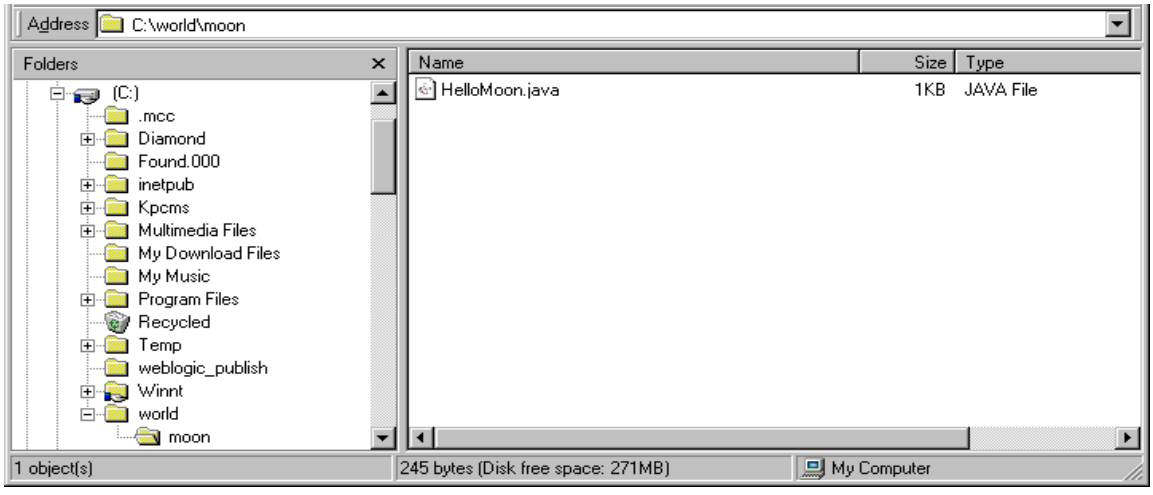


Figure 4. HelloMoon in world.moon package

Although we add a subpackage under package world, we still don't have to change anything in our CLASSPATH. However, when we want to reference to the HelloMoon class, we have to use world.moon.HelloMoon as its fully-qualified class name.

11.3 How to use packages

There are 2 ways in order to use the public classes stored in package.
1. Declare the fully-qualified class name. For example,

```
...
world.HelloWorld helloWorld = new world.HelloWorld();
world.moon.HelloMoon helloMoon = new world.moon.HelloMoon();
String holeName = helloMoon.getHoleName();
...
```

2) Use an "import" keyword:

```
import world.*;  // we can call any public classes inside the world package
import world.moon.*;  // we can call any public classes inside the world.moon package
import java.util.*;  // import all public classes from java.util package
import java.util.Hashtable;  // import only Hashtable class (not all classes in java.util package)
```

Thus, the code that we use to call the HelloWorld and HelloMoon class should be

```
...
  HelloWorld helloWorld = new HelloWorld(); // don't have to explicitly specify world.HelloWorld anymore
  HelloMoon helloMoon = new HelloMoon(); // don't have to explicitly specify world.moon.HelloMoon anymore
...
```

Note that we can call public classes stored in the package level we do the import only. We can't use any classes that belong to the subpackage of the package we import. For example, if we import package world, we can use only the HelloWorld class, but not the HelloMoon class.

Example Step1:
md Test
cd Test
create first.java and save in Test

compile first.java
Test is the package name
……..

```
package Test;
public class first
{
public void view()
{
System.out.println("I am from Test package");
}
}
```

Step2:
change to root directory
create mainpack.java
and import package Test in that file
Create an object of class first and view the result from the package Test

```
import Test.*;
class mainpack
{
public static void main(String args[])
{
first f=new first();
f.view();
}
}
```


Step 3: javac mainpack.java
java mainpack
o/p I am from the package Test

Importing classes from other packages

Step1:create two directories pack1 &pack2
create first.java and save in package pack1
compile first.java from the folder pack1
Also create second.java and save in package pack2. compile second.java from the folder pack2

```
//first.java
package pack1;
public class first
{
public void view()
{
System.out.println("I am from package");
```

```
}}
-----------------------------------
//second.java
package pack2;
public class second
{
public void view2()
{
System.out.println("I am from package2");
}}
-------------------------------------------
```

Step2:

change to root directory
create mainpack1.java

```
//mainpack1.java
import pack1.*;
import pack2.second;
class mainpack1
{
public static void main(String args[])
{
first f=new first();
f.view();
second s=new second();
s.view2();
}}
```


Step 3: javac mainpack1.java
java mainpack1
o/p I am from the package1
I am from the package2

Subclasses an Imported class

It is possible to subclass a class that has been imported from another package.
For example
Step1
create package pack2.create second.java and save in package pack2. compile second.java from the folder pack2

```
//second.java
package pack2;
public class second
{
public void view2()
{
```

```
System.out.println("I am from package2");
}
}
```

Step2:

change to root directory
create mainpack2.java.

```
//mainpack2.java
import pack2.second;
class three extends second
{
void view3()
{
System.out.println("I am from pack2 extended in class three");
}
}
class mainpack2
{
public static void main(String args[])
{
three t=new three();
t.view2();
t.view3();
}
}
```


Step 3:

```
javac mainpack2.java
java mainpack2
```

o/p

```
I am from the package2
I am from pack2 extended in class three
```

Comparison with Java and C++

- Java does not support pointersSecurity applied by java program can be eliminated by the concept of pointers.
- Java does not include structures or unions.
- Java does not support operator overloading
- Java does not support operator overloading,
- Java does not perform any automatic type conversions that result in a loss of precision.
- Java does not allow default arguments
- Java supports constructors, it does not have destructors.It does that by finalize function.
- Goto,delete is not available in Java
- In java objects are passed by reference only.multithreading,packages and interfaces are in Java.