OBJECT ORIENTED PROGRAMMING

USING

C++

INTRODUCTION

Objectives

In this section, you will learn to:

- Describe the limitations of the structured methodology
- Describe the evolution of the object-oriented methodology
- Define and describe the principles of Object-Orientation
- Describe the features of Abstract Data Types

The Structured Approach

- Gained currency with the advent of structured programming.
- Was among the first methodologies that brought in rigour and discipline to the documentation of Systems Analysis and Design through the use of graphical notations.
- Used the Top-Down approach to the task of Systems Development.
- Laid more emphasis on the functions in the system.

The Structured Approach

- Was ideal to the analysis, design, and development of systems of smaller scope.
- Could not handle large and complex systems.
- Systems developed using this approach were not resilient to change.
- Could not therefore, evolve and adapt to changing business requirements.

The Structured Approach

- This would escalate system maintenance costs.
- A small change in system functionality would entail rewriting huge portions of code.
- This is because modules in a system developed using this approach were tightly coupled, thereby increasing the interdependency between modules.
- It was this interdependency that prevented the reusability of generic modules across systems.

Object-Orientation – A Paradigm Shift

- Uses real-world concepts when dealing with the issue of complexity inherent in large systems.
- System structured along “objects”.
- An object can be defined as a real-world entity (physical or conceptual), that has attributes, and exhibits well-defined behaviours.
- It has a state, has a well-defined boundary, and has a unique identity.

Classification

- One of the ways of dealing with the myriad of objects is to classify them.
- Classification is done based on common attributes and behaviours.
- Objects within the scope of a system are classified.
- Classification leads to the definition of a set of classes.

Class

- A Class is a general description of a set of objects with common attributes and common behaviours.
- Classes generally tend to be related in the form of a hierarchy.
- Classes at a certain level in the hierarchy (except the class at the top of the hierarchy) are derived from the classes at the immediately preceding level.

Inheritance

- A class that is derived from an already existing class is called as a Derived Class/Subclass.
- The class from which other classes are derived is called the Base class/Superclass
- A subclass not only inherits attributes and behaviours from its superclass, but also adds its own unique attributes and behaviours giving it its unique identity.

Generalization/Specialization

- Superclasses and subclasses help implement the Generalization/Specialization mechanism that is so very typical and characteristic of a hierarchy.
- Increasing levels of generalization are observed as we ascend the hierarchy.
- Increasing levels of specialization are observed as we come down the hierarchy.

Abstract Classes

- Classes at the very top of a class hierarchy are pure abstractions, instances of which do not exist in the real world.
- They generally have one or more behaviours that cannot be implemented.
- But their subclasses can have concrete instances, which provide a concrete implementation of abstract behaviours defined in abstract classes.

Overriding Behaviours

- There are times when a subclass may choose to provide for its own specific implementation of an inherited behaviour.
- In such a scenario, the subclass chooses to keep the same nomenclature of the behaviour as defined in the superclass, but chooses to keep the semantics different specific to its own needs.
- This is a case where a subclass is said to have overridden a behaviour inherited from its superclass.

Overridden Behaviours

- Overridden behaviours across subclasses in a class hierarchy lead to the definition of a generic behaviour in a superclass that is inherited and overridden by each of the subclasses.

- Overridden functions lay the groundwork for the “Single Interface, Multiple Implementations” paradigm that is the very essence of Polymorphism

Polymorphism

- Polymorphism literally means “anything that is capable of existing in multiple forms. (the root words “Poly” and “Morph” are Greek words that stand for “many” and “forms” respectively.

- Polymorphism in OO environments is typically associated with overridden behaviours across subclasses in a class hierarchy, each of which has the same name as that in its superclass, but chooses to keep its implementation specific to its own needs.

Abstraction

- Unable to comprehend complexity of real-world objects while attempting to classify them, we look for what is essential relative to our perspective or understanding. This is Abstraction.

- Abstraction leads to the definition of a well-defined set of public interfaces using which the external world can interact with the object.

- By interacting with these interfaces, the external world can draw out the behaviours of the object, while choosing to ignore the internal implementation of the object.

Encapsulation

- Encapsulation is the process of hiding the implementation-level details of an object.
- Implementation-level details refer to the object's attributes as well as to the implementation of the object's behaviours.
- The internal implementation can only be accessed through the object's public interfaces.
- Keeping attributes and related behaviours together is another way of implementing encapsulation.

Abstraction Vs. Encapsulation

- Abstraction and Encapsulation are two sides of the same coin.

- If you start abstracting by providing a set of well-defined public interfaces for an object, using which the external world can interact with the object, you start hiding or encapsulating the implementation-level details of the object.

Abstraction Vs. Encapsulation

- Or, if you start encapsulating the implementation-level details of an object, you invariably end up defining a set of well-defined interfaces for accessing the internal implementation of the object.

- Therefore, the conclusion is that if you start abstracting, you end up encapsulating, and vice versa.

OOPL – Implementation Issues

- A traditional programming language like C provided a set of pre-defined data types (primitive data types), and pre-defined operations on those data types to cater to the computational requirements of a wide cross-section of applications.

- To address the need for specific data types mandated by applications, the C language provided the **struct** construct that allowed the programmer to define a data type specific to his/her application.

User-Defined Data Types

- Using a **struct** construct, a programmer can define a blueprint for a data type.
- The programmer can then choose to create a variable (an implementation) based on the template defined by him.
- The **struct** therefore, facilitates the definition and implementation of **user-defined data types** in C.

Operations on User-Defined Data Types

- Operations on user-defined data types have to be defined by the programmer in the form of user-defined functions.

- In the case of pre-defined data types, the compiler would automatically associate legitimate operations on a specific predefined data type.

Operations on User-Defined Data Types

- The onus is on the programmer to associate a user-defined function with its respective user-defined data type through an explicit call to the user-defined function.

- This is in sharp contrast to primitive data types where the compiler can bind legitimate operations to its data type.

- The need was to somehow evolve a mechanism wherein the binding of a user-defined function to its corresponding user-defined data type could be achieved so that only these functions could operate on the data type.

Operations on User-Defined Data Types

- But the problem with a struct declaration in C is that it allows only data members, and user-defined functions cannot be defined as part of the **struct** declaration itself.

- This was the problem with the C compiler that confronted Bjarne Stroustrup when he was working on the design of an Object-Oriented Programming Language in 1980.

- He adapted the **class construct** that he had used in **Simula-67** (an earlier language that he developed) into the C language by rewriting the C compiler that he called **"C with Classes"**

Abstract Data Types (ADT)

- C with classes supported the **class** construct using which one could define not only data in the type, but also operations on the data type as part of the type declaration itself.

- A **class** therefore, satisfies the requirement of an Abstract Data Type (ADT).

- The advantage of an Abstract Data Type is that only operations defined as part of the data type can access the data in the type.

Class

- Class is a way to bind the data and the associated functions together.
- Class declaration is similar to structure declaration.
- Creates a new data type that can be treated as any other built-in data type.

Classes and Objects

- An object is a physical implementation of a class created in memory by a program.
- An object therefore, represents class instantiation.
- An object is therefore, called as an instance of a class.

The Object-Message Paradigm

- Objects communicate with one another using messages.
- An object needing a service initiates a message to an object providing the service.
- The object requesting the service is called the client object and the object providing the service is called the server object.
- Sending a message to the server object results in an invocation of a service provided by the server object.

The Object-Message Paradigm

- The components of the message initiated by the client to the server object consists of the server object name, the service name, and any arguments needed by the service.
- The service being invoked in the server object by the client returns a value to the client.
- The client can use this value to determine successful execution, or otherwise of the invoked service.

Object-Based, Class-Based and Object-Oriented Languages

- If a language supports only pre-defined objects, it is **Object-Based**.
- Support for classes renders the language **Class-Based**.
- If in addition to classes, a language supports the basic principles of Abstract Data Types and Inheritance, and the enabling principles of Encapsulation and Polymorphism respectively, it is **Object-Oriented**.

Procedural vs OOP

Procedure oriented programming

- Emphasis on algorithms
- Functions
- Global data
- Top down approach

Object oriented programming

- emphasis on data abstraction
- objects
- Functions and data are grouped into structures
- Bottom up approach

Summary

In this section, you learnt to:

- Describe the limitations of the structured methodology
- Describe the evolution of the object-oriented methodology
- Define and describe the principles of Object-Orientation
- Describe the features of Abstract Data Types

A tour of C++

Objectives

In this section, you will learn to:

- Describe the basics of a C++ Program
- Describe data types, variables and control structures
- Define default arguments
- Define strong typing
- Define function overloading
- Define Inline function
- Describe the new and the delete operator
- Define references

Features of C++

- The term object-oriented originated during the development of **Smalltalk**.
- Originated at Xerox's Palo Alto Research Centre.
- **Smalltalk** is a pure object-oriented programming language, in that everything is viewed as an object.
- **Simula-67** was a primary influence upon the C++ language, though some ideas were also taken from the language **Flex.**

History

- **1970 --** Thompson designs new B language
- **1972 --** Dennis Ritchie at Bell Labs designs C and 90% of UNIX is then written in C
- **Late 70's --** OOP becomes popular
- Bjarne Stroustrup at Bell Labs adds features to C to form "C with Classes"
- **1983 --** Name C++ first used

Functions

- Functions are building blocks of C++ programs.
- main() is the first function to be executed and has to return a value in C++(exit value)
- format

  return_type function_name(argument list)

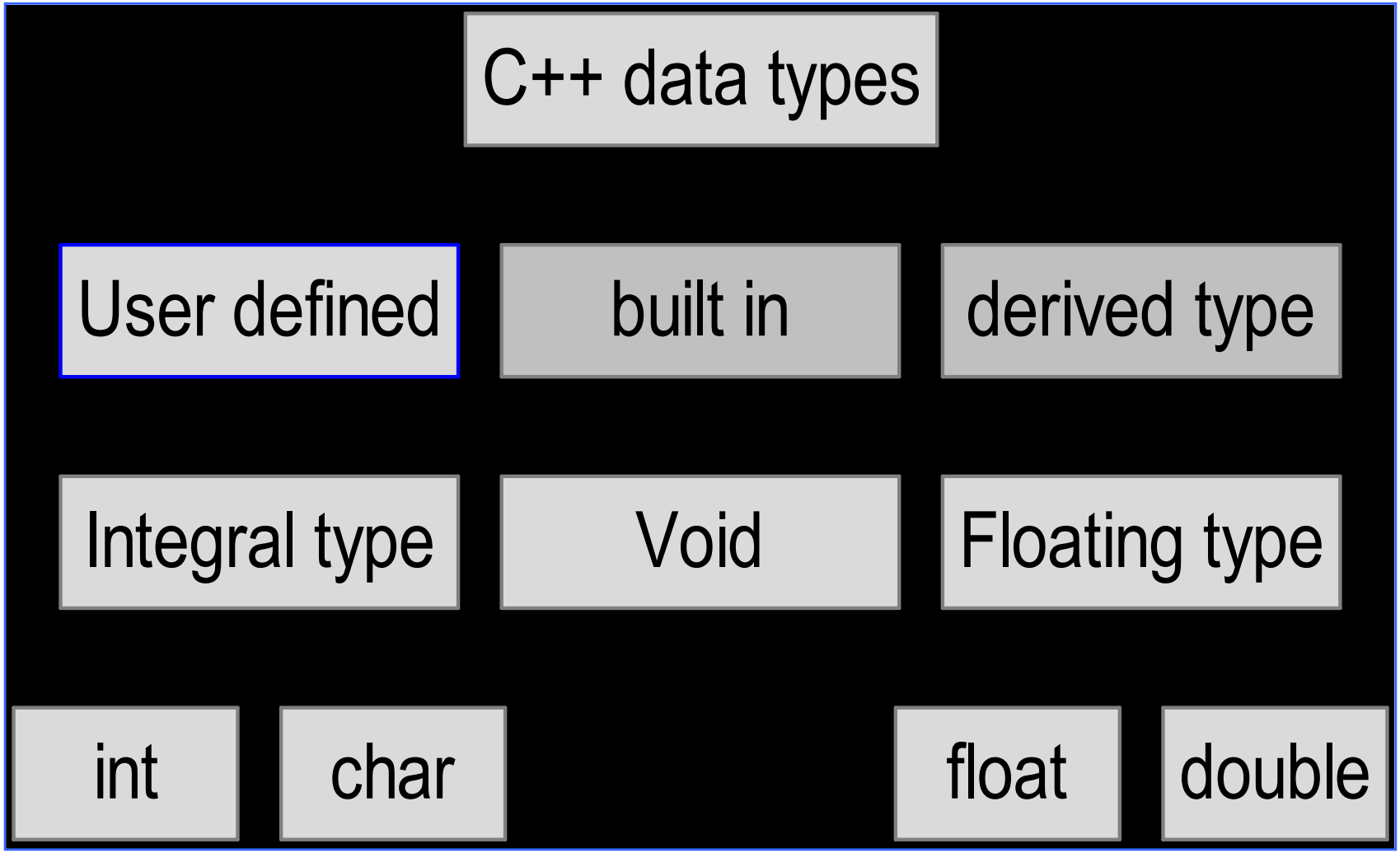
- Prototyping required but variable name not necessary

Preparing to Program

```
//My First Program            /*Program Comments */

 #include <iostream.h>         //Include the header file
 void main()                   //Execution starts from main()
 {                             //Beginning of the block
   cout << "Welcome to Wipro"; //Output statement
 }                             //End of the Block
```

Like C, C++ programs are collections of functions

Data Types


C++ data types
User defined
built in
derived type
Integral type
Void
Floating type
int
char
float
double

User-Defined Data types

- **Structures and Classes**
  - Structures remain same as C
  - Classes are similar to Structures with subtle differences
- **Enumerated Data Types**

  - an **enum** is a set of integer constants

  - compiler's default value assignment starts with 0, and each subsequent enumerator increments by 1

    - enum { RED, BLUE, GREEN };

  can also be explicitly assigned value in declaration (which doesn't necessarily have to be unique)

    - enum { RED=42, BLUE, GREEN=47 };

Data Types

- **Arrays**
  - Array size cannot be exactly equal to length of string. ex. char str[5]="Hello" not allowed but char str[6]="Hello" is allowed

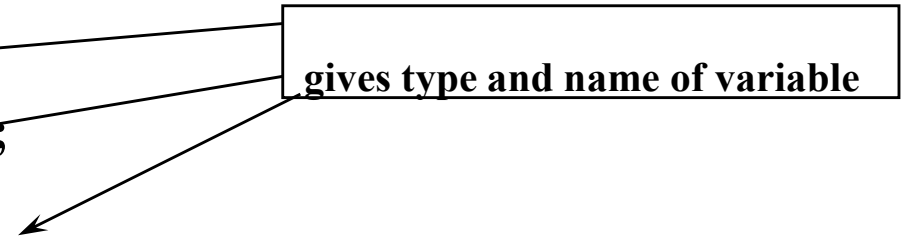
- **Pointers**
  - Constant Pointer and pointer to a constant
  - char * const constptr = "Hello";
  - int const *ptrconst=&a;

Variables

- symbols that represent values in a program
- have datatype and name

  type-specifier identifier [= initial-value];

```
char cMyChar;
unsigned long nObjectID;
int nHours, nMinutes, nSeconds;
int nAnswer = 42;
float fMyTemp = 98.6;
```



- variable must be declared before it can be used

Control Structures

- Control structures
  - selection
    - if - else (Two way branch)
    - switch (Multiple branch)
  - loop
    - do - while (Exit control)
    - while (Entry control)
    - for (------do--------)

Default Arguments

- An argument value that is specified at the time of defining the function prototype. This value will be automatically passed to a function when no explicit argument is specified in the function call.

- Allows calling a function without specifying all its arguments.

```
float discount( int size, int quantity=0); //function prototype
discount(10); // quantity=0 default argument assumed
discount(10,4); //quantity=4 actual argument overrides default
```

Strong Typing

- C++ is a strongly typed language.
- The C++ compiler enforces type checking of each assignment made in a program at compile time.
- Both the argument list and the return type of each function are type checked during compilation. An implicit conversion will be applied if possible.
- If this is not possible, or if the number of arguments is incorrect, then a compile time error is issued.

Function Overloading

- C++ supports two functions to coexist with the same name. Such functions are said to be overloaded. Calls to an overloaded function are resolved using function signatures.

- When an overloaded function is invoked, the compiler chooses the appropriate function by examining the number, data types, and the order of arguments present in that function.

```
int n=power(10,5)          //first function is called
float f=power(10.5,6.3)    //second function is called
```

Ambiguity in function overloading

- Main cause of ambiguity involves C++'s automatic type conversions as well as providing default arguments

- C++ automatically attempts to convert the arguments used to call a function into the type of arguments expected by the function.
  - For ex: func(10) call will cause an ambiguity if the function is actually receiving a float or double.
  - func(int x=10, int y=100) may conflict with func()

Const Qualifiers

These are read-only variables

- must be initialized in declaration

```
const float NORMAL_TEMP = 98.6;
```

- better than using pre-processor **#define** to declare a constant (goes in compiler symbol table)

- not just values, but pointers, references and member functions can be declared **`const`** also

Inline Functions

- If a function is declared as inline, at the time of compilation, the body of the function is expanded at the point at which it is invoked.

- For small functions, the inline function provides modularity, and removes all the overheads of function calls.

There are two ways of declaring a function as inline:

- Declaring the body of the function within the class
- Using the inline keyword.

Inline Vs. Macros

- Inline functions are functionally similar to #define macros. In both the cases, during compilation time, the body of the macro or the function is expanded.

- But inline functions are preferred over macros because of two main reasons:

- First, in the case of inline functions, the types of the arguments are checked against the parameter list in the declaration for the function.

Inline Vs. Macros

- As a result, any mismatch in the parameters can be detected at the time of compilation.
- This allows inline functions to be overloaded, which is not possible in the case of macros.
- Second, there are certain situations where a macro does not behave in the same manner as a function call, which may lead to unpredictable results.

Inline Vs. Macros

- To compute the square of a given number, either a macro or an inline function can be used as follows:

```
#define square(x) x*x
        OR
inline int square(int x)
   {    return (x*x);   }

void main( )
 {
  cout << square( 1 + 2) << "\n";
 }
```

The new operator

- allocates memory on the heap at runtime for an object, or a primitive data type, and returns a pointer to the object or the primitive data type thus allocated.

Eg:

```
int *p = new int;
int *pia = new int[4];
```

allocates an array of four integer elements.

- In addition to allocating memory, **new** also creates an object by calling the object's constructor.

The delete operator

- Frees the memory occupied by an object on the heap previously allocated to it using **new**.
- **delete** pint; //deletes a single object

  delete [] pia; //deletes an array of objects

- In addition to de-allocating memory occupied by an object, **delete** also destroys the object by calling the object's destructor.

malloc() / free() Versus new / delete

- Easier syntax and ability to work with a variety of data types without being required to do some clumsy typecasting.

- **new** automatically determines the size of the data type when allocating memory for an object or variable of the data type. No need to use **sizeof** operator.

- **new** and **delete** create and destroy objects. **malloc()** and **free()** merely allocate and de-allocate memory.

- **new** and **delete** can be overloaded.

