OOPs & C++

Student Reference Book

Contents

Procedural Programming

In this lecture you will understand:

* Disadvantages of Procedural Programming
* The need for Object-Oriented Programming
* How structure in C++ differ from that in C

Agenda

In this course our aim is to learn Object-Oriented language called C++. We shall learn the various concepts of Object-Oriented language. In order to learn C++, one should be familiar with C programming. By saying 'familiar with C programming' it is expected that one should have thorough knowledge of important elements of C language like arrays, strings, pointers and structures. Here we would not be dealing with all these features. (You can refer Appendix at the end of this book to brush up some of the basics of C programming). While starting with C++ we would like to have a gradual transition from the C language. Hence we would start with the discussion on structures.

Structures

To begin with let us discuss how structures are created and used in a C program. Consider following example:

```
struct emp
{
    char n[ 20 ] ; int a ; float s ;
} ;
```

Here, **emp** is the structure name and **n**, **a** and **s** are the structure elements (members). A structure is a user-defined data type, which is used whenever we wish to deal with a collection of dissimilar data types or whenever we need to create our own type having dissimilar elements. A structure is a blueprint or a design or a template by which we just tell the compiler what members the structure variable would have when it would be created. A structure declaration never occupies memory, rather memory gets allocated for the variables of the structures. For example,

```
struct emp e1 = { "Anil", 23, 5000 } ;
struct emp e2 = { "Amol", 24, 6000 } ;
```

Here, **e1** and **e2** are two variables of type **emp**, each of which holds different set of data. A variable of a structure is also called an instance of a structure. Structure members are always stored in contiguous memory locations and they can be accessed using the structure operators '.' and '->' as shown below:

```
struct emp *p ;
p = &e2 ;
printf ( "%s %d %f", e1.n, e1.a, e1.s ) ;
printf ( "%s %d %f", p -> n, p -> a, p -> s ) ;
```

To access structure members using a variable of that structure, '.' operator is used. However, to access structure members using a pointer to the structure variable, '->' operator is used. The value of one structure variable can be assigned to another using the assignment operator.

To understand the basics of object-oriented programming and why it came into being, we need to discuss the problems with the procedural approach towards programming. To make a comparison we would take an example of how stack data structure can be implemented with procedural and object-oriented programming approaches.

Stack

Earliest programs used to be merely a collection of data and statements that processed this data. In this approach, if one job to be performed several times we were simply required to rewrite the same code those many times at the places where we wanted to get it repeated. This obviously was not the best way of going about it. So there was a shift to another style of programming called Procedure-oriented programming whose cornerstone was functional abstraction.

Let us consider the working of a stack. In stack, addition of new element or deletion of an existing element always takes place at the same end. This end is known as **top** of the stack. This is similar to how plates are organized in a cafeteria where every new plate is added to the top. Similarly, every new plate is taken off from the top. Adding a new item to the stack is known as **push** and removing the top-most element is known as **pop**.

Let us now see how stack can be implemented with procedural programming approach.

Procedural Programming

Let us develop a program to maintain stack of integers. Had this problem been solved using old style of programming, to pop 2 to 3 elements from a stack same set of statements would have been required causing repetition of code. This repetition of code would have led to long programs that were difficult to comprehend. Hence people abandoned this style of programming and separated program into two entities called **data** and **functions**.

Functions/subroutines/procedures were introduced in procedural languages (like C, Pascal, etc.) to make the programs more comprehensive to their human creators. A program could now be divided into functions, with each function having a clearly defined purpose and a clearly defined interface to the other functions in the program. It used control instructions like sequence, decision, repetition or case. Local variables came into existence with procedural programming.

A local variable is said to born when it is declared in a function, lives its life till the control is in the function and dies the moment control goes out of the function in which it is declared. A new concept came into existence with procedural programming—**Abstraction**. Abstraction permitted the programmer to look at something without being concerned with its internal details. In a procedural programming it is enough to know which task is performed by function. It does not matter to the programmer how that task is performed so long as the function does it reliably. This is called **functional abstraction** and is the corner stone of procedural programming. However, even this approach has many limitations.

The first and foremost is that no importance is given to data. The whole emphasis is on doing things that is on functions/routines/procedures. Functions do things just as single program statements do. What they do may be more complex, but the emphasis is still on doing things. Data is given step-motherly treatment even though it is the reason for program's existence. For example, in a payroll processing application the important part is not only the functions, which display the data and checks for correct input, but also the payroll data, which is important.

Another limitation of procedural programming is that its primary components—functions and data structures—don't model the real world very well. For example, suppose you are writing a program to create the elements of a GUI such as menus, windows, etc., there are no obvious program elements to which a menu or a window would correspond.

Thirdly, procedural programming languages permit us to create structures and functions that can be used at several places in the program, leading to reuse. However, the reuse is limited only this. Object-oriented programming takes this reuse a step further through concept called Inheritance and Templates. We would be covering these two topics later.

Since the procedural approach is a poor representation of the real world we need to first understand the real-world model well.

Real-World Model

If you look around you would find that the world is full of objects. Thus a car is an object. A window that you get on the screen is also an object. Each object may contain several smaller objects as listed in the slide. Let's now see what is common amongst all objects.

Object Contents

Each object, say, a car contains several attributes like oil temperature, oil viscosity, revolution per minute etc. and functions that would access/modify these attributes. Similarly, a toolbar would have several attributes that can be accessed/modified using different functions present in it. Thus any object in real-world would contain attributes (data) and functions that would access/modify this data.

More Objects

This slide shows a few more examples of objects. The **stack** object would contain data (array and top) and functions like **push( )** and **pop( )** for addition and removal of elements from the stack respectively.

Similarly, there can be a **DateTime** object having data members date, month, year, hours, minutes and seconds. The functions like **getdate( )**, **gettime( )**, etc. can then be used to get information about the date, current time, etc.

Note that in this type of program design importance is given to both a function that works on data and the data for which the functions are written. Without data, functions are meaningless and without functions, data has no meaning.

If we look at the procedural solution for such a design we would be able to appreciate the limitation of procedural programming.

Object Oriented Solution

In this lecture you will understand:

* Basics of Classes and Objects
* Access modifiers, data members and member functions

Procedural Solution

Let us see a procedural solution for maintaining a stack. Consider the program given in the slide, that maintains a stack (under heading 'stack.c').

Here we have kept the stack data and functions that operate on this data separate from the program that uses the stack. In the program we have directly accessed the array **a** and data member **top**. If this can be done then what is the use of functions. If the data can be processed without calling a function why write a function at all. This is a limitation of a procedure-oriented programming, as data is given less importance. Stress is given more on procedures that work on data.

As against this, let us see the object-oriented solution.

OO Solution

Consider the program given in the slide, that gives Object-Oriented solution.

Here, the data and functions are enclosed within the structure. Moreover, some access modifiers are given indicating how the end-user will have access to them. The access modifier **private** protects data/functions from outside world. They can be accessed within the structure but not outside the structure. The access modifier **public** makes available both data/functions to the outside world. This is what is the fundamental change in OOP—a program is designed around the data being operated upon rather than upon the operations themselves.

Some interesting facts…

(a) Programming languages like C++, Java, SmallTalk, etc. are all object-oriented programming languages.

(b) C++ was developed by Bjarne Stroustrup in 1980 at AT & T Bell Labs. His picture (bottom) is shown in the slide along with that of Dennis Ritchie (top).

Difference

C++ extends the reach of structures by allowing the inclusion of functions within structures. The functions defined within a structure have a special relationship with the structure data members. Placing data and functions (that work upon the data) together into a single entity is the central idea in object-oriented programming and is known as **Encapsulation**.

There is another entity in C++ called **class** that too can hold data and functions. There is only one difference between a structure and a class—structure members are by default **public**, whereas, class members are by default **private**. Hence at least in principle they can be used interchangeably. But most C++ programmers use structures to exclusively hold dissimilar data and classes to hold both data and functions.

Classes are generic, whereas objects are specific. Thus stack can be a class whereas **s1** and **s2** can be objects of that class. This means that each object can hold specific values in it that may be different than the values stored in another object.

Classes In C++

To begin with let us see a program that demonstrates the syntax and general features of classes in C++. Consider program given in the slide.

Here the class **a** contains an **int** and a **float** as two data items. An object is an instance of a class, and the process of creating an object is called **instantiation**. In our program we have defined two objects **z1** and **z2** which are instances of the class **a**. The objects **z1** and **z2** are often known as instance variables. Remember that the specification of the class does not create any objects. It only describes

how the objects will look like when they are created. It is the definition that actually creates objects, which can then be used by the program. Thus, defining an object is similar to defining a variable of any data type—space is set-aside for it in memory. Note that it is not necessary to use the keyword **class** while defining an object.

In simplest terms a class is like a data type, whereas, an object is like a variable of that data type. Like a structure, the body of a class is delimited by braces and terminated by a semicolon.

Like structure members, class members can also be accessed using the'.' (dot) or an '->' operator.

Note that we have used **cout** (console output) to display the values on the screen. Whenever we use **cout** it is necessary to include 'iostream.h' at the beginning of the program. The operator << is called an insertion operator since using it we insert values in the output stream that goes to the console. In C << was a bitwise left-shift operator. In C++ too this operator exists but in addition to the bit operations the << operator can also be used to send output to the console. This facility of using the same operator for performing different operations is called **Operator Overloading**.

cout is better than **printf( )** on two counts:

(a) We are not required to remember and use the format specifiers '%d', '%f', etc.

(b) We can output the contents the entire object **z** by simply saying **cout << z**. For this to work we would have to understand an implement a concept called operator overloading, which we are going to cover later.

Since class members are by default private they cannot be accessed from outside the class. Hence the program shown in the slide would not work. Let us now see what we need to do to make it work.

Making It Work

In the previous program our aim was to create an object **z1** of class **a** and initialize its data members **i** and **j** with data 10 and 3.14 respectively. Then using **cout** we wished to print the data on screen. If we try to execute this program it would generate errors. This is because we are trying to initialize the data members **i** & **j** outside the class. This is not possible, because the default access modifier for a class is **private**, which means that the data members cannot be accessed outside the class. A solution to this is to make access modifier for data as **public**. However, this is not a good solution, as it does not protect data. A better solution to this is shown in the next slide.

Running C++ Programs

This slide shows the steps that you should carry out to create and execute the C++ programs using the Visual C++ 6.0 compiler.

Once the program is typed and saved you can compile and execute the same using either Ctrl + F5 or by clicking the **!** button present in one of the toolbars in the VC++ ID.

Classes in C++

In this lecture you will understand:

(k) How to print data

Still Better Way…

Observe carefully the program given in the slide.

Here, the body of the class contains two keywords—**private** and **public**. They are used in C++ to implement a concept called data hiding. It means that data is concealed within a class so that it cannot be accessed by functions outside the class even by mistake. The mechanism used to hide data is to put it in a class and make it **private**. The **private** data and functions can be accessed from within the class. The **public** data and functions on the other hand, are accessible from outside the class. In the class given in the slide, the data items **i** and **j** follow the keyword **private**, so they can be accessed from within the class, but not from outside it.

A function declared within a class is called a 'member function' or 'method'. These two terms are interchangeable. Member function is a C++ term, whereas, method is a general object-oriented programming term. The member functions provide controlled access to the data members of class.

Usually the data within a class is **private** and the functions are **public**. The data is kept hidden from external use so that it remains safe from accidental manipulation, while the functions that operate on the data are **public** so they can be accessed from outside the class. However, there is no rule that data must be **private** and functions **public**. In fact in some cases you may be required to use **private** functions and **public** data.

Now, consider following statement,

```
e1.setdata ( 10, 3.14 ) ;
```

Why is the object name **e1** connected to the function using a '.'? This syntax is used to call a member function that is associated with the specific object. Since **setdata( )** is a member function of the class **a**, it must always be called in connection with an object of this class. It won't make much sense if we say,

```
setdata ( 10, 3.14 ) ;
```

This is because a member function is always called to act on a specific object, not on the class in general. Not only does this statement not make much sense, but the compiler would also flash an error message if you use it. Moral is, member functions of a class can be accessed only by an object of that class.

To use a member function, the dot operator connects the object name and the member function. The syntax is similar to the way we refer structure elements, but the parentheses signal that we are accessing a member function rather than a data item. The dot operator is also called 'class member access operator'.

Which Is Better

Suppose some value in the data member of a class has to be stored, but the condition is that the value should be greater than 0. A solution to this can be as shown in the slide.

In the program (the first program from left side of the slide), before assigning the value we are checking whether it is greater than 0 and if so then only we are assigning it to the data member of class **sample**. This code suffers from two problems. First, we have to keep the **age** data member as **public** so that after checking the entered value we can directly assign it to the data member. Secondly, every time we wish to set the value in **age**, we need to check the value before assigning it. A better solution would be as the second program (to the right-side) as shown in the slide.

Here, the data member **age** is kept **private** to the class. We have added a function called **setdata( )** to set value in **age**. This function checks the value before assigning it. This is a better solution because not only is **age** protected from accidental use, there is no need to check for its validity again and again.

Printing The Data

What if **private** data has to be printed? Add a member function called **printdata( )** to the class and call it from **main( )**. This is shown in the program given in the slide.

this Pointer

In this lecture you will understand:

* The this pointer and its utility
* How a console-based application is created in VC++IDE

How Many Copies

Now a question arises as to how many copies of data members and member function would get created if multiple objects were created? It is obvious that each object would have its copy of data members since values of data members in each object are likely to be different. Each object would occupy different locations in memory. Size of any object is the sum of sizes of data members present in the class (except **static** data members, which would be discussed later).

But what about member functions? The member functions are loaded only once in memory and are shared amongst objects. This makes sense, as member functions do not maintain state on behalf of objects. As against this, data members maintain state on behalf of objects. Thus, though an object is a collection of data and functions, this definition of the object is only conceptually true. In practice all objects share a common set of member functions.

How come member function would know on which object it has to work with. This is explained in the next slide.

The *this* Pointer

Any member function can be called only through an object of that class. When a member functions is called using an object the address of the object is implicitly passed to the member function. The member function collects this address in a pointer named **this**. When we compile a class, the compiler attaches **this** -> before each and every data member used in the member function. Since all data members are prefixed with **this** ->, (where this contains the address of the object) the data member of an appropriate object gets accessed. For example, in the slide **this** -> **i** would refer to **e1**'s **i** or **e2**'s **i** depending upon using which object (**e1** or **e2**) is the **setdata( )** function called.

Is *this* Necessary

The program given in the slide illustrates use of **this** pointer.

In this program the **setdata( )** function contains **i** and **a** as its parameters. The purpose of **setdata( )** is to set the value in the object's data members, whose names are **i** and **a** too. But when we compile the class, the compiler uses those **i** and **a**, which are most local. Since **i** and **a** defined as parameters are most local, they are used and compiler doesn't prefix **this** -> before any **i** and **a**. Hence data members of an object are not assigned any value.

To tell the compiler to use the data member **i** and **a** we need to explicitly attach **this** -> before them as shown in the slide.

Access Specifier And Constructor

In this lecture you will understand:

* How guaranteed initialization is possible with constructors

Recap

In the last lecture we had our first tryst with classes in C++. Just to reiterate, a class is a collection of data and functions that operate upon this data. Both data and functions can be **private** or **public**, which essentially decide the access to the data and functions within the class.

To develop a new application we need to create a new project. A question arises as to why should we create a project? A project can consist of group of related files that are required to develop an application. When a project is compiled all these files get compiled at one shot. This avoids the need of compiling each file independently and then linking them together. Till now the project that we built contained only one file. But as we go along we would be required to build multiple files holding different classes that would be required to build a single application.

When we create and execute a 'Win32 Console application', it directs the output on a console window i.e. DOS window. Hence we use **cout** (console out) to print the output.

Hiding vs Security

Protecting or hiding data from external usage is popularly known as data hiding. Don't confuse data hiding with the security techniques used to protect computer data. Security techniques prevent illegal users from accessing data. Data hiding, on the other hand, is used to protect data from accidental usage through honest mistakes. If we are bent upon then we can access even **private** data from outside the class as shown in the program given in the slide.

Here, **s** is an object of class **sample** and **p** is an **int** pointer. We have stored the address of **s** in pointer **p** by typecasting it to **int***. This makes **p** point to **s**. Now if we try to access the value at the address stored in **p**, first four bytes of **s** would be accessed, which is the value of data member **i**. And if we try to store some value, let's say 10, value of **i** would get overwritten by 10. Thus if we are bent upon, using pointers we can access even **private** data from outside the class.

C vs C++

Let us take the example of stack and see how it can be implemented in C and C++. First let us try to implement the stack in C. Observe carefully the two programs given in the slide. The first program (to the right of slide) implements stack in C whereas the second program (to the left of slide) implements the stack in C++ using a class.

Since in a class all members (data and functions) are private by default we may as well drop the keyword **private**. In the interest of clarity we recommend you to use the keyword **private** explicitly.

In the C++ class there is one member function which has the same name as the class. Such a function is called **constructor**. This function automatically gets called when we create an object of the stack class.

Though both programs would be able to maintain the stack the C++ code is superior for following reasons:

(a) In the C program we are required to call **init( )** function to specifically initialize **top** to 0. If we forget to call **init( )** then **top** would have a garbage value. This might prove disastrous. As against this, the constructor offers guaranteed initialization of top as it always gets called when we create a stack object

(b) There is no need to pass the address of the object explicitly to the member functions. The address automatically gets passed when we call the member function using an object.

Thus using encapsulation (keeping data and functions grouped together in a **class**) has obvious advantages over the keeping the data and functions separate from one another (as done in C).

Constructors

In Lecture 1 we had used the **setdata( )** function to initialize the data members of the object. We were required to call **setdata( )** explicitly using the object of the class. Forgetting to call **setdata( )** would leave the object un-initialized and hence prone to bugs. To ensure guaranteed initialization of an object's data members C++ provides a feature called constructor.

