Object Oriented Programming

Unit 6

Unit Outcome

- Exception Handling
- Types of Errors
- Benefits of exception handling
- try, catch, throw keywords
- Throwing an exception
- 'try' block
- Catching an exception
- Exception Handling Mechanism
- Catching all exceptions
- Nested try blocks

File Handling in C++

Every member function of the class is born with the pointer called "THIS"; which points to the object with which member function is associated.

When a member function is invoked it comes into existence with the value of this set to the address of the object for which is it called. File handling is used for store a data permanently in computer. Using file handling we can store our data in secondary memory (Hard disk).

How to achieve the File Handling?

For achieving file handling we need to follow the following steps:-

- STEP 1-Naming a file
- STEP 2-Opening a file
- STEP 3-Writing data into the file
- STEP 4-Reading data from the file
- STEP 5-Closing a file.

This Pointer

- Every member function of the class is born with the pointer called "THIS"; which points to the object with which member function is associated.
- When a member function is invoked it comes into existence with the value of this set to the address of the object for which is it called.

Using this for Returning Values

- This pointer can be use to return the values from the member function.
- Since it points to the address of the object which is called the member function, we can return the object by value with the help of this pointer.

Using this for specifying memory address

- This pointer is created automatically inside the member function whenever the member function is invoked by the object.
- It hold the memory address of the object so it can be used to access the memory address of the object.

Using this for accessing the data member

- This pointer can also be used to access the data members inside the member function.
- It can be done with the help of arrow operator $(->)$.
- Arrow operator is the combination of hyphen $(-)$ and greater than operator $(>)$.

Static Data Member and Member Function

Static Data Member A data member of a class can be qualified as static. The properties of a static member variable are similar to that of C-programmings static variable. A static data member has certain special characteristics. They are:-

- It is initialized to zero when the first object of its class is created. No other initialization is permitted.
- Only one copy of that member is created for the entire class and is shared by all the objects of that class, no matter how many objects are created.
- It is visible only within the class, but its lifetime is the entire program.

Static Data Member and Member Function

A static variable is normally used to maintain value common to the entire class. For e.g, to hold the count of objects created. Note that the type and scope of each static member variable must be declared outside the class definition. This is necessary because the static data members are stored separately rather than as a part of an object.

Declaration

static data-type member-name;

Defining the static data member

It should be defined outside of the class following this syntax:

data-type class-name :: member-name =value;

If you are calling a static data member within a member function, member function should be declared as static (i.e. a static member function can access the static data members)

Static Data Member and Member Function

Static Member Function

like a static member variable, we can also have static member functions. A member function that is declared static has the following properties

- A static function can have access to only other static members (function or variable) declared in the same class.
- A static member function can be called using the class name (instead of its object) as follows-
  Class-name::Function-name();

C++ Inheritance

- In C++, inheritance is a process in which one object acquires all the properties and behaviors of its parent object automatically. In such way, you can reuse, extend or modify the attributes and behaviors which are defined in other class.

- In C++, the class which inherits the members of another class is called derived class and the class whose members are inherited is called base class. The derived class is the specialized class for the base class.

C++ Inheritance

Advantage of C++ Inheritance

- Code reusability: Now you can reuse the members of your parent class. So, there is no need to define the member again. So less code is required in the class.

Types Of Inheritance: C++ supports five types of inheritance:

- Single inheritance
- Multiple inheritance
- Hierarchical inheritance
- Multilevel inheritance
- Hybrid inheritance

C++ Inheritance

The Syntax of Derived class:

```
class derived_class_name : visibility-mode base_class_name
{
    // body of the derived class.
}
```

Details:

- derived_class_name: It is the name of the derived class.
- visibility mode: The visibility mode specifies whether the features of the base class are publicly inherited or privately inherited. It can be public or private.
- base_class_name: It is the name of the base class.

C++ Single Inheritance

- Single inheritance is defined as the inheritance in which a derived class is inherited from the only one base class.

- When one class inherits another class, it is known as single level inheritance.

Protected: Access Specifier

- Protected: Protected access modifier is similar to that of private access modifiers, the difference is that the class member declared as Protected are inaccessible outside the class but they can be accessed by any subclass(derived class) of that class.

C++ Multi Level Inheritance Example

- When one class inherits another class which is further inherited by another class, it is known as multi level inheritance in C++.
- Inheritance is transitive so the last derived class acquires all the members of all its base classes.

C++ Multiple Inheritance

- Multiple Inheritance is a feature of C++ where a class can inherit from more than one classes. i.e one sub class is inherited from more than one base classes.