Type Cast Operator

The C++ cast operator is used to explicitly request a type conversion. The cast operation has two forms.

```
int     intVar;
float   floatVar  =  104.8 ;

intVar  =  int ( floatVar ) ;        // functional notation, OR

intVar  = ( int ) floatVar ;         // prefix notation uses ()
```

Function Argument Passing Mechanisms

C/C++ call convention is *pass-by-value*

```
void swap ( int x, int y )
{
  int nTemp = x;
  x = y;
  y = nTemp;
 }
```

***pass-by-reference* traditionally done with pointers in C**

```
void swap ( int* px, int* py )
{
  int nTemp = *px;
  *px = *py;
  *py = nTemp;
}
```

Reference

- **A reference is essentially an implicit pointer.**

- Three ways of using a reference:
- Function Parameter
- Function return value
- Standalone reference

- **The most important use for a reference is to allow creation of functions that automatically use call-by-reference parameter passing.**

Reference Parameters

- C++ allows two ways to achieve call-by-reference parameter passing.
- First, you can explicitly pass a pointer to the argument as in C.
- Second, you can use a reference parameter, which proves often to be the best way.

Reference Parameters

- Revisiting call-by-reference using pointer parameters, or manual call by reference:

- Function negate( ), which reverses the sign of the integer variable pointed to by its argument.
- // manually create a call by reference using a parameter
- #include<iostream>
- using namespace std;
- void negate (int *i); // function prototype
- int main( )
- {

Manual Call-By-Reference

- int x
- x = 10;
- cout << x << " negated is ";
- negate (&x);
- cout << x << "\n";
- return 0;
- }
- void negate (int *i)
- {
- *i = -*i;
- }

Automatic Call-By-Reference

```
// Use a reference parameter
#include<iostream>
using namespace std;
void negate (int &i); // i is now a reference
int main( )
 {
   int x
    x = 10;
    cout << x << " negated is ";
    negate (x); // no longer need the & operator
 cout << x << "\n";
 return 0;
}
void negate (int &i)
{
  i = -i; // i is now a reference; no longer need *
}
```

Summary

In this section, you learnt to:

- Describe the basics of a C++ Program
- Describe data types, variables and control structures
- Define default arguments
- Define strong typing
- Define function overloading
- DefineInline function
- Describe the new and the delete operator
- Define references

CLASSES AND OBJECTS

Objectives

In this section, you will learn to:

- Define a class
- Implement an object based on a class
- Describe the access specifiers Private, Public, & Protected
- Describe the scope resolution operator
- Describe the **this** pointer
- Describe the accessibility of class members Vs Struct members
- Describe Constructors and Destructors
- Describe static class members, both data and member functions

Class

- General form:

```
class  class name
{
access specifire:
 data declaration
 function declaration
};
```

Demonstration: difference between Class and structure

- In C++, Functions can be defined inside the stucture .
- Example: To represent a point on a two-dimensional plane as a user-defined data type. A point on a two-dimensional plane is represented by its x-coordinate and y-coordinate.
- The most basic operation on this Point data type would be to store valid screen coordinates into the data in the type. There may be need for operations which need to retrieve the x and y coordinates of a particular point object.

Example: Structure and its objects

- Let us now start with the declaration of a struct point, which incorporates the x-coordinate and y-coordinate, and then proceed to define the functions to set and get the x-coordinate and the y-coordinate.

```
struct point
 {
   int x_coord;
   int y_coord;
 };
```

- define the set of functions to set and get the values of the x and y coordinates of point.

Example: Structure and its objects

```
void setx( int x)
  {
   x_coord = (x > 79 ? 79 : (x < 0 ? 0 :x));
  }
void sety (int y)
  {
   y_coord = (y < 24 ? 24 : (y < 0 ? 0 : y));
  }
int getx( void)
  {
   return x_coord;
  {
int gety( void)
{ return y_coord;}
```

Example: Structure and its objects

- These functions need to be bound to the point data type. C++ provides a simple means of ensuring this. This is done by simply including the functions as part of the struct declaration point. The following is the modified declaration for the struct point:

```
struct point
 {
   int x_coord;
   int y_coord;

```

Example: Structure and its objects


```
void setx( int x)
  {    x_coord = (x > 79 ? 79 : (x < 0 ? 0 :x)); }
void sety (int y)
  {
    y_coord = (y < 24 ? 24 : (y < 0 ? 0 : y));
  }
int getx( void)
  {
    return x_coord;
   {
int gety( void)
{ return y_coord;}
}; // end of struct
```

Example: Structure and its objects

- For example, the following code can go in main( ):

```
struct point
 {
   int x_coord;
   int y_coord;
   void setx( int x)
    { x_coord = (x > 79 ? 79 : (x < 0 ? 0 :x)); }
void sety (int y)
  { y_coord = (y < 24 ? 24 : (y < 0 ? 0 : y)); }
int getx( void)
  {  return x_coord; }
int gety( void)
  { return y_coord;}
}; // end of struct
```

Example: Structure and its objects


```
main( )
 {
  int a, b;
  // a structure variable p1 of point type, struct keyword not required
  point p1;
  p1.setx(22); // set the value of x_coord  of p1
  p1.sety(44); // set the value of y_coord of p1
  a = p1.getx( ); // return the value of the x_coord member of p1
  b = p1.gety( ); // return the value of the y_coord member of p1
  }
```

- Variables and methods declared within a struct are freely accessible to functions outside the structure declaration.

Scope

- Every variable has an associated scope. The scope defines the context of the variable. The scope of a variable specifies where the variable may be accessed.

- C++ provides for three types of scope: **file**, **class** and **local** scope.

File Scope

- File scope is considered to be the outermost scope.
- It is defined as the space outside all functions.
- Variables that have file scope are accessible to all the functions in the program file, and are referred to as global variables.
- Such variables are defines outside main( ).

Local Scope

- Local scope is defined as being limited to the braces of a function, or control structure like if, while, and for.
- Variables that have local scope are not accessible outside the function or the control structures.
- Nesting of local scope is possible.

Class Scope

- The **class** keyword was borrowed from Simula, and incorporated into C++ by Stroustrup.
- But for some fundamental differences, the **class** keyword is similar in usage to the **struct** keyword in C++.
- Shown below is the class declaration for the data type point:

Class Declaration

```
class point
 {
   int x_coord;
   int y_coord;
   void setx( int x)
   { x_coord = (x > 79 ? 79 : (x < 0 ? 0 :x)); }
void sety (int y)
   { y_coord = (y < 24 ? 24 : (y < 0 ? 0 : y)); }
int getx( void)
   { return x_coord; }
int gety( void)
   { return y_coord;}
}; // end of class
```

```
main( )
    {
   int a, b;
   point p1;//class variable(object)
   p1.setx(22); // not being accessible
   p1.sety(44); // not being accessible
   a = p1.getx( ); // not being accessible
   b = p1.gety( ); // not being accessible
     }
```

Accessibility of Struct Members

- Variables and methods declared within a struct are freely accessible to functions outside the structure declaration.

- **Therefore, all members in a structure are by default public.**

Accessibility of Class Members

- On the other hand, when a class declaration is used for the Point data type as depicted earlier, the data members and the member functions are accessible from only within the class.

- Data and methods within the class declaration will no longer be visible to functions outside the class point. The member functions to get and set the x and y coordinates can no longer be called from main( )

- **Therefore, all members in a class are by default private, thereby not being accessible outside the class.**

Access Specifiers

- C++ offers the designer of the class the flexibility of deciding which member data or methods should be accessible from outside the class, and which should not.

- Access specifiers serve the important purpose of drawing the line between the accessible and the inaccessible parts of a class.

- It is the class designer's prerogative to demarcate the part of the class that needs to be hidden, and the part that needs to be offered to the user as an interface to the class.

Access Specifiers

- The **private** access specifier is generally used to encapsulate or hide the member data in the class.
- The **public** access specifier is used to expose the member functions to the outside world, that is, to outside functions as interfaces to the class.
- The modified code for the class point is presented in the following slides:

Class Declaration for Point

```
class point
 {
  private:
   int x_coord;
   int y_coord;

  public:
   void setx( int x)
   {    x_coord = (x > 79 ? 79 : (x < 0 ? 0 :x)); }
   void sety (int y)
    { y_coord = (y < 24 ? 24 : (y < 0 ? 0 : y)); }
   int getx( void)
    {  return x_coord; }
   int gety( void)
    { return y_coord;}
}; // end of class
```

Class Declaration for Point

```
main( )
 {
  int a, b;
  // an object p1 of class type point, class keyword not required
  point p1;
  p1.setx(22); // set the value of x_coord  of object p1
  p1.sety(44); // set the value of y_coord of object p1
  a = p1.getx( ); // return the value of the x_coord member of object p1
  b = p1.gety( ); // return the value of the y_coord member of p1
  }
```

Initializing Member Data in an Object of a Class

- Every time an object is created, memory is allocated for it. But allocation of memory does not automatically ensure initialization of member data within the object.

- Referring to our earlier example of the class point, whenever an object of the class point is created, we would want to initialize the x_coord and y_coord member data with (0,0), or any specific values.

Constructors

- While designing a class, the class designer can define within the class, a special member function that is automatically invoked whenever an object of the class is created.

- Such functions are called constructors. A member function can be designated as a constructor by assigning to it the same name as the name of the class.

- A class can contain more than one constructor, thereby allowing multiple ways of initializing an object.
- Constructors can therefore be overloaded.

Class With Constructors

```
class point
 {
  private:
   int x_coord;
   int y_coord;

  public:
  point( )
   { x_coord = y_coord = 0;}
  point( int x, int y)
   {
    x_coord = (x > 79 ? 79 : (x < 0 ? 0 :x));
    y_coord = (y < 24 ? 24 : (y < 0 ? 0 : y));
   }
```

Class With Constructors

```
int getx( void)
      {  return x_coord; }
     int gety( void)
      { return y_coord;}
};// end of class
main( )
  {
    point p1;
   point p2(10,20);
    }
```

Constructors: Features

- A special member function which is invoked automatically when an object is created.
- Generally declared in the public section.
- Has the same name as the class.
- Does not have return types.
- Can be overloaded.

Destructors

- Destructors are functions that are complimentary to constructors. They serve to de-initialize objects when they are destroyed.

- A destructor is invoked when an object of the class goes out of scope, or when the memory occupied by it is de-allocated using the **delete** operator.

- A destructor, like a constructor, is identified as a function that has the same name as that of the class, but is prefixed with a '~' (tilde).

The this pointer

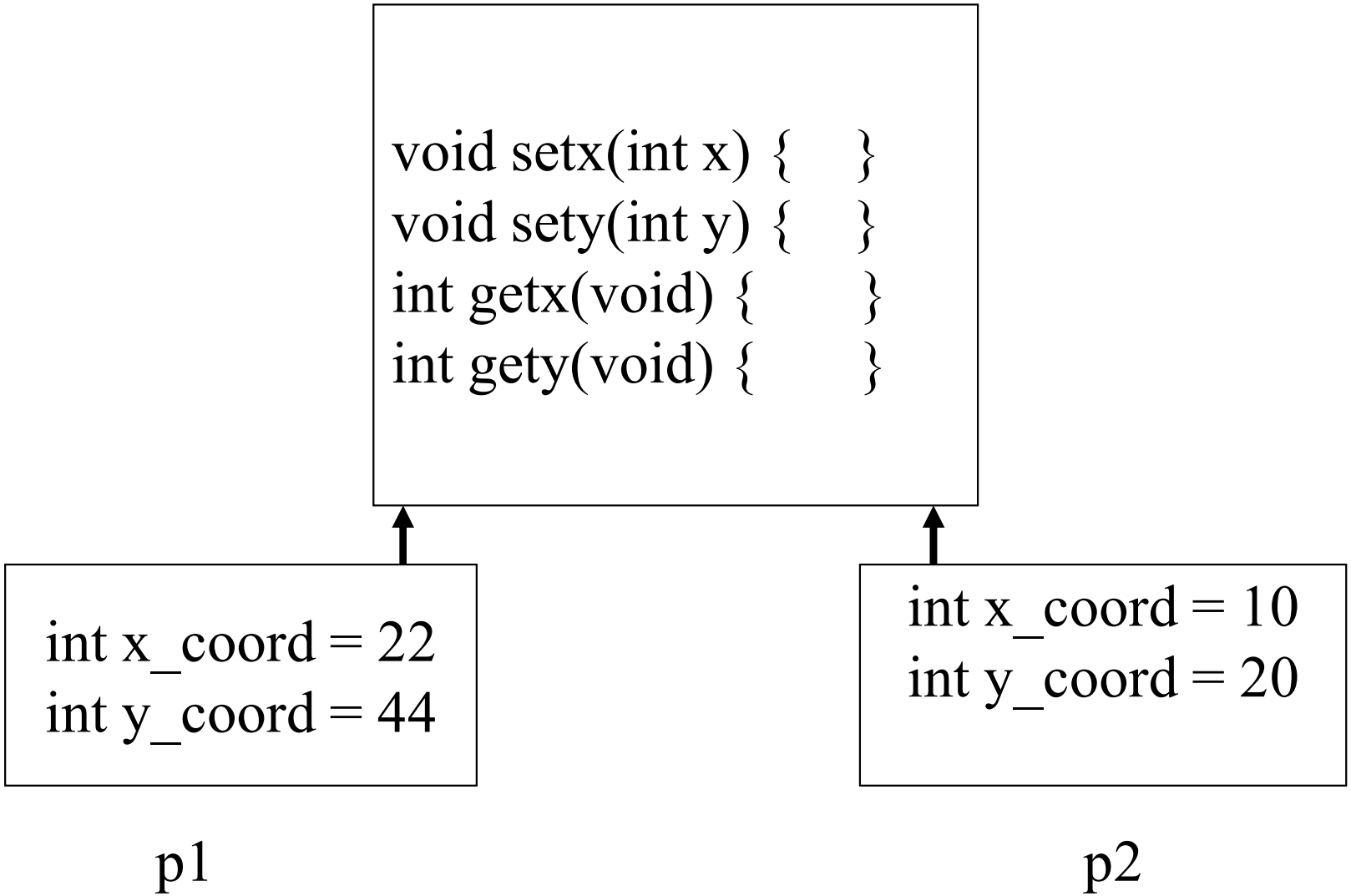
- **this ponter** holds the address of the object which invokes the function
- Consider the following code:

```
class point
 {
  private:
   int x_coord;
   int y_coord;

  public:
   void setx( int x)
   {    x_coord = (x > 79 ? 79 : (x < 0 ? 0 :x)); }
   void sety (int y)
    { y_coord = (y < 24 ? 24 : (y < 0 ? 0 : y)); }
   int getx( void)
    {  return x_coord; }
   int gety( void)
    { return y_coord;}
}; // end of class
```

The this pointer

```
main( )
 {
  int a, b;
   // an object p1 of class type point, class keyword not required
   point p1,p2;
   p1.setx(22); // set the value of x_coord  of object p1
   p1.sety(44); // set the value of y_coord of object p1
   a = p1.getx( ); // return the value of the x_coord member of object p1
   b = p1.gety( ); // return the value of the y_coord member of p1
   p2.setx(10);
   p2.sety(20);
   }
```

The this pointer



The this pointer

- Each class member function contains an implicit pointer of its class type, named **this.**
- The **this** pointer, created automatically by the compiler, contains the address of the object through which the function is invoked.
- Therefore, when the member function setx( ) is invoked through p1, the function setx( ) implicitly receives the address of the object p1 (***this**), and therefore, the x_coord of p1 is set.

Scope Resolution Operator ::

- Defining member functions within the body of the class will be inline function bydefault.

- C++ provides the scope resolution operator :: that allows the body of the member functions to be separated from the body of the class.

- Using the :: operator, the programmer can define a member function outside the class definition, without the function losing its connection to the class.

Scope Resolution Operator ::

Consider the following example:

```
class point
  {
    private:
     int x_coord;
     int y_coord;
    public:
     point (int x, int y);
     void setx (int x);
  };
point::point (int x, int y)
 {
   x_coord = x;
    y_coord = y; }
void point::setx( int x)
  { x_coord = x; }
```

Static Class Members – Static Data Members

- Both function and data members of a class can be made **static**.

- When you precede a member variable's declaration with the keyword static, you are telling the compiler that only one copy of that variable will exist, and that all objects of that class will share that variable.

Static Data Members

```
#include <iostream>
using namespace std;
class static_demo
 {
 private:
  static int a;
  int b;
 public:
  void set ( int i, int j)
  {a = i; b = j; }
  void show( );
 };
int static_demo::a; // define the static variable a
void static_demo::show( )
 {
  cout << “this is static a:  “ << a;
  cout << this is non-static b:  “ << b; << ‘\n’;   }
```

Static Data Members

```
int main( )
 {
  static_demo x, y;
  x.set(1, 1); //set a to 1
  x.show( );
  y.set(2, 2); // change a to 2
  y.show( );
  x.show( ); /* Here, a has been changed for both x and y because a is shared
  by both objects */
  return 0;
 }
```

Static Data Members – Uses

- An interesting use of a static member variable is to keep track of the number of objects of a particular class type that is in existence. Consider the following example:

```
#include <iostream.h>
class counter_test
  {
   public:
    static int count;
    counter_test ( ) { count++; }
    ~counter_test ( ) { count--;}
  };
```

Static Data Members – Uses

```
int counter_test::count;
void f( );
int main( )
 {
  counter_test ob1;
  cout << objects in existence:  “ << counter_test::count << “\n”;
  counter_test ob2;
  cout << objects in existence:  “ << counter_test::count << “\n”;
  f( );
  cout << objects in existence:  “ << counter_test::count << “\n”;
  return 0; }
void f( )
 {
   counter temp;
   cout << objects in existence:  “ << counter_test::count << “\n”;
   // temp is destroyed when f( ) returns
 }
```

Static Member Functions

- Member functions may also be declared static.
- Static member functions are subject to several restrictions.
- They may only directly refer to other static members of the class.
- A static member function does not have a **this** pointer.

Static Member Functions

- There cannot be a **static** and a **non-static** version of the same function.
- A static member function may not be **virtual**.
- Finally, they cannot be declared as **const** or **volatile**.
- One good use for them is to “pre-initialize” private static data before any object is actually created.

Static Member Functions

```
#include <iostream>
using namespace std;
class static_type
  {
   private:
    static int i;
   public:
    static void init ( int x)
      { i = x; }
    void show ( )
     { cout << i;}
    };
int static_type::i; // define i
```

Static Member Functions


```
int main( )
 {
  // initialize static data before object creation
  static_type::init(100);
  static_type x;
  x.show( ); // displays
  return 0;
 }
```

Summary

In this section, you learnt to:

- Define a class
- Implement an object based on a class
- Describe the access specifiers Private, Public, & Protected
- Describe the scope resolution operator
- Describe the **this** pointer
- Describe the accessibility of class members Vs Struct members
- Describe Constructors and Destructors
- Describe static class members, both data and member functions

Inheritance

Objectives

- In this lesson, you will learn to:
- Derive a class from an existing class
- Use base class access control when deriving a class
- Describe the workings of protected members in a base class vis-à-vis derived class objects
- Describe the order of invocation of constructors and destructors in an inheritance hierarchy
- Pass parameters to base-class constructors from a derived class constructor

Inheritance

- Inheritance is one of the cornerstones of OOP because it allows for the creation of hierarchical classifications.

- Using inheritance, you can create a general class that defines traits common to a set of related objects, i.e, objects with common attributes and behaviours.

- This class may then be inherited by other, more specific classes, each adding only those attributes and behaviours that are unique to the inheriting class.

Code Reuse

- Inheritance leads to the definition of generalized classes that are at the top of an inheritance hierarchy. Inheritance is thus the implementation of generalization.

- Inheritance, in an object-oriented language like C++ makes the data and methods of a superclass or base class available to its subclass or derived class.