The constructor is a special member function that has a same name as that of the class and which gets called automatically when an object is created. Moreover, the constructor is never required to return any value, hence its return type is implicitly **void**. The following program given in the slide shows the constructor at work:

In C++, object creation is a two-step process:

(a) Memory is allocated for the object

(b) Constructor is called to initialize the data members of the object.

In the program given in the slide, we have created two objects. When we create **e1** the two-argument constructor gets called, and when we create **e2** the zero-argument constructor gets called. In previous programs we had not created zero-argument constructor in the class. But still we could successfully create the objects from the class. This is because C++ compiler provides an empty zero-argument constructor if we don't provide one.

However, if a class contains constructors other than a zero-argument constructor and if we try to build a zero-argument object thinking that the compiler anyway provides one, an error would occur. This is because compiler doesn't provide a zero-argument constructor if it finds any other constructor inside the class.

Note that the class created above contains two constructors having same names but different number of arguments. Such constructors are known as **overloaded constructor** functions. Function overloading permits us to have multiple functions with same name.

Function Overloading

In this lecture you will understand:

* The advantages of function overloading, default arguments & operator overloading
* How overloaded constructors simplify code

Function Overloading

In C, every function in a program has to have a unique name. At times this becomes annoying. For example, in C there are several functions that return the absolute value of a numeric argument. Since a unique name is required, there is a separate function for each numeric data type. Thus, there are three different functions that return the absolute value of an argument:

```
int  abs ( int  i ) ;
long  labs ( long  l ) ;
double  fabs ( double  d ) ;
```

All these functions do the same thing, so it seems unnecessary to have three different function names. C++ overcomes this situation by allowing the programmer to create three different functions with the same name. This is called **function overloading**. The program given in the slide illustrates this.

How does a C++ compiler know which **set( )** function to be called? It decides that from the type of the arguments being passed to the function.

However, if you execute this program it would generate an error. Why? The reason is that the last two definitions of **set( )** function receive the same type of argument but differ only in the value that they return. Moral is that the overloaded functions must at least differ in the type, number or order of parameters they accept. Don't rely on the return values to differentiate them.

Why C++ compiler cannot differentiate on the basis of return types? Consider the previous example. Can we decide by seeing the last two calls which function definition should get called? The answer would obviously be no, because both the last two definitions qualify for this call. This becomes an ambiguous situation for the C++ compiler. Hence it flashes an error.

Two In One

C++ also allows its programmer to specify default arguments for the function parameters. This simplifies calling functions having large number of parameters. Consider the program given in the slide.

Here, the two-argument constructor has default values for its parameters. Given below are a few examples indicating which constructor would get called while creating the object:

```
ex e1 ( 10, 3.14 ) ;     // calls 2-arg. cons.
ex e2 ;                  // calls 2-arg. cons. & Initializes both data members with default values
ex e3 ( 15 ) ;           // calls 2-arg cons. & Initializes i with 15 and j with 0
ex e4 = 10 ;             // calls 2-arg cons. & Initializes i with 10 and j with 0
```

However, the following statement would generate an error.

```
ex e5 = 2, 1.1 ;
```

We cannot create and initialize objects in this manner.

Calling Constructor Explicitly

In this lecture you will understand:

* How to initialize array of objects by calling constructors explicitly

Calling Constructor Explicitly

Till now we saw that constructors are called implicitly to make guaranteed initialization of objects. There are situations when we are required to call even constructors explicitly. Consider the second statement given in **main( )** of the program given in the slide.

This statement initializes an object **e1**. To initialize the object we have called the two-argument constructor. Since constructor is used merely to initialize the allocated object's data, calling the constructor directly without allocating memory for the object would lead to an error. Hence when the C++ compiler finds that constructor is being called explicitly it first makes provision to allocate space for the object. As we have not specified any name to the objects, they are known as nameless objects.

Note that once nameless objects' values are assigned to object to be initialized with, they became useless in the program. If they don't die, unnecessarily they would keep occupying the memory. Hence C++ compiler ensures that nameless objects die after they are assigned to the object.

Pointer To An Object

What if we want to change the value of a local object from another function? Like primitive variables we can pass the address of the local object to another function, collect the address in a pointer of the same type and change the value of the object. This is shown in the program given in the slide.

Here, the two-argument constructor gets called to initialize object **e**. Next, we have called **set( )** function to set new values for the data members of **e**. One more way is given in the slide to set new values. We have called function **fun( )**. To this function we have passed address of **e** (i.e. **this** pointer). Then in **fun( )** using pointer **p** (which contains address of **e**) we have called **set( )** function to set new values.

Which Gets Called

In this lecture you will understand:

* Determine which constructor gets called
* The overloaded assignment operator & copy constructor

Readymades

The C++ compiler provides a zero-argument constructor by default. However, if we provide either the zero-argument constructor or multiple-argument constructor then the compiler would not provide the zero-argument constructor.

However, the C++ compiler provides copy constructor even if some other constructor is available. The default copy constructor is not provided if we defined one explicitly. The compiler also provides a default overloaded assignment operator. We can override the default one if we want.

Which Gets Called

This slide shows which functions in the class are called while creating, assigning, passing and returning objects. Let us analyze the program statement by statement.

(a) The following statements call two-argument constructor which is also a default-argument constructor:

```
ex  e1 ( 1, 2.5 ) ;
ex e2 ;
```

(b) The following statement calls overloaded assignment operator function of the class. Since we have not provided it, it would be added to our class by the C++ compiler.

```
e2 = e1 ;
```

The overloaded assignment operator gets called when data of members of an existing object of one type is to be copied to members of other exiting object of the same type.

(c) The following statement calls a copy constructor (special type of constructor that is used to initialize the object with the object of same type).

```
ex  e2 = e1 ;
```

Unlike zero-argument constructor, which is provided only when no other type of constructor is present, C++ compiler always provides a copy constructor. The copy constructor gets called when data of members of an existing object of one type is to be copied to members of the new object of the same type. Thus, here **e2** is a new object, which gets created first and then, gets initialized with data members of **e1**.

(d) In the following code snippet, the copy constructor is called when the **set( )** function gets called. This is because when **set( )** is called, object **x** is created and initialized with the same type of object.

```
e2.set ( e1 ) ;

void  set ( ex  x )
{
}
```

(e) In the following code snippet the two-argument constructor with default arguments would get called when we call **set( )**:

```
e2.set ( 1 ) ;

void  set ( ex  x )
{
```

```
}
```

On calling **set( )**, **x** object would be created and **x** would be initialized to 1. It would be like **ex x (1)**, which invokes two-argument constructor with default arguments provided in the class.

(f) In the following code snippet when the **fun( )** function would return, the object returned by it would get collected in a nameless object. Since nameless object would get created first and then it would get initialized, a copy constructor would get called. Next the contents of nameless object would get copied member-by-member to existing object **e2**, through an overloaded assignment operator.

```
e2 = fun( ) ;

ex fun( )
{
    ex t ;
    return t ;
}
```

Operator Overloading

In this lecture you will understand:

* What is operator overloading

Operator Overloading

C++ provides a facility called operator overloading that makes the operations on objects intuitive. Suppose we wish to store the result of concatenation of two strings, **s1** and **s2**, into **s3**. Instead of using **strcat**( ) for concatenation and **strcpy**( ) for copying, the statement **s3** = **s1** + **s2** would be more intuitive. The program given in the slide uses the concept of operator overloading to perform addition of two objects.

Here, the statement **c** = **a** + **b**, can be interpreted as **c** = **a.operator** + **(b)**. Here a call to **operator** + **( )** function is being made using the object **a**, whereas, object **b** is being passed to it. We can even cascade operators, for example, **a** + **b** + **c** +**d** or **a** + **b** * **c** / **d**, etc. is permissible. The priorities of operators remain same as that for primitive data types and cannot be changed through operator overloading.

The result of the operation **a** + **b** gets collected in a nameless object, which is then assigned to **c**.

Tips About Overloading

If you want to extend the reach of C++ operators you can always do so using operator overloading. However, there are some limitations too. We cannot overload operators like . , :: , ? and : , because they work on names and not on operands (values).

Separating Things

In this lecture you will understand:

* What is nameless object

Separating Things…

In real-life programming using C++, the definition of class member functions is kept separate than the declaration of the class. This slide contains the declaration of class **comp**. The declaration is kept in '.h' file.

Slide Number 04

This slide contains the definition of class **comp.** The class definition is kept in '.cpp' file. The class information is provided to '.cpp' file by **#includ**ing 'comp.h' file.

Note that the default values if any are to be given in a function, then they must be specified in '.h' file, where function declaration is given. These values should not be specified in function definition.

Slide Number 05

This slide contains the program that uses class **comp**. We have **#include**d 'comp.h' but not 'comp.cpp'. The 'comp.cpp' contains class implementation, which should be protected from the end-user. The '.cpp' file should never be **#include**d. When the definition (i.e. '.cpp') is compiled it generates the object code. This object form and the '.h' file is then given to the other programmer. This doesn't expose the code written by the creator of the class. The programmer who wants to use such a class has to include the '.h' file and link the object file to his program.

More Variety

In this lecture you will understand:

* Need of friend function
* How to write and use a friend function

More Variety

Now we know what sort of flexibility is provided by the C++ language when compared with C in terms of functions and user-defined data types. Let us now discuss more on operator overloading and some issues related with it. Till now we have used the overloaded **operator +( )** function to add same type of objects. What if we add a user-defined object with a primitive type? Consider the program given in the slide.

Here, **c = a + d**, would be expanded to **c = a.operator + (d)**, **d** would be collected in **c2,** which would become **comp c2 (d)**. This becomes a call to one-argument constructor. Since **comp** class contains a two-argument constructor with default parameters, this constructor would get called and **c2** would get initialized. The overloaded **operator +( )** function would then work without any problem. The conversion from **double** to **comp** that happened here is called implicit conversion. We can prevent such implicit conversions by specifying **explicit** keyword before the constructor. If we still want **a + d** to work we need to write it this way, **a + comp (d).** Here **comp (d)** is an explicit call to the constructor.

Also, **c = d + a**, would be expanded to **c = d.operator + (a)**. This time **operator + ( )** function of **comp** class would not get called. Instead **operator + ( )** of **double** class would get called. Since **double** class has no overloaded **operator + ( )** function that can collect a **comp** object, the call would result into an error. To make such arithmetic statements work, the user-defined class whose object is taking part in computation should provide a **friend** function as shown in the next slide.

Making It Work

To make **d + a** work, we need to provide a global **operator + ( )** function as shown in the slide.

When compiler sees a global overloaded **operator + ( )** function it converts the call to **operator + (d, a)**. Here both operands are being passed to function. The value of **d** would get converted to **comp** using one-argument constructor and value of **a** would get copied with the help of copy constructor. But if we try to access **private** data members of **c1** and **c2** objects to actually perform the addition, it would result in an error. This is because the global function is not a member of the class. To allow the global function access **private** data members of the class, the class creator needs to declare the global function as **friend** in the class. Thus a **friend** function is actually a global function with rights to access **private** data members of the class in which it is declared.

Now, all the three statements involving addition operation would work. However, if we use **explicit** keyword with the constructor, then the last two statements, **c = a + d ;** and **c = d + a ;** would not work. In order to make it work we would require to add two more **friend** functions, one that works on a **double** and object of **comp**, and the other that works on object of **comp** and **double**.

Pre, Post and References

In this lecture you will understand:

* Overloading of pre and post incrementation operators
* Need of references

Pre & Post

Pre and Post increment and decrement operators are unary operators. This means they operate only on one operand. For example, in **++i**, **i** is the only operand. Also, **++i**, would get expanded to **i.operator++( )**. Here no value is passed to **operator++( )** function. The program given in the slide demonstrates unary operator at work.

Here we have two overloaded operator ++ functions. One is **index operator ++( )** and the other is **index operator ++ (int)**. The first one is used for prefix notation, whereas, the second one is used for postfix notations. The only difference is the **int** in the parentheses. This **int** isn't really an argument. It's simply a signal to the compiler to call the postfix version of the overloaded operator. On similar lines we can implement the pre and post decrement operators as well.

References

Suppose we have a structure containing numerous elements. If we are to pass a variable of this structure to a function then the function will have to collect it in another structure variable. This would lead to unnecessary duplication of data as well as overheads in terms of time required to pass a big structure. This difficulty can be eliminated in C by passing address of the structure variable. C++ provides a more elegant solution by providing a reference. A reference is a **const** pointer, which gets de-referenced automatically. Let us understand this with the help of examples given in the slide.

Here, **j** is a **reference** of **i**, whereas **i** is a referent. When we write **j = 20**, **j** gets de-referenced automatically and becomes ***j**. Hence 20 gets assigned to ***j**. Since **j** holds the address of **i**, value stored in **i** gets overwritten with **20**.

Reference is more elegant than pointer in the way that we do not have to de-reference it explicitly using ***,** as well as, -> while accessing structure members. Also, we do not have to store the address of the variable in it by specifying **&**. The address of the variable gets stored automatically. For example, the statement **int &p = i**, is expanded to **int * const p = &i**, automatically.

But once a reference (which is a **const** pointer) is tied with a variable, it cannot be tied in any way with another variable. As against this, a pointer has flexibility that it can store address of one variable at one time and later on store address of another variable.

Subtleties

A few points to note...

(a) A reference must always be initialized. Thus the following set of statements produce an error.

```
int i = 10 ;
int &j ; // error
j = i ;
```

(b) A variable can have multiple references. Changing the value of one of them effects a change in all others.

(c) Though an array of pointers is acceptable, an array of references is not.

Are References Necessary

In this lecture you will understand:

* Dos and don'ts about references

Different Calls

What is the advantage of referencing? Referencing goes a long way in removing untidiness in code, making it more readable, as the program given in the slide would justify.

Here, the call to **fun1( )** is a normal call where we have passed the variable **d**, which gets collected in **x**. Changing the value of **x** in this function, has no effect on the value **d**. The **fun2( )** function demonstrates a call by address. In this function using pointers we could change the values of **d** in the calling function. The **fun3( )** achieves the same effect more elegantly by using the reference. While calling the function that accepts a reference, we do not need to pass the address explicitly by specifying **&d**. The address of the variable is automatically passed.

Are References Necessary

Consider the code snippet given in the slide.

Here, whenever the function **operator +( )** gets called a new object **c2** gets created. We can avoid the creation of new copy by defining **c2** as a reference. Here if we define **c2** as pointer then the user of class would be required to write **c = a + &b.** This expression would work but is not a good idea since it gives a feeling that **a** is being added to the address of **b**.

Dynamic Memory Allocation

In this lecture you will understand:

* How to achieve guaranteed cleanup of objects using destructors
* Dynamic Memory Allocation
* How memory is allocated on stack and heap
* The differences between static and dynamic memory allocation

Dynamic Memory Allocation

While doing dynamic memory allocation in C the memory is allocated from **heap**. Thus heap is a pool of memory from which standard library C functions like **malloc( )** and **calloc( )** allocate memory. The memory allocated from system heap using **malloc( )**, **calloc( )** and **realloc( )** is vacated (deallocated) using the function **free( )**.

C++ offers a better way to accomplish the same job through the use of the **new** and **delete** operators. The **new** operator allocates memory from free store (in the C++ lexicon, heap is called **free store**), whereas, the **delete** operator returns the allocated memory back to the free store. Thus the **new** and **delete** operators perform the job of **malloc( )** and **free( )**.

Instead of using the **new** operator to allocate memory had we used **malloc( )** the allocation statements would have looked like this:

```
i = ( int * ) malloc ( sizeof ( int ) ) ;
a = ( float * ) malloc ( sizeof ( float ) ) ;
e = ( emp * ) malloc ( sizeof ( emp ) ) ;
```

Note that since **malloc( )** returns a **void** pointer it is necessary to typecast it into an appropriate type depending on the type of pointer we have on the left hand side of the assignment operator. This gets completely avoided when we are using the **new** operator as shown below:

```
p1 = new int ;
p2 = new employee ;
```

Static V/s Dynamic Allocation

The variables defined in a program gets allocated on stack, whereas, dynamically created variables are allocated on heap. Allocations in stack are in adjacent locations. Same may not be true in case of a heap.

Allocation

Consider the code snippet given the slide.

On compilation of a program, a compiler creates a symbol table. In this symbol table it stores information like name of the variable, its scope, and size of the variable. It also stores the offset from data segment for global variables and from stack segment for local variables. Then it stores the instruction for local variables and the statement that cause a call to **malloc( )** function. Finally after creation of OBJ code (Object code) compiler discards the symbol table. The OBJ code of our program is then linked with the OBJ code of library (if there is any call to a library function in our program). Finally, an EXE file gets generated. This EXE file contains some header information, which is used by the loader program (one which loads an EXE into memory).

If there were lots of global variables used in a program, then the loader would fail in loading in the variables. This is because after loading an EXE file into memory first job is to load all global variables. And if enough memory is not available then at this stage loader will fail. Secondly, if too many local variables were present in our program then it would cause stack overflow. Similarly, if too much of dynamic memory were required to be allocated, then it would cause heap overflow.

Nameless Objects

In this lecture you will understand:

* Difference between **new** and **malloc( )**
* Difference between **delete** and **free( )**
* How to avoid memory leaks and dangling pointers and make your program robust
* Allocating memory dynamically for an array

Named & Nameless Objects

Look at the code snippet given in the slide.

Here, the first statement creates an object **p** of class **shape**. The second statement declares a pointer to an object of class **shape**. The third statement, creates an object of class **shape** dynamically and stores its address in **q**. Actually, **new** allocates memory for a nameless object and the address of this nameless object gets stored in the pointer **q**.

Are *new* & *malloc( )* Same

No! **new** not only allocates space, it also calls the constructor. Similarly, **delete** before freeing the space calls the destructor (function that gets called automatically just before the object is destroyed). In contrast, **malloc( )** & **free( )** merely allocate and de-allocate memory. The program given in the slide, justifies this.

Avoid Memory Leaks - I

Consider the program given in the slide.