- Multiple inheritance is the process of deriving a new class that inherits the attributes from two or more classes.

C++ Hierarchical Inheritance

- In this type of inheritance, more than one sub class is inherited from a single base class. i.e. more than one derived class is created from a single base class.

- Hierarchical inheritance is defined as the process of deriving more than one class from a base class.

C++ Hybrid Inheritance

- Hybrid Inheritance is implemented by combining more than one type of inheritance. For example: Combining Hierarchical inheritance and Multiple Inheritance.

- Hybrid inheritance is a combination of more than one type of inheritance.

Visibility of Inherited Members

Base class visibility	Derived class visibility		
	Public	Private	Protected
Private	Not Inherited	Not Inherited	Not Inherited
Protected	Protected	Private	Protected
Public	Public	Private	Protected

Figure 1. Visibility of Inherited Members

Visibility of Inherited Members

Type of Inheritance: When deriving a class from a base class, the base class may be inherited through public, protected or private inheritance. The type of inheritance is specified by the access-specifier. We hardly use protected or private inheritance, but public inheritance is commonly used. While using different type of inheritance, following rules are applied:

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

EXCEPTION HANDLING

EXAMPLE

```
class MyException : public exception
{
  char * what () const throw ()
  {
    return "C++ Exception";
  }
};
```

```
void main()
{
  int a=0;
  try
  {
     if (a==0)
    throw MyException();
  }
  catch(MyException& e)
  {
   cout << "MyException caught" ;
   cout << e.what();
  }
  catch(exception& e)
  {
    //Other errors
  }
}
```

EXCEPTION HANDLING

- **An exception is a problem that arises during the execution of a program.**
- **Exceptions provide a way to transfer control from one part of a program to another. C++ exception handling is built upon three keywords: try, catch, and throw.**

TRY, CATCH & THROW

- **throw:** A program throws an exception when a problem shows up. This is done using a **throw** keyword.
- **catch:** A program catches an exception with an exception handler at the place in a program where you want to handle the problem. The **catch** keyword indicates the catching of an exception.
- **try:** A **try** block identifies a block of code for which particular exceptions will be activated. It's followed by one or more catch blocks.

TRY & CATCH

```
try
{
  // protected code
}
catch( ExceptionName e1 )
{
  // catch block
}
catch( ExceptionName e2 )
{
  // catch block
}
catch( ExceptionName eN )
{
  // catch block
}
```

You can list down multiple catch statements to catch different type of exceptions in case your try block raises more than one exception in different situations.

THROWING EXCEPTIONS

- Exceptions can be thrown anywhere within a code block using **throw** statements.
- The operand of the throw statements determines a type for the exception and can be any expression and the type of the result of the expression determines the type of exception thrown.

EXAMPLE

```
double division(int a, int b)
{
  if( b == 0 ) // type of exception
  {
    throw "Division by zero condition!";
  }
  return (a/b);
}
```

CATCHING EXCEPTIONS

- **The catch block following the try block catches any exception.**
- **You can specify what type of exception you want to catch and this is determined by the exception declaration that appears in parentheses following the keyword catch.**

EXAMPLE WITHOUT EXCEPTION HANDLING

```
int a=20,b=0;
c=a/b;


// displays error
```

EXAMPLE WITH EXCEPTION HANDLING
- DIVIDE BY ZERO EXCEPTION

```
int a=20,b=0;
try
{
 if( b == 0 )
   {
     throw "Division by zero condition!";
   }
else
  c=a/b;
}
catch (char* msg)
{
    cout << msg << endl;
   }
```

