- Data Abstraction
- Data Encapsulation
- Interfaces
- Exception Handling

Data Abstraction in C++

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

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

In C++, classes provides great level of data abstraction. They provide sufficient public methods to the outside world to play with the functionality of the object and to manipulate object data, i.e., state without actually knowing how class has been implemented internally.

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

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

Benefits of Data Abstraction

Data abstraction provides two important advantages –

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

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

```
int main() {
  Adder a;

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

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

Data Encapsulation in C++

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

- **Program statements (code)** – This is the part of a program that performs actions and they are called functions.
- **Program data** – The data is the information of the program which gets affected by the program functions.
- Encapsulation binds together the data and functions that manipulate the data, and that keeps both safe from outside interference and misuse.
- Data encapsulation led to the important OOP concept of data hiding.
- **Data encapsulation** is a mechanism of bundling the data, and the functions that use them and **data abstraction** is a mechanism of exposing only the interfaces and hiding the implementation details from the user.

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

```
int main() {
  Adder a;

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

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

Interfaces in C++ (Abstract Classes)

- An interface describes the behavior or capabilities of a C++ class without committing to a particular implementation of that class.
- The C++ interfaces are implemented using **abstract classes** and these abstract classes should not be confused with data abstraction which is a concept of keeping implementation details separate from associated data.
- Abstract classes cannot be used to instantiate objects and serves only as an interface. Attempting to instantiate an object of an abstract class causes a compilation error.

```
class Shape {
  public:
     virtual int getArea() = 0;
    void setWidth(int w) {
      width = w;    }
void setHeight(int h) {
      height = h;   }
protected:
    int width;
    int height;
};
class Rectangle: public Shape {
  public:
    int getArea() {
      return (width * height); }
};
class Triangle: public Shape {
  public:
    int getArea() {
      return (width * height)/2;  }
};
```

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

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

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

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

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

  return 0;
}
```

C++ Exception Handling

Exceptions are runtime anomalies or abnormal conditions that a program encounters during its execution.

There are two types of exceptions: a)Synchronous, b)Asynchronous (i.e., exceptions which are beyond the program's control, such as disc failure, keyboard interrupts etc.).

C++ provides the following specialized keywords for this purpose:

The try statement allows you to define a block of code to be tested for errors while it is being executed.

The throw keyword throws an exception when a problem is detected, which lets us create a custom error.

The catch statement allows you to define a block of code to be executed if an error occurs in the try block

C++ Exception Handling

- An exception is a problem that arises during the execution of a program.
- A C++ exception is a response to an exceptional circumstance that arises while a program is running, such as an attempt to divide by zero.
- 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**.

- **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.

advantages of exception handling

1) Separation of Error Handling code from Normal Code:
2) Functions/Methods can handle only the exceptions they choose:
3) Grouping of Error Types

C++ Exception Handling

Assuming a block will raise an exception, a method catches an exception using a combination of the **try** and **catch** keywords. A **try/catch** block is placed around the code that might generate an exception. Code within a **try/catch** block is referred to as protected code, and the syntax for using **try/catch** as follows –

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

Throwing Exceptions

Throwing Exceptions

Exceptions can be thrown anywhere within a code block using throw statement. The operand of the throw statement 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.

```
double division(int a, int b) {
  if( b == 0 ) {
    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.

```
try {
  // protected code
    }
catch( ExceptionName e )
  {
 // code to handle ExceptionName exception
 }
```

Above code will catch an exception of ExceptionName type.

Catching Exceptions

If you want to specify that a catch block should handle any type of exception that is thrown in a try block, you must put an ellipsis, ..., between the parentheses enclosing the exception declaration as follows −

```
try {
   // protected code
} catch(...) {
  // code to handle any exception
}
```

```
try {
  int age = 15;
  if (age >= 18) {
    cout << "Access granted - you are old enough.";
  } else {
    throw (age);
  }
}
catch (int myNum) {
  cout << "Access denied - You must be at least 18 years old.\n";
  cout << "Age is: " << myNum;
}
```

```
int main() {
  try {
    int age = 11;
    if (age >= 18) {
      cout << "Access granted - you are old enough.";
    } else {
      throw 505;
    }
  }
  catch (int myNum) {
    cout << "Access denied - You must be at least 18 years old.\n";
    cout << "Error number: " << myNum;
  }
  return 0;
}
```

```
int main() {
  try {
    int age = 15;
    if (age >= 18) {
      cout << "Access granted - you are old enough.";
    } else {
      throw 505;
    }
  }
  catch (...) {
    cout << "Access denied - You must be at least 18 years old.\n";
  }
  return 0;
}
```

Exception Handling

```
int main ()
{
  int x = 50;
  int y = 0;
  double z = 0;
  try {
    z = division(x, y);
    cout << z << endl;
  } catch (const char* msg) {
   cout << msg << endl;
  }
  return 0;
}
```

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

Exception Handling in C++

```
int main()
{
int x = -1;
// Some code
cout << "Before try \n";
try {       cout << "Inside try \n";
            if (x < 0)
            {
                        throw x;      cout << "After throw (Never executed) \n";
            }
}
catch (int x ) {
            cout << "Exception Caught \n";
}

cout << "After catch (Will be executed) \n";
return 0;
}
```

Output

```
Before try
Inside try
Exception Caught
After catch (Will be executed)
```

- There is a special catch block called the ‘catch all’ block, written as catch(…), that can be used to catch all types of exceptions

```
int main()
{
        try {
        throw 10;
        }
        catch (char *excp) {
                cout << "Caught " << excp;
        }
        catch (...) {
                cout << "Default Exception\n";
        }
        return 0;
}
```

Output:

```
Default Exception
```

Implicit type conversion doesn’t happen for primitive types. For example, in the following program, ‘a’ is not implicitly converted to int.

```
int main()
{
        try {
        throw 'a';
        }
        catch (int x) {
                cout << "Caught " << x;
        }
        catch (...) {
                cout << "Default Exception\n";
        }
        return 0;
}
```

Output:

```
Default Exception
```

If an exception is thrown and not caught anywhere, the program terminates abnormally. For example, in the following program, a char is thrown, but there is no catch block to catch the char.