Here, we have called function **f( )**, which creates object **e** of class **ex**. The constructor allocates memory for an **int** and **float** and stores the addresses in **p** and **q** respectively. The object **e** dies when control returns to **main( )** from function **f( )**. As a result, the destructor gets called. However, the memory that we have allocated for **p** and **q** still remains allocated causing a memory leak. To avoid this memory leak we must de-allocate memory using **delete** operator in the destructor as shown in the slide.

Array Allocation

Let us now see how memory is allocated dynamically, for an array: Consider the code given in the slide.

Here, we have allocated memory dynamically for an array of 10 integers and have made pointer **p** to point to this array. Then we have accessed the array and store some values in it.

Avoid Memory Leaks - II

One more situation is there which can cause memory-leak. Consider the code snippet given in the slide.

Here, too we have called function **f( )** through **main( )**. In this function, **z** is a pointer, which points to an array of objects of type **ex**. The statement, **delete z**, should de-allocate memory pointed to by **z**. Yes it de-allocates the memory pointed by **z**, but does not call destructor for all the objects in an array. Hence the memory allocated in the constructor does not get cleaned up. To make **delete** call destructors for all objects we need to use the form **delete[ ] z**.

Static Objects

In this lecture you will understand:

* Role of **static** data members and member functions of a class

Static

We know that each object contains its own separate data members, whereas, the member functions are shared amongst all objects. However, there is an exception to this rule. If a data member of a class is declared as **static**, then only one such item is created for the entire class, irrespective of the number of objects created from that class. We can have a **static** member function as well.

A **static** data member is useful when all objects of the same class must share a common item of information. A **static** data member is available only within the class, but it continues to live till the time program execution doesn't come to an end. In that sense a **static** data member is similar to the ordinary **static** variable. However, their utility is different. While a normal **static** variable is used to retain information across function calls, **static** data members of a class are used to share information among the objects of a class. For example, if we develop a **calendar** class that displays a calendar with Month name, days and dates. If we want to display multiple calendars at a time and allow users to manipulate them, we would be required to create multiple objects. But each would share month names and day names since they would be common for all calendars.

Likewise we can even count number of objects created from the class. The program given in the slide shows how to do so.

The class **sample** has two data members, **i**, which is a normal **int**, and **count**, which is of type **static int**. The constructor for this class causes **count** to be incremented. In **main( )** we have defined two objects of class **sample**. Each time an object is created the constructor gets called. Hence, **i** would be initialized to 0 and **count** would get incremented. We have created a function **objects( )** of type **static** to display the current value of **count**. Note the way this function has been called. We can call this function in two ways as shown below:

```
sample s1 ;
s1.objects( ) ;
sample::objects( ) ;
```

The first way is a little clumsy. We shouldn't need to refer to a specific object when we're accessing something, which is not at all part of the object. The second way is more elegant. It's more reasonable to use the name of the class itself with the scope resolution operator. Also note that it is necessary to initialize the static variable and that too outside the class.

Difference

Observe carefully the table given in the slide.

The **static** functions cannot access non-static data members or call non-static member functions because **static** functions are never called through objects and hence object's address is never passed to them. If non-static data members and member functions were referred from within **static** functions, the compiler would fail to resolve the statements, as it would never come to know which object's data members to access and on which object's data the member functions are going to work.

Like **static** member functions, **friend** functions too are not called through objects. Unlike **static** functions, **friend** functions are always out of the class scope.

Singleton Class

In this lecture you will understand:

* What is mean by Singleton class

Problem

Static data and functions can be used to create a class of which only one object can be created. Such a class is known as **singleton** class. This is a frequent requirement in an Object-Oriented program. Such a common requirement is often called a **Design Pattern**.

Singleton Class

The slide shows a class **sample** containing a **private** zero-argument constructor and a **public static** function, **create( )**. When the **create( )** function is called it first checks whether any object of type **sample** has been created or not. If **p** contains NULL it means no object has been created. In such a case it creates an object of the **sample** class dynamically, stores the address in **p** and returns **p**. If an object of class **sample** already exists then a new object is not created. Instead the address of the existing object is returned.

The client application cannot create the object of the **sample** class directly because, the zero-argument constructor is private and no other constructor is available. The client has no other choice but to create the object by calling the static function **create( )**.

Reuse

In this lecture you will understand:

* Ways to reuse the existing code
* When to use 'Has' a (container) or 'Is' a (inheritance) relationships

What Next

The slide shows the different features of a C++ class that we have done so far.

Reuse

Reusability of the code is a main feature of OOPs. Reuse of code can be divided into two categories—Source-code level reuse and Object-code level reuse. Source-code level reuse is achieved through templates. Templates are used for generating functions and classes based on type of parameters supplied. By using templates we can design a single class or function that operates on data of many types, instead of having to create separate classes and functions for each type.

Object-code level reusability involves containership and inheritance. For example, a MFC (Microsoft Foundation Class Library) class **CString** is used to work on strings of variable lengths. But if list of strings is to be maintained then a class called **CStringList** is used. **CStringList** class holds objects of **CString** class. Thus, **CStringList** is a container holding **CString** objects.

Inheritance allows one class to reuse the state and behavior of another. For example, a MFC class **CButton** inherits properties and functionalities of **CWnd** class, thereby reusing the **CWnd** class. The class that inherits the properties and method implementations of another class can extend the methods and properties it has inherited by overriding methods and providing additional properties and methods.

Thus, when B class is derived from A class we say that B is like A. In other words, B has 'Like a' relationship with A. In case of containership it is said that the class has 'Has a' relationship with another class.

Relationships

Let us now examine some relationships that exist in real world. In real world too we have same sort of relationships like containership and inheritance. For example, car and scooter are vehicles. This means that car and scooter inherits properties and functionalities of a vehicle. This is an inheritance relationship. As against this, a car and a scooter contain an engine. So the relationship between the car and engine or scooter and engine is that of containership.

Similarly, the UI (User-Interface) elements in Windows depict such relationships. For example, toolbar, status bar, button, edit control, combo box and list box are windows (i.e inheritance), whereas, toolbar contains buttons, combo box contains edit box and list box (i.e. containership).

Containership and Inheritance

In this lecture you will understand:

* How to write highly reusable classes with the help of different types of inheritance
* Why construction of an object of derived class always proceeds from base class to derived class

Containership

Consider the code snippets given in the slide.

Here, **string** is a class that contains a character array **str** used to store a string. The **print( )** function prints the string. This class can have functions to carry out conversion operations like lower to upper or upper to lower, and other operations like concatenation, comparison, etc. What if we wish to maintain a collection of string objects and perform operations on them like printing all strings, adding a new string, deleting a string, retrieving a string, etc. The solution is to create a container class **stringlist** as shown in the slide.

The **stringlist** class contains an array of objects of string. The **add( )** function adds an object of string to the array **s[ ]**. Similarly, **printall( )** function can be used to print the strings in **s[]**.

Inheritance

Inheritance is the process of creating a new class, called **derived** class from an existing class. The existing class is called a **base** class. The **derived** class inherits all the capabilities of the **base** class but can add new features and refinements of its own. By adding this refinement the base class remains unchanged. Inheritance not only promotes reuse of existing code but also helps in the original conceptualization and design of the program. We can derive a new class from an existing class even if we do not have the source code of it. Let us now see inheritance at work:

Consider the program given in the slide. Here **index** is the base class and **index1** is the derived class. The base class serves the purpose of a general-purpose counter. However, it can only increment the counter and not decrement it. We would not add the decrement facility in the **index** class for two reasons:

(a) We may not have access to source code of **index**.

(b) If **index** class is a thoroughly tested class, inserting new code in it may demand additional testing.

Instead we would prefer to derive **index1** from **index** and add the decrement facility to it. The class **index1** doesn't need the **operator** ++ **( )** function, since it is already present in the base class. When we create an object **i** or when we increment **i** the functions in the base class get called. Note that for the data member **count** to be available in the derived class it must be declared **protected** in the base class. A **protected** member can be accessed in the derived class but not from outside the class. Thus, we have increased the functionality of the **index** class without modifying it.

A few important tips are in order...

(a) A derived class function can call a base class function through a statement like,

```
base::function( ) ;
```

(a) A base class function can never call a derived class function except the function is defined **virtual** in the base class. Virtual functions would be covered in next lecture.

Object Size

In this lecture you will understand:

* How to access base class data members in derived class using protected access modifier
* Private, Public & Protected Inheritance

Object Sizes

Consider the code given in the slide.

Here, class **b** contains 3 **int**s **i**, **j** and **k** as data members. The class **d** is derived from class **b** and contains its own data members **x**, **y** and **z** all **int**s. Now, if we create an object of derived class **d** then the size of the object would be the total size of the data member of its own class as well as those that are inherited from the base class. All data members of the base class are inherited irrespective of access modifiers. Hence, the size of an object of class **d**, would be 24.

Access

Considering the same example discussed in earlier slide, let us see how class **d** can access members of class **b**. Class **d** has access to all the **protected** as well as **public** members of class **d**. We can access the **public** members of class **b** from outside the class. Similarly, we can access only **public** members of class **d** from outside the class.

Which Gets Called

Consider the program given in the slide.

Here, after creating an object **z** of class **b** the first statement would print **Hello** by calling function **f1( )** inherited by derived class **b** from class **a**. The second statement would print **Hi** by calling function **f2( )** of derived class **b**. The third statement would print **GM** by calling **f3( )** function of derived class **b**. The fourth statement would print two strings in the order **ByeAdieu**. This is because in the **f4( )** function of derived class **b** we have firstly called function **f2( )** of base class **a**. In the last statement we have explicitly called the function **f2( )** of class **a** through the statement **z.a::f2( )**, hence the output would be **Bye**.

This program demonstrates that while inheriting a class we can:

(a) Use the existing functionality of the base class.

(b) Extend the existing functionality by providing new functions in the derived class.

(c) Override the existing functionality of the base class by providing new functions in the derived class bearing the same name as those in the base class.

(d) Combine the existing functionality of the base class with the new functionality provided in the derived class.

Calls Differ

In this lecture you will understand:

* Difference in calling mechanism for normal member functions and constructor functions

Calls Differ

This slide shows the difference in calling mechanism for normal member functions and constructor functions.

When a derived class object calls a function, the compiler first checks for the function in the derived class. If it finds the function, it links the call to the definition of the function. If the compiler doesn't find the function in the derived class then it searches for the function in the base class. If function is found then it is linked, otherwise it is searched in super base class. If function is still not found or super base class doesn't exist then an error occurs.

When an object of a derived class is created, the compiler searches for a proper constructor in the derived class. If proper constructor is not available the compiler straightaway flashes an error. If a proper constructor is found, then the compiler searches for a zero-argument constructor in the base classes (irrespective of how many argument object is created). If zero-argument constructor is not found in any of the base classes an error occurs. But if zero-argument constructor is found, the constructor of the top-most base class is called followed by the constructor of the derived class. Thus, the order of calls to constructors is always from base class to derived class.

Slide Number 04

Suppose during the construction of a derived class object if we wish to call a one-argument constructor of the base class. For this we need to explicitly call the one-argument constructor of the base class, as shown below:

```
class b
{
    public :
        b( int  k ) : a( k )   // a is the base class containing one-argument constructor
        {
            cout  << "b's  constructor " ;
        }
}
```

Here, we are directing the compiler to call one-argument constructor of the base class.

Why Base To Derived (B To D)

As we saw in the last slide the order of construction should be from base towards derived. Let us now try to understand the reason for this. This order is important because of two reasons:

(1) We may want to overwrite the values of data members of the base class in the derived class.

(2) We may want to use the base class data members in the derived class constructor.

The program given in the slide demonstrates this.

Types of Inheritance

In this lecture you will understand:

* The Types of inheritances—Multi-level Inheritance, Multiple Inheritance

Types Of Inheritance - I

The derived class creator can inherit base class **public**ly, **protected**ly or **private**ly. This leads to three types of inheritance:

Public Inheritance—So far we have used only this type of inheritance. In this type of inheritance the **public** and **protected** members of the base class remain **public** and **protected** members for the derived class.

Protected Inheritance - When a class is inherited **protected**ly all **public** members of it become **protected** in the derived class. Hence, the derived class object would not be able to access the **public** members of the base class from outside the derived class. But the **public** as well as **protected** data members of the base class would be accessible from within the derived class.

Private Inheritance - When a class is inherited **privately** all **public** and **protected** members of it become **private** in the derived class. Hence, the derived class object would not be able to access the **public** and **protected** members of the base class from outside the derived class.

If a class **d** is derived **private**ly from class **b** then **protected** and **public** members of class **b** become **private** in class **d**. Hence an object of class **d** object will not be able to access the **public** members of class **b** from outside the class **d**. Moreover, if a class **c** is derived from class **d** then from class **c** we will not be able to access the **public** as well as **protected** members of the class **b**, even if we inherit class **c** **public**ly.

Types Of Inheritance – II

There are two more types of inheritance relationships:

If there are multiple classes in an inheritance chain then it is know as **multiple-level of inheritance**. For example, **class b** is derived from **class a** and **class c** is derived from **class b**.

If we derive a class from more than one class then it is known as **multiple-inheritance**.

If we derive a class **b** from class **a**, then we can further derive class **c** from class **b**, class **d** from class **c** and so on. This is known as **multiple levels of inheritance**.

A **class c** can be derived from two independent classes called **class a** and class **b**. This is known as **multiple inheritance**.

Suppose class **b** is derived from class **a** and class **c** is derived from class **b**. If we create an object of class **c** the constructors would be called in the order **a**, **b**, **c**. This means construction proceeds from base class towards derived class. The destruction proceeds from derived class towards base class.

Polymorphism

In this lecture you will understand:

* What is Polymorphism
* Need of polymorphism
* The difference between compile-time & runtime polymorphism

Problem

Consider a problem where we want to draw circles and rectangles on the screen. After the screen is erased, we would like to repaint or redraw all the shapes drawn before. For this we would have to create two arrays, one for circle and another for rectangle. Each array would then store the coordinates & color of the respective shapes that are drawn on the screen. Not only this, it would be necessary to store the order in which the shapes were created. If we the order is not used, then, a circle that was drawn later may overlap a rectangle drawn earlier. To avoid this we would have to maintain one more array in which we would store say 'R' for rectangle and 'C' for circle for example, in the order in which they were drawn. This solution is not elegant because we need to maintain a separate array for each shape. If anytime a new shape is added, a new array needs to be created to store the information. Moreover, array if allocated statically imposes limitation of being non-resizable.

Solutions

The solution could be to create a linked list of **void** pointers. When a new shape is drawn an object of it would be created dynamically and its address would be stored in the array of **void** pointers. Here we would need only one array to store both **circle** as well as **rectangle** object pointers. But once we store the address of any object its type would be lost and while retrieving the address it would be difficult to know which object's address is stored where. Hence again we would be needed to maintain one more linked list that would store the indicator like 'R' and 'C', which would help us in knowing whose address is stored in which location. Thus this solution has an overhead of an extra linked list that stores 'R' and 'C'.

Is there any solution whereby we are now required to maintain multiple arrays and that the type information remains intact? We need to create a class called **shape** and derive the classes—**circle** and **rectangle** from it. Then we need to create a linked list of **shape** pointers and store in it the addresses of **circle** and **rectangle** objects. This process of storing address of derived class object in pointer to base class object is called **upcasting**. But by doing this the type information is again lost. That's when a feature called Runtime Polymorphism can be used to access the correct type information and work with different **circle** and **rectangle** objects. The philosophy and working of this concept is discussed in next few slides.

Polymorphism

The meaning of the word polymorphism is one thing existing in several different forms. Such type of polymorphism we had seen in the form of function and operator overloading. There is one more type of polymorphism. In this type on performing the same action a different activity takes place. In practice we notice many cases of this type of polymorphism. For example, using mouse if we click on a menu then a menu pops-up, whereas, if we click on close button the window closes. This means the action that we take (left clicking the mouse) is same but its effect is different. Similarly while driving a car on carrying out the same action of shifting gears the car moves either in forward or in reverse direction.

In C++ polymorphism can either be Compile-time polymorphism or Runtime polymorphism. The compile-time polymorphism is implemented through function and operator overloading, whereas, runtime polymorphism is implemented using **virtual** functions.

Slide Number 6

In this lecture we would try to understand what's the need of runtime polymorphism. Then we would discuss how virtual function mechanism works.

Consider the program given in the slide.

Here **circle** and **rectangle** are classes derived from the base class **shape**. Each of these three classes has a member function **draw( )**. In **main( )**, having created the objects **c**, **r** of **circle** and **rectangle** respectively and a pointer **p** to base class, we have assigned the address of a derived class object to the base class pointer through the statement,

```
p = &c ;
```

Should this not give us an error, since we are assigning an address of one type to a pointer of another? No, since in this case the compiler relaxes the type checking. The rule is that pointers to objects of a derived class are type-compatible with pointers to objects of the base class. Taking the address of a derived class object and storing it as the address of the base class object is called **upcasting**.

When we execute the statement,

```
p -> draw( ) ;
```

which function gets called—**draw( )** of **circle** or **draw( )** of **shape**? The function in the base class gets called. This is because the compiler ignores the contents of the pointer **p** and chooses the member function that matches the **type** of the pointer. Here, since **p**'s type matches the base class, the **draw( )** of base class gets called. Same thing happens when we call **draw( )** for the second time.

Sometimes this is what we want, but it doesn't provide the facility of accessing functions of different classes using the same statement. Let's make a small change in our program. Let's precede the declaration of the function **draw( )** in the base class with the C++ keyword **virtual**.

If we execute the program now the output would be:

```
circle
rectangle
```

As can be seen from the output, this time instead of the base class, the member functions of the derived classes are executed. Thus the same function call,

```
p -> draw( ) ;
```

executes different functions, depending on the contents of **p**. The rule here is that the compiler selects the function to be called based on the **contents** of the pointer **p**, and not on the **type** of the pointer. Problem is how does the compiler know which function to link, when it doesn't know which object's address **p** may contain at runtime. It could be the address of an object of the **circle** class or of the **rectangle** class. Which version of **draw( )** does the compiler call?