- Inheritance has many advantages, the most important of them being the reusability of code. Once a class has been created, it can be used to create new subclasses.

Generalization/Specialization

- In keeping with C++ terminology, a class that is inherited is referred to as a base class. The class that does the inheriting is referred to as the derived class.

- Each instance of a derived class includes all the members of the base class. The derived class inherits all the properties of the base class.

- Therefore, the derived class has a large set of properties than its base class. However, a derived class may override some of the properties of the base class.

Code Syntax for Inheritance

- Class inheritance uses this general form:
- class derived-class-name : **access** base-class-name

- The access status of the base class members inside the derived class is determined by **access**

- The base class access specifier must either be **public, private, or protected.**

- If no access specifier is present, the access specifier is private by default.

Access Specifier - Public

- When the access specifier for a base class member is public, all public members of the base class become public members of the derived class.
- All protected members of the base class become protected members of the derived class.
- In all cases, the base class' private elements remain private to the base, and are not directly accessible by members of the derived class.

Access Specifier – Public (Code)

```
#include<iostream>
using namespace std;
class base
{
 private:
  int i, j;
 public:
   void set( int a, int b)
    { i = a;
      j = b; }
void show( )
  {
   cout << i << "  " << j << "\n";
  }};
```

Access Specifier – Public (Code)

```
class derived : public base
  {
   private:
    int k;
   public:
    derived (int x)
     { k = x; }
void showk( )
  { cout << k << "\n";} };

int main( )
  {
   derived ob(3);
   ob.set(1,2); // access member of base from derived object
   ob.show( ); // access member of base
   ob.showk( ); // uses member of derived class
   return 0; }
```

Access Specifier - Private

- When the base class is inherited by using the private access specifier, all public and protected members of the base class become private members of the derived class.

- The following program will not even compile because both set( ) and show( ) are now private members of derived:

```
#include<iostream>
using namespace std;
class base
{
 private:
  int i, j;
 public:
   void set( int a, int b)
    { i = a;
      j = b; }
```

Access Specifier - Private

```
void show( )
  { cout << i << "  " << j << "\n"; }
};
class derived : private base
  {
   private:
    int k;
   public:
    derived (int x)
     { k = x; }
void showk( )
  { cout << k << "\n";}
};
```

Access Specifier - Private


```
int main( )
  {
    derived ob(3);
    ob.set(1,2); //error, can't access set( ) from outside derived
    ob.show( ); // error, can't access show( ) from outside derived
    return 0;
  }
```

Inheritance and Protected Members

- The protected keyword is included in C++ to provide greater flexibility in the inheritance mechanism.

- With one important exception, access to a protected member is the same as access to a private member. **The sole exception is when a protected member is inherited.**

- If the base class is inherited as public, then the base class' protected members become protected members of the derived class, and are therefore, accessible by the derived class.

Inheritance and Protected Members

```
#include<iostream>
using namespace std;
class base
{
  protected:
   int i, j;
  public:
    void set( int a, int b)
     { i = a;
       j = b; }
void show( )
  {
    cout << i << “  “ << j << “\n”; }
};
```

Inheritance and Protected Members

```
class derived : public base
  {
   private:
    int k;
   public:
Void setk (  )
 { k = i * j; } // access to protected members
void showk( )
  { cout << k << "\n";} };
int main( )
  {
   derived ob(3);
   ob.set(1,2); //OK, known to derived
   ob.show( ); // OK, known to derived
   ob.setk( );
   ob.showk( );
   return 0;  }
```

Inheritance and Protected Members

- When a derived class is used as a base class for another derived class, any protected member of the initial base class that is inherited by the first derived class may also be inherited as protected again by a second derived class.

- For example, the following program is correct, and **derived2** does indeed have access to **i** an **j**
- #include<iostream>
- using namespace std;
- class base
- {
- protected:
- int i, j;

Inheritance and Protected Members

```
public:
    void set( int a, int b)
     { i = a;
       j = b; }
void show( )
  {
    cout << i << "  " << j << "\n"; } };
class derived1 : public base
  {
    private:
     int k;
    public:
Void setk (  )
 { k = i * j; }
void showk( )
  { cout << k << "\n";} };
```

Inheritance and Protected Members

```
class derived2 : public derived1
 {
  private:
   int m;
  public:
   void setm ( )
    { m = i – j; }
   void showm( )
    {cout << m << “\n”;} };
int main( )
 {
  derived1 ob1;
  derived2 ob2;
  ob1.set(2, 3);
  ob1.show( );
```

Inheritance and Protected Members

- ob1.setk( );
- ob1.showk( );
- ob2.set(3,4)
- ob2.show ( );
- ob2.setk( );
- ob2.setm( );
- ob2.showk( );
- ob2.showm( );
- return 0;
- }

Inheritance and Protected Members

- **If however, base were inherited as private, then all members of base would become private members of derived1, which means they would not be accessible by derived2**

This is illustrated by the following program

```
#include<iostream>
 using namespace std;
 class base
 {
 protected:
 int i, j;
 public:
  void set( int a, int b)
   { i = a;
     j = b; }
 void show( )
 {cout << i << "  " << j << "\n"; }
};
```

Inheritance and Protected Members

```
class derived1 : private base
  {
   private:
    int k;
   public:
void setk (  )
 { k = i * j; }
void showk( )
  { cout << k << "\n";} };

class derived2 : private derived1
 {
  private:
   int m;
  public:
   void setm ( )
    { m = i – j; } // i and j private in derived1, will not compile
```

Inheritance and Protected Members

```
void showm( )
    {cout << m << "\n";}
  };

int main( )
 {
   derived ob1;
   derived ob2;
   ob1.set (1, 2); // error, can't use set( )
   ob1.show( ); //error, can't use show( )
   ob2.set(3, 4); // error, can't use set( )
   ob2.show( ); // error, can't use show( )
   return 0;
  }
```

Protected Base Class Inheritance

- It is possible to inherit a base class as protected. When this is done, all public and protected members of the base class become protected members of the derived class.

```
#include<iostream>
using namespace std;
class base
{
  protected:
   int i, j;
  public:
    void setij( int a, int b)
     { i = a;
       j = b; }
```

Protected Base-Class Inheritance

```
void showij( )
  {
    cout << i << “  “ << j << “\n”; } };
class derived : protected base
  {
    private:
     int k;
    public:
    void setk(  )
    {
      setij(10,12);
      k = i * j;  }
void showall( )
  { cout << k << “\n”;
     showij( ); }
};
```

Protected Base-Class Inheritance


```
int main( )
 {
  derived ob;
  ob.setij( ); // illegal, setij( ) is protected member of derived
  ob.setk( ); // OK, public member of derived
  ob.showall( ); // ok, public member of derived
  ob.showij( ); illegal, showij( ) is protected member of derived
   return 0;
 }
```

Constructors, Destructors & Inheritance

- It is possible for a base class, a derived class, or both to contain constructor and/or destructor functions.

- **Constructors are the only exception to the inheritance rule in that they are not inherited by the derived class. The same holds true for destructors.**

- The language therefore, has to provide for the explicit invocation of a base class constructor at the moment of creation of a base class object.

Constructors, Destructors & Inheritance

- It is important to understand the order in which the constructors and destructors are invoked when an object of the derived class comes into existence, and when it ceases to exist.

- To begin, consider this short program:
- #include<iostream>
- using namespace std;
- class base
- {
- public:
- base( )
- { cout << " Constructing base\n"; }
-

Constructors, Destructors & Inheritance


```
~base ( )
   { cout << "Destructing base\n"; } };

class derived : public base
  {
   public:
     derived( )
      { cout << "Constructing derived\n"; }
     ~derived( )
       { cout << "Destructing derived\n"; }};

int main( )
  {
    derived ob;  // do nothing but construct and destruct ob
     return 0;  }
```

Constructors, Destructors & Inheritance

- **Therefore, constructor functions are executed in their order of derivation.**

- **Destructor functions are executed in reverse order of derivation.**

Passing Parameters to Base Class Ctors

- Calling constructor explicitly

  -when base class contains parameterised constructor

- use an expanded form of the derived class' constructor declaration that passes along arguments to one or more base class constructors. The general form of this expanded derived class constructor is as follows:

- derived-constructor (arg-list) : base-constructor (arg-list)

  { body of derived class constructor}

Passing Parameters to Base Class Ctors

- Consider the following program:
- #include<iostream>
- using namespace std;
- class base
- {
- protected:
- int i;
- public:
- base (int x)
- { i = x;
- cout << “Constructing base\n”; }

Passing Parameters to Base Class Ctors

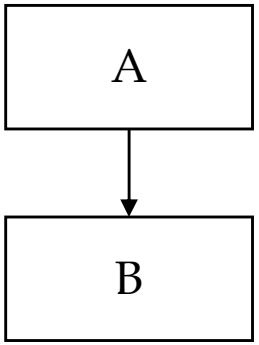

```
~base( )
  { cout << "destructing base\n"; } };

class derived : public base
 {
  private:
   int j;
  public:
   // derived uses x; y is passed along to base
   derived (int x, int y) : base(y)
    { j = x;
      cout << "Constructing derived\n"; }
```

Passing Parameters to Base Class Ctors

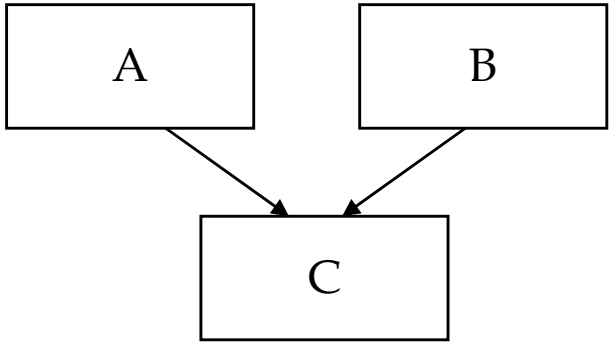
```
~derived( )
   { cout << "destructing derived\n"; }
void show( )
   { cout << i << " " << j << "\n"; } };

int main( )
 {
   derived ob(3,4)
   ob.show( ); // displays 4,3
   return 0; }
```

Types of inheritance

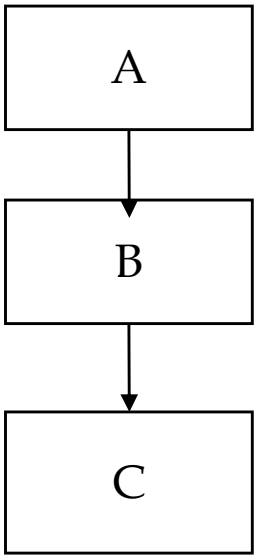
- **Single Inheritance**



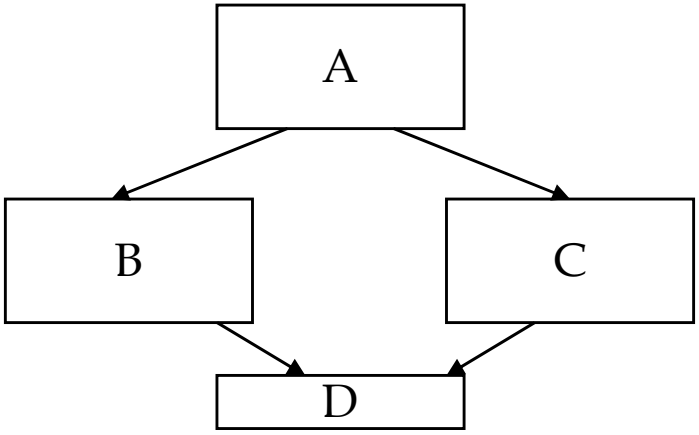
Multiple Inheritance



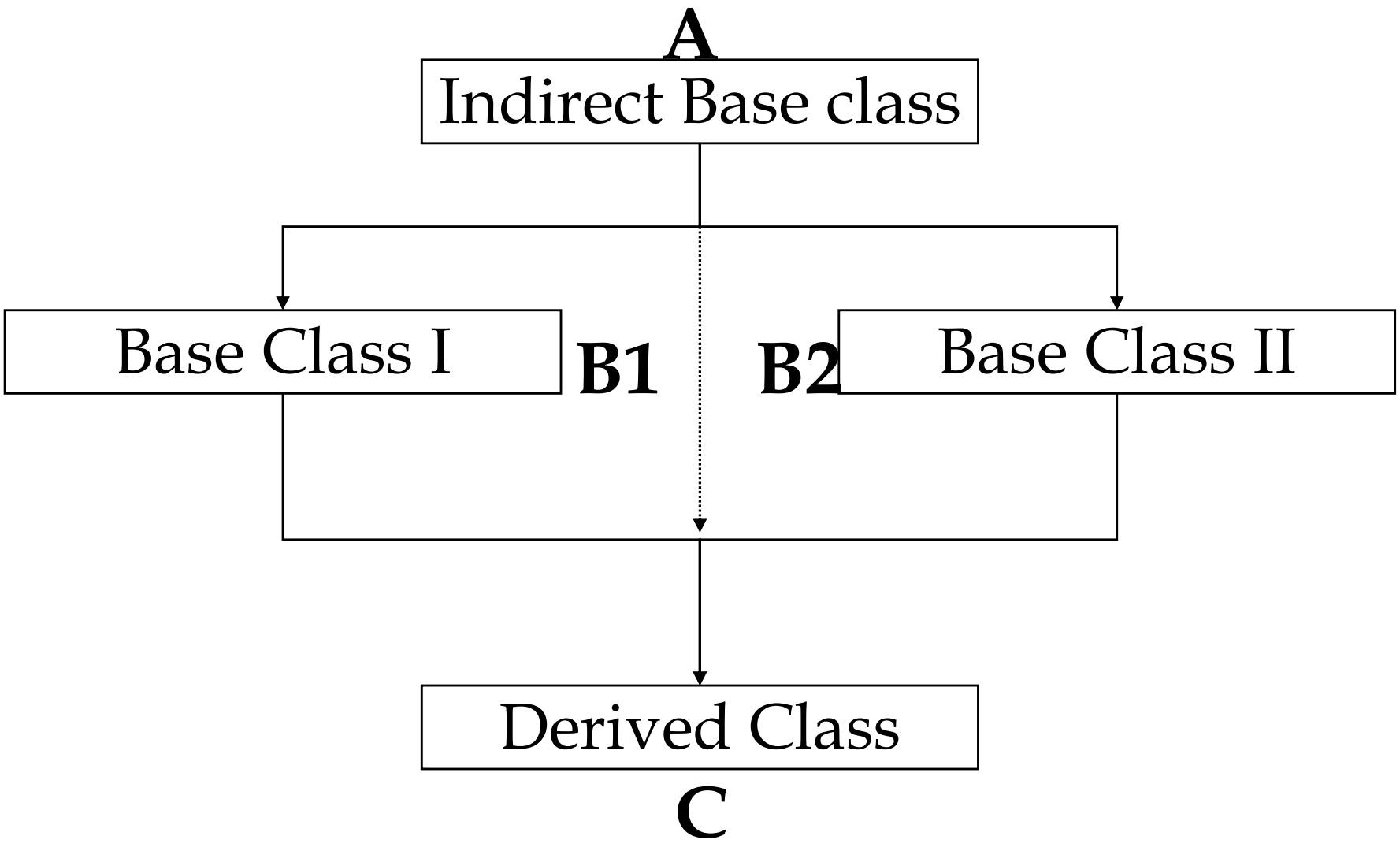
Multilevel Inheritance



Hybrid Inheritance



Hybrid inheritance



Virtual Base Class

- Properties of the virtual base class are made virtual for inheritance in the subsequent derived classes.

- **class B1:public virtual A**
- **class B2:public virtual A**

- **class C:public B1, public B2**

  - only one copy of the properties of class A will be inherited by class C via class B1 and class B2**.**

Inheritance Vs. Composition

- Mechanism to make use of existing classes to define new and more complex classes.
- The internal data (the state) of one class contains an object of another class.
- The host class has access to public members of contained class.
- It can be an alternative to multiple inheritance.
- Provides an additional layer of data protection.

Summary

- In this lesson, you learnt to:
- Derive a class from an existing class
- Use base class access control when deriving a class
- Describe the workings of protected members in a base class vis-à-vis derived class objects
- Describe the order of invocation of constructors and destructors in an inheritance hierarchy
- Pass parameters to base-class constructors from a derived class constructor

Arrays, Pointers, References, and Special Constructors

Objectives

In this lesson, you will learn to:

- Use two special forms of constructors
- Create arrays of objects
- Access an object through a pointer
- Access an object through a reference

Constructor With One Parameter

- If a constructor only has one parameter, there is a special way to pass an initial value to that constructor. For example, consider the following short program:

```
#include<iostream>
using namespace std;
class x
 {
  private:
   int a;
  public:
    x( int j)
      { a = j; }
```

Constructor With One Parameter

```
int geta ( )
  {  return a;  } };
int main( )
 {  x ob = 99; // passes 99 to j;
   cout << ob.geta( ); // outputs 99
   return 0;  }
```

- In general, whenever you have a constructor that requires only one argument, you can use either **ob(i)** or **ob = i** to initialize an object.

- **The reason for this is that whenever you create a constructor that takes one argument, you are implicitly creating a conversion from the type of that argument to the type of the class.**

Copy Constructor

- Whenever one object is used to initialize another, C++ performs a bitwise copy. That is, an identical copy of the initializing object is created in the target object.

- Although this is particularly adequate in most cases, there are situations in which a bitwise copy should not be used.

- One of the most common situations is when an object allocates memory when it is created.

Copy Constructor

- The same type of problem can occur in two additional ways:
  - when a copy of an object is made when it is passed as an argument to a function
  - when a temporary object is created as a return value from a function.

- C++ allows you to create a copy constructor, which the compiler invokes when one object initializes another.

- When a copy constructor exists, the default bitwise copy is bypassed.

Copy Constructor

- The common form of a copy constructor is:

```
classname (const classname &o)
 {    // body of constructor }
```

- There are two distinct situations in which the value of one object is assigned to another.
  - assignment.
  - initialization

Copy Constructor

- Initialization, which can occur in any one of the following three ways:

  -When one object explicitly initializes another, such as in a declaration.

  -When a copy of an object is made to be passed to a function.

  -When a temporary object is generated (most commonly as a return value of a function).

- **The copy constructor applies only to initializations.**

Copy Constructor

```
#include<iostream>
#include<stdlib>
Using namespace std;
Class array
 {
  int *p;
  int size;
  public:
   array (int sz)
    { try {
          p = new int [sz];
         }
      catch (bad_alloc xa)
       { cout << “allocation failure\n”;
         exit(EXIT_FAILURE); }
    size = sz; }
```

Copy Constructor

```
~array ( )
 { delete [ ] p; }
array (const array &a); // copy constructor
void put (int i, int j)
 { if (i >= 0 && i < size)
    p[i] = j; }
int get (int i)
 {  return p[i]; } };
array:: array (const array &a) //Copy constructor
 { int i;
   try {
      p = new int[ a.size];
      }catch (bad_alloc xa)
       {cout << "Allocation Failure\n";
         exit (EXIT_FAILURE); }
     for (i=0; i < a.size; i++)
      p[i] = a.p[i]; }
```

Copy Constructor

```
int main( )
 {
  array num(10);
  int i;
  for (i = 0; i < 10; i++)
    num.put (i ,i);
  for (i = 9; i >= 0; i--)
    cout << num.get(i);
  //create another array and initialize with num
  array x(num); // invoke copy constructor
  for (i = 0, i < 10; i++)
   cout << x.get( i);
   return 0;
 }
```

Arrays of Objects

- In C++, it is possible to have arrays of objects. The syntax for declaring and using an object array is exactly the same as it is for arrays of primitive data types.