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

int main()
{
        try {
        throw 'a';
        }
        catch (int x) {
                cout << "Caught ";
        }
        return 0;
}
```

```
//If we put the base class first then the derived class catch block will never be reached
class Base { };
class Derived : public Base { };
int main()
{
        Derived d;
        try {
                // Monitored code
                throw d;
        }
        catch (Base b) {
                cout << "Caught Base Exception";
        }
        catch (Derived d) {
                // This 'catch' block is NEVER executed
                cout << "Caught Derived Exception";
        }
        getchar();
        return 0;
}
```

Output:

```
Caught Base Exception
```

A derived class exception should be caught before a base class exception

```
class Base { };
class Derived : public Base { };
int main()
{
        Derived   d;
        try {
                throw   d;
        }
        catch (Derived d) {
                cout << "Caught Derived Exception";
        }
        catch (Base b) {
                cout << "Caught Base Exception";
        }
        getchar(); // To read the next character
        return 0;
}
```

Output:

```
Caught Derived Exception
```

- the C++ library has a standard exception class which is the base class for all standard exceptions. All objects thrown by the components of the standard library are derived from this class. Therefore, all standard exceptions can be caught by catching this type
- in C++, all exceptions are unchecked, i.e., the compiler doesn’t check whether an exception is caught or not. So, it is not necessary to specify all uncaught exceptions in a function declaration. Although it’s a recommended practice to do so

```
// This function signature is fine by the compiler, but not recommended. Ideally, the function should
specify all uncaught exceptions and function   signature should be "void fun(int *ptr, int x) throw (int *,
int)"
void fun(int *ptr, int x)
{               if (ptr == NULL)
                                throw ptr;
                if (x == 0)
                                throw x;
                /* Some functionality */
}
int main()
{
                try {
                fun(NULL, 0);
                }
                catch(...) {
                                cout << "Caught exception from fun()";
                }
                return 0;
}
```

```
// Here we specify the exceptions that this function throws.
void fun(int *ptr, int x)    throw (int *, int)                    // Dynamic Exception specification
{
        if (ptr == NULL)
                throw ptr;
        if (x == 0)
                throw x;
        /* Some functionality */
}
int main()
{
        try {
        fun(NULL, 0);
        }
        catch(...) {
                cout << "Caught exception from fun()";
        }
        return 0;
}
```

In C++, try/catch blocks can be nested. Also, an exception can be re-thrown using “throw; “.

```
int main()
{
    try {
        try {
            throw 20;
        }
        catch (int n) {
            cout << "Handle Partially ";
            throw; // Re-throwing an exception
        }
    }
    catch (int n) {
        cout << "Handle remaining ";
    }
    return 0;
}
```

```
Handle Partially
Handle remaining
```

When an exception is thrown, all objects created inside the enclosing try block are destroyed before the control is transferred to the catch block.

```
class Test {
public:
        Test() { cout << "Constructor of Test " << endl; }
        ~Test() { cout << "Destructor of Test " << endl; }
};

int main()
{
        try {
                Test t1;
                throw 10;
        }
        catch (int i) {
                cout << "Caught " << i << endl;
        }
}
```

```
Constructor of Test
Destructor of Test
Caught 10
```

Define New Exceptions

You can define your own exceptions by inheriting and overriding exception class functionality. Following is the example, which shows how you can use std::exception class to implement your own exception in standard way –

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

int main() {
  try {
    throw MyException();
  } catch(MyException& e) {
    cout << "MyException caught" <<endl;
    cout << e.what() <<endl;        //what() is a public method provided by exception class and it
                                    //has been overridden by all the child exception classes.
                                    //This returns the cause of an exception.
  } catch(exception& e) {
    //Other errors
  }
}
```

Output

```
MyException  caught
C++ Exception
```

```
int main()
{
    try
    {
        int x, y;
        cout << "Enter the two numbers : \n";
        cin >> x >> y;
        if (y == 0)
        {   MyException z;    throw z;  }
        else
        {     cout << "x / y = " << x/y << endl;  }
    }
    catch(exception& e)
    {
        cout << e.what();
    }
}
```

```
class MyException : public exception{
    public:
        const char * what() const throw()
        {
            return "Attempted to divide by
zero!\n";
        }
};
```

When an exception is thrown, all objects created inside the enclosing try block are destroyed before the control is transferred to the catch block.

```
class Test {
public:
        Test() { cout << "Constructor of Test " << endl; }
        ~Test() { cout << "Destructor of Test " << endl; }
};
int main()
{       try {
                Test t1;
                throw 10;
        }
        catch (int i) {
                cout << "Caught " << i << endl;
        }
}
```