In fact the compiler doesn't know what to do, so it arranges for the decision to be deferred until the program is running. At runtime, when it is known what object is pointed to by **p**, the appropriate version of **draw( )** gets called. This is called **late binding** or **dynamic binding**. (Choosing functions in the normal way, during compilation, is called **early**, or **static** binding.) Late binding requires some overhead but provides increased flexibility.

Instead of pointers had we used references the effect would have been same.

Pure Virtual Functions

In this lecture you will understand:

* How to achieve runtime polymorphism using virtual functions
* Advantage of pure-virtual functions over virtual functions

Pure Virtual Functions

We can add another refinement to the virtual function declared in the base class of the last program. Since the function **draw( )** in the base class never gets executed we can easily do away with the body of this **virtual** function and add a notation *=0* in the function declaration, as shown in the slide.

The **draw( )** function is now known as a **pure** virtual function. Thus, a pure virtual function is a virtual function with no body and a *= 0* in its declaration. The = sign here has got nothing to do with assignment; the value 0 is not assigned to anything. It is used to simply tell the compiler that a function will be pure virtual function, i.e. it will not have any body.

If we can remove the body of the virtual function can we not remove the function altogether. That would be too ambitious and moreover it doesn't work. Without a function **draw( )** in the base class, statements like

```
p -> draw( ) ;
```

would be invalid.

Since **draw( )** of the base class was never getting called we made it a pure virtual function. There is another side to it. At times we may want that a user should never be able to create an object of the base class. Consider the same example, if there is base class called **shape** from which three classes **line**, **circle** and **rectangle** have been derived. We would never make an object of the **shape** class; we would only make objects of the derived classes to draw specific shapes. A class from which we would never want to create objects is called an **abstract** class. Such a class exists only as a parent for the derived classes. Now how do we communicate to users who are going to use our classes that they should never create an object of the base class? One way is to document this fact and rely on the users to remember it. That's a sloppy way. Instead, a better way would be to write the base class such that any object creation from it becomes impossible. This can be achieved by placing at least one pure virtual function in the base class. Now anybody who tries to create an object from such a base class would be reported an error by the compiler. Not only will the compiler complain that you are trying to create an object of the abstract class, it will also tell you the name of the virtual function that makes the base class an abstract class.

Whenever a pure virtual function is placed in the base class, we must override it in all the derived classes from which we wish to create objects. If we don't do this in a derived class then the derived class becomes an abstract class.

Consider the program given in the slide.

Here, **shape** is an abstract class containing **draw( )** as a pure virtual function. The class **circle** and **rectangle** have been derived from **shape** class. In **main( )** we have created few objects of **circle** and **rectangle**. Then we have initialized an array of pointers to **shape** with the addresses of derived class objects. Then through a loop we have called **draw( )** function through **p**. At runtime depending on the **contents** (i.e. address of a derived class) stored in **p[ ]** the **draw( )** function would get called. Thus the way we call **draw( )** remains same, but either **circle**'s **draw( )** or **rectangle**'s **draw( )** would get called. Thus the interface is same but the implementation of it is different in different classes.

Summary

This slide summarizes all that we learnt so far in this lecture.

Virtual Functions & Cases

In this lecture you will understand:

* How virtual function mechanism is achieved using VTABLE and VPTR

Surprised?

Go through the code snippet given in the slide.

When the first program is executed the output would be 1 1. As there are no data members in **ex** the C++ compiler forces the object **e1** and **e2** to be of nonzero size (in our case 1 byte) because each object must have a distinct address. If you imagine indexing into an array of zero-sized objects, you'll be able to appreciate why the size has to be nonzero. Nonzero is fine, but why **1**? This is because the smallest nonzero positive integer is **1**.

Now, modify the class **ex** as shown in the second program given in the slide.

Here, we expect the same output. However, the output in this case would be 4 4. Surprised? The reason is simple. The size of the object **e1** is the size of a pointer. The compiler inserts this pointer (called VPTR) if you have one or more virtual functions in the class.

Slide Number 4

Having established the fact that the compiler silently adds a pointer to an object of a class which contains virtual functions let us see what this pointer points to and when is it set up. Consider the program given in the slide.

Here, shape is a base class containing two virtual functions. Class circle is derived from shape in which we have overridden both the virtual functions of base class. In **main( )** we have declared a pointer **p** and an object **q** of the base class **shape** and an object **c** of derived class **circle**. Then, first in **p** we have stored address of **q** and called **draw2( )** function. This would call **draw2( )** function of **shape**, as **p** contains address of object of **shape** class. Next, in **p** we have stored address of **c** and again called **draw2( )**. This time it would call **draw2( )** function of **circle** class. This is what is late binding. Let us discuss how this is achieved.

To accomplish late binding, the compiler creates a table called VTABLE (stands for virtual table) for each class that contains virtual functions. In this table, the compiler places addresses of all virtual functions in the order in which they are defined in the class. If the class is a derived class, then the compiler first places addresses of virtual functions defined in the base class and then adds the address of virtual functions defined in the derived class. If the derived class has redefined or overridden virtual functions of the base class then the compiler would override the addresses of the base class virtual functions with the derived class virtual functions in the VTABLE. Thus each class has its own VTABLE.

When objects of a class are created, each object in its first four bytes contains the address of the class's VTABLE. Suppose the address of the object is assigned to the pointer of the base class. Then using the pointer a function is called. When compiler sees such a call, it checks the type (class) of pointer and then checks whether the function exists in that class or not. If the function doesn't exist, the function is searched in the base class. If the compiler doesn't find such a function in the base class then an error is reported. But, if the function is found, then the compiler checks whether the function is defined as virtual or not. If the function is not defined as virtual then the compiler would directly link the call with the function definition. This linking/binding is known as early binding and at compile time it gets decided as to which function is going to get called when the program is executed.

Now if the function is defined as virtual, the compiler doesn't link it with any definition. Instead, it simply resolves it to VPTR + offset, where offset is the VTABLE's slot number that contains address of the function being called.

Now consider the statements

```
p = &q ;
p -> draw2( ) ;
```

The call to **draw2( )** has already been resolved into Address of VTABLE (VPTR) + 1, since address of **draw2( )** is in slot number 1 of the VTABLE. When this code is executed address of object **q** is found in the pointer. From the first four bytes of the object the value of VPTR would be extracted. Since this value would be address of VTABLE of shape class **draw2( )** of **shape** class would get called.

Let us now consider the statements

```
p = &c ;
p -> draw2( ) ;
```

This call to **draw2( )** has already been resolved into Address of VTABLE (VPTR) + 1. When this code is executed address of object **c** is found in the pointer. From the first four bytes of the object the value of VPTR would be extracted. Since this value would be address of VTABLE of **circle** class, **draw2( )** of **circle** class would get called.

Show Me The Cases

Let us now discuss the cases where it cannot be decided at compile time as to which function should get called.

Consider the program given in the slide that comes under **Case I**.

Here, if user enters value of **a** as 1 then we have passed address of **q**, an object of base class **shape**. As a result, **draw2( )** function of base class would get called. However, if the value entered for **a** is not 1 then we have passed address of derived class object to **fun( )**. This would cause **draw2( )** function of derived class to be called. Since at different times different class's **draw2( )** are getting called the compiler cannot decide whether it should bind the call to **draw2( )** of **shape** class or **draw2( )** of **circle** class. Hence it does late binding for this call.

Consider the program given in the slide that comes under **Case II**.

Here addresses of objects of base class and derived class are stored in an array of pointers to base class. In the **for** loop we have called **fun( )** function and passed to it the address stored in **p[ ]**. In each iteration of **for** loop the function **fun( )** would get called, and based on the contents (i.e. address of object) stored in **p**, the **draw2( )** function of base or of derived class would get called.

Different Cases

In this lecture you will understand:

* Study different Cases of which Function Gets called

Case I

In the next few slides we would put the virtual function mechanism through a lot of hoops to test your understanding of the mechanism. Consider following program (which is also given in the slide).

```
class base
{
    public :
        void h( ) { }
        virtual void f( ) { }
        virtual void g( ) { }
} ;

class der1 : public base
{
    public:
        void h( ) { }
        virtual void f( ) { }
} ;

class der2 : public der1
{
    public :
        virtual void x( ) { }
        virtual void g( ) { }
        virtual void f( ) { }
} ;

void main( )
{
    base b ;
    b.h( ) ;
    b.f( ) ;
    b.g( ) ;
}
```

Here, in **main( )** we have created an object b of the base class and called the functions through this object. As a result, the functions of base class get called. All functions whether they are defined as virtual or not get early bound because they are being called using an object. In other words, late binding comes into play when a virtual function is called through a pointer.

Considering same class declarations, observe second case of the program.

```
void main( )
{
    der1 d1 ;
    d1.h( ) ;
    d1.f( ) ;
    d1.g( ) ;
}
```

Here, in **main( )** we have created **d1** an object of derived class **der1**. Then we have called functions **h( )**, **f( )** and **g( )** through this object. The functions **h( )** and **f( )** of class **der1** get called. Since **der1** has not overridden function **g( )** the function of base class **base** gets called.

Now, consider one more case.

```
void main( )
{
    der2 d2 ;
    d2.h( ) ;
    d2.f( ) ;
    d2.g( ) ;
    d2.x( ) ;
}
```

Here, in **main( )** we have created **d2** an object of derived class **der2**. Then we have called functions **h( )**, **f( )** and **g( )** and **x( )** through this object. The function **h( )** of **der1** class gets called as **der2** has inherited it from **der1**. However, since **der2** has overridden functions **f( )** and **g( )** and added **x( )** as its own virtual function, the overridden versions of function **f( )** and **g( )** would get called and **x( )** of **der2** would get called.

What is important in each of the three cases discussed above is that the type of object through which the function has to be called is known at compile time. Hence the compiler can early bind all the calls made in this slide.

Case II

To understand the discussion that follows please take a look at the virtual tables shown in the slide. Once you know how they are created it would be much easier to follow the discussion.

Looking at the class declarations done in this slide let us discuss some cases.

```
void main( )
{
    base *b ;
    der1 d1 ;
    b = &d1 ;
    b -> h( ) ;
    b -> f( ) ;
    b -> g( ) ;
}
```

Here, in **main( )** we have declared a pointer to base class and created an object **d1** of class **der1**. Then in **b** we have stored address of **d1**. Thus **b** is an upcasted pointer. Next, through this upcasted pointer we have called functions **h( )**, **f( )** and **g( )**. Since **h( )** is a non-virtual function, the base class version gets called. Next, we have called **f( )**, which is a virtual function and is overridden in derived class **der1** hence the **der1** version of the function gets called. But, since **der1** has not overridden virtual function **g( )**, the base class version of **g( )** gets called as its address would be found in the VTABLE.

Consider one more variation of this program.

```
void main( )
{
    base *b ;
```

```
    der2 d2 ;
    b = &d2 ;
    b -> h( ) ;
    b -> f( ) ;
    b -> g( ) ;
    b -> x( ) ;
}
```

Here the first three calls are similar to the previous case. Let us consider the call to **x( )**. This code would generate an error. Here, through an upcasted pointer we have called the function **x( )**. To decide how to link this call the compiler checks the function's existence in the class **base**. Since **x( )** function is not present in the class **base**, the compiler would try to search it in the base class of **base**, which does not exist. Hence the error.

Case III

Let us discuss some more cases related to virtual functions. Consider following program:

```
void main( )
{
    der1 *d ;
    der2 d2 ;
    d = &d2 ;
    d -> h( ) ;
    d -> f( ) ;
    d -> g( ) ;
    d -> x( ) ;
}
```

This code too would generate an error. Let us see how?

Here, first we have declared a pointer **d** to the base class **der1** and created an object **d2** of class **der2**. Thus, **d** is an upcasted pointer. Then through this upcasted pointer we have called functions **h( )**, **f( )**, **g( )** and **x( )** respectively. The function **h( )** is a non-virtual function hence **h( )** of **der1** should get called. Next, **f( )** and **g( )** are virtual functions overridden in derived class **der2**, so functions of **der2** should get called. Then comes a call to **x( )** function, which does not belong to base class **der1** so we cannot call it using a base class pointer. Hence, an error would get generated.

Case IV

Considering same declarations of **base** and **der1** class as in the previous slide. But, this time class **der1** has not been derived from **base** class. Both are independent classes. Let us discuss one more case.

```
void main( )
{
    base *b ;
    der1 d1 ;
    b = ( base * ) &d1 ;
    b -> h( ) ;
    b -> f( ) ;
    b -> g( ) ;
}
```

Here even though **der1** is not derived from **base**, we have stored the address of **d1** object in the pointer **b** by forcibly typecasting address of **d1** to **base** *. The compiler early binds call to the **h( )**, since it is not defined as virtual. But when the compiler finds that **f( )** and **g( )** are defined virtual in base, it late binds these calls.

At runtime, address of **d1** is found in the pointer **b**. Hence **d1**'s first four bytes are accessed, which contain address of the **der1**'s VTABLE. The **der1**'s VTABLE contains address of the virtual function **f( )** defined in **der1**. (See the declaration of classes in Case I). Since **f( )** in base is first in the order, it would have offset 0. Hence, 0 would have been added by the compiler to the VPTR that would be found at execution time. The VPTR that has been found here is of **der1**'s VTABLE. Adding 0 this extracts address of **der1**'s **f( )**. Hence the call materializes to **der1**'s **f( )**.

But for **g( )** the offset 4 would have been added by the compiler. When this offset is added to **der1**'s VPTR, a garbage value is extracted and hence exception (runtime error) occurs.

If we want that the compiler should point out such errors perform the typecasting using **static_cast** as shown in the slide.

Output?

Consider the program given in the slide, where virtual function play a major role:

When the compiler compiles the program it attaches **this** -> before call to **f1( )** in the function **f2( )**. The **f1( )** function being virtual and is being called through a pointer (**this**) it would be late bound. Similarly, the statement **b -> f1( )** in **main( )** would also be late bound. But **b -> f2( )** would be early bound since **f2( )** is non-virtual.

When this program is executed the call to **b -> f1( )** would end up calling the **der::f1( )** since **b** contains address of **d**. The call to **b -> f2( )** would end up calling the **base::f2( )**. The **this** pointer in the **f2( )** function would now contain address of the object **d**. Hence the call to **this -> f1( )** would also end up calling the **der::f1( )**.

There are three uses of late binding:

(a) Runtime polymorphism

(b) Calling derived class functions from base class

(c) Keeping the interface (abstract class) separate than implementation (class that implements abstract class).

Virtual Destructor

In this lecture you will understand:

* What is virtual destructor

Slide Number 1

This slide shows how the data members would be visible in case of inheritance. Here, **base** is a class that contains **private**, **protected** and **public** data members. If **derived1** class is derived publicly from **base** class then **protected** members would remain **protected** and **public** members would be **public** for the derived class. Moreover, only the **public** members of **base** class would be accessible through an object of **derived1** class.

Similarly, if class **derived2** is derived **private**ly from **base** then **protected** members of **base** class would remain **protected** for derived2 but **public** members would become **private**. Moreover no members of the base class would be accessible through an object of **derived2** class.

Virtual Destructors

Consider program given in the slide.

Here **shape** class contains a pointer to an **int** as its data member. The memory for this member is allocated in the constructor and deallocated in the destructor. Furthermore, **circle** is a class derived from **shape** class. The data member **p** of **shape** class is used both in the constructor as well as destructor of **circle** class.

In **main( )** we have created **s** as a pointer to **shape** class and **c** as a pointer to **circle** class. We have allocated memory for circle class dynamically and then store it in **s**, an upcasted pointer. At the end we have called **delete** to de-allocate memory pointed to by **s**. Syntactically, the program is correct, however, the statement **delete s** ; would simply call the destructor of shape class, though s is containing address of object of circle class. This is dangerous. To avoid this we need to write the base class destructor as a virtual. On doing so, in case of example given in the slide, **delete s;** would now call **circle** class destructor and then it would call **shape** class destructor.

I/O File Class

In this lecture you will understand:

* Console I/O using functions of 'stdio.h'.
* Limitations of functions of 'stdio.h'
* The hierarchy of **ios** class

Input File Class

There are several assorted functions in C for carrying out I/O. Instead of using the assorted functions we can encapsulate these I/O functions in different classes. This would make it easier, safer and more efficient to perform input/output.

Consider the program given in the slide. Here, we have declared a class **ifile**. The class has a data member **fp** of type **FILE ***. In the one argument constructor we have opened the file sent by the user. We have collected the return value of **fopen( )** in **fp**. Since we are going to read from this file we have opened the file in **rb** (read binary) mode. We have provided a member function **read( )** through which we can read the file. The user has to pass the buffer and size of buffer to the **read( )** function. Lastly, we have defined a member function **close( )** to close the file.

We must **#include** 'stdio.h' file, which is required for the I/O functions.

Output File Class

Consider the class definition of **ofile** given in the slide.

In this class, again we have declared **fp** as **private** data member of type **FILE ***. In the constructor we have opened the specified file in the **wb** mode. Next, we have defined **write( )** function so that the user would be able to write data into the file. Again to the **write( )** function user has to pass the buffer containing the contents to be written in the file and its size. In the member function **close( )** we have closed the file.

Using File Classes

Let us now use the file classes **ifile** and **ofile** that we developed in the last two slides, in a program. We have declared a structure **emp**. We have created two objects **e1** and **e2** of this structure in **main( )**. Next, we have created an object of the **ofile** class. While creating the object we have passed name of the file, which we want to open. Once the file gets opened we have called the **write( )** function. We have passed to it the address of an object which is of type **emp** and size of the **emp** structure. We have then closed the file by calling the **close( )** function of **ofile** class.

To read the file we have created an object of the **ifile** class and called the **read( )** function. To the **read( )** function we have passed the pointer to an object of type **emp**. The structure elements would get filled with the contents read from the file. We have then closed the file and printed the values read from the file.

Limitations

The way we wrote **ifile** and **ofile** classes for file I/O we can write similar classes for console I/O and for reading/writing to a block of memory. However, there is a catch here. All these classes would use **printf( )** and **scanf( )** functions. This leads to some limitations:

(a) Remembering all the format specifiers in **printf( )/scanf( )** is not an easy job.