- This example uses a three-element array of objects:
- #include<iostream>
- using namespace std;
- class c1
- {
- private:
- int i;
- public:
- void set_i (int j)
- { i = j; }

Arrays of Objects

```
int get_i( )
  { return i; } };
int main( )
  {
   c1 ob[3];
   int i;
   for (int i = 0; i < 3; i++)
    ob[i].set_i(i+1);
   for ( i = 0; i < 3; i++)
      cout << ob[i].get_i( ) << "\n";
   return 0;
  }
```

Arrays of Objects

- If a class defines a parameterized constructor, you may initialize each object in an array by specifying an initialization list, just like you do for arrays of primitive data types.

- However, the exact form of the initialization list will be decided by the number of parameters required by the object's constructor function.

- For objects, whose constructor functions have only one parameter, you can simply specify a list of initial values, using the normal array initialization syntax.

Arrays of Objects

- As each element in the array is created, a value from the list is passed to the constructor's parameter. The following example illustrates this point:

```
#include<iostream>
using namespace std;
class c1
 {
   private:
    int i;
   public:
    c1( int j)
     { i = j; }
```

Arrays of Objects

```
int get_i( )
  { return i; } };

int main( )
 {
   c1 ob[3] = {1, 2, 3};
   int i;
   for ( i = 0, i < 2, i++)
    cout << ob[i].get_i( ) << "\n";
   return 0; }
```

Arrays of Objects

- The following example illustrates passing list to a two parameter constructor

```
#include<iostream>
using namespace std;
class c1
 {
  private:
   int h, i;
  public:
   c1( int j, int k)
    { h = j;
      i = k; }
   int get_i( )
    { return i; }
```

Arrays of Objects

```
int get_h( )
  { return h; }

int main ( )
 {
  c1 ob[3] = {c1(1, 2), c1(3, 4), c1(5, 6)} // initialize
  int i;
  for ( i = 0; i < 3; i ++)
   {
    cout << ob[i].get_h( ) << “, “<< ob[i].get_i( ) << “\n”;
   }
  return 0
 }
```

Creating Initialized & Non-Initialized Arrays

- A special case situation occurs if you intend to create both initialized and non-initialized arrays of objects. Consider the following case:

```
class c1
 {
  private:
   int i;
  public:
    c1( int j)
     { i = j; }
    int get_i( )
     { return i; }
 };
```

Creating Initialized & Non-Initialized Arrays


```
class c1
 {
  private:
    int i;
  public:
    c1( )
     { i = 0; } //called for non-initialized arrays
    c1( int j)
      { i = j; }// called for initialized arrays
    int get_i( )
      { return i; } };
```

- Given the preceding class declaration, both of the following statements are permissible:
- c1 a1[3] = {3, 5, 6}; //initialized
- c1 a2[34]; //un-initialized

Pointers to Objects

- Just as you can pointers to primitive data types, you can have pointers to objects as well. When accessing members of a class given a pointer to an object, use the arrow operator (->)

- When a pointer is incremented, it points to the next element of its type in an array. All pointer arithmetic is relative to the base type of the pointer, i.e., it is relative to the type of data that the pointer is declared as pointing to.

- The same is true of pointers to objects.

Pointers to Objects

```
#include<iostream>
using namespace std;
class c1
 {
  private:
   int i;
  public:
   c1 ( )
    { i = 0; }
   c1 (int j)
    { i = j; }
   int get_i( )
     { return i; }
  };
```

Pointers to Objects

```
int main( )
 {
  c1 ob[3] = {1, 2, 3 };
  c1 *p; // pointer p of class type c1
  int i;
  p = ob; // get starting offset of array
 for ( i = 0; i < 3; i++)
  {
    cout << p->get_i( ) << "\n";
    p++; // point to the next object in the array
  }
  return 0;
 }
```

Pointers to Derived Types

- Assume two classes B and D. Further, assume that D is derived from the base class B. In this situation, a pointer of type B may also point to an object of type D.

- **To generalize, a base class pointer can also be used as a pointer to an object of any class derived from that base. The reverse, however, does not hold true.**

- Using a base class pointer, you can access only the members of the derived type that were inherited from the base. But, using a base class pointer, you cannot access members added by the derived class.

Pointers to Derived Types

- Consider the following program:

```
#include<iostream>
using namespace std;
class base
 {
  private:
   int i;
  public:
   void set_i( int num)
    { i = num; }
   int get_i( )
    { return i; }
};
```

Pointers to Derived Types

```
class derived : public base
 {private:
    int j;
  public:
    void set_j( int num)
     { j = num; }
    int get_j( )
     { return j; }  };

int main( )
 { base *bp;
  derived d;
  bp = &d; // base pointer points to derived object
  bp->set_i( 10); //access inherited members from derived object
  cout << bp->get_i( ) << " ";
  bp->set_j(22); //ERROR
 cout << bp->get_j( ); // ERROR
 return 0; }
```

Passing Object References

- When an object is passed as an argument to a function, a copy of that object is made. When the function terminates, the copy's destructor is called.

- If you do not want the destructor function to be called, simply pass the object by reference. When you pass by reference, no copy of the object is made.

- This means that no object used as a parameter is destroyed when the function terminates, and the parameter's (object's) destructor is not called.

Passing Object References

```
#include<iostream>
using namespace std;
class c1
 {
  int id;
  public:
   int i;
   c1( int num)
   ~c1( );
   void negate(c1 &ob)
     {ob.i = -ob.i; } };
c1::c1(int num)
  {
   cout << "constructing " << num << "\n";
   id = num;
  }
```

Passing Object References

- c1::~c1( )
- { cout << “destructing “ << id << “\n”; }
- int main( )
- {c1 o(1);
- o.i = 10;
- o.negate(o);
- cout << o.i << “\n”;
- return 0; };

- Here is the output of the program
- Constructing 1
- -10
- Destructing 1

The Dot . Operator

- **When you access a member of a class through a reference, you use the dot operator.**
- The arrow operator is reserved for use with pointers only.
- Passing all but the smallest objects by reference is faster than passing them by value. Arguments are usually pushed onto the stack.
- Thus, large objects take a considerable amount of CPU cycles to push onto, and pop from the stack.

Returning References

- A function may return a reference. This has the startling effect of allowing a function to be used on the left hand side of an assignment statement. Consider the following program:

```
#include<iostream>
using namespace std;
char& replace (int i); //returns a reference to a character
char s[80] = “hello there”;
int main( )
 {replace(5) = ‘x’ //assign x to space after hello
   cout << s;
   return 0; }
char& replace( int i)
 { return s[i]; }
```

Independent References

- You can declare a reference that is simply a variable. This type of reference is called an independent reference. An independent reference is another name for an object variable.

- All independent variables must be initialized when they are created. Apart from initialization, you cannot change what object a reference variable points to.

Independent References

```
#include<iostream>
using namespace std;
int main( )
{ int a;
 int &ref = a; // independent reference
 a = 10;
 cout << a << " " << ref << "\n";
 ref = 100
 cout << a << " " << ref << "\n";
 int b = 19;
 ref = b; // this puts b's value into a
cout << a << " " << ref << "\n";
ref - -
cout << a << " " << ref << "\n";
 return 0;
}
```

References to Derived Types

- **A base class reference can be used to refer to an object of a derived class of that base class.**
- The most common application of this is found in function parameters.
- A base class reference parameter can receive objects of the base class as well as any other type derived from that base.

Restrictions on References

- You cannot reference another reference. In other words, you cannot obtain the address of a reference.
- You cannot create arrays of references.
- You cannot create a pointer to a reference.
- A reference variable must be initialized. Null references are prohibited.

Summary

In this lesson, you learnt to:

- Use two special forms of constructors
- Create arrays of objects
- Access an object through a pointer
- Access an object through a reference

Virtual Functions and Polymorphism

Objectives

In this lesson, you will learn to:

- Describe static, or early, or compile-time binding
- Describe dynamic, or runtime, or late binding
- Describe a virtual function
- Employ a virtual function to implement dynamic binding
- Achieve runtime polymorphism through dynamic binding, a base class pointer, and virtual functions

Virtual Functions

- A virtual function is a member function that is declared within a base class, and is redefined by a derived class.
- To create a virtual function, precede the function's declaration in the base class with the keyword **virtual**.
- When a class containing a virtual function is inherited, the derived class redefines or overrides the virtual function to fit its own needs.

Virtual Functions

- Virtual functions implement the “single method; multiple implementations” paradigm intrinsic to polymorphism.
- The virtual function within the base class defines the form of the interface to the function.
- Each redefinition of the virtual function by a derived class implements its operation as it relates specifically to the derived class. That is, the redefinition creates a specific method.

Virtual Functions

- When a base class pointer points to a derived class object that contains a virtual function, C++ determines which version of that function to call based upon the type of the object pointed to by the pointer.

- And this determination is made at runtime.

- Thus, when different derived class objects are pointed to by the base class pointer at different points of time, different versions of the virtual function are executed.

Virtual Functions


```
#include<iostream>
using namespace std;
class base
 {
  public:
   virtual void vfunc( )
     { cout << “this is base’s vfunc( )\n”; } };

class derived1: public base
 {
  public:
   void vfunc( )
     {  cout << “this is derived1’s vfunc( )\n”; } };
```

Virtual Functions

```
class derived2 : public base
 {
  public:
   void vfunc( )
     { cout << “this is derived2’s vfunc( )\n”; } };

int main( )
 { base *p, b;
  derived1 d1;
  derived2 d2;
  p = &b;
  p->vfunc( ); //access to base’s vfunc( )
  p = &d1;
  p->vfunc( ); // access derived1’s vfunc( )
  p = &d2;
  p->vfunc( ); // access derived2’s vfunc( )
  return 0;  }
```

Virtual Functions

- The key point here is that the kind of object to which p points to determines which version of vfunc( ) is executed.
- Further, this determination is made at runtime, and this process forms the basis of runtime polymorphism.
- Although you can call a virtual function in the normal manner by using a object's name and the dot operator, *it is only when access is through a base-class pointer (or reference) that runtime polymorphism is achieved.*

Virtual Functions

- A function declared as virtual in the base class, and redefined in the derived classes is the implementation of overridden functions.
- The prototype for a redefined or overridden virtual function must exactly match the prototype specified in the base class.
- Prototype encompasses not only signature, but also the return data type of a function.

Calling a Virtual Function Through a Base Class Reference

- **The polymorphic nature of a virtual function is also available when called through a base class reference.**

- Thus a base class reference can be used to refer to an object of the base class, or any object derived from that base.

- When a virtual function is called through a base class reference, the version of the function executed is determined by the object being referred to at the time of call.

Calling a Virtual Function Through a Base Class Reference

- The most common situation in which a virtual function is invoked through a base class reference is when the reference is defined as a function parameter. Consider the following program:

```
#include<iostream>
using namespace std;
class base
 {
  public:
   virtual void vfunc( )
     {  cout << "this is base's vfunc( )\n"; } };

class derived1: public base
 {
  public:
   void vfunc( )
    { cout << "this is derived1's vfunc( )\n"; } };
```

Calling a Virtual Function Through a Base Class Reference


```
class derived2 : public base
 {
  public:
   void vfunc( )
     { cout << "this is derived2's vfunc( )\n"; } };
void f(base &r)
 {  r.vfunc( ); }
int main( )
 { base b;
  derived d1;
  derived d2;
  f(b); // pass a base object to f( )
  f(d1); // pass a derived object to f( )
  f(d2); // pass a derived object to f( )
  return 0; }
```

The Virtual Attribute is Inherited

- When a virtual function is inherited, its virtual nature is also inherited.
- This means that when a derived class that has inherited a virtual function is itself used as a base class for another derived class, the virtual function can still be overridden.
- *No matter how many times a virtual function is inherited, it remains virtual.*

The Virtual Attribute is Inherited


```
#include<iostream>
using namespace std;
class base
 {
  public:
   virtual void vfunc( )
     { cout << “this is base’s vfunc( )\n”; } };

class derived1: public base
 {
  public:
   void vfunc( )
     { cout << “this is derived1’s vfunc( )\n”; } };
```

The Virtual Attribute is Inherited

```
class derived2 : public derived1
 {
  public:
   void vfunc( )
     { cout << “this is derived2’s vfunc( )\n”; } };
int main( )
 { base *p, b;
  derived1 d1;
  derived2 d2;
  p = &b;
  p->vfunc( ); //access to base’s vfunc( )
  p = &d1;
  p->vfunc( ); // access derived1’s vfunc( )
  p = &d2;
  p->vfunc( ); // access derived2’s vfunc( )
  return 0;  }
```

Virtual Functions are Hierarchical

- When a function is declared as virtual by a base class, it may be overridden by a derived class. However, the function does not have to be overridden.

- *When a derived class fails to override a virtual function, then, when an object of the derived class accesses that function, the function defined by the base class is used.*

- Consider this program in which class derived2 does not override vfunc( )

Virtual Functions are Hierarchical


```
#include<iostream>
using namespace std;
class base
 {
  public:
   virtual void vfunc( )
     { cout << “this is base’s vfunc( )\n”; } };

class derived1: public base
 {
  public:
   void vfunc( )
     {  cout << “this is derived1’s vfunc( )\n”; } };
```

Virtual Functions are Hierarchical


```
//derived2 inherits virtual function from derived1
class derived2 : public derived1
 {  // vfunc( ) not overridden by derived2; base's is used  };

int main( )
 { base *p, b;
   derived1 d1;
   derived2 d2;
   p = &b;
   p->vfunc( ); //access to base's vfunc( )
   p = &d1;
   p->vfunc( ); // access derived1's vfunc( )
   p = &d2;
   p->vfunc( ); // access base's vfunc( ) since derived2 does not override vfunc( )
   return 0; }
```

Pure Virtual Functions

- When a virtual function is not redefined by the derived class, the version defined in the base class will be used. However, in many situations, there cannot be any meaningful definition of a virtual function within a base class.

- For example, a base class may not be able to define an object sufficiently to allow a base class virtual function to be created.

- Further, in some situations, you will want to ensure that all derived classes compulsorily override a virtual function.

Pure Virtual Functions

- To handle these two situations, C++ supports the **Pure Virtual Function.** A pure virtual function is a virtual function that has no definition in the base class.

- To declare a pure virtual function, use this general form:
- virtual type func_name (parameter_list) = 0;

- When a virtual function is declared pure, any derived class **must** provide its own definition of the virtual function. If the derived class fails to override the pure virtual function, a compile-time error will ensue.

Pure Virtual Functions

- The base class **number** contains an integer called **val,** the function setval( ) and the pure virtual function show( )

- The derived class hextype, dectype, and octtype inherit number, and redefine show( ) so that it outputs the value of val in each number base (i.e., hexadecimal, decimal or octal).
- #include<iostream>
- using namespace std;
- class number
- {
- protected:
- int val;
- public:
- void setval(int i)
- { val = i;}
- virtual void show( ) = 0; };

Pure Virtual Functions


```
class hextype : number
 {
  public:
   void show( )
    { cout << hex << val << "\n"; } };

class dectype : number
 { public:
   void show( )
    { cout << val << "\n"; } };

class octtype : number
 { public:
   void show( )
    { cout << oct << val << "\n"; } };
```

Pure Virtual Functions

```
int main( )
 {
  number *p;
  dectype d;
  hextype h;
  octtype o;
  p = &d;
  d.setval(20);
  p->show( ); // displays 20 – decimal
  p = &h;
  h.setval(20);
  p->show( ); // displays 14 – hexadecimal
  p = &o;
 o.setval(20);
 p->show( ); // displays 24 –octal
 return 0; }
```

Virtual Function Mechanics – The Virtual Table

- C++ implements late binding by setting up a **vtable.** The keyword **virtual** tells the compiler it should not perform early binding.

- Instead, it should automatically install all the mechanisms necessary to perform late binding. The compiler creates a single table called VTABLE for each class that contains virtual functions.

- The compiler places the addresses of the virtual functions for that particular class in the VTABLE. A virtual table is therefore an array of virtual function pointers stored by the compiler as a table called as VTABLE.

Virtual Function Mechanics – The Virtual Table

- Each instance of the class has a pointer to its class wide VTABLE. Using this, the compiler can achieve dynamic binding.

- When you make a virtual function call through a base-class pointer, the compiler quietly inserts code to fetch the VPTR and look up the function address in the VTABLE, thus calling the right function and causing late binding to take place.

- With virtual functions, the proper function gets called for an object, even if the compiler cannot know the specific type of the object.

Abstract Classes

- A class that contains at least one pure virtual function is said to be abstract. An abstract class cannot be instantiated since it has one or more pure virtual functions.

- Instead, an abstract class constitutes an incomplete type that is used as a foundation for derived classes. Although you cannot create objects of an abstract class, you can create pointers and references to an abstract class.

- This allows abstract classes to support runtime polymorphism, which relies upon base class pointers or references to select the proper virtual function.

Using Virtual Functions

- One of the central aspects of Object-Oriented Programming is the principle of “one interface, multiple methods”.

- This means that a general class of actions can be defined, the interface to which is constant, with each derivation defining its own specific operations.

- In concrete C++ terms, the base class can be used to define the nature of the interface to a general class.

Using Virtual Functions

- Each derived class then implements the specific operations as they relate to the type of data used by the derived type.

- One of the most powerful and flexible ways to implement the “one interface, multiple methods” approach is to use abstract base classes, pure virtual functions, and base class references or pointers.

- Using these features, you can define a class hierarchy that moves from general to the specific (base to derived).

Using Virtual Functions

- define all common features and interfaces in a base class. In cases where certain actions can be implemented only by the derived class, use a virtual function.

- Therefore, in the base class, you create and define everything you can that relates to the general class. The derived class fills in the specific details.

Using Virtual Functions

- Consider the following example. A class hierarchy is created that performs conversions from one system of units to another (for example, litres to gallons).

```
#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;};
```

Using Virtual Functions

```
//litres to gallons
class l_to_g : public convert
 {  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; } };
```

Using Virtual Functions


```
int main( )
 {
  convert *p // pointer to abstract base class
  l_to_g lgob(4);
  f_to_c fcob(70);
 // use virtual function mechanism to convert
  p = &lgob;
  cout << p->getinit( ) << litres is “;
  p->convert( );
cout << p->getconv( ) << “gallons\n”;
p = &fcob;
cout << p->getinit( ) << in fahrenheit is “;
p->compute( );
cout << p->getconv( ) << “celsius\n”;
return 0;
 }
```

Early Vs. Late Binding

- Early binding refers to events that occur at compile time. In essence, early binding occurs when all information needed to call a function is know at compile time.

- In other words, early binding means that an object and a function call are bound during compilation.

- Examples of early binding includes normal function calls (including standard library functions), overloaded function calls, and overloaded operators.

Early Vs. Late Binding

- The advantage of early binding is efficiency. Because all information necessary to call a function is determined at compile time, these types of function calls are very fast.

- The opposite of early binding is late binding. As it relates to C++, late binding refers to function calls that are not resolved until runtime.

- **Virtual functions are used to achieve late binding.**

Early Vs. Late Binding

- When access is via a base class pointer or reference, the virtual function actually called is determined by the type of object pointed to by the pointer.
- Because in most cases, this cannot be determined at compile time, the object and the function are not linked until runtime.
- The main advantage to late binding is flexibility.

Early Vs. Late Binding

- Unlike early binding, late binding allows you to create programs that can respond to events occurring while the program executes, without having to create contingency or conditional code.

- Because a function is not resolved until runtime, late binding can make for somewhat slower execution times.