MULTIPLE CATCH BLOCKS

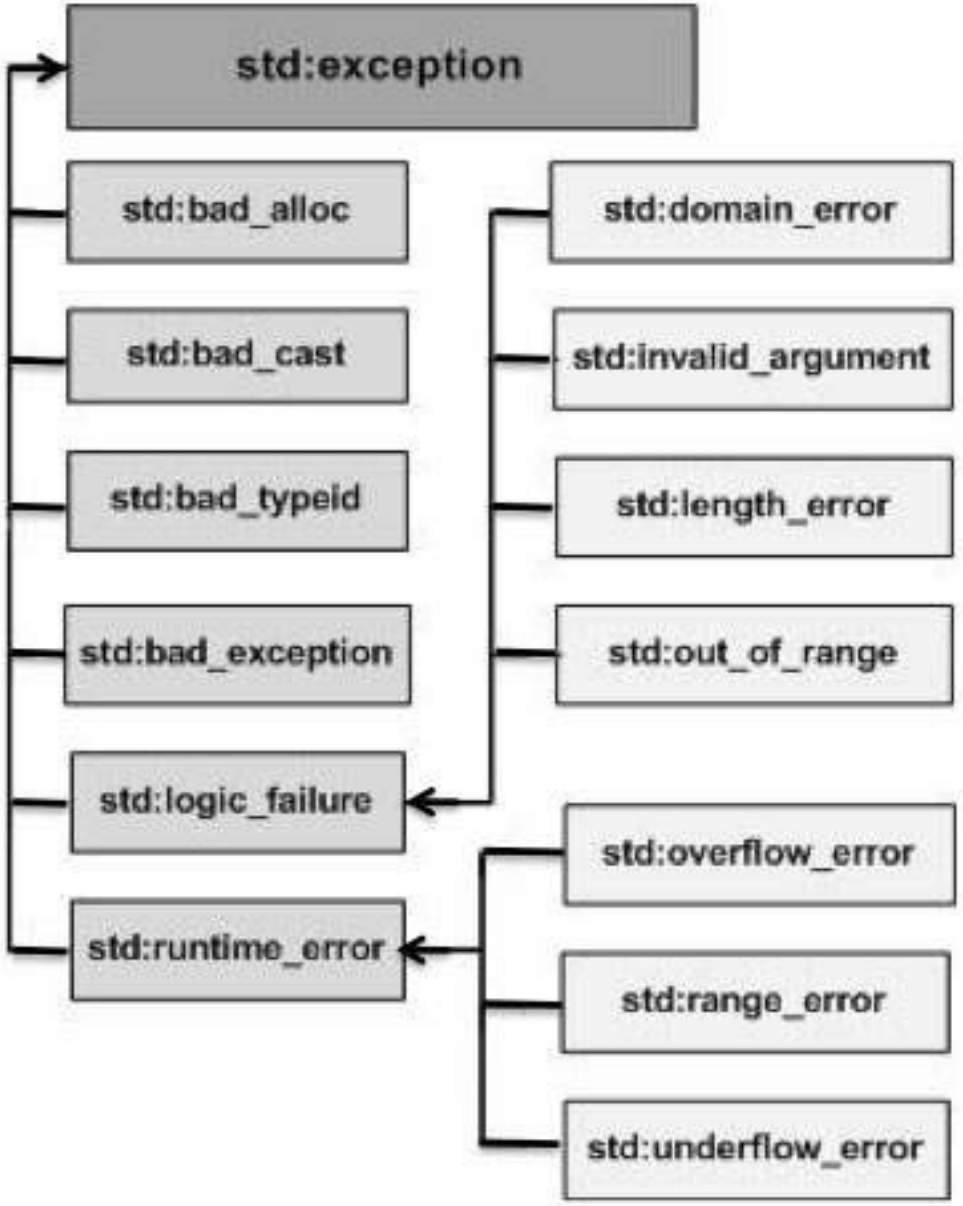
```
void test(int x)
{
  try
  {
          if(x>0)
            throw x;
      else
            throw 'x';
  }

  catch(int x)
  {
          cout<<"Catch a integer and that integer
is:"<<x;
  }

  catch(char x)
  {
          cout<<"Catch a character and that character
is:"<<x;
  }
}
```

```
void main()
{
  clrscr();
  cout<<"Testing multiple catches
n:";
  test(10);
  test(0);
  getch();
}
```

C++ STANDARD EXCEPTIONS:



std::exception	**An exception and parent class of all the standard C++ exceptions.**
std::bad_alloc	**This can be thrown by new.**
std::bad_cast	**This can be thrown by dynamic_cast.**
std::bad_exception	**This is useful device to handle unexpected exceptions in a C++ program**
std::bad_typeid	**This can be thrown by typeid.**
std::logic_error	**An exception that theoretically can be detected by reading the code.**
std::domain_error	**This is an exception thrown when a mathematically invalid domain is used**
std::invalid_argument	**This is thrown due to invalid arguments.**
std::length_error	**This is thrown when a too big std::string is created**
std::out_of_range	**This can be thrown by the at method from for example a std::vector and std::bitset<>::operator[]().**
std::runtime_error	**An exception that theoretically can not be detected by reading the code.**
std::overflow_error	**This is thrown if a mathematical overflow occurs.**
std::range_error	**This is occured when you try to store a value which is out of range.**

DEFINE NEW EXCEPTIONS

- You can define your own exceptions by inheriting and overriding **exception** class functionality.
- Allows you can use std::exception class to implement your own exception in standard way