(b) **printf( )/scanf( )** do not carry out conversions logically. For example, 3.5 print using %d neither produces 3 nor 4.

(c) In case of a mismatch between the specifier and the type to be printed/scanned the **printf( )/scanf( )** functions do not report any warning.

(d) We cannot extend **printf( )** to accommodate new data types. Thus the primary goal of C++ -- the ability to add new data types with ease -- gets defeated.

(e) **printf( )/scanf( )** are variable-argument list functions. An interpreter is loaded for such functions at runtime. This increases overheads because if we print only a character still logic that prints out **long**, **double**, etc. gets printed. This leads to wastage of memory space.

Hierarchy

C++ offers a fresh approach for performing I/O. This approach is easy (no remembering of format specifiers), clean (no clutter in the program), safe (no unexpected manipulations of class elements), efficient (no overheads for doing small jobs) and adaptable (accommodate new datatypes easily). This approach uses a library called **iostream** library. This slide shows the organization of **iostream** library.

Since in this lecture we are going to discuss Console I/O, the slide shows only the Console I/O classes and objects.

As seen in the slide **ios** class is at the root of the **iostream** class library. This class includes features like flags for formatting the string data, the error status flags and the file operation mode.

The **ios** class has a pointer to the **streambuf** class as a data member. The **streambuf** class manages the buffer into which the data is read or written. It also contains functions to handle the buffered data.

Two classes **istream** and **ostream** are derived from the **ios** class. They are used for input and output respectively. **istream** class provides overloaded >> operator function that reads data from input stream. **ostream** class provides overloaded << operator that writes data to the output stream.

A class called **istream_withassign** is derived from the **istream** class. **ostream_withassign** class is derived from **ostream** class. We have been using **cin** and **cout**. They actually are stream objects. **cin** and **cout** are the objects of the **istream_withassign** and **ostream_withassign** classes respectively. That is why **cin** and **cout** objects are used for input and output respectively.

Stream is a general name given to the flow of data. Different streams are used to represent different kinds of data. For example, the standard output stream flows to the screen display, the standard input stream flows from the keyboard.

Formatting Flags & Manipulators

In this lecture you will understand:

* What are formatting flags
* How to use formatting flags in a program
* What are manipulators
* How to use manipulators in a program

Formatting Flags

The **ios** class contains formatting flags that help users to format the stream data. Formatting flags are a set of **enum** definitions. There are two types of formatting flags:

(1) On/Off flags

(2) Flags that work in group

The On/Off flags are turned on using the **setf( )** function and are turned off using the **unsetf( )** function. To set the on/off flags the one argument **setf( )** function is used. The flags working in groups are set through the two-argument **setf( )** function .

Using Formatting Flags

Let us discuss a program that shows how to use both types of formatting flags. Consider the program given in the slide.

Here, firstly, we have called the one-argument **setf( )** function with flags **ios::showbase** and **ios::uppercase**. These flags are ORed and passed to the **setf( )** function. If we display an octal or hexadecimal values the **ios::showbase** flag shows **0** before the octal value and **0x** before the hexadecimal value. The **ios::uppercase** flag ensures that the characters in octal or hexadecimal values appear in upper case. We have declared two integer variables and stored **32767** and **2000** respectively in them. We intend to print these values in hexadecimal form. For this, we have set the **ios::hex** flag by calling two-argument **setf( )** function as shown in the slide.

There is a flag for each numbering system (base) like decimal, octal and hexadecimal. Collectively, these flags are referred to as **basefield** and are specified by **ios::basefield** flag. If we set the **hex** flag as **setf (ios::hex)** then we will set the **hex** bit but we won't clear the **dec** bit resulting in undefined behavior. The solution is to call **setf( )** as **setf (ios::hex, ios::basefield)**. This call first clears all the bits and then sets the **hex** bit.

Next, we have displayed values of **i** and **j**. Since we have set the **hex** flag, values would get printed in hexadecimal. Again, since we have set the **showbase** and **uppercase** flags the output will be:

0x7FFF

0x7D0

If we had not set these flags the output would have been:

7fff

7d0

We have then reset the **dec** flag by calling the two-argument **setf( )** function and printed **i** again. This time 32767 gets printed in decimal. To unset the **showbase** and **uppercase** flags we have called the **unsetf( )** function. The **unsetf( )** function takes only one argument.

If we want that a string should get displayed in 40 columns we should specify the number of columns using a member function **width( )** of **ios** class. When we print the string using **cout** you will see the right justified string printed in the output window. If we want to left justify the string we can set the **left** flag using **setf( )**. This is shown in the next slide.

Slide Number 5

To left justify the string we have called **setf( )** function with **left** flag. **ios::adjustfield** is a collective name for the justification flags. If we want the constant width the **width( )** function should be used before each insertion or extraction statement. Hence, we have called **width( )** function again and

displayed the string. Now, you will see the string left justified. To set the default flag again we have called the **setf( )** function with **ios::right** flag.

Manipulators

Every time calling **ios** member functions to display formatted output becomes tedious. Manipulators provide a clean and easy way for formatted output. Using manipulators the formatting instructions are inserted directly into the stream. Also, we can create our own manipulators. **endl** is an example of manipulator.

Manipulators are of two types, those that take an argument and those that don't. The slide shows few examples of manipulators.

Using Manipulators

Consider the program given in the slide, which is the same program that we discussed earlier but makes use of manipulators for formatting. To set the **showbase** and **uppercase** flags we have called a manipulator **setiosflags( )**. **setiosflags( )** is a manipulator that takes one argument. Next, to display values of **i** and **j** in hexadecimal form, we have used **hex** flag directly in **cout** statement. To print the value of **i** in decimal form we have reset the number flag to **dec**.

To display a string in specified number of columns instead of calling the **width( )** function we have used a manipulator called **setw( )**. As stated earlier, by default string gets displayed right justified. To left justify it we have used the **setiosflags( )** manipulator with **left** flag and displayed the string again. To reset the default flags, we have used another manipulator **resetiosflags( )**.

Creating Manipulators

In this lecture you will understand:

* How to create no-argument manipulator
* How to create one-argument manipulator

Creating No Argument Manipulator

Though the list of manipulators provided by the **iostream** library is quite impressive, at times we may want to create our own manipulators. Let us see how to create a zero-argument manipulator.

In the program given in the slide, we have created a manipulator that places the cursor at the beginning of the same line. Let's understand this.

In **main( )** we have written **cout << ret ;** Since << is an overloaded operator internally this statement becomes **cout.operator << (ret)**. The << operator has been defined in 'iostream.h' as follows:

```
ostream& ostream :: operator << ( ostream& ( *_f ) ( ostream& ) )
{
    return ( *_f ) ( *this ) ;
}
```

In the call to the **operator <<( )** function, **ret( )** being a function what is being passed to a function is the pointer to the function. Hence, when we pass the address of **ret( )** to the **operator<<( )** overloaded function it collects the address of **ret( )** function in a pointer to a function. This pointer to a function receives an **ostream** reference and returns an **ostream** reference. Hence, we have defined **ret( )** with the prototype that receives the **ostream** reference and returns **ostream** reference. Now, when the control reaches in **operator<<( )** function a call is made to our **ret( )** function with ***this** as the parameter through the statement **(*_f) (*this)**. Here, ***this** represents **cout** object because the operator function is called through the **cout** object. We have collected this object in our definition of **ret( )** function and used this object to insert carriage return in the output stream through the statement **o << '\r' ;**

Creating One-argument Manipulator

We saw how to create a zero-argument manipulator. However, creation of manipulators with argument is pretty convoluted. Here we have created a manipulator called **string**, which receives an **int** as an argument and outputs its string equivalent. To implement the string manipulator we have defined a class called **string**. This class consists of three data members, **str** to hold the final string. **arr** to hold the string equivalents of digits. **num** to hold the number to be converted to string. The class also consists of a constructor and an overloaded **operator<<( )** function. This function is implemented as a **friend** function. It is because this function is intended to work on objects of two different types.

Slide Number 5

Here, in **main( )** we have used the **string** manipulator in the statement **cout << string (105)**.

string (105) is a call to the one-argument constructor. This creates a nameless object. Now the statement will become **cout <<** the nameless object of type **string**. Since no overloaded operator that takes **string** as an argument is defined in **ostream** class our **operator <<( )** function gets called. Within the **operator << ( )** function we have separated the digits of the specified number. The separated digit is used as an index in the array to retrieve the string equivalent of it. All the strings are concatenated in the data member **str**. Next, the **str** is displayed through **o**. Lastly, **o** is returned back to **main( )**. This is necessary because there can be a cascaded **cout** statement like **cout << string (105) << endl ;**

Buffered & File I/O

In this lecture you will understand:

* What is buffered I/O
* The **iostream** library
* File I/O using stream
* The modes available for opening a file
* What are error flags

Buffered I/O

Suppose we wish to read the contents of some file, say CH1PR1.CPP and display them on the screen. The approach that we may follow to do this could be as follows:

We can read a character, display that character on the screen, again read a character again display it on the screen. This can be continued till we do not reach the end of file. This seems to be logical, but is pretty inefficient, as it would involve lot of disk reads. Moreover, every time we intend to read the disk has to rotate, the character to be read must come below the read/write head, such that the head can advance and carry out the reading. A lot of time would be spent in doing this for every character. Instead, it would make more sense to read the entire file, store it at some place in memory called buffer and then read the characters from the buffer rather than from the disk. This would be more efficient.

Similarly, while writing characters to the disk there is no point in doing a write operation for every character. Instead, it would be more efficient to first store all the characters in the buffer and then write this buffer to the disk.

This type of input/output is known as buffered I/O.

Hierarchy

The **iostream** library provides classes for performing console I/O as well as file I/O. We have already seen the Console I/O classes in the last lecture. This slide shows classes that are used for file I/O. As seen earlier, **ios** is the root class of **iostream** library. In addition to the classes **istream** and **ostream** another class called **fstreambase** is also derived from the **ios** class. The **fstreambase** class contains an object of the **filebuf** class, which manages file-oriented buffer. The **filebuf** class is derived from **streambuf** class, which we have already seen.

A class **iostream** is derived from both the **istream** and **ostream** classes by multiple inheritance. So, the object of **iostream** class can be used for both input and output.

To enable users to work on disk files the **iostream** library provides a set of three classes viz. **ifstream**, **ofstream** and **fstream**. The **ifstream** class is for input, **ofstream** for output and **fstream** for both input and output. The **ifstream** class is derived by multiple inheritance from **istream** and **fstreambuf** classes. The **ofstream** class is derived by multiple inheritance from **ostream** and **fstreambuf** classes. The **fstream** class is derived by multiple inheritance from **iostream** and **fstreambuf** classes. The **ifstream**, **ofstream** and **fstream** classes are declared in 'fstream.h' file.

File I/O

File I/O can be done in two modes - text and binary. Text I/O is further divided into formatted I/O and unformatted I/O. In the formatted I/O, as the name suggests, we can use various formatting flags and manipulators to format the output. In short, we can decide how the output should look like. In unformatted I/O we can perform input/output character by character. The output cannot be formatted. The major difference between text I/O and binary I/O is that in text I/O numbers are also stored as series of characters, whereas, in binary I/O numbers are stored as bytes. For example, in text I/O 435 is stored as three characters 4, 3 and 5, whereas, in binary mode it is stored as a 2-byte integer.

Formatted File I/O

In formatted file I/O numbers are stored on a disk as a series of characters. Thus, 12.35 will get stored as characters '1', '2', etc. and so instead of occupying 4 bytes as a **float** it will occupy 5 bytes in file. The program given in the slide demonstrates how to perform formatted file I/O to write an **int**, **float**, **char** and a string into the file and read them back.

Here, we have defined an object **o** of class **ofstream**. We have initialised this object with file 'data.dat'. When we pass a filename to the **ofstream** constructor this file gets opened. If the file does not exist it gets created. If the file exists, it gets overwritten and the old data is replaced by the new data. We have seen in the previous slide that **ofstream** class is derived from the **ostream** class. So, we can use << operator to write data in the file using the object of **ofstream** class. Since the numbers are stored as characters and not as a fixed-length field we have to specify explicitly where the numeric data ends, otherwise, when we read the data from file, the extraction operator would not know where the one number ends and another begins. For this, it is necessary to give a space as a delimiter between the numeric data. For the same reason, two strings must be separated with a space. This restricts the use of embedded spaces between a string.

After writing data in the file we have closed it using the **close( )** member function. We have then opened the same file for reading. For this, we have defined an object of **ifstream** class and passed to the constructor the name of the file. Since, **ifstream** class is derived from **istream** class we can use >> operator to read the data from a file. The data read is collected in the variables and displayed using **cout**.

When we include the 'fstream.h' file we don't need to include 'iostream.h' file because 'iostream.h' is included in 'fstream.h' file.

Strings With Embedded Blanks

In the program given in the slide, the user can input as many strings as he wants from keyboard. These strings are saved in a file 'strings.dat' and later on displayed on the screen.

We have defined an object of **ofstream** class and passed to it the name of the file. We have declared a **char** array and a **char** variable. We have run a loop, which would run till the variable **ans** contains 'y', i.e, till the user wants to enter more strings. To get the multiword string we have called a function **getline( ),** which is a member function of **istream** class. We have concatenated a newline character at the end of each string. This is necessary because **getline( )** function reads characters until it encounters '\n' and we are going to read the strings from file using **getline( )** function. So, we must mark the end of line by '\n'. We have written the strings in the file using << operator. We have then prompted the user to enter 'y' if he wants to enter more strings otherwise 'n'. We have obtained user's input in **ans** and then called an **istream** function **ignore( )** to skip the Enter character. Once control comes out of the loop we have closed the file.

Slide Number 6

Next, we have opened the file by creating an object of **ifstream** class. We have read the strings from the file and displayed them on the screen. We have read strings in a loop, which runs until end of file is reached. We have checked for the end of file using **eof( )** function of **ios** class. We have used **getline( )** function to read strings from file. When the end of file would be encountered **getline( )** function would read the end of file character. This would set the **ios::eofbit** error status flag. The **eof( )** function would read this flag and would return non-zero value. Thus, the loop would get terminated. The strings are displayed using **cout**. At the end we have closed the file.

Opening Modes

In the previous programs we had not specified file-opening modes while creating the **ifstream** or **ofstream** objects but still the files got opened in appropriate modes. This is because both **ifstream** and **ofstream** classes use default opening modes, which are set as **ios::out** in **ofstream** and **ios::in** in **ifstream**. This slide shows various file opening modes available in the **iostream** library. Since these modes are defined as **enum**s in the **ios** class we must access them using **ios::**. By default the file gets opened in text mode. To open the file in binary mode we must specify **ios::binary** while creating the stream objects.

Error Flags

We have seen three main features of the **ios** class viz. formatted flags, opening modes and error status flags. We have discussed formatted flags and opening modes. Let us now see what are the error flags provided by **ios**.

The error status flags are the enumerated data and are provided to report errors that occurs in input and output operations like opening file, reading from file, writing to file reaching end of file, etc.

For every error flag there is a corresponding function, which we can call and check if the error flag is set. The error flags, their meanings, corresponding functions and the purpose of functions are shown in this slide.

Character & Binary I/O

In this lecture you will understand:

* How character I/O is performed
* How to perform binary I/O
* How to position a file pointer
* How to write data to printer
* C++ names for hardware devices

Character I/O

So far we have used **ofstream** class to write the data in the file and **ifstream** class to read it back from the file. Instead of using two separate classes we can perform input and output by using only one class i.e. **fstream** class. Since **fstream** class is derived from **iostream** class it can perform both input and output operations. The program given in the slide, shows how to input and output contents character by character.

We have defined an object **f** of the **fstream** class and passed to it the file name. In **fstream** constructor we have to specify modes because there is no default mode defined in this constructor. Hence, we have specified the opening modes as **in** and **out** so that we can write in as well as read from the file. We have then written a string character by character in the file by calling the **put( )** member function of **ostream** class.

Next, to read the characters from first character onwards, we must place the file pointer at the beginning of the file. For this, we have used **seekg( )** function, which is a member function of the **istream** class. We would see the file pointer positioning functions in detail in the slides to follow. To read the characters we have used the **get( )** function of **istream** class. We have read the characters until end of file is encountered. We have then displayed the characters read from the file.

Note that we have not closed the file. This is not necessary because if we don't close the file, destructors of the **ifstream**, **ofstream** and **fstream** classes close the file for us. So, as soon as object of **fstream** class would go out of scope its destructor would get called and the file would get closed.

Binary I/O

We have seen a program to write an **int**, **float** and a **char** in a file in the text mode. We have also seen the limitations of writing numeric fields in text mode. In this mode each digit in a number is stored as a character resulting in occupying more memory than necessary. This becomes inefficient at the time of storing records that contain several numeric fields. The solution is to use binary I/O. In binary I/O the data is stored in a file in the way it is stored in RAM. So, writing an integer in a file in binary mode would occupy only 2 bytes irrespective of number of digits of the number.

The program given in the slide writes a record in a file and reads it back. We have defined an array of structure **book**. We have also defined an object of **ofstream** class to open the file 'data.dat' in binary mode. We have written records in this file using a **ofstream** member function **write( )**. Then we have passed the buffer and its size to the **write( )** function. This function merely transfers the buffer passed to it from memory to the disk file without taking care about how the data is formatted. After writing records we have closed the file.

Slide Number 3

To read records from the file we have defined an **ifstream** class object. Since we have written data in binary mode we must read it in binary mode. This time to read records from file we have used the **read( )** member function of the **ifstream** class. After reading the record we have displayed it using **cout**. We have repeated this until end of file is reached.

Note: In formatted text mode, we should use << operator to write data and >> operator to read data. In unformatted text mode, we should use **put( )** and **get( )** functions to read and write data respectively. In binary mode, we should use **read( )** and **write( )** functions.

Positioning File Pointer

If we are working with a file we may be required to go back to the beginning of the file and write something there, or we may need to modify the existing contents. In such a situation we need to move

a pointer called 'file pointer' at appropriate position. In the program given in the slide, we would see how to move file pointer in the file.