Virtual Destructor

- When using dynamic binding all the instances of a class may not be properly disposed off. As delete to a pointer to a base class will call the destructor of the base class only.

- But if destructor is declared virtual it will call the inherited class destructor as well, thus properly disposing the class instances.

Summary

In this lesson, you learnt to:

- Describe static, or early, or compile-time binding
- Describe dynamic, or runtime, or late binding
- Describe a virtual function
- Employ a virtual function to implement dynamic binding
- Achieve runtime polymorphism through dynamic binding, a base class pointer, and virtual functions

Friend Functions

Objectives

At the end of this Lesson, you will learn to:

- Define friend function
- Implementing friend function
- Define friend class

Friend Functions

- An external function cannot access the non-public members of an object.
- The only way to access the data within objects is through messages, which involve a call to a public member function that in turn accesses the non-public data.
- These function calls to access the non-public data slows down the execution of code owing to the overhead of additional function calls.

Friend Functions

- To enable the programmer to avoid the overhead of using function calls, C++ provides the **friend** facility.

- Syntax:

  friend return type Function Name()

  { statements}

- Using friend function we can access all the members of a class from outside the class

Friend Functions

- Example:
- class mks_distance
- {private:
- int meter;
- int cm;
- public:
- mks_distance(int, int);
- void disp_distance(void);
- friend int compare(mks_distance &, mks_distance &); };
- mks_distance:: mks_distance(const int mt = 0, const int cmt = 0)
- {meter = mt;
- cm = cmt; }

Friend Functions


```
• void mks_distance::disp_distance( void)
• { cout << "the distance = " << meter << '.' << cm << "metres\n"; }

• int compare( mks_distance &m1, mks_distance &m2)
• {
•  // Accessing private members of objects of mks_distance
•  int m_diff = m1.meter – m2.meter;
•  int cm-diff = m1.cm – m2.cm;
•  return( m_diff * 100 + cm_diff);
• }
```

Friend Functions

- Delaring a function as a friend within a class does not make it a member of the class.
- Since a friend function is not a member of the class, it can be declared in the public, private, or protected section of the class without affecting its visibility.
- A function that is a friend of a particular class could be either a global function, or a member function of another class.

Building Bridges of Friendship

- Friend function can access the non-public members of more than one class.

- **A friend function can act as a bridge between unrelated classes**

  - By declaring a function as a friend in more than one class, it can be made to access their non-public members.

Building Bridges of Friendship

Example:

- **class mks_distance; // forward declaration**

```
class fps_distance
 {private:
    unsigned int feet;
    float inch;
 public:
   fps_distance(unsigned int, float); // constructor
  void disp_distance (void);
/* the friend function feet_to_meter( ) can access the private members of
     this class */
friend void feet_to_meter( fps_distance &, mks_distance & ); };
fps_distance :: fps_distance( unsigned int ft = 0, float in = 0)
 {feet = ft;
   inch = in;}
```

Building Bridges of Friendship


```
void fps :: disp_distance( void)
 {cout << "The distance = " << feet << "'" << inch << "," << '\n'; }

class mks_distance
  {private:
     int meter;
     int cm;
public:
   mks_distance( int, int);
   void disp_distance( void);
};
mks_distance:: mks_distance( const int mt = 0, const int cmt = 0)
 {meter = mt;
   cm = cmt; }
```

Building Bridges of Friendship

- void mks_distance::disp_distance(void)
- {cout << “the distance = “ << meter << ‘.’ << cm << “metres\n”;}
- friend void feet_to_meter(fps_distance &, mks_distance &);
- };

- /*Function to convert feet and inches to meter and centimeters. The function receives an object of the fps_distance and the mks_distance class as parameters. */
- void feet_to_meter(fps_distance &fps, mks_distance &mks)
- {
- int temp = (((fps.feet 12 + fps.inch) / 39 * 100);
- mks meter = temp / 100;
- mks.meter = temp % 100;
- }

Forward Declaration

- **Forward Declaration** : Declaring a class, without having it's implementation immediately followed by.
- Forward declaration tells the compiler that class definition comes later in the program.
- The forward declaration is made globally (outside any class declaration) before a reference is made to the class.

Friend class

- Class can be declared as a friend of another class.
- Friend class can access all the members (including private and protected) of another class directly using object
- Friend declarations can go in either the public, private, or protected section of a class.it doesn't matter where they appear.
- Access is granted for a class using the class **keyword** as **friend class aclass**;

Example:Friend class

Example:

```
class A
{
private:
int X;
void setx(int p)
{
X=p;
}
void disx()
{
cout<<X;
}
friend class B;
};
```

Example: Friend class

```
class B
{
private:
int Y;
public:
void setxy()
{
A obj;
Obj.setx(10);//accessing private data of class A
Obj.disx();
}
};
```

```
main()
{
B Obj;
Obj.setxy();

}
```

Summary

At the end of this Section, you learnt to:

- Define friend function
- Implementing friend function
- Define friend class

Operator Overloading

Objectives

At the end of this lesson, you will be able to:

- Describe operator overloading and its functionality
- Implement overloaded operators
- Overload unary and binary operators
- Use conversion functions
- Friend operator function

Operator Overloading

- Most fundamental data types have pre-defined operations associated with them.
- To make a user-defined data type as natural as a fundamental data type, the appropriate set of operators must be associated with it.
- C++ allows to Define additional meaning to a existing operator without changing its basic meaning

  -Operator overloading

Operator Overloading

- To add two objects of the same class , a member function needs to be invoked.
  - Obj3.addFun(obj1,obj2)
- User have to remember functions names for all operations

- Understanding of operations will be more easier when we use operators

Operator Overloading

- adding the two objects and storing the result in a third object could be carried out as follows:
  - object3 = object1 + object2;

- The user can understand the operation more easily as compared to the function call because it is closer to a real-life implementation.

- Thus by associating a set of meaningful operators, manipulation of an ADT can be done in a conventional and simple form. Associating operators with ADTs involves overloading the operators.

Operator Overloading

- Only the predefined set of C++ operators can be overloaded, no new operator can be introduced.
- An operator can be overloaded by defining a function for it.
- The function for the overloaded operator is declared using the **operator** keyword.
- For example, if the = = operator is overloaded in the fps_distance class, the declaration for the operator function is as follows:

Operator Overloading

- int fps_distance :: operator = =(fps_distance);
- This instruction tells the compiler to call the operator = =( ) function whenever the = = operator is encountered, provided the left hand side operand is of type fps_distance.
- The right hand side operand is passed on to the operator function as an argument.
- For example, the expression:
  - X = R = = F translates to
  - X = R.operator= =(F);
  - The return value of type int of the operator function is stored in X.

Precautions When Overloading Operators

- When operators are overloaded, it does not change the predefined sequence of execution of the operators. This is true regardless of the class or implementation.

- The expression syntax of the C++ operators cannot be overloaded. For example, it is not possible to overload the '%' operator as a unary operator.

- Neither is it possible to overload the ++ operator as a binary operator.

Overloading Unary Operators

- Unary operators act on one operand. As an example, you will now overload the + + operator for the fps_distance to increment the data member feet by 1.
- #include(iostream>
- class fps_distance
- { private:
- int feet;
- float inch;
- public:
- fps_distance (int f, float i)
- {feet = f;
- inch = i; }
- operator ++( )
- { feet++; }
- void disp_distance( )
- { cout << “distance = “ << feet << “ ‘” << inch << “ “ “; } };

Overloading Unary Operators

```
void main ( )
 {
  fps_distance fps(10,10)
  ++fps;
  fps.disp_distance( );
 }
```

- The data member feet of the fps object is incremented using the overloaded operator ++.

- Since the operator ++ is a member function in class fps_distance, it can access all members of the class.

Overloading Prefix & Postfix Unary Operators

- Unary operators work with one operand. Hence, the operator function does not need any parameter. Note that the + + operator has been used as a prefix operator.

- When used with fundamental data types, the prefix application of the operator causes the variable to be incremented first before it is used in an expression.

- The postfix application of the operator causes the value to be incremented after the value is used in an expression.

Distinguishing Between Postfix & Prefix Unary Operators

- The ++ operator used in the fps_distance class earlier cannot be used as a postfix operator.
- **An operator function with no arguments is invoked by the compiler for the prefix application of the operator.**
- **The compiler invokes the operator function with an int argument for the postfix application of the operator.**
- **int operator ++(int)**

Distinguishing Between Postfix & Prefix Unary Operators

- Overloading the - - operator for the fps_distance class:
- Prefix Implementation:
- fps_distance operator - - ( )
- {feet --;
- fps_distance temp(feet, inch);
- return temp; }

- Postfix Implementation:
- fps_distance operator - - (int)
- { //create a temporary object with the original value of feet
- fps_distance temp (feet, inch);
- feet- -;
- // return temp; }

Nameless Objects

- The solution to the earlier exercise can be modified as follows:

```
fps_distance operator - - ( )
 {
  feet - -;
  return fps_distance(feet, inch);
 }
```

- This method saves us one step of creating a temporary object called **temp**. Since the function is not accessing the new object created, it does not have a name.

Overloading Binary Operators

- Overloading a binary operator is similar to overloading a unary operator. Binary operators are those operators which work on two operands.

- The only difference is that the binary operator function requires only one more parameter.

- You will now overload the binary “+” operator for adding two objects of the fps_distance class.

Overloading Binary Operators

```
#include<iostream>
class fps_distance
 {private:
   int feet;
   float inch;
  public:
   fps_distance( int, float);
   fps_distance operator +( fps_distance&);
   void disp_distance( ); };
fps_distance :: fps_distance( int f = 0; i = 0.0)
{ feet = f;
 inch = i; }
```

Overloading Binary Operators

```
fps_distance fps_distance :: operator + (fps_distance &fps2)
 {int f = feet + fps2.feet;
  float i = inch + fps2.inch;
  if ( i >= 12)
   { i - = 12;
     f++; }
    return fps_distance( f, i); //creating an unnamed object and returning it }

void fps_distance :: disp_distance( )
 { cout << "distance = " << feet << inch; }
void main( )
 {fps_distance fps1(11, 11), fps2(10, 10), fps3;
 fps3 = fps1 + fps2;
 fps3.disp_distance( ); }
```

Overloading Binary Operators

- When an overloaded binary operator is invoked, the object on the left hand side of the operator is the object of which the operator is a member, and the variable or the constant on the right hand side is the parameter to the operator function.

- In general, the declaration of an overloaded operator always requires one less argument than the number of operands, as one operand is the object of which the operator function is a member.

Overloading Overloaded Operators

- A class can have more than one function for the same operator, i.e., the same operator can be overloaded.

- For example, you might want to add a constant distance to the object of the fps_distance class. The constant distance can be in inches.

  fps2 = fps1 + 20; //adding a constant

- The program for overloading the '+' operator given earlier cannot handle this case. We have to define another + operator function with a different kind of parameter.

Overloading Overloaded Operators

- The following are the two operator functions:

- fps_distance fps_distance :: operator + (fps_distance &fps2)
- {int f = feet + fps2.feet;
- float i = inch + fps2.inch;
- if (i >= 12)
- { i - = 12;
- f++; }
- return fps_distance(f, i); //creating an unnamed object and returning it }

- fps_distance fps_distance :: operator + (int inches)
- {int f = feet + (inches / 12);
- int i = inch + (inches % 12);
- if (i >= 12.0)
- {i- = 12;
- f++; }
- return fps_distance(f, i); }
-

Data Conversion

- The assignment operator (=) can be used to assign a value from one variable to another, or assign a constant value to a variable.

- The assignment operator can also be used to assign one object to another object provided the objects are of the same data type or class type.

- Thus, assignment between same types (fundamental and user-defined) are handled by the compiler on the condition that the data types on both the left-hand side and the right-hand side of the operator are the same.

Fundamental to User-Defined Conversion

- The compiler takes care of conversion for only fundamental data types. You may want to convert between user-defined and fundamental types.

- The following program illustrates the conversion of a fundamental data type to a user-defined data type. This is accomplished through a single-argument constructor.

- The program allows assigning distance in inches (integer) to an object using the assignment (=) operator.

Fundamental to User-Defined Conversion


```
#include<iostream>
class fps_distance
 {private:
   int feet;
   float inch;
  public:
   fps_distance( int inches)
    {feet = inches /12;
     inch = inches % 12; }

fps_distance( int f, float i)
 {feet = f;
   inch = i; }
```

Fundamental to User-Defined Conversion


```
void disp_distance( )
  {cout << feet << inch << "\n"; } };

void main( )
 {
   fps_distance dist1 = 39;
   dist1.disp_distance( );
   dist1 = 78;
   dist1.disp_distance( );
}
```

- When an object is created with one argument, the compiler calls the constructor with one argument.

User-Defined to Fundamental Conversion

- The following program converts a user-defined data type to a fundamental data type.
- #include<iostream>
- class fps_distance
- {private:
- int feet;
- float inch;
- public:
- fps_distance(int f, float i);
- {feet = f;
- inch = i; }
- void disp_distance( )
- {cout << feet << "-" << inch << "\n"; }
-

User-Defined to Fundamental Conversion

```
    operator int( )
•   {
•     return (feet * 12 + inch);
•   }
• };
• void main( )
•   {
•     fps_distance height( 6,3);
•     height.disp_distance( );
•     int inches = int( height);
•     cout << inches << "\n";
•   }
```

User-Defined to Fundamental Conversion

- When you want to convert from a user-defined data type to a fundamental data type, the trick is to overload the **cast** operator.

- *The overloaded cast operator is called the conversion operator that performs the conversion from a user-defined data type to a fundamental data type.*

- The operator takes the value of the object of which it is a member, and converts it into an inch. The int value is returned.

Overloading the Assignment Operator

- Consider the following RationalNum class:
- class RationalNum
- {private:
- int numerator;
- int denominator;
- public:
- void display( )
- { cout << numerator << "/" << denominator; }
- RationalNum(int x, int y)
- { numerator = x;
- denominator = y; } };
- Consider the following code segment:
- RationalNum a(5,9), b;
- b = a; // assignment
- b.display( ); // outputs 5/9

Overloading the Assignment Operator

- In our sample case, the default assignment operator function translates the statement b = a; as follows:
- b.numerator = a.numerator;
- b.denominator = a.denominator;

- **This type of assignment is called member-wise assignment.**

- In the absence of an assignment operator defined in the class, the C++ compiler builds this type of default assignment operator for each class.

Overloading the Assignment Operator

- If the class does not contain any pointer data members, this default assignment operator is adequate, and it causes no problems.

- If the class has pointer data members, the default assignment operator, supplied by the compiler, could lead to serious problems.

Overloading the Assignment Operator

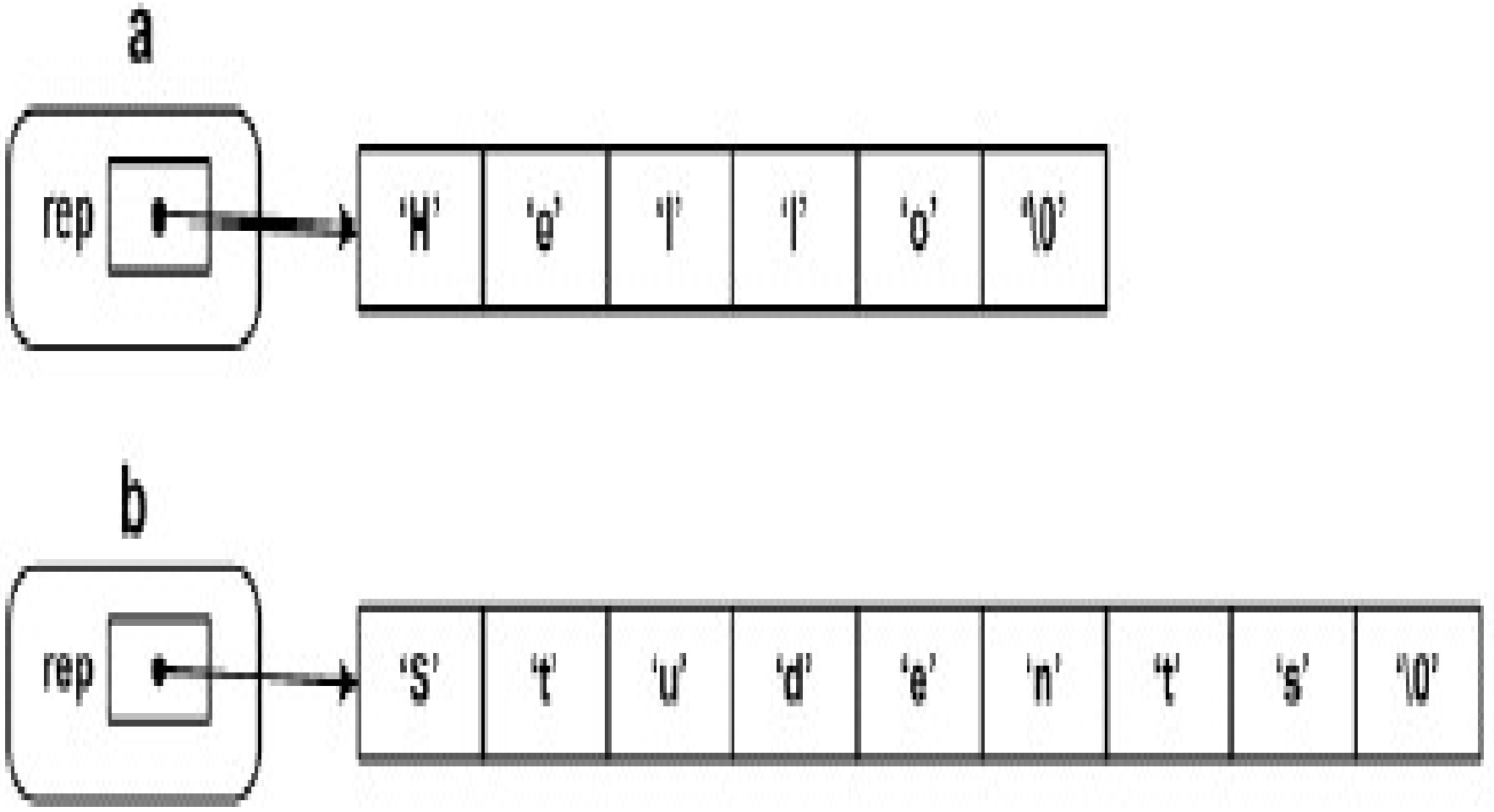
- illustrate it with a class that contains a pointer data member:

```
class MyString
 {private:
   char* rep;
  public:
   MyString( char* str)
    {rep = new char[ strlen(str) + 1];
     strcpy( rep, str); } };
```