We have defined an array of structure **book**. We have defined an object of **ofstream** class and written the records in the 'bnames.dat' file using the **write( )** member function of the **ofstream** class.

Slide Number 5

When we write in a file the file pointer associated with that file also moves. So, after writing a few records in a file, the pointer would be pointing at the end of the last record. Now, if we write some more record, it will get written at the end of this last record. Here, we have created one more object **b1** of the **book** structure and written that object in the file. Obviously, it would get added after the last record. However, if we want to delete the previously added record and write new record in its place we must position the pointer at the beginning of that record. To position the pointer we have used the **ostream** member function **seekp( )**.

The **seekp( )** function is overloaded to take either one or two parameters. If we use one argument **seekp( )**, we can mention the position in bytes relative to the beginning of the file. In the two argument version we can specify the position in bytes relative to the end of file, beginning of the file and current position. For this, we can use **end**, **beg** and **cur** flags respectively. Here, we have specified that the pointer should move two records back from the end. Then we have written another record in the file. This record will overwrite the previous record. We have then closed the file.

Similarly, we can get the current position of the file pointer by using **tellp( )** member function of **ostream** class. The **tellp( )** function returns position in bytes.

To read the file we have again opened the file, this time with the object of **ifstream** class. We have positioned the pointer before the second last record and read the record by calling the **read( )** function. We have then displayed the record read from the file.

Slide Number 6

Here we have first moved the file pointer at the end of the file. This time we have used the **istream** class member function **seekg( )** to move the pointer. Now, if we obtain the pointer position we can get the total bytes written in the file. We have used **tellg( )** member function of **istream** class to get the same. We have then calculated the number of records contained in the file by dividing total bytes by the size of one record.

Now, if we have to print all the records of the file we must position the file pointer at the beginning of the file. We have done so using the statement **i.seekg (0)**. We have displayed all the records and then closed the file.

Remember that we cannot call **seekg( )** and **tellg( )** functions using **ofstream** object and **seekp( )** and **tellp( )** functions using **ifstream** object. However, if we use **fstream** for file I/O we can use any of them.

Printer Output

The **iostream** library has made it easy to read data from various input devices and write data to the output devices. The program given in the slide shows, how to print contents of disk file, 'data.dat' on the printer.

The file must be present on the disk so that we can open it for reading. If the file does not exist then rest of the code should not get executed. Hence, in the program we have checked whether the file is opened or not using the **fail( )** function, where **fail( )** is an error flag.

Operating system has given familiar names to the hardware devices. We can use these names as file names for accessing these devices. The various names are discussed in the next slide. The printer is

generally connected to the first parallel port. So, the file name for the printer should be **PRN** or **lpt1**. In the program, we have defined the object of **ofstream** class. We have called a member function **open( )** of class **ofstream** which opens the specified port. We have specified **PRN** to open the port on which printer is connected. Since we have called **open( )** using object of **ofstream** class the port would get opened for writing. Next, using the **while** loop we have read the contents of file character by character and sent the same to the printer. At the end we have put a character '\x0C'. This character causes the page to eject from the printer.

Hardware Device Names

This slide shows the various device names, which we can use as file names in the **ifstream** or **ofstream** classes.

Error Handling

In this lecture you will understand:

* What is exception handling
* How to handle runtime errors
* How to catch exception

Types Of Errors

Exceptions are errors that occur at run time. The reasons why exceptions occur are numerous. Some of the more common ones are:

(a) Falling short of memory

(b) Inability to open a file

(c) Exceeding the bounds of an array

(d) Attempting to initialize an object to an impossible value

When such exceptions occur, the programmer has to decide a strategy according to which he would handle the exceptions. The strategies could be, displaying the error messages on the screen, or displaying a dialog box in case of a GUI environment, or requesting the user to supply proper data or simply terminating the program execution.

Handling Runtime Errors

Usually C programmers deal with exceptions in two ways:

(a) Following the function calls with error checks on return values to find whether the function did its job properly or not.

(b) Using the **setjmp** and **longjmp** mechanism. This approach is intended to intercept and handle conditions that do not require immediate program termination. For example, if a recursive descent parser detects an error, it should report it and continue with further processing.

Let us look at these methods more closely.

Checking Function Return Value

In C programs a function usually returns an error value if an error occurs during execution of that function. For example, file-opening functions return a NULL indicating their inability to open a file successfully. Hence, each time we call these functions we can check for the return value. This is shown for some fictitious functions **fun1( )**, **fun2( )** and **fun3( )** in the slide.

There are three problems with this approach:

(a) Every time we call a function we must check its return value through a pair of **if** and **else**. Easier said than done! Surrounding every function call with a pair of **if** and **else** results in increase in code size. Also, too many **if-else**s make the listing lose its readability.

(b) This approach cannot be used to report errors in the constructor of a class as the constructor cannot return a value.

(c) It becomes difficult to monitor the return values in case of deeply nested function calls. Especially so if the functions belong to a third-party library.

Catching Exception

C++ provides a systematic, object-oriented approach to handling run-time errors generated by C++ classes. The exception mechanism of C++ uses three new keywords: **throw**, **catch**, and **try**. Also, we need to create a new kind of entity called an **exception class**.

Suppose we have an application that works with objects of a certain class. If during the course of execution of a member function of this class an error occurs, then this member function informs the application that an error has occurred. This process of informing is called **throwing** an exception. In the application we have to create a separate section of code to tackle the error. This section of code is

called an **exception handler** or a **catch block**. Any code in the application that uses objects of the class is enclosed in a **try block**. Errors generated in the **try** block are caught in the **catch** block. Code that doesn't interact with the class need not be in a **try** block. The program given in the slide shows how to throw and catch exception:Here **math** is a class containing a function **fun( )** in which runtime errors might occur. An exception class called **excep** has been specified. In **main( )** we have enclosed the part of the program that calls the **fun( )** function. If the condition **d == 0** gets satisfied then function **fun( )** throws an exception, using the keyword **throw** followed by the exception calls object. When an exception is thrown control goes to the **catch** block that immediately follows the **try** block. Here the value of the object **e** would be passed, which would get collected in **p**.

How The Whole Thing Works

Let's summarize the events that take place when an exception occurs:

(a) Code is executing normally outside a **try** block.

(b) Control enters the **try** block.

(c) A statement in the **try** block causes an error in a member function.

(d) The member function throws an exception.

(e) Control transfers to the exception handler (**catch** block) following the **try** block.

Note that there can be multiple calls to different functions in the single **try** block. If each function throws the same exception then only one **catch** block would be required after the **try** block. You can appreciate how clean the code would be. Just about any statement in the **try** block can cause an exception, but we don't need to worry about checking a return value for each one. The **try-throw-catch** arrangement handles it all for us, automatically.

If different calls throw different types of exceptions then there can be multiple **catch** blocks after one **try** block to handle those exceptions.

Exception Handling Tips

The slide shows various points about the exception handling.

Constructors in MI

In this lecture you will understand:

* How constructors in Multiple Inheritance are defined
* The need of **virtual** base classes
* **virtual** destructors

Constructors In MI

Let us see how constructors are handled in multiple inheritance. Consider the program given in the slide.

Here, the class **d1** is derived from two classes **b1** and **b2**. In class **d1**, the constructors of the base classes are called in the order in which the classes have been inherited. In this program the classes have been inherited in the following order:

class d1 : public b1, public b2

Note that the destructors in case of multiple inheritances are called in exactly the reverse order of the constructors.

It doesn't matter the order in which constructors are called:

d1 ( ) : b2( ), b1( )

Why is the order of calling constructors governed by the order in the class declaration? This is because if you change the order of constructor calls while defining the constructor, you may have two different call sequences in two different constructors, but the poor destructor wouldn't know how to properly reverse the order of calls for destruction.

Conflict?

Now consider the program given in the slide. Here, time and date are two different classes each having function **show( )** with same prototype. The class **scheduler** is derived publicly from class **time** and **date** respectively. Through the **display( )** function of scheduler class we have called **show( )** function twice. Then in **main( )** we have created an object of **scheduler** class and called **display( )** and **show( )** functions through this object.

On compilation the compiler reports an error since **scheduler** inherits two copies of **show( )**, one via **time** and another via **date**. Hence when we attempt to call **show( )** at two places one in **scheduler::display( )** and second in **main( )** as **x.show( )**, the compiler would not know whether we intend to call the copy of **time** or that of **date**.

What's Wrong?

The program given in the slide shows one more situation that causes a conflict.

Here, **b** is a base class. **d1** and **d2** are two classes each derived from base class **b**. Furthermore, **der** is a class derived from two classes **d1** and **d2** respectively.

On compilation the compiler reports an error. The class **der** inherits two copies of **data**, one via **d1** and another via **d2**. This causes a conflict, since in **der::showdata( )** we are trying to display value of **data**. This is an ambiguous situation for the compiler, hence it flashes an error.

In Memory View

This slide gives the in-memory view of the classes discussed in earlier slide.

Here, the class **b** has a data member **data**. Since the two classes **d1** and **d2** are derived from the class **b** these two classes have inherited **data** of the class **b**. Then the class **der** derived from the classes **d1** and **d2**, too has inherited **data** from both **d1** and **d2**. So, when the object of **der** will be created it will have two copies of **data** having different addresses.

The solution for such ambiguous situation is to make **d1** and **d2** as virtual base classes. In the next slide we will see how to use the virtual base class.

Virtual Base Class

Consider the program given in the slide.

Here, we have made **d1** and **d2** as virtual base classes by adding a keyword **virtual**. Making the base classes virtual cause them to share single copy of the data members inherited from the base class. It means that **d1** and **d2** would have only one copy of data members of class **b** and so if we now access **b**'s data members in **der** there will be no ambiguity.

Virtual Destructors

While destroying an object the current destructor always knows that the base-class members are alive and active. It should not so happen that the base class members get destroyed through the base class destructor and then the derived class destructor tries to access them. This can be ensured by first calling the derived class destructor followed by the base class destructor. Thus, the destructor can perform its own cleanup, then call the base class destructor, which will perform its own cleanup. This it can do because it knows what it is derived from, but not what is derived from it.

Consider a situation where a derived class object is created using **new**. The address of this object can be assigned to a pointer to a base class object. Now if we delete the pointer, since the pointer is a base class pointer this would result in a call to the base class destructor. Ideally, firstly the derived class destructor should be called followed by the base class destructor. This can be ensured by using a **virtual destructor** in the base class. The program given in the slide shows how this can be implemented.

Inline Functions

In this lecture you will understand:

* **inline** Functions
* Usage of **const** qualifier
* The **const** functions
* When to use **mutable** keyword

---

inline Functions

One of the important advantages of using functions is that they help us save memory space. As all the calls to the function cause the same code to be executed, the function body need not be duplicated in memory.

Imagine a situation where a small function is getting called several times in a program. As you must be aware, there are certain overheads involved while calling a function. Time has to be spent on passing values, passing control, returning value and returning control. In such situations to save the execution time you may instruct the C++ compiler to put the code in the function body directly inside the code in the calling program. That is, at each place where there's a function call in the source file, the actual code from the function would be inserted, instead of a jump to the function. Such functions are called inline functions. The in-line nature of the individual copy of the function eliminates the function-calling overhead of a traditional function. The program given in the slide shows inline function at work.

Note that the function must be declared to be inline before calling it. On compilation the contents of the **reporterror( )** function would get inserted at two places within our program. These obviously are the places where **reporterror( )** is being called.

What Is p?

The keyword const (for constant), if present, precedes the data type of a variable. It specifies that the value of the variable will not change throughout the program. Any attempt to alter the value of the variable defined with this qualifier will result into an error message from the compiler.

The const qualifier ensures that your program does not inadvertently alter a variable that you intended to be a constant. It also reminds anybody reading the program listing that the variable is not intended to change. Variables with this qualifier are often named in all uppercase, as a reminder that they are constants.

The pointers when declared as const, ensures that the program does not inadvertently alter the address in pointer variable that you intended to be a constant. Consider the code snippet given in the slide.

```
char *p = "Hello" ; // string is fixed pointer is not
*p = 'M' ; // error
p = "Bye" ; // works
```

Here, p is a character pointer holding base address of string Hello. The string is constant here, but not the pointer p. Hence the second statement here would cause error, but the third statement would work as we can make p (here), to point to some other string.

```
const char *q = "Hello" ;  // string is fixed pointer is not
*q = 'M' ;  // error
q = "Bye" ; // works
```

Here too, the string Hello is fixed and the pointer q is variable. Hence trying to change the string pointed to by q would cause error. However, q can be made to point to some other string.

```
char const *s = "Hello" ;  // string is fixed pointer is not
*s = 'M' ;  // error
s = "Bye" ;  // works
```

This is similar to above two cases.

```
char * const t = "Hello" ;  // pointer is fixed string is fixed
*t = 'M' ;  // error
t = "Bye" ;  // error
```

Here, both, the char pointer t and the string to which it is pointing to are fixed. This means neither we can change the contents of string nor we can make t to point to some other string. The same can be achieved as shown below:

```
const char * const u = "Hello" ;  // string is fixed so is pointer
*u = 'M' ; // error
u = "Bye" ; // error
```

Const Functions

A **const** member function guarantees that it will never modify any of its class's member data. The program given in the slide shows this.

Here, function **modifydata( )** has been declared as **const** function. Hence it cannot modify data. If it tries to, a compiler error 'Cannot modify a const object' results.

Note that to make **modifydata( )** constant the keyword **const** is placed after the declarator but before the function body. If the function is declared inside the class but defined outside it then it is necessary to use **const** in declaration as well as definition. Member functions that do nothing but acquire data from an object are obvious candidates for being made **const**. In our program one such function was **showdata( )**, which too could have been made a constant function.

mutable Keyword

When we create a **const** object none of its data members can change. In a rare situation, however, you may want some data member should be allowed to change despite the object being **const**. This can be achieved by the **mutable** keyword as shown in the program given in the slide.

When the car is sold its owner would change, rest of its attributes would remain same. Since the object **c1** is declared as **const** none of its data members can change. An exception is however made in case of owner since its declaration is preceded by the keyword **mutable**. The change is brought about through the function **changer( )**. Trying to change other members would result in an error as the data members have not been declared as **mutable** and hence cannot be changed. Had **c1** been a non-const object we would have been allowed to change the owner as well as the model.

Data Conversion

In this lecture you will understand:

* How to carry out various data conversions

Data Conversion

We are already aware that the = operator assigns a value from one variable to another in statements like

```
int a, b ;
a = b ;
```

We have also used the = operator in context of user-defined data types. Here, = assigns the value of one user-defined object to another, provided they are of the same type, in statements like

```
matrix3 = matrix1 + matrix2 ;
```

Thus assignments between types, whether they are basic or user-defined, are handled by the compiler with no effort (i.e. implicit) on our part, provided that the same data type is used on both sides of the assignment operator. At times we may want to force the compiler to convert one type of data to another. This can be achieved by using typecasting. Typecasting provides an explicit conversion.

Data Conversion

The program given in the slide shows how to convert between a basic type and a user-defined type and vice versa. In this program the user-defined type is a string class and the basic type is int. The program shows conversion from string to int and from int to string.

Here, to convert an int to a user-defined type string we have used a constructor with one argument. It is called when an object of type string is created with a single argument. The function assumes that this argument represents an int, which it converts to a string and assigns it to str using the itoa( ) function. Thus the conversion from int to string is carried out when we create an object in the statement

```
string s3 = 789 ;
```

A similar conversion is carried out when the statement

```
s1 = 150 ;
```

is executed. Here we are converting an int to a string, but we are not creating a new object. The one argument constructor is called even in this case. When the compiler comes across a statement that needs a conversion, it looks for any tool that can carry out this work for it. In our program it finds a constructor that converts an int to a string, so it uses it in the assignment statement by first creating an unnamed temporary object with it's str holding the value corresponding to the integer 150 and then assigns this object to s1. Thus if the compiler doesn't find an overloaded = operator it looks for a constructor to do the same job.

To convert a string to an int the overloaded cast operator is used. This is often called a conversion function. This operator takes the value of the string object of which it is a member, converts this value to an int value and then returns this int value. This operator gets called in two cases:

```
i = int ( s2 ) ;
```

and

```
i = s3 ;
```

In the second assignment the compiler first searches for an overloaded assignment operator. Since this search fails the compiler uses the conversion function to do the job of conversion.

One might think that it would not be a sound programming practice to routinely convert from one type to another. However, the flexibility provided by allowing conversions often outweighs the dangers of making mistakes by allowing mixing of data types.

Data Conversion

Let us now see how do we go about converting data between objects of different user-defined classes? The same two methods used for conversion between basic types and user-defined types apply to conversions between two user-defined types. That is, we can use a one-argument constructor, or we can use a conversion function. The choice depends on where we want to put the conversion routine: in the class declaration of the source object or of the destination object.

When the conversion routine is in the source class, it is commonly implemented as a conversion function as shown in the program given in the slide. The two classes used in the program are **date** and **dmy**. Both classes are built to handle dates, the difference being the date **class** handles it as a string, whereas the **dmy** class handles it as three integers representing day, month and year.

Suppose **d1** is an object of the type **date**, which is not initialized, and **d2** of the type **dmy**, which has been initialised. Next an assignment is carried out through the statement **d1** = **d2**.

Since **d1** and **d2** are objects of different classes, the assignment involves a conversion, and as we specified, in this program the conversion function **date( )** is a member of the **dmy** class. This function transforms the object of which it is a member to a **date** object, and returns this object, which **main( )** then gets assigned to **d1**.

Data Conversion

Let's now see how the same conversion is carried out when the conversion routine is present in the destination class. In such cases usually a one-argument constructor is used. However, things are complicated by the fact that the constructor in the destination class must be able to access the data in the source class to perform the conversion. That is, since the data **day**, **mth** and **yr** in the **dmy** class is **private** we must provide special functions like **getday( )**, **getmth( )** and **getyr( )** to allow direct access to it. The program in the slide implements this.