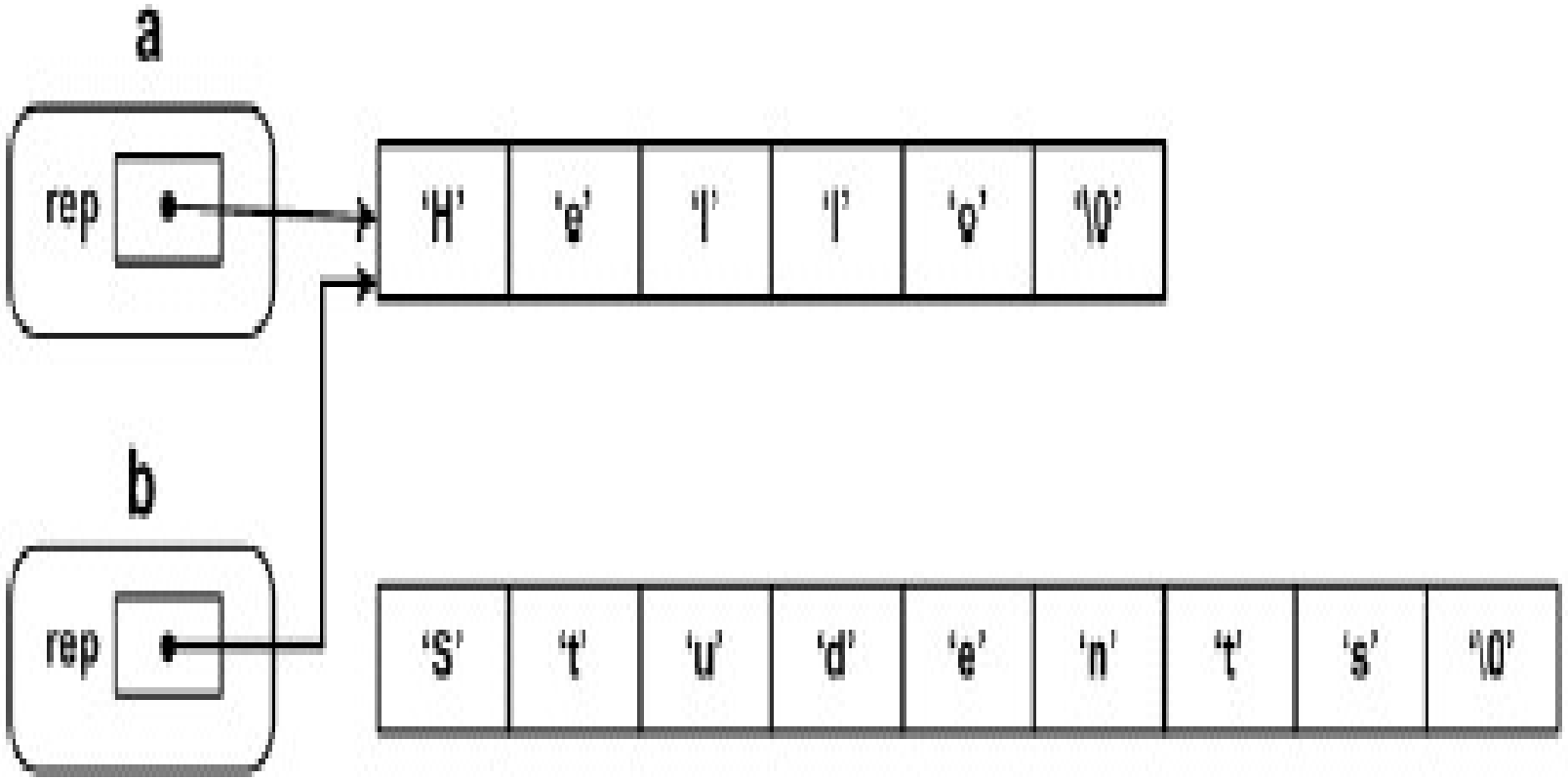
- The following statement:
- MyString a("Hello"), b("Students");
- Invokes the constructor MyString (char *) twice.

- The two calls, one for object **a**, and the other for object **b**, build two MyString objects as follows:

Overloading the Assignment Operator

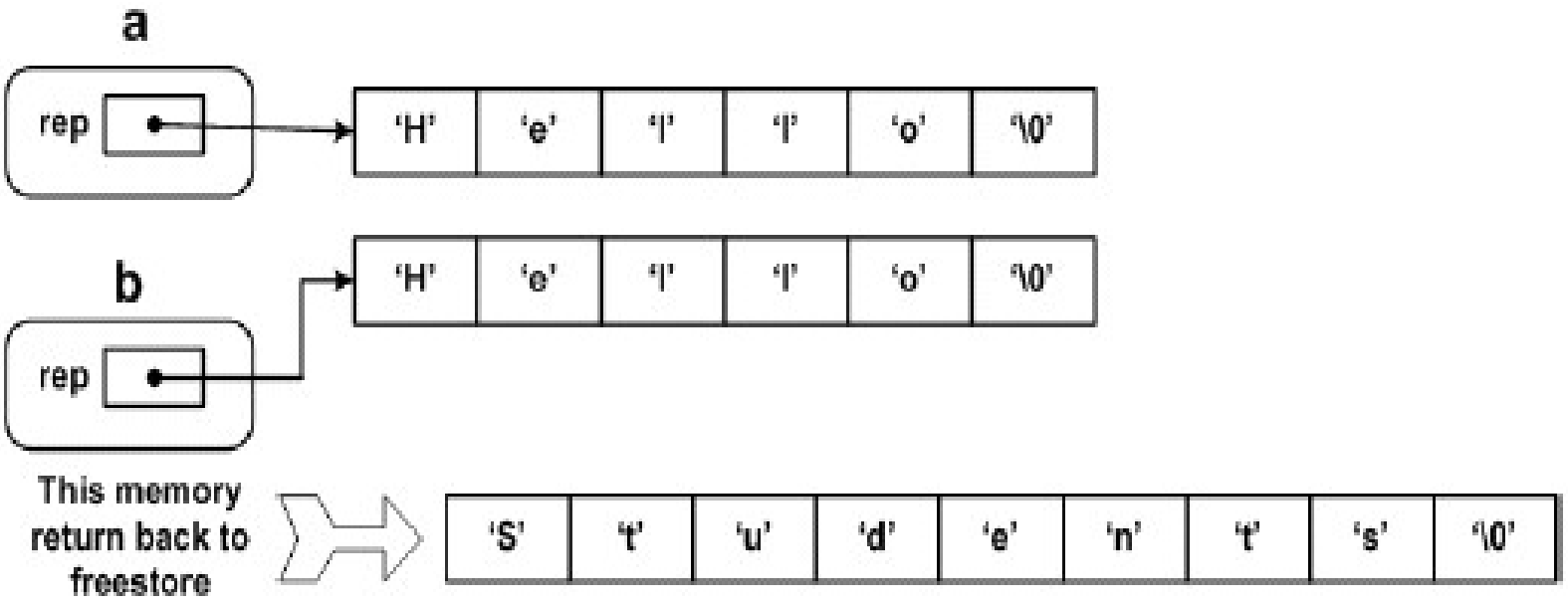


Overloading the Assignment Operator



Overloading the Assignment Operator

- What is really desired by the expression b = a; is that the strings themselves be copied, but not their pointers as shown below:



Overloading the Assignment Operator

- The assignment operator is overloaded for a given class with the following declaration in the class, where ClassName is the identifier for the class:
- const ClassName& operator= (const ClassName& rhs);

- Let's decode this a piece at a time, starting from the right.

- First, **rhs** (short for right hand side) is simply the identifier for the parameter which will appear to the right of the assignment operator.

Overloading the Assignment Operator

- It is standard to pass the parameter as a constant reference parameter which is indicated by the const and & in the parentheses.
- This is done for the usual reason that we do not want to needlessly copy a large structure.
- The name of this function is operator =. It is a member function, so if anObj and someObj were declared to be an objects of class ClassName, we could make an assignment as follows:
- anObj.operator=(someObj);

Overloading the Assignment Operator

- The following statement is the more common equivalent of the one above:
- anObj = someObj;

- The left part of the declaration of the operator = specifies the return type.

- Why a return type for the assignment operator when its work is done by assigning the value of one expression to another variable?

Overloading the Assignment Operator

- When we overload the assignment operator for a new class, we want the same syntax to work, so it must return the value assigned, which has type ClassName.

- However, to avoid extra copying and an implicit call to the copy constructor, we make the return value a constant reference parameter, hence the "const ClassName &".

Overloading the Assignment Operator

```
const MyString& operator= (const MyString& rhsObject)
 {
  if (this != &rhsObject)   // do not copy self
   {
     delete [ ] rep; // return memory pointed to by rep back to freestore
     rep = new char[ strlen( rhsObject.rep + 1]; // allocate new array
     strcpy( rep, rhsObject.rep);
     return *this;
   }
```

Overloaded Assignment Operator and the Copy Constructor

- On the face of it, the overloaded assignment operator looks functionally similar to a copy constructor but with two significant differences:
  - it returns something: a reference
  - it must cleanup on the *this* object before assigning

Friendly Operator

- While overloading the binary operator ‘ + ‘ the left side argument must be the object of which the overloaded operator is a member.

- Assume that the + operator is overloaded in the fps_distance class. Then it can be invoked as:
- height + 3 // height is an object of fps_distance class
- height + height2;
- // height and height are objects of the fps_distance class

- The conversion from int type to fps_distance is done by the single argument constructor.

Friendly Operator

- If this overloaded operator is a member of the fps_distance class, then it cannot be invoked as:
- 3 + height;

- This statement will not work because the object of which the + operator is a member must be on the left hand side.

- This property where A + B = B + A is know as the commutative property which is a natural property of the + operator when used with fundamental data types.

Friendly Operator

- If the operator is defined globally, then the operands on either side of the operator are sent down as parameters to to the operator function.
- If the operator + is global, then the same expression translates to operator +(f1, f2);
- Thus, the global operator can have any operand of any data type on the left hand side of an operation.
- Since the global operator function needs to access the member data of the operands, it is usually declared a friend of the operand class/classes.

Friendly Operator

```
class fps_distance
 {private:
  int feet;
  int inch;
 public:
  fps_distance (const int);
  fps_distance(int float);
  friend fps_distance operator + (fps_distance &, fps_distance &);
  void disp_distance( ); };
fps_distance :: fps_distance (const int i)
 { feet = i /12;
   inch = i % 12; }
fps_distance::fps_distance (int f = 0, float i = 0.0)
 {feet = f;
  inch = i; }
```

Friendly Operator

```
fps_distance operator + ( fps_distance &fps1, fps_distance fps2)
 {int f = fps1.feet +fps2.feet;
  float i = fps1.inch + fps2.inch;
  if (i >= 12.0)
   {i-= 12;
    f++;
   return fps_distance(f,i); }

void fps_distance :: disp_distance( )
 {cout << feet << "-" << inch << "'" << '\n'; }
void main( )
 {fps_distance f1(6,2), f2 = 78; f3
   f3 = f1+f2
   f3 = f2 + 2;
   f3 = 2 + f2; }
```

Summary

At the end of this lesson, you learnt to:

- Describe operator overloading and its functionality
- Implement overloaded operators
- Overload unary and binary operators
- Use conversion functions
- Friend operator function

C++ Streams

Objectives

At the end of this Lesson you will learn to:

- Define Stream and Stream classes
- Describe Stream extraction –input/output
- Describe Implicit file opening and closing
- Describe Explicit file opening and closing
- Use File open mode bits
- Use Stream Status Bits
- Use File Randam Access Functions
- Use Formated I/O Functions
- Describe Manipulators

Streams

- A stream is a source or a destination for a collection of characters, or a flow of data. Output streams allow you to store (write) characters, and characters are fetched (read) from input streams.

- The stream classes form a powerful set of classes that can be modified, extended, or expanded to include support for user-defined data types or objects.

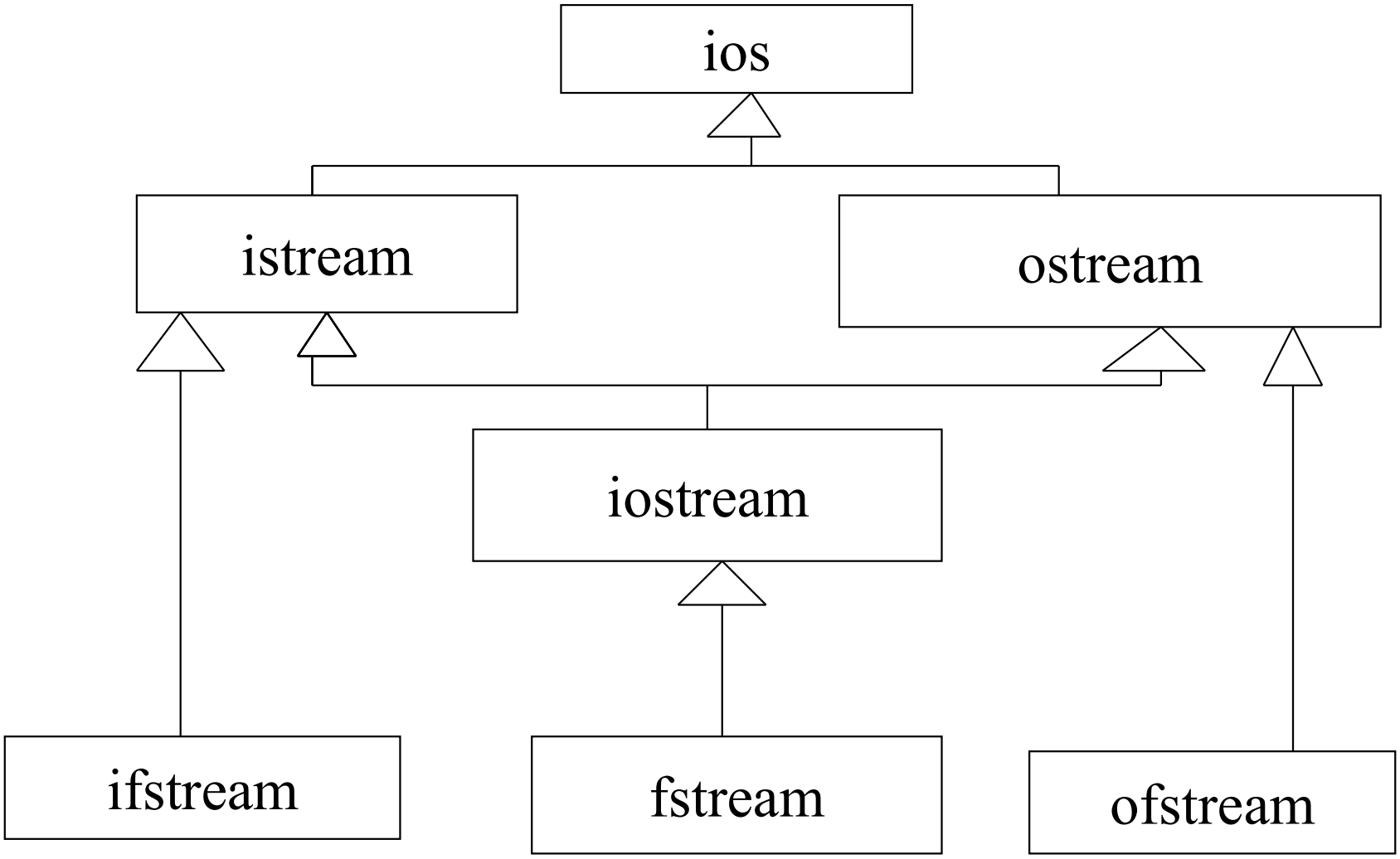
- They are fully buffered to reduce disk access.

Advantages of the Stream Classes

- They encapsulate their internal workings from the user. Thus, the programmer need not specify the type of data that is to be input or output; it is automatically determined by the stream class.

- They offer a rich set of error handling facilities.

The Stream Class Hierarchy



Standard Output

- Stream classes have their member data, functions, and definitions.
- Class ostream contains functions defined for output operations. These operations are called stream insertions. The << operator is called the inserter.
- **cout** is an object of type ostream defined in the header file iostream, and is attached to the standard output device, i.e., the screen.

Stream Extraction & Standard Input

- The opposite of insertion is extraction, which is the fetching of data from an input stream. Input stream operations are defined in istream class.

- The overloaded operator >> (right-shift bitwise operator) is called the extractor.

- The >> operator is overloaded to accept three data types, namely integers, floating points, and characters.

Integer Extraction

- cin is an object of type istream defined in the header file iostream, and associated with the standard input device, i.e., the keyboard.

- int i;
- cin >> i;

- If the input from the user through the keyboard is <space> <space> 787w33, then i will contain 787.

- The input buffer will now contain w33, and the pointer will be positioned at w. this is because integer extraction bypasses white spaces, and reads input characters until it encounters a character that cannot be part of that type.

Character Extraction

- The character extractor might not work the way one might expect it to. It reads the next character in the stream after skipping white spaces.
- If you enter <space> A, the extractor would ignore the space and return ‘A’.
- A direct consequence is that char * (or string) extractors may often yield unpredictable results.

Character Extraction

- char str1[20], str2[20];
- cin >> str1;
- cin >> str2;

- If the user enters <space> james <space> gosling <return>, then str1 will contain “james”, and str2 will contain “gosling”.

- This is because extraction terminates the moment a white space is encountered.

Implicit File Opening and Closing

- In c++ , file can be opened without using additional open function.

  ofstream out("OBJ.TST");
- It means that we create an object called **out** of the ofstream class, and its constructor is invoked with the value "OBJ.TST".
- When the scope of an object is over, its destructor is automatically called.
- Likewise, when the scope of a stream object in a program gets over, the destructor of the stream class is automatically called, and the file is closed.

Extraction & Insertion of Fundamental Data Types

- File I/O using Fundamental Data Types:

- Consider a situation where you want to write some integers to a file called INT.TST

```
#include<iostream>
using namespace std;
void main( )
 {
  ofstream out("int.tst");
  out << 25 << ' ' << 4567 << ' ' << 8910
}
```

Extraction & Insertion of Fundamental Data Types


```
#include<iostream>
using namespace std;
void main( )
 {
  ifstream in("int.tst");
  int i, j, k;
  in >> i >> j >> k;
  cout << i << ' ' << j << ' ' << k;
 }
```

Extraction & Insertion of Fundamental Data Types

- Strings in File I/O
- #include <fstream.h>
- void main( )
- { ofstream out("STR.TST");
- out < "This is a test string"; }

- #include <fstream.h>
- void main( )
- {
- ifstream in("STR.TST");
- char str[30];
- in >> str;
- cout << str;
- }

File I/O Using Objects

- Consider the following code:
- #include<fstream>
- #include<string.h>
- class vehicle
- {
- public:
- int serialno;
- char model[8];
- double price; };
- void main( )
- { ofstream out("obj.tst");
- vehicle car;
- car.serialno = 22;
- strcpy(car.model, "astra");
- car.price = 600000.00;
- out << car.serialno << car.model << car.price; }

File I/O Using Objects

- The following program shows the input of objects from a file:
- #include<fstream>
- class vehicle
- {
- public:
- int serialno;
- char model[8];
- double price; };

- void main( )
- { ifstream in("obj.tst");
- vehicle car;
- in >> car.serialno;
- in >> car.model;
- in >> car.price;
- cout << '\n' << car.serialno << '\n' << car.model << '\n' << car.price; }

Binary I/O

- The I/O operations discussed so far are text or character-based. That is, all information is stored in the same format as it would be displayed on screen.

- So, ‘A’ would be written as ‘A’ on to the file. And the number -12345.678 will be written as -12345.678. This means that you can type the file and see the contents.

- Binary I/O entails that the number -12345.678 will be written as a float representation taking up to 4 bytes of storage.

Binary I/O

- When reading text files using the >> operator, certain translations will occur. For example, white space characters are omitted.

- The C++ binary I/O functions such as get( ), getline( ), and read( ) can be used when the programmer wants to read white space characters as well.

- So with binary I/O, the problem of accepting and writing character strings with white space characters will be solved.

Binary I/O – Character I/O

- The stream classes have many member functions for file I/O. Two of these are get( ) and put( ).