When we execute the statement **d1** = **d2** the one-argument constructor in the **date** class (whose argument is a **dmy** object) gets called. This constructor function gets the access to the data of **d2** by calling the **getday( )**, **getmth( )** and **getyr( )** functions. Finally it converts this data into a string. The output of this program is similar to the earlier one. The difference is behind the scenes. Here a constructor in the destination object, rather than a conversion function in the source object, handles the conversion.

That brings us to the important question: when should we use a one-argument constructor in the destination class, and when should we use a conversion function in the source class? Often this choice is simple. If you have a library of classes, you may not have access to its source code. If you use an object of such a class as the source in a conversion, then you'll have access only to the destination class, and you'll need to use a one-argument constructor. Or, if the library class object is the destination, then you must use a conversion function in the source. What if we use a conversion function as well as a one-argument constructor? The compiler would of course flash an error since this becomes an ambiguous situation.

Object Slicing

In this lecture you will understand:

* What is object slicing
* What is Run Time Type Identification (RTTI)
* How to get RTTI information
* The usage of **reinterpret_cast**, **const_cast**, **static_cast**, etc.

Object Slicing

The virtual functions ensure that the code that manipulates objects of a base type can without change manipulate derived-type objects as well. Virtual functions should however always be called using either a pointer or a reference. If we try to do so using an object a phenomenon called **object slicing** takes place. The program given in the slide throws more light on this effect.

When we use an object instead of a pointer or reference as the recipient of the upcast, the object is sliced until all that remains is the subobject that corresponds to the recipient. That is, if an object of a derived class is assigned to a base class object, the compiler accepts it, but it copies only the base portion of the object. It slices off the derived portion of the object. Hence when we make the call **b.fun( )** only the member function in the base class gets called.

Object slicing actually removes part of the object rather than simply changing the meaning of an address as when using a pointer or reference. Because of this, upcasting into an object is not often done: in fact, it's usually something to watch out for and prevent. You can explicitly prevent object slicing by putting pure virtual functions in the base class: this will cause a compile-time error when we try to create a base class object.

RTTI

RTTI stands for Run Time Type Identification. In an inheritance hierarchy, using RTTI we can find the exact type of the object using a pointer or reference to the base class. The idea behind virtual functions is to upcast the derived class object's address into a pointer to a base class object and then let the virtual function mechanism implement the correct behavior for that type. Does this mean that an attempt to know the type of the derived class object from the base class pointer (RTTI) a step backward? No. At times it is useful to know the exact type of the object from the base class pointer. You may require this information to perform some specific operation more efficiently.

There is also a practical reason for providing RTTI as a language feature. Most class libraries were using RTTI of some form internally. So if RTTI is made a language feature you would have a consistent syntax for each library and would not be required to worry whether it is built into a new library that you intend to use.

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

(a) Using **typeid( )** operator

(b) Using the **dynamic_cast** operator

Let us explore them one by one.

The **typeid( )** operator takes an object, a reference or a pointer and returns a reference to a global const object of the type **typeinfo**. The code snippet given in the slide uses it.

We have a base class called **base**. We have derived two classes from it: **der1** and **der2**. Then we have created two pointers to the base class objects: **p1** and **p2**. The return value of **typeid( )** can be compared using == as shown in the slide. If you execute this code you would get the output as 'of same type' for both the comparisons.

RTTI should be used only with polymorphic classes i.e. those, which have a virtual function in the base class. In absence of polymorphism the static type information is used. Also runtime type identification doesn't work with **void** pointers, because a **void *** truly means no type information at all.

RTTI

Another way to obtain type information at runtime is by using the dynamic_cast operator. The code snippet given in the slide shows its usage.

The **dynamic_cast** operator attempts to convert the pointer **p1**, which can contain either the address of a **der1** object or the address of **der2** object or the address of the **base** object. If the result is non-zero then **p1** was indeed pointing at **base**. If the result is zero it means it pointed to something else.

Although we have used the **dynamic_cast** and **typeid** with pointers they work equally well with references.

reinterpret cast

This casting mechanism is the least safe and more often than not a source of bugs. If for some unusual reason you need to assign one kind of pointer type to another, you can use **reinterpret_cast**.

The **reinterpret_cast** can also be used to convert pointers to integers or vice versa as shown in the slide.

The use of **reinterpret_cast** is not recommended, but sometimes it's the only way out. Whenever you feel the need to use the explicit type conversion you should take time to reconsider it. You would find that in many situations it could be completely avoided. In others it can be localized to a few routines within the program. Always remember that whenever you are using a cast you are breaking the type system. And that is fraught with dangers.

const cast

The **const_cast** permits us to convert a **const** to a non-const as shown in the slide.

If we are to assign address of a **const** object (**a** in our program) to a pointer to a non-**const** (**ptr**) we should use the **const_cast**.

Another place where we can use a **const_cast** is when we wish to change a class member inside a **const** member function. However, this is a workable way. A better way is to define data as mutable. This way in the class definition itself it would be clear the data member might change in a **const** member function.

Static Cast

A **static_cast** is used for conversions that are well-defined. These include:

(a) castless conversions

(b) narrowing conversions

(c) conversions from void *

These are shown in the slide.

The advantage of using **static_cast<>** is that we can easily search for all the type casting done in the program This is useful while debugging when something has gone wrong.

Example

Microsoft COM (Component Object Model) is a specification to build language independent components. The slide shows a COM component consisting of a class with **Add( )** function. This class implements an interface containing pure virtual **Add( )** function. Each COM component provides code known as class factory that is responsible for creating component class objects. From the client program we need to call a function in the COM library, which calls the function in the class factory of the COM component and returns the address of the newly created COM object. We need to

store the address in the interface pointer. This is because we never come to know about the class name of the component class. The client has only the interface pointer. The interface is the base class of the component class.

Here the **CoCreateInstance( )** function is a global function hence call to it is non-OO.

Smart Pointers

Consider the code snippet given in the slide.

For making a call to the **CoCreateInstance( )** function in OO style, we use a smart pointer class. This class overloads an -> operator. It also provides a member function **CreateInstance( )** that internally calls the **CoCreateInstance( )** function. In the client program we create an object of the smart pointer class and calls the **CreateInstance( )** function. The **CreateInstance( )** function stores the address of the component object in the interface pointer, which it obtains by calling the **CoCreateInstance( )** function.

Then we call the **Add( )** function using the object of the smart pointer class. We call this function using the -> operator. This calls the overloaded -> operator function, which returns the address stored in interface pointer. Using this address the **Add( )** function is then called. Hence the **Add( )** of the COM component is called.

The important point to note here is that the smart pointer class object works like an object as well as pointer. Here the **CoCreateInstance( )** call is made in object-oriented style as well the call to function inside the COM component is made using the pointer notation, which was the past experience.

Class Libraries

In this lecture you will understand:

* What are different class libraries
* What is STL
* Containers, Iterators and Algorithms
* What are different categories of containers

Class Libraries

Reuse of existing code is one of the primary goals that C++ addresses. Existing classes can be made available for reuse by packaging them in a library. Several such class libraries are available. They are Standard C++ Library, Standard Template Library and Microsoft Foundation Class Library.

A C++ program can call on a large number of functions from the Standard C++ Library that provide efficient implementations of frequently used operations such as input and output. Each and every entity in the library is declared in one or more header files. To use the classes declared in these header files we need to **#include** them in out program. We have been using header files like 'iostream.h' and 'iomanip.h' in our programs. The Standard C++ Library consists of 50 headers files. These headers together host implementation of the C++ library.

Microsoft Foundation Class (MFC) library contains hundreds of C++ classes. These classes are used for programming Windows Operating System. Using these classes we can do virtually everything that you would want to do under Windows.

We would see STL in the next slide.

Standard Template Library (STL)

The C++ Standard Template Library referred to, as STL is a C++ programming library that has been developed by Alexander Stepanov and Meng Lee at the Hewlett Packard laboratories in Palo Alto, California. The Standard Template Library (STL) is a general-purpose C++ library of algorithms and data structures. It is a built-in container class library, used to store and process data. It is a part of the standard ANSI/ISO C++ library.

The STL is implemented by means of the C++ template mechanism, hence the name Standard Template Library. The STL can be applied in a very straightforward way, facilitating reuse of the sophisticated data structures and algorithms it contains. It provides many of the basic algorithms and data structures of computer science. The STL consists of various types of entities most important of which are containers, algorithm, and iterators. We would learn more about containers and iterators in the coming slides. Let us see what algorithms are.

Algorithms in STL are certain procedures that are applied to containers to process the data. STL provides algorithms for operations like search, sort, merge, copy, etc. The algorithms are provided by means of a template function. These template functions are not written as part of the container class, but are provided as standalone template functions. Moreover, these functions can also be used on ordinary C++ data members like arrays, or user-defined container classes.

Containers

A **container** is a template class that is used to store objects of other types such as **int**, **float**, **double**, **char**, etc. It actually manages the objects of different data types. By saying managing the objects, we mean that the container as a template class provides constructor, destructor, necessary operator functions and additional member functions. Container classes form one of the most crucial components of STL.

The container classes are divided into three categories, depending on the way the elements are arranged. The categories are Sequence Containers, Associative Containers and Derived Containers.

A **sequence** container is a kind of container that organizes a finite set of objects in a linear arrangement, provided that the elements are of the same type. Each element is related to the other elements by its position. STL provides three basic kinds of sequence containers—**vector**, **list** and **deque**.

STL provides three derived containers. They are— **stack**, **queue** and **priority_queue**. They are also known as container adapters. Creation of these containers is based on sequence containers. The derived containers do not support iterators and therefore they cannot be used for data manipulation.

Associative Containers are the one that provide fast retrieval of data from the collection, which is based on keys. Associative container is not sequential and hence uses keys to access data directly. The size of the collection can vary at runtime. The collection is maintained in order. There are four types of associative containers—**set**, **multiset**, **map** and **multimap**.

Iterators

Iterators are special STL objects that are used to represent positions of elements in various STL containers. More clearly, iterators play a role similar to that of a subscript in a C++ array. Iterators are like references that allow the programmer to access a particular element, and to traverse through the container.

The slide shows few more points about iterators.

Types of Iterators

There are many different kinds of iterators depending on the type of container with which they are associated. At any given time, an iterator object is associated with only one container object.

In addition to the types given in the slide, the iterators can be **const** (e.g., "const_iterator") or non-const. Constant iterators can be used to examine container elements, but cannot be used to modify the elements in the container. Non-constant iterators cannot be used with constant container objects.

Using Stack, Queue & Vector

In this lecture you will understand:

* How to use sequence containers
* How to use derived containers
* How to use associative containers

Using *stack*

A **stack** object is a sequential container that allows insertion and deletion of elements only at one end. It follows Last In First Out system for adding and retrieving the stack elements. This slide shows the program that maintains a stack of integers.

A stack can be implemented using **vectorlist** or **deque**. We can specify the type of the underlying container as the second parameter in the constructor of **stack** as shown below:

```
stack <int, vector<int> > stk ;
```

The default value is the class **deque**.

In the program we have created a stack to maintain integers. The object **stk** is a stack implemented as a **deque**. Then using **push( )** function we have added elements to the stack. The **size( )** function returns the total number of elements present in the stack. The **top( )** function returns an element present at the top of the stack. Hence we have called this function through a **while** loop that runs until the stack becomes empty. This we have checked using **empty( )** function. After displaying the element at the top we have called **pop( )** to remove an element at the top of the stack.

Finally **#include** the necessary header files. To use the **stack** template class we must **#include** the <stack> header file. We can skip giving the '.h' extension. In this file the **stack** class is defined, enclosed in the **std** namespace.

Using the **queue** class is same as using the **stack** class, hence we won't write separate program implementing the queue.

Using *priority_queue* (1/3)

Consider some jobs, which are to be processed by the CPU. The job, which has to be processed first, depends on its priority. CPU should get these jobs in the ascending order of their priority number, i.e. lower the priority number higher is the priority. There should be some function, which would arrange these jobs as per their priorities. Let us see how this can be done with the help of **priority_queue** container.

In this program a priority queue holds objects of a class *tasks* in the form of a vector. Note the statement that builds a priority queue.

```
priority_queue<tasks, vector<tasks>, prioritizetasks> pq ;
```

The above statement builds a priority queue **pq** to hold objects of class **tasks** as a vector. Furthermore, the class **prioritizetasks** provides a function to decide the order in which the tasks should get placed. We would define these classes later.

Here, firstly we have created an array **t** of **tasks** and initialized it at the same place. Then through a **for** loop we have added the objects to the priority queue **pq,** by calling function **push( )**. Finally, we have displayed the contents of **pq** through a loop.

Using *priority_queue* (2/3)

This slide shows the declaration of **tasks** and **prioritizetasks** classes.

The **tasks** class contains two data members out of which **prno** specifies the priority of the task. The class also contains two-argument constructor that is used to initialize the data members. The class overloads the << operator so that we can directly pass the object of the class to **cout**.

The **prioritizetasks** defines only the overloaded **operator( )** function. This function would get called while adding objects to **pq** in statement **pq.push (t [i]) ;** This function is used to sort the objects of **tasks** according to the **prno** of the object. It means that this function must have access to the **private**

data member of the **tasks** class. To allow the **prioritizetasks** class to access the **private** data members of the **tasks** class we have declared the **prioritizetasks** as a **friend** of the **tasks** class.

#include the necessary header files and declare the namespace **std** as shown in the slide. We have extended the **std** namespace and have included **tasks** and **prioritizetasks** classes in it to avoid ambiguity between the overloaded << operator declared in **tasks** class and **std** namespace.

Using *priority_queue* (3/3)

This slide shows the definition of the member functions of the **tasks** and **prioritizetasks** class.

In the constructor of the **tasks** class we have initialized the **pname** and **prno** data members. In the **operator<<( )** function we have simply displayed the values of **pname** and **prno**.

The **operator( )** function compares the priorities of the two objects **t1** and **t2** and returns 0 if **t1** is smaller and 1 if it is greater. Thus, at the time of pushing values to the queue, the tasks get sorted according to their priority and get stored in the queue.

Algorithms in STL are certain procedures that are applied to containers to process the data. STL provides algorithms for operations like search, sort, merge, copy, etc. The algorithms are provided by means of a template function. These template functions are not written as part of the container class, but are provided as standalone template functions. Moreover, these functions can also be used on ordinary C++ data members like arrays, or user-defined container classes.

Using *vector* (1/4)

Let us now see how to use the **vector** template class as a container of integers.

In this program we have created an object **vec** of the **vector** class and added elements to it by calling the **push_back( )** function. Then in the statement **vector < int >::iterator vitr ;** we have declared an iterator **vitr**. When an iterator to a **vector** or a **deque** is created it is automatically created as a random access iterator. Using this iterator we have traversed through the vector. The **begin( )** and **end( )** member functions return the random access iterator to the starting and ending elements of the container respectively. We have initialized the **for** loop with iterator to the starting position by calling the **begin( )** function. The loop will run until it reaches the end of the container. The iterator is incremented using the ++ operator and is dereferenced using the * operator.

Next we have displayed the first element of the container by calling the **front( )** member function. Then we have called the **at( )** function to obtain the 2nd element of the vector. Lastly, we have obtained the last element of the vector by calling the **back( )** function.

The **vector** container class is declared in **std** namespace defined in the **<vector>** header file. So, write the **#include** statements as shown in the slide.

Using *vector* (2/4)

Next, in the program, we have again obtained the iterator to the start position again and have stored the value **35** at that position. Iterators also allow using the **[ ]** operator to access the elements. Using this operator we have stored **20** at second position in the vector. Then we have incremented the iterator by 4 and stored **99** at that position.

Then we have displayed all the elements of the vector using iterator in a **for** loop.

To insert an element in the vector we have called the **insert( )** member function and passed to it the position and the value to be inserted. After inserting the value, we have again iterated the vector and displayed all the elements.

Using *vector* (3/4)

In this part of the program, we have firstly deleted an element from the specified position and then popped an element from the back of the vector.

We have deleted the element at the second position using the **erase( )** member function. For deleting the elements at the end of the vector, we have called the **pop_back( )** function.

Using *vector* (4/4)

Lastly, we have cleared the vector by deleting all the elements at one shot by calling the **clear( )** member function. Using the **empty( )** function we have checked whether the vector is empty. We have displayed the appropriate message if the vector is empty.

Using List

In this lecture you will understand:

* How to use **list** container

Using list (1/5)

The list is a container, which implements a classic list data structure. Lists are implemented as doubly linked list structures in order to support bi-directional iterators. Each element in the list contains a pointer to the preceding and the next element in the list. Lists are better used when we want to add or remove elements to or from the middle of the list. The header file required to be **#include**d in a program for lists would be <list>. In the next few slides we would see how to implement **list** container.

In this program, we wish to create list **ls** of integers. We have called two functions such as **push_back( )** and **push_front( )** to add elements at the end or at the beginning of the list respectively. While adding elements at the beginning of the list the existing elements would get shifted one place to the right.

The elements of the list are accessed using the iterator as we saw in the last lecture. The functions **front( )** and **back( )** displays element at the first and last position of the list respectively.

Using list (2/5)

The functions **pop_back( )** and **pop_front( )** remove element at the end and at the beginning of the list respectively. In case of **pop_front( )** function, after removing the element at the beginning of the list, the remaining elements are shifted one place to the left of their existing positions.

Using list (3/5)

The iterator provided for the list is a bi-directional iterator, which can move, sequentially in a forward or backward direction. As a result, random access to the elements in a list is not possible. This is the reason why list does not support subscript operator **[]**. Thus to insert an element at the 6^{th} position (in a list of 7 elements), we have used the statement **litr = ls.end( ) ;** which returns an iterator referring to the element at the 7^{th} position. Next, we have decremented the value of iterator by 1, to refer **litr** to the element at the 6^{th} position. The **insert( )** function then adds element -20 at this position. Similar steps are carried out to insert element at the 4^{th} and 5^{th} position.

The **erase( )** function called next erases an element to which the iterator **litr** is referring to.

Using list (4/5)

The **clear( )** function removes all the elements from the list. After deleting all the elements we have added new elements to the list. For this, we have initialized an array and used the array elements to populate the list.