- The get( ) function reads a character from a file, while the put( ) function writes a character on to a file.

```
#include<fstream>
#include<string.h>
void main( )
 {
   ofstream outfile("chrfile.tst");
   char str[ ] = "this is a test";
   for (int i = 0; i < strlen(str); i++)
   outfile.put(str[i]);
   outfile.put('\0');
 }
```

Binary I/O – Character I/O

- Program to read the string back from the file:
- #include<fstream.h>
- void main(void)
- {
- const int MAX = 80;
- ifstream infile(“CHRFILE.TST”);
- char chstr[MAX];
- while (infile) // **EOF check**
- {
- infile.get(chstr[i++];
- }
- cout << chstr;
- }

Binary I/O – String I/O

- The get( ) function has the following overloaded form that can read a complete string:

get(char * str, int len, char delim = '\n');

- Fetches characters from the input stream into the array str.
- Fetching is stopped if len number of characters have been fetched, or the delimiter is encountered, whichever is earlier.
- The terminating character is not extracted.

Binary I/O – String I/O

```
#include<iostream>
void main( )
 {
  char str[20];
  cin.get (str, 20);
  cout << str;
 }
```

- This program gets a string (along with white spaces) from the standard input, and displays the string on standard output.

- Another function available for string input is:
- getline(char * str, int len, char delim = '\n')

Binary I/O – String I/O

The getline( ) function:

- is similar to the get( ) function
- extracts the terminator also

- Here is a program segment that uses the getline( ) function.

```
#include<iostream>
Void main( )
 {
  char str[20];
  cin.getline(str,20);
  cout << str;
 }
```

- In this case, str will also contain the '\n' character.

Binary I/O – Integer I/O

- Methods:
- read( ) called with two arguments
  - The address of the buffer into which the file is to be read
  - The size of the buffer

- write( ) requires similar arguments.

- Program to get a series of values from the user, and write it to a file.

Binary I/O – Integer I/O

```
#include<fstream>
void main( )
 {
  ofstream outfile("intfile.tst");
  int number = 0;
  while ( number != 999)
   {
    cout << "please input an integer";
    cin >> number;
    outfile.write((char *) &number, sizeof(number));
   }
```

Binary I/O – Integer I/O

```
#include<fstream>
void main( )
 {
  ifstream intfile("INTFILE.TST");
  int number = 0;
  intfile.read((char *)&number, sizeof(number));
  while(intfile) // EOF check
   {
    cout << number;
    intfile.read((char *)&number, sizeof(number));
   }
 }
```

Explicit File Opening and Closing

- The open( ) function: There is another way to open files. Each file stream has an open( ) member function.
- ifstream ifile; // create an unopened input stream
- ifile.open("file"); // and associate it to a file
-
- Each file stream has a close( ) function.
- ofstream ofile;
- ofile.open("obj.tst");
- ofile.close( );

Opening a File for Input and Output

- If we want to open a file in both input as well as output mode, an fstream object is used.
- fstream inherits from iostream, that in turn inherits from both istream and ostream, and hence supports read/write capability.
- The statement fstream file("IOFILE"); will not be enough for the file to be used for both input and output.
- We need to explicitly state this by using the Open Mode bits.

The Open Mode Bits

- Each stream has a series of bits, or flags associated with its operations. These bits are defined in the **ios** class.

- Some of these bits are used for formatting input and output. Some others are used for error handling. By inheritance, these are available for the **fstream** classes.

- Some of these bits are associated with the opening of files. Associated with every stream is a series of bits called the Open Mode Bits.

The Open Mode Bits

- These open mode bits represent the mode in which the file is opened. Collectively, these bits state the mode in which the file is to be opened.

- fstream file("IOFILE", ios::in | ios::out);

- When the program encounters the aforesaid statement, these two bits are set on for the stream.

- Therefore, the file is available for input as well as output.

The Open Mode Bits

Mode	Explanation
app	All data written out is appended to the stream
ate	The file pointer starts at the end of the stream, i.e., causes a seek to the end of the file. I/O operations can still occur anywhere within the file
in	The stream is opened for input.
out	The stream is opened for output
trunc	If the file exists, it is truncated, i.e., all data is erased before writing or reading

The Open Mode Bits

Mode	Explanation
nocreate	Operation fails when opening, if the file does not already exist. If the file we are opening does not already exist, the open fails.
noreplace	Operation fails when opening for output, if the file already exists, unless app or ate is set.

Disk I/O Using Member Functions


```
class Drug
 {
  private:
   unsigned int batchno;
   char drugCode[5];
   char category;
   char units[4];
   float qtyProduced;
  public:
   void getdata( )
    { cout << "\nEnter Batch Number:";
      cin >> batchno;
      cout << "\nEnter Drug Code :"
      cin >> drugCode;
      cout << "\nEnter Category :";
      cin >> category;
      cout << "\nEnter Unit of Measurement :";
```

Disk I/O Using Member Functions

- cin >> units;
- cout << "\nEnter Quantity Produced :";
- cin >> qtyProduced; }

- Void show data( )
- { cout << "\n Displaying Data";
- cout << "\n The Batch Number is " << batchno;
- cout << "\n The Drug Code is " << drugCode;
- cout << "\n The Category is " << category;
- cout << "The Unit of Measurement is " << units;
- cout << "\n The Quantity Produced is " << qtyProduced;
- } };
-

Disk I/O Using Member Functions


```
Class Drug_file
 {
  private:
    drug buffer;
  public:
     void diskout( void)
     void diskin( void) };
```

Disk I/O Using Member Functions


```
void drug_file :: diskout (void)
 {ofstream ofile("drugs.dat", ios::app);
  char loop;
  do {
     buffer.getdata( );
     ofile.write((char *)&buffer, sizeof(buffer);
     cout << "\n any more records (y/n) ?";
     cin >> loop;
   } while (loop = = 'y'); }

void Drug_file :: diskin( void)
 { ifstream ifile("DRUGS.DAT");
  ifile.read((char *)&buffer, sizeof(buffer));
  while (ifile) // EOF check
   {
    buffer.showdata( );
    ifile.read((char *)&buffer, sizeof(buffer)); } }
```

Disk I/O Using Member Functions


```
void  main( )
 {
  drug_file fileobj;
  //get data and write to the file
  fileobj.diskout( );
  //read file and display records
 fileobj.diskin( );
 }
```

- In the above example, we examined the error state of a stream object to find out whether we had reached the end of file.

```
while (infile)
 { // file processing }
```

Disk I/O Using Member Functions

- Here, infile becomes zero when EOF is reached. The ios::nocreate flag reports an error when opening, if the file does not already exist. The modified diskin( ) funciton is shown below:
- void drug_file :: diskin(void)
- {
- ifstream ifile("DRUGS.DAT", ios::nocreate);
- if (ifile)
- {
- ifile.read((char *)&buffer, sizeof(buffer);
- while(ifile)
- { buffer.showdata( );
- ifile.read((char *)&buffer, sizeof(buffer)) }
- }
- else
- { cout << "\n file does not exist"; } }
-

The Stream Status Bits

- Every stream has a state that indicates if an error has occurred, and what the error is.
- The C++ I/O system maintains status information about the outcome of each I/O operation.
- The state is indicated by the setting (0 or 1) of a set of bits, and these bits are defined by the io_state variable declared in the ios class.

The Stream Status Bits

Name	Description
eofbit	Set when an input stream is at its end. Indicates that no more characters are available for extraction.
failbit	Set when the last insertion or extraction has failed. This is a recoverable error. The stream is still in a usable state.
badbit	Set when an illegal insertion or extraction is attempted. The stream is not suitable for further use. Usually set when opening a non-existent file, or seeking when a past EOF has occurred.
hardfail	Set when a hardware failure has occurred. The error that sets hardfail is usually unrecoverable.

Member Functions for Accessing Stream Status Bits

Name	Description
int good( )	returns a non-zero value if the stream is ok, zero otherwise.
int bad( )	returns a non-zero value if the badbit or hardfail bits are set, else returns 0.
int eof( )	returns a on-zero value if the eofbit is set, else returns 0.
int fail( )	returns a non-zero value if the failbit, badbit, or hardfail bit
int rdstate( )	returns the current value of the iostate variable
void clear(int ef=0)	set the error flags equal to ef. by default, ef equals 0, which resets all the error bits

Random Access

- The C++ I/O system manages two integer values associated with a file. One is the **get pointer**, which specifies where in the file the next input or read operation will occur.

- The other is the **put pointer**, which specifies where in the file the next output or write operation will occur.

- In other words, these are the **current positions for read and write** respectively.

Random Access

- The seekg( ) and the tellg( ) functions allow you to set and examine the get pointer.
- The seekp( ) and the tellp( ) functions allow you to set and examine the put pointer.
- One can, therefore, access the file in a random access mode using these functions.
- All objects of the iostream classes can be manipulated using either the seekg( ), or the seekp( ) member functions.

Random Access

- The seekg( ) member function takes two arguments:
- file.seekg(10, ios::beg); means position the get pointer 10 bytes relative to the beginning of the file

- The first argument is a long integer, specifying the number of byte positions (also called offset). The second argument is the reference point.

Random Access

- There are three reference points defined in the ios class:

Name	Description
ios::beg	The beginning of the file
ios::cur	The current position of the file pointer
ios::end	The end of file

Random Access

- If supplied with only one argument, ios::beg is assumed by default. For example, in the statement: file.seekg(16), ios::beg will be the reference point by default.

- The tellg( ) member function, on the other hand, does not have any arguments. It returns the current byte position of the get pointer relative to the beginning of the file.

- long position = file.tellg( );
- Would result in the variable **position** taking the value of the current position of the get pointer.

Random Access Scenario

Example : to find out the how many Drug records are there in the file.

```
void main( )
 {
  drug drugobj;
  fstream ifile("drugs.dat", ios::in);
  ifile.seekg(0, ios::end);
  long endpos = ifile.tellg( );
  cout << "\n the size of the file is " << endpos;
  cout << "\n the size of a Drug record is " << sizeof(Drug);
  int n = endpos/ sizeof(drug)
  cout << "\n" << n << "drug records are in the the file";
}
```

Random Access – Querying a File

- Example: to retrieve a particular record from the file DRUGS.DAT.

```
void main( )
 {
  drug drugobj;
  fstream ifile("drugs.dat", ios::in);
  ifile.seekg(0, ios::end);
  long endpos = ifile.tellg( );
```

Random Access – Querying a File

- cout << "\n the size of the file is " << endpos;
- cout << "\n the size of a Drug record is " << sizeof(Drug);
- int n = endpos/ sizeof(drug)
- cout << "\n" << n << "drug records are in the the file";
- cout << "\n drug record query";
- cout << "\n which record do you want to query";
- cout << "\n please enter the record number";
- int num;
- cin >> num;
- int seekpos = (num – 1) * sizeof(drug);
- ifile.seekg(seekpos);
- ifile.read((char *)&drugobj, sizeof(drug);
- drugobj.showdata( ); }

Formatted I/O - ios format functions

- By default console I/O is unformatted, but using following ios functions, we can format them:

setf()	Sets format flags that can control the form of output display.
unsetf()	Resets format flags.
width()	Sets the required field size for displaying output.
fill()	Sets the padding used to fill unused portions of a field.
precision()	Sets the number of digits to be displayed after the decimal point of a float value.

Format flags

Format required	Flag (arg1)	Bit-field (arg2)
Left-justified output	ios::left	ios::adjustfield
Right-justified output	ios::right	ios::adjustfield
Scientific notation	ios::scientific	ios::floatfield
Fixed point notation	ios::fixed	ios::floatfield
Decimal base	ios::dec	ios::basefield
Octal base	ios::oct	ios::basefield
Hexadecimal base	ios::hex	ios::basefield

Manipulators

- setw (int w) Set the field width to w.
- setprecision(int d) Set the floating point precision to d.
- setiosflags(long f) Set the format flag f.
- resetiosflags(long f) Clear the flag specified.
- endl Insert newline and flush stream.

Summary

At the end of this Session you learnt to:

- Define Stream and Stream classes
- Describe Stream extraction –input/output
- Describe Implicit file opening and closing
- Describe Explicit file opening and closing
- Use File open mode bits
- Use Stream Status Bits
- Use File Randam Access Functions
- Use Formated I/O Functions
- Describe Manipulators

Templates

Objectives

At the end of this session, you will be able to:

- Describe function templates and their implementation
- Describe class templates and their implementation
- Get an overview to containers and their implementation

Template Functions

- 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 upon is passed to it as a parameter.
- Through a generic function, a single general procedure can be applied to a wide range of data.

Template Functions

- Once you have defined a generic function, the compiler will automatically generate the correct code for the type of data that is actually used when you execute the function.

- In essence, when you create a generic function, you are creating a function that can **automatically overload itself**.

- A generic function is created using the keyword **template**. The normal meaning of the word “template” accurately reflects its use in C++.

Template Functions

- It is used to create a template that describes what a function will do, leaving it to the compiler to fill in the details as needed.
- When the compiler creates a specific version of this function, it is said to have created a specialization.
- This is also called a generated function. The act of generating a function is referred to as instantiating it.
- This is also called as **template instantiation**.

Template Functions

- By using templates, you can reduce this duplication to a single function template:

- template <class T> T min(T a, T b)
- return (a < b) ? a : b;

- Templates can significantly reduce source code size, and increase code flexibility without reducing type safety.

Template Functions - Implementation

```
#include <iostream.h>
template <class T>
void swap(T &a, T &b)
{
    T temp=a;
    a=b;
    b=temp;
}
```

```
void main()
{
        int x=10,y=20;
        swap(x,y);
        cout<<x<<"   "<<y<<endl;
        char *s1="Hello",*s2="Hi";
        swap(s1,s2);
        cout<<s1<<"  "<<s2<<endl;

}
```

Multiple Generic Types

```
template<class T, class S, class Z>
void fun(T a, S b, Z c)
{
    cout<<a<<endl<<b<<endl<<c;
}
void main()
{
    int i=10;
    float j=3.14;
    char ch='A';
    fun (i, j, ch);
}
```

Explicitly Overloading a Generic Function

- Even though a generic function overloads itself as needed, you can explicitly overload one, too. This is formally called **explicit specialization**.

- If you overload a generic function, that overloaded function overrides (or hides) the generic function relative to that specific version.

- Consider the following example:

Explicitly Overloading a Generic Function


```
// Overriding a template function
#include <iostream>
using namespace std;
template <class X> void swapargs (X &a, X &b)
  {X temp;
    temp = a;
    a = b;
    b = temp;
    cout << "Inside template swapargs \n"; }

//This overrides the generic version of swapargs ( ) for ints.
void swapargs( int &a, int &b)
  {int temp;
    temp = a;
    a = b;
    b = temp;
    cout << "Inside swapargs int specialization\n"; }
```

Explicitly Overloading a Generic Function


```
Int main( )
 {int i = 10, j = 20;
  double x = 10.1, y = 23.3;
  char a = 'x', b = 'z';
  cout << "Original i, j:  " << i << " " << j << '\n';
  cout << "Original x, y:  " << x << " " << y << '\n';
  cout << "Original a, b:  " << a << " " << b << '\n';
  swapargs( i, j); // calls explicitly overloaded swapargs
  swapargs( x, y); // calls generic swapargs
  swapargs( a, b); // calls generic swapargs
  cout << "Swapped i, j:  " << i << " " << j << '\n';
  cout << "Swapped x, y:  " << x << " " << y << '\n';
  cout << "Swapped a, b:  " << a << " " << b << '\n';
  return 0; }
```

Overloading a Template Function

- **template specification itself can be overloaded**

- To do so, simply create another version of the template that differs from any others in its parameter list. For example,

```
// Overload a function template declaration
#include <iostream>
using namespace std;
//First version of f( ) template
template <class X> void f( X a)
 { cout << “Inside f( X a)\n”; }
```

Overloading a Template Function


```
//Second version of f(  ) template
Template <class X, class Y> void f( X a, Y b)
  {
    cout << Inside f( X a, Y b) \n";
  }

Int main( )
  {
    f(10); // calls f( X)
    f( 10, 20); // calls f( X, Y)
    return 0;
  }
```

Templates Vs Macros

- Macros in C were also independent of data type but this feature of template is more bug free.

- In macros there is no type checking. The type of return value isn't specified, therefore the compiler cannot check whether we are assigning it to the correct return type or not.

Template Classes

- Like generic functions, you can also define a generic class.
- When you define a generic class, you create a class that defines all the algorithms used by that class.
- However, the actual type of the data being manipulated will be specified as a parameter when objects of that class are created.
- Generic classes are useful when a class uses logic that can be generalized.

Template Classes

- The compiler will automatically generate the correct type of object, based upon the type you specify when the object is created.

- The general form of a generic class declaration is shown here:
- template <class Ttype> class class-name
- {….. };

- Once you have created a generic class, you create a specific instance of that class using the following general form:
- class-name <type> ob;

Template Classes

- Member functions of a generic class are themselves automatically generic. You need not use the keyword **template** to explicitly specify them as such.

- In the following example, a generic stack class is used to store objects of any type.
- #include <iostream>
- using namespace std;
- const int size = 10;
- // create a generic stack class
- template <class StackType> class stack
- {private:
- StackType stck[size];
- int tos; // index to top-of-stack

Template Classes

```
public:
    stack( )
     { tos = 0; // initialize stack }
void push( StackType ob); // push object on stack
StackType pop(  ); // pop object from stack };

template <class StackType> void stack< StackType>::push( StackType ob)
 {if (tos = =  size)
    {cout << "Stack is full\n";
     return; }
     stck[ tos ] = ob;
     tos + +; }

template <class StackType> StackType stack<StackType>::pop( )
 {if (tos = = 0)
    {cout << "Stack is empty\n";
     return 0; // return null on empty stack }
     tos - -;  return stck[ tos ]; }
```

Template Classes


```
int main( )
 {
   //demonstrating character stacks
   stack<char> s1, s2; // create two character stacks
   int i;
   s1.push( 'a');
   s2.push( 'x');
   s1.push( 'b');
   s2.push( 'y');
   s1.push( 'c');
   s2.push( 'z');
   for (int i = 0; i < 3; i+ +) cout << "pop s1:  " << s1.pop( ) << "\n";
   for (int i = 0; i < 3; i+ +) cout << "pop s2:  " << s2.pop( ) << "\n";
```

Template Classes


```
// demonstrate double stacks
stack<double> ds1, ds2; // create two double stacks
ds1.push(1.1);
ds2.push(2.2);
ds1.push(3.3);
ds2.push(4.4);
ds1.push(5.5);
ds2.push(6.6);
for (int i = 0; i < 3; i+ +) cout << “pop ds1:  “ << ds1.pop( ) << “\n”;
for (int i = 0; i < 3; i+ +) cout << “pop ds2:  “ << ds2.pop( ) << “\n”;
return 0;
}
```

Template Class

- Generic class independent of data type.
- Can be instantiated using type-specific versions.
- Can create an entire range of related overloaded classes called template classes.
- Usually used for data storage (container) classes like stacks etc.
- Class templates cannot be nested.
- Template classes can be inherited

Inheritence and Template class

- Template class can be inherit
- Three ways of template class inheritece
  - Template class(base) to template class(Derived)

  - Template class (base)to normal class(Derived)

  - normal class(Base) to Template class(Derived)

Inheritence and Template class

Example :Template class to template class

```
template<class type>
class base
{
type x;
public:
void getx()
{ cin>>x; }

void putx()
{cout<<x; }
};
```

```
template<class type>
class Der:public base<type>
{
type y;
public:
void gety()
{cin>>y;}

void puty()
{cout<<y; }
};
```

```
main()
{
Der<int> obj;
obj.getx();
obj.putx();
obj.gety();
obj.puty();
}
```

Inheritence and Template class

Example :Template class to normal class

```
template<class type>
class base
{
type x;
public:
void getx()
{ cin>>x; }

void putx()
{cout<<x; }
};
```

```
class Der:public base<int>
{
int y;
public:
void gety()
{cin>>y;}

void puty()
{cout<<y; }
};
```

```
main()
{
Der obj;
obj.getx();
obj.putx();
obj.gety();
obj.puty();
}
```

Inheritence and Template class

Example :Normal class to Template class

```
class base
{
int x;
public:
void getx()
{ cin>>x; }

void putx()
{cout<<x; }
};
```

```
template<class type>
class Der:public base
{
type y;
public:
void gety()
{cin>>y;}

void puty()
{cout<<y; }
};
```

```
main()
{
Der<int> obj;
obj.getx();
obj.putx();
obj.gety();
obj.puty();
}
```

Standard Template Library (STL)

- Constructed from template classes. The algorithms and data structures can be applied to any type of data.

- Based on three fundamental items:

  -*Containers*

  template data structures

  - *Algorithms*

  data manipulation, searching, sorting, etc

  - *Iterators*

  like pointers, access elements of containers

Containers

- There are three types of containers
  - Sequence containers
    - Linear data structures (vectors, linked lists)
  - Associative containers
    - Non-linear, can find elements quickly
    - Key/value pairs
  - Container adapters

- All Containers have some common functions

Containers cont..

- Sequence containers
  - Vector , deque , list
- Associative containers
  - Set ,multiset ,map ,multimap
- Container adapters
  - Stack ,queue ,priority_queue

Common STL Member Functions

- Member functions for all containers
  - Default constructor, copy constructor, destructor
  - empty
  - max_size, size
  - = < <= > >= == !=
  - swap
- Functions for first-class containers
  - begin, end
  - erase, clear

Iterators

- Iterators are similar to pointers
  - Point to first element in a container
  - Iterator operators same for all containers
    - **`*`** dereferences
    - **`++`** points to next element
    - **`begin()`** returns iterator to first element
    - **`end()`** returns iterator to last element

Types of Iterators

- Input
  - Retrieval of elements from container , can only move forward
- Output
  - Used for storing elements to container, forward
- Forward
  - Combines input and output
- Bidirectional
  - Like forward, but can move backwards as well
- Random access
  - Like bidirectional, but can also jump to any element

Iterator Operations

- For all iterators
  - `++p, p++`
- Input iterators
  - `*p`
  - `p == p1, p != p1`
- Output iterators
  - `*p`
  - `*p=100;`
- Forward iterators
  - Have functionality of input and output iterators

Iterator Operations cont..

- Bidirectional iterator
  - `--p, p--`
- Random access iterator
  - `p + i, p += i`
  - `p - i, p -= i`
  - `p[i]`
  - `p < p1, p <= p1`
  - `p > p1, p >= p1`

Basic Steps to Use STL

- Decide the type of container that you wish to use.
- Use its member functions to add, access, modify, or delete elements.
- Members within a container can be accessed with an iterator.
- The container can be manipulated using one or more algorithms.

Example : Vector container

- Vector
  - Have to include the following header file **<vector>**
  - Data structure with contiguous memory locations
    - Access elements with **[]**
  - Use when data must be sorted and easily accessible
- When memory exhausted
  - Allocates larger, contiguous area of memory
- Has random access iterators

Vector container operations

- vector class member functions
  - push_back(value)
    - Add element to end (found in all sequence containers).
  - size()
    - Current size of vector
  - capacity()
    - How much vector can hold before reallocating memory
  - insert(*iterator*, *value*)
    - Inserts *value* before location of *iterator*

Vector container operations cont..

- erase(iterator)
  - Remove element from container
- erase(iter1, iter2)
  - Will give a range to delete between iter1 to iter2
- clear()
  - Will erase the entire vector.
- begin()
  - Will return the iterator to the beginning.
- end()
  - Will return the iterator to the end of vector.

Summary

At the end of this session, you learnt to:

- Describe function templates and their implementation
- Describe class templates and their implementation
- Get an overview to containers and their implementation

Exception Handling

Objectives

- In this session you will learn to:
- Define Exception
- Describe Error handling in C
- Describe Implementation of exception handling using try catch, throw
- Handling uncaught exception
- Restricting Exception and re-throwing exception
- Handling Derived class Exception
- Use of Standard Library Exception Hierarchy

What is an Exception?

- An exception is an error that occurs during runtime.
- Such runtime errors can cause the applications to behave unpredictably, or can cause the application to crash.
- Exception handling allows you to manage runtime errors in an orderly fashion.
- Using exception handling, your program can automatically invoke an error-handling routine when an error occurs.

Common Exceptions

- Falling short of memory
- Inability to open files
- Exceeding bounds of an array

Exception Handling

- Provides a structured means by which your program can handle abnormal events.
- Automatically invokes an error handling routine when an error occurs.
- Can handle only synchronous exceptions like “overflow”, “out of range” etc.

Error handling in C

- Checking function return value. Using the setjmp and longjmp mechanism.

- Coupling of error handling code to the function corrupts the normal logic.

- Too many conditional checks reduce readability.

- setjmp/longjmp uses a data structure jmp_buf, which is system-dependent.

- Does not handle the destruction of objects

Exception Handling in C++

- C++ uses a special language-supported exception handling feature to signal such anomalous events. The exception mechanism uses three new keywords:
  - try
  - catch
  - throw

- The function in which error is expected is kept inside the **try** block. Exception is **thrown** from the **try** block.

- This is handled by the **catch** block immediately following the try block.

Exception Handling - Mechanics

- When an exception is thrown, it is caught by the corresponding **catch** statement, which processes the exception.
- There can be more than one **catch** statement associated with a **try.**
- Which **catch** statement is used is determined by the type of the exception.

The **throw** Statement

- The general format of the throw statement is as follows:
  throw *exception*

- throw generates the exception specified by *exception*.

- If the exception is to be caught, then **throw** must be executed from either within **try** block itself, or from any function called from within the **try** block.

Uncaught Exceptions

- If an exception is thrown for which there is no applicable **catch** statement, an abnormal program termination occurs.
- Throwing an unhandled exception causes the standard library function **terminate( )** to be invoked.
- By default, **terminate( )** calls **abort( )** to stop your program.

Restricting Exceptions

- The **throw** clause in a function definition specifies the type of exceptions the function is expected to throw.

- Using the throw clause, you can restrict the type of exceptions that a function can throw outside of itself by specifying a comma-separated list of exception types within the throw clause in the function definition.

- Throwing any other exception will cause abnormal program termination.

Restricting Exceptions

- Special function **unexpected( )** is called when you throw something other than what appears in the exception specification.
- By default, **unexpected( )** causes **abort( )** to be called, which causes abnormal program termination.
- Presence of **throw( )** with empty parenthesis indicates that function is not expected to throw any exception.

Catching Any Exception

- If your function has no exception specification, any type of exception can be thrown. One solution to this problem is to create a handler that catches any type of exception.

- This is done by using the ellipses in the argument list of catch().
  - catch(...) {
  - cout << "an exception was thrown" << endl }

Re-throwing an exception

- Sometimes, you'll want to rethrow the exception that you just caught, particularly when you use the ellipses to catch any exception, because there's no information available about the exception.

- This is accomplished by saying **throw** with no argument:

```
catch(...)
{
  cout << "an exception was thrown" << endl;
   throw;
 }
```

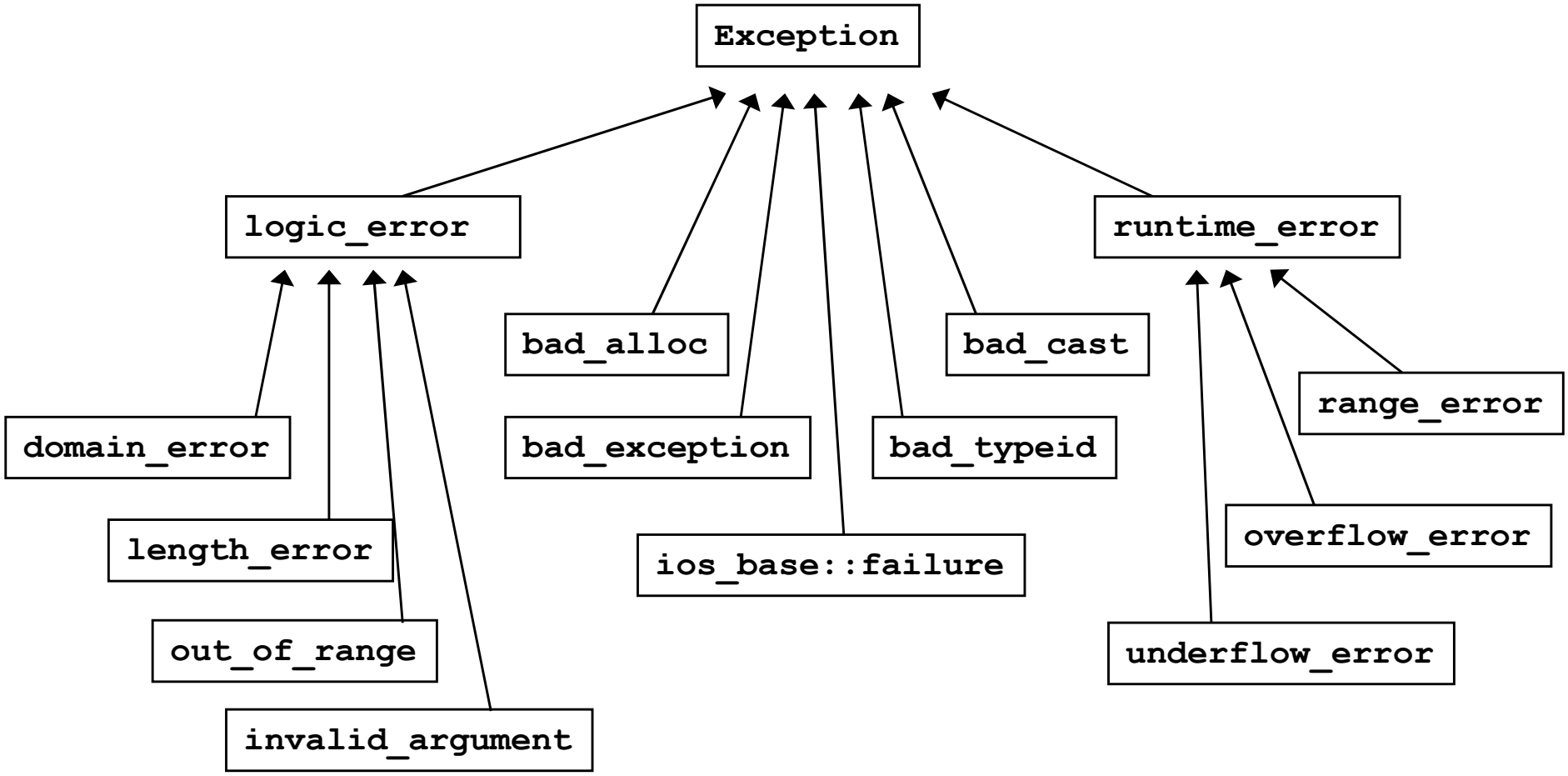
- the **throw** causes the exception to go to the exception handlers in the next-higher context.

Handling Derived Class Exceptions

Since a catch's argument type applies to objects of base, as well as classes derived from that base, the order of catch handlers is important

```
catch( CBaseException& ex )
{
        // handle a base type of exception
}
catch( CDerivedException& ex )
{
        // this handler would never "run" because its
       // base would catch everything it would…
}
```

Standard Library Exception Hierarchy



Standard Library Exceptions

- Declares many std exception objects, e.g. :

  bad_alloc, thrown when **new** fails

  bad_cast, thrown when **dynamic_cast** fails

  bad_typeid, thrown if **dynamic_cast** is applied to a **NULL** pointer

  bad_exception, thrown when an unexpected (non-specified) exception is thrown

- also in **std** namespace
  - **throw std::range_error;**

Programming With Exceptions

Avoid exceptions :

- for asynchronous events.
- for ordinary error conditions.
- for flow-of-control.

Use exceptions to:

- fix the problem and resume the program.
- do whatever you want in the current context, and throw a different exception to a higher context.
- terminate the program.

Summary

In this session you learnt to:

- Define Exception
- Describe Error handling in C
- Describe Implementation of exception handling using try catch, throw
- Handling uncaught exception
- Restricting Exception and re-throwing exception
- Handling Derived class Exception
- Use of Standard Library Exception Hierarchy

Run Time Type Identification (RTTI)

Objectives

In this session you will learn to:

- Define RTTI
- Describe Application of RTTI
- Use dynamic_cast operator

RTTI

- In polymorphic languages such as C++, there can be situations in which the type of an object is unknown at compile time, i.e., when the program is written.
- It is not always possible to know in advance what type of object will be pointed to by a base class pointer at any given point in time.
- This determination must be made at runtime, using runtime type identification.

What is RTTI?

- Mechanism that allows the type of an object to be determined during program execution.

- Used only with polymorphic classes (i.e. those which have a virtual function in the base class) to find the exact type of an object when you have a pointer or reference to the base type.

- In the absence of polymorphism, the static type information is used

RTTI

- RTTI is possible only with virtual functions and base class pointers. While using virtual functions, you must have access to the base class source code.

- If the base class is a part of a library and does not contain the virtual function that you need, you are stuck up.

- At such times you should use RTTI. You can derive a new class from the base class and add your extra member function to it.

- Then, you can detect your particular type (using RTTI) and call the member function.

C++ RTTI support

C++ provides two ways to obtain the information about the object class at runtime. These are:

- Using **typeid**() operator and **type_info** class.
- Using **dynamic_cast** operator

The type_info class

It defines the following public members:

- bool operator==(const type_info *&ob*);
- bool operator!=(const type_info *&ob*);
- bool before(const type_info *&ob*);
- const char *name();

A Simple Application of RTTI


```
#include<iostream>
using namespace std;
class Figure
 {public:
   virtual void draw( ) = 0; };

class Rectangle : public Figure
  {public:
     void draw( )
      { cout << "Rectangle's draw\n"; } } ;

class Circle : public Figure
 {public:
   void draw( )
    { cout << "Circle's draw\n"; } };
```

A Simple Application of RTTI


```
Class Triangle : Public Figure
 {
  public:
   void draw( )
   { cout “Triangle’s draw\n”; } };
Figure* factory( ) // a factory for objects derived from Figure
  {switch( rand( ) % 3)
    {case 0 : return new Rectangle;
     case 1 : return new Circle;
     case 2 : return new Triangle; }}
int main( )
 {
  Figure *ptr; // pointer to base class
  int i;
```

A Simple Application of RTTI


```
int r, c, t;
for (i = 0; i < 10, i + +) // generate and count objects
 {ptr = factory( ); // generate an object
   cout << "Object is " << typeid(*ptr).name( ) << endl;
   if (typeid(*ptr) == typeid(Rectangle) r++;
   if (typeid(*ptr) == typeid(Circle) c++;
   if (typeid(*ptr) == typeid(Triangle) t++; }
   cout << "figures generated:\n";
   cout << "rectangles: " << r << endl;
   cout << "circles: " << c << endl;
   cout << "triangles: " << t << endl;
   return 0;
}
```

The dynamic_cast Operator

- The purpose of dynamic_cast is to perform casts on polymorphic types.
- For e.g., given two polymorphic classes B and D, with D derived from B, a dynamic cast can always cast a D* pointer into a B* pointer.
- This is because a base pointer can always point to a derived object.

The dynamic_cast Operator

- But a dynamic_cast can cast a B* pointer to a D* pointer only if the object being pointed to actually is a D object.
- If the cast fails, then dynamic_cast evaluates to NULL if the cast involves pointers.
- If a dynamic_cast on reference types fails, a **bad_cast** exception is thrown.

Dynamic_cast -*example*

```
Base *bp, bobj;
Derived *dp, dobj;

 bp=&dobj;   //base pointer points to derived object
 dp= dynamic_cast<Derived *> (bp);
 if (dp) cout<<“Cast OK”;

 bp=&bobj;
 dp= dynamic_cast<Derived *> (bp);
 if (!dp) cout<<“Cast fails”;
```

Summary

In this session you learnt to:

- Define RTTI
- Describe Application of RTTI
- Use dynamic_cast operator

Namespaces

Objectives

In this session you will learn to:

- Describe namespace with an example
- Use ‘using’ keyword with namespace
- Unnamed namespace
- New C++ header

Namespaces – An Introduction

- The purpose of namespaces is to localize the names of identifiers to avoid name collisions.
- The C++ environment has seen an explosion of variable, function, and class names.
- Prior to the invention of namespaces, all of these names competed for slots in the global namespace, and many conflicts arose.

Namespaces – An Introduction

- The creation of the **namespace** keyword was a response to these problems.

- Since it **localizes the visibility of names** declared within it, a namespace allows the same name to be used in different contexts without conflicts arising.

- The most notable beneficiary of **namespace** is the C++ standard library. Prior to **namespace**, the entire C++ was defined within the global namespace (which was of course the only namespace).

Namespace Basics

- The namespace keyword allows you to partition the global namespace by creating a declarative region.

- In essence, a namespace defines a **scope**. The general form of namespace is as follows:

```
namespace name
  {
    // declarations
  }
```

- Anything defined within a namespace is within the scope of that namespace.

Namespace – An Example

```
namespace CounterNameSpace
 {int upperbound;
  int lowerbound;
  class counter
    {int count;
     public:
      counter (int n)
       {if (n <= upperbound)
          count = n;
        else
          count = upperbound;  }
        void reset ( int n)
         {if (n <= upperbound)
          count = n; }
```

Namespace – An Example


```
int run( )
 {if (count > lowerbound)
    return count--;
   else
    return lowerbound; } }; // end of class
} // end of namespace
```

- Code outside CounterNameSpace:
- CounterNameSpace::upperbound = 10; /* assign a value to a variable defined within a namespace */
- CounterNameSpace:: counter ob; /* declare an object of a class defined within a namespace */

Namespace – An Example

- However, since namespace defines a scope, you need to use the scope resolution operator (**::**) to refer to declarations declared within a namespace from outside that namespace.

- *In general, to access a member of a namespace from outside its namespace, precede the member's name with the name of the namespace followed by the scope resolution operator.*

Namespace – Additional Points

- Syntax is similar to that of a class.
- Definition can only appear at the global scope.
- Declarations that fall outside all namespaces are still members of global namespace.
- Alternative names can be given called namespace-alias.
- Members of a namespace may be defined within or outside the namespace.
- Definition can be nested within another namespace definition.

Using a Namespace

A name within a namespace can be referred in two ways:

- Using the scope resolution operator.
- Better solution comes in the form of *using* keyword.

- The *using* keyword declares all the names in the namespace to be in the current scope, and can be used without qualification.

- Using one namespace does not override another. When you bring a namespace into view, it simply adds its names to whatever other namespaces are currently in effect.

Unnamed Namespaces

- Allows you to create identifiers unique within a file.
- Unnamed namespaces allow you to establish unique identifiers that are known only within the scope of a single file.
- Within the file that contains the unnamed namespace, the members of the namespace may be used directly, without qualification.
- But outside the file, the identifiers are unknown.

The New C++ header

- Do not specify filenames.
- Do not have a .**h** extension
- Solely consist of header name contained within angle brackets. Eg: <iostream>
- included using the #include statement.
- Contents of the header contained in the **std** namespace.

Summary

In this session , you learnt to:

- Describe namespace with an example
- Use ‘using’ keyword with namespace
- Define Unnamed namespace
- Use New C++ header