The **sort( )** function called in the program sorts the elements in the list in ascending order.

Using list (5/5)

Here, we have called the member function of the list container called **reverse( )** that reverses the order of elements in the list **ls**. Then we have displayed the elements using the iterator.

Using Set Multiset

In this lecture you will understand:

* How to use **map** and **multimap** containers
* How to use **set** and **multiset** containers

Using set/multiset (1/6)

There are four types of associative containers—**set**, **multiset**, **map** and **multimap**. In the coming slides we would see programs using these containers.

The **set** container stores number of items and access them using a value as a key. These keys must be unique. Entries in **set** are kept in order. Set can be used to store the objects of user-defined classes or to hold simple data objects like **int**, **char**, **float**, **string**, etc. Set supports bi-directional iterators.

In this program we have used a set to hold integers. To create and store elements in a set we have given the following statement,

```
set < int, less <int> > set1 ( arr1, arr1 + 10 ) ;
```

It indicates that the set would store integer values, which would get arranged in an ascending order. The function **less<int>** given in the statement is a predefined algorithm operation, which compares two elements (integers in our case) and arranges them in ascending order. To arrange elements in descending order we can use **greater<type>** algorithm operation, or name of any user-defined function can be given. The values to be stored and compared are read from the integer arrays.

Thus two sets **set1** and **set2** store the values of **arr1** and **arr2** respectively in ascending order. The set **set3** has been kept empty since we want it to store results of various functions carried on sets **set1** and **set2**.

Then to traverse through the set we have declared an iterator **sitr** and have iterated the list.

Using set/multiset (2/6)

Next, we have used the **find( )** function to search an element in the list. To this function we have passed the range in which search should be done and the element. The functions like **insert( )**, **empty( )**, etc. work in same manner on sets also.

Next, in this program the algorithm specific to sets have been used. The **includes( )** function compares elements to check if the sequence of elements in the set **set1** and **set2** are same. In our case, this function would evaluate to **false** as the elements in two sets are not identical.

Using set/multiset (3/6)

The **set_union( )** function extracts those elements which are common as well as uncommon to both the sets. These elements would then get copied to the third set **set3**. The template function **inserter( )** copies these values to the set **set3**. The elements in the set **set3** too, would get arranged in an ascending order as the function **less<>** is mentioned while declaring **set3**.

The function **set_intersection( )** extracts elements which are common to both the sets **set1** and **set2**. These elements are copied to **set3**. As **set3** is being used to store the results of various functions, we have called **clear( )** function each time before using it in a function.

Using set/multiset (4/6)

The function **set_diiference( )** extracts such elements of **set1** which are not present in the **set2**. These elements too would then get copied to the **set3**. The **set_symmetric_difference( )** function on the other hand extracts all those elements which are either present in the **set1** or **set2** but not in both. Note that the elements that would get copied to the **set3** (as a result of algorithm operations) would be unique.

Using set/multiset (5/6)

A **multiset** also stores a key value, but as against a set, **multiset** can store duplicate values. Rest of the working of multiset is similar to set.

In the same program we have used a **multiset** of integers, which can hold duplicate or multiple key values. The statement,

```
multiset < int, greater < int > > mset1 ( arr3, arr3 + 10 ) ;
```

creates a multiset **mset1** to hold integer values. Furthermore, the elements are arranged in descending order, as the algorithm operation given is **greater<>**. We have copied the elements of an array **arr3** to **mset1**. To iterate through the container we have declared an iterator **msitr**. In the similar manner we have created another multisets **mset2** and **mset3**. The multiset **mset2** is initialized with the array **arr4**.

An important point to note here is that an iterator declared to traverse a set can be used on multisets or vice versa. Similarly, if an iterator is associated with a set of **int**s arranged in ascending order, then it can be used with the set of **int**s arranged in descending order or any other order specified by some user-defined function.

Using set/multiset (6/6)

Here, we have combined the two multisets by calling the **set_union( )** function in **mset3**.

The functions **find( )**, **empty( )**, etc. work in the same manner on a **multiset** as they work on other containers.

Using map/multimap (1/3)

A **map** stores a pair of values, where the pair consists of a key object and the value object. The key object can be a data such as **string**, **int**, **float** or any other object of user-defined class. The key object contains a key for which the map can be searched for. The value object stores additional data. The value object usually stores numbers or strings but it can even store the objects of other classes. For example, if the key object in a map holds a word, then the value object could be the length of the word, or the number of times the word has been repeated, or even the meaning of the word.

The data in a map always gets stored in a sorted order of the key object. The order of arranging data is decided by the function given in the syntax while creating the map. A map always stores a unique pair of key and value. A multi-map on the other hand stores multiple pairs of key and value in a sorted order. The various operations, which can be performed, on a map or multi-map are demonstrated in this program.

In this program we have created a map to store students name and the total marks obtained by the student. Here the key is the name of a student and a total mark scored by the student is the value for the key. The elements in the map are stored in an alphabetical order. To create the map we have given the statement,

```
map < string, int, less < string > > map1 ;
```

which indicates that the map **map1** would store a pair of **string** and an **int** in an order sorted on the key object **string**. We have copied the elements to the map **map1** from the string array **name1** and an **int** array **marks1** through a loop. Using the subscript operator **[]** the elements are copied to the map.

Using map/multimap (2/3)

Here, firstly we have displayed elements stored in *map1* through a loop. The statement given in the loop to display data is slightly different. Till now we simply used an iterator to display the data. But, since a map stores a pair of key and a value, **(*mitr).first** is used to display the key value and **(*mitr).second** is used to display the value associated with the key.

The algorithm function **find( )** called next searches for the specified name in the map. If the specified name is found then we are displaying the marks obtained by the student, otherwise an appropriate message is displayed.

Using map/multimap (3/3)

In the same program we have created a multimap **map2**. The statement,

```
multimap <string, int, less < string > > map2 ;
```

indicates that the multimap would contain a **string** as a key object and an **int** as a value, and the elements would get arranged in ascending order of the key object **string**. A multimap does not support the subscript **operator []**. Hence, to add elements to the multimap we have created an object **p** of the template class **pair**, which stores a pair of objects first and second.

The **insert( )** function adds pair **p** to the multimap **map2**. Then through a loop we have displayed the elements stored in **map2**.

MFC Collection Classes

In this lecture you will understand:

- The limitations of array and linked list
- Various MFC collection classes
- Usefulness of MFC collection classes

Store n Integers

Use of an appropriate Data Structure at appropriate place goes a long way in building efficient programs. If we are to store **n** integers we can either choose to declare an array of **n** integers or use a linked list.

If we are to use an array we must know its dimension beforehand, which may not always be possible. Moreover, once we commit the size of an array we cannot change it during execution.

If we use a linked list the difficulties that we have with arrays gets avoided, however, understanding and maintaining a linked list is difficult.

So best solution is to use MFC collection classes.

MFC Collection Classes

Use of an appropriate Data Structure at appropriate place goes a long way in building efficient programs. If you use an array in place of a linked list or vice-versa the program is likely to fire in some situation. But while writing a VC++ program the last thing on our mind is how to build a linked list or how to check bounds of an array. There are more important things to attend to. In such situations we can either use the Standard Template Library (STL) or MFC Collection Classes (MFC). Most MFC programmers prefer to use the collection classes than STL because linking another library (STL) may lead to code bloat. MFC provides collection classes of three categories:

- MFC Array Classes
- MFC List Classes
- MFC Map Classes

The slide lists the array, list and map collection classes.

Since we are using MFC classes for the first time, let us get familiar with how they look like. All the MFC classes begin with letter 'C', for example, **CByteArray**, **CObList**, etc. MFC uses Hungarian notation for class names, whereas, structure names are given in capital letters.

We would discuss the collection classes mentioned above one by one. But before we take up their discussion let us understand a concept called 'Returning By Reference'. This concept is used at number of places in MFC collection classes and a good understanding of this concept would go a long way in mastering MFC classes.

First Console Application

In this lecture you will understand:

- Returning a reference of a variable and its utility in MFC classes
- How **CUIntArray** class can be used to maintain an array of unsigned integers.

Returning By Reference

Here, the class **array** contains an array **arr** of 10 **int**s and an overloaded **operator [ ]( )** function. The array is initialized (code not given in slide) with some values. We wish to store some different value at 5th position in the array. Observe the second statement given in **main( )**. It would get interpreted as,

```
a.operator[ ] ( 5 ) = 25 ;
```

The value **5** would get passed as a parameter to **operator[ ]( )** function, which would get collected in **j**. From the **operator[ ]( )** function we have returned reference of **arr[j]** i.e. **arr[5]**, which would get collected in a temporary reference of type **int**. This temporary reference refers to **arr[5]**. To this temporary reference value 25 would get assigned. As a result, **arr[5]** would now hold 25.

Similarly, while displaying the value stored at 5th position the same steps as explained above would get carried out and the value of the referent i.e. **arr[5]** would get displayed.

Steps to Create a Console App

To use the MFC collection classes we would create a Console Application using Visual Studio editor. A console application is one that gives output in a console. For creating the console application, start Visual C++ 6.0. Select 'File | New' menu option from the VC++ editor. Then select 'Win32 Console application'. Give some project name, say, **sample**. A folder of this name gets created in which all the files related to this project are stored. In step 1 of the AppWizard, select 'An application that supports MFC' option and click the 'Finish' button.

The source file 'sample.cpp' gets created wherein you can type the program.

Array of UINTs

We know that arrays in C++ suffer from the limitation that their bounds have to be checked by the programmer. Exceeding the bounds causes an access violation. This can be prevented by creating array classes that perform bounds checking internally. MFC provides several such array classes as we saw in the 'MFC Collection Classes' section.

This slide shows a program that demonstrates usage of the MFC class **CUIntArray**. Open the **_tmain( )** function in the '.cpp' file which is the entry point of a Console Application. Some 'AppWizard' generated code would already be there. **_t** preceding **main( )** indicates that this program can use Unicode characters. The arguments passed to **_tmain( )** represent command-line arguments and environment variables. Since for our program these arguments are not relevant we would not go into their details here.

This program demonstrates how the various functions of **CUIntArray** class can be used to maintain an array of **UINT**s, where **UINT** is a **typedef** for **unsigned int**. Here, we have declared an array of ten **UINT**s and then initialized it through a loop. The size of the array is set using the **SetSize( )** member function.

The **[ ]** operator is overloaded in the **CUIntArray** class and so the expression **a [i]** calls the overloaded operator function as shown in the slide.

Slide Number 7

To get the size i.e. the number of elements present in **a** we have called **GetSize( )** function. Then to get an index of the last element of array i.e. to get the maximum index we have called **GetUpperBound( )** function. Remember an index of **Array** class always begins with 0. The difference between **GetSize( )** and **GetUpperBound( )** is that the first one returns total number of elements, whereas, the second returns index of the last element hence result of both always differ by one.

To get an element at a specified index we have called **GetAt( )** function. We have collected element placed at index 6 in an **UINT** data item. Similarly, to replace an element at index 6 with new element we have called **SetAt( )** function and then retrieved the new element by calling **GetAt( )**. The new element at index 6 now would be 99.

Slide Number 8

To insert new element we can make use of **InsertAt( )** function which first shifts elements from the specified index one place to the right and then inserts given element. Thus, **InsertAt( )** function increases the size of array as it inserts element in the array whereas **SetAt( )** simply replaces the element at specified index. Next, we have displayed all elements of the array by running a loop. Note that **GetUpperBound( )** this time returns index as 10 (as the size of array is increased by 1).

Next, to delete an element from the array present at specified index we have called **RemoveAt( )** function. This function will remove **2** elements starting from index **3**. Then, we have again displayed the modified array by running a loop.

Now, we wish to append a new array to an existing array **a**. So, first we have created a new array **b** having five elements.

Slide Number 9

To add elements of **b** to **a** we have called **Append( )** function. On appending an array to another array, a new array is created, elements from older array are copied to it and older array is deleted. To display elements of array again we have run a loop. The **GetUpperBound( )** this time returns maximum index as 13.

Lastly, we have removed elements of both the arrays **a** and **b** by calling **RemoveAll( )** function. Since **a** does not contain any element now, **GetSize( )** returns **0**.

2-D Array

This slide shows how the **CUIntArray** class can be used to create a 2-D array.

Here, we have created two **CUIntArray** objects—**a[0]** and **a[1]**. Each object holds a different number of elements. The statement,

```
a[ i ][ j ] = j ;
```

gets interpreted as **a[i].operator[ ] (j)** = **j**, which returns a temporary reference. The value of **j** is then placed at the referent to which the temporary reference is referring.

Now that you know how to use the array class **CUIntArray**, you can use the other MFC array classes on similar lines.

Linked List of Strings

In this lecture you will understand:

- How **CStringList** class can be used to maintain a linked list of strings.

Linked List of Strings

Though the **InsertAt( )** and **RemoveAt( )** functions make it easy to add/delete elements to/from array, the performance suffers since array elements may have to be shifted upward or downward in memory. This can be avoided using MFC list classes that maintain an ordered list of items using the linked list data structure. When items are maintained using linked list the insertion or deletion of an item doesn't require any items to be shifted upward or downward in memory. It simply involves readjustment of pointers stored in the items before and after the insertion point.

This slide shows a program that creates a list of strings. in this program firstly we have created an object of **CStringList** class and then added 4 strings to it by calling **AddTail( )** function. This function appends a new node to the linked list.

To print the contents of the list first we should have a pointer to the first node of the list. This we have retrieved here by calling **GetHeadPosition( )** function. This function returns the position of the first node, which we have collected in **p**, a data item of type POSITION. POSITION is a pointer to a node suitably **typedef**ed. Then through a loop we have retrieved a string stored at each node and displayed it. To retrieve a string we have called **GetNext( )** function by passing it the address of first node i.e. **p** which returns a string present at current position and changes **p** to make it point to the next position i.e. address of the second node. The value in **p** can be changed because it is received as a reference in **GetNext( )**.

Note the statement given in the code that displays the strings retrieved. The **GetNext( )** function returns a **CString** object, and as mentioned earlier **CString** is an MFC class, which **cout** doesn't know. However, it knows how to display a string pointed to by a long pointer. This is the reason why we have typecasted the value returned by **GetNext( )** to LPCTSTR (Long Pointer To Constant String). **CString** provides an overloaded typecast operator for LPCTSTR.

Slide Number 4

To replace a particular string present in the list first we need to get the position (i.e. an address) of the node containing the string. This we have done here by calling the **Find( )** function. Then to replace the string we have called **SetAt( )** function and passed to it the position **p** and the new string. We have also displayed the newly replaced string.

Next we have inserted a string after the specified string. Here again, first we need to get the position of that node after which we want to insert string. This we have done by calling **Find( )** function. Next we have called the **InsertAfter( )** function to which we have passed the position **p** (after which new node has to be inserted) and the string to be inserted. This function returns the position of the newly added node, which we have collected in **p**. Then we have displayed the string at position **p**.

Slide Number 5

On similar lines, to remove a string from the list we need to get position of the node containing particular string. So, here too we have called **Find( )** function and collected the position of node to be deleted in **p**. To remove the node we have called **RemoveAt( )** function and passed it the position **p**.

Lastly, we have printed the contents of the list thus updated. Here first we have retrieved the position of first node by calling **GetHeadPosition( )** and collected it in **p**. Then, the loop runs till **p** has not reached end of the list, i.e. till **p** is not NULL.

MAP Strings

In this lecture you will understand:

- How to create map of items keyed by other item with the help of MFC **CMapStringToString** Class.
- How to set multiple values with a key in a map

Map Strings

A map is a table of items keyed by other items. Maps are designed such that given a key the corresponding item can be found in the table quickly. Maps make locating an item in a large volume of data efficiently.

Here we have stored in the map Hindi names for days of week. Each item is keyed by a string specifying its English-Language equivalent. The **[]** operator inserts an item and it's key into the map. Internally what is maintained is a linked list of pairs of item and its key. Note that the order of insertion may be different than order of storage. Which order is used depends upon the hashing scheme used by MFC.

Now, when we say **m["Sun"] = "Rav" ;** it becomes **m.operator [ ] ("Sun") = "Rav" ;** The overloaded operator **[]** returns a reference to a **CString** belonging to the node where "Sun" is stored. Now it becomes **reference.operator = ("Rav") ;** This reference is a **CString** reference, hence = operator of **CString** gets called and "Rav" gets stored in a **char** array that is part of the **CString** object and the **CString** object is part of the node.

Next we have called **Lookup( )** to retrieve an item. If **Lookup( )** returns zero it means no item is keyed by the key specified in **Lookup( )**'s first parameter. Here, we have searched for an item associated with the key Wed. If it is found then we have displayed the value of the item collected in **s**.

Slide Number 4

To display the value of key and its associated item, we need to get the position of first pair added in map. This we have retrieved by calling **GetStartPosition( )** function and collected in **p**. Then the way we traversed through the linked list here too we are required to traverse through the list of pairs. Note that here instead of **GetNext( )**, map class has provided **GetNextAssoc( )** function. To this function we have passed the position **p**, and two **CString**s **key** and **item** in which the value of the key and its associated item would get collected for the current position. This function also changes position **p** to the next node. **GetNextAssoc( )** takes all three arguments as references.

One Key – Multiple Values

The program given in the slide shows how multiple values can be set for one key.

Here first we have created an array of objects of **CStringList** class. Then we have added two strings in each of the **CStringList** object of array **s**.

Next we have created an object of **CMapStringToOb**, which maps a string with an address of object of class **CObject** or of class derived from **CObject**. We have then stored in the map an alphabet and address of **CStringList** objects of **CStringList** array **s**. In other words with every **CString** (i.e an alphabet) we have associated multiple values stored in **CStringList** object.

Now, to retrieve the key and its associated multiple values we have run a loop. First we have retrieved the position of first pair by calling **GetStartPosition( )**. Then in every iteration of **for** loop we have called **GetNextAssoc( )** which returns the value of key and its associated item. To retrieve items from the **CStringList** object collected in **str** we have retrieved the position of the first node and then traversed through the list till position **strpos** is not NULL.