girmiti software™

Introduction to C++

- Exception Handling
- try,catch,throw statements
- Catch all exceptions using '...'
- Rethrowing the Exception
- Exception Specification
- Exceptions & Constructor/Destructor
- Stack Unwinding
- C++ standard Exceptions
- User-defined Exception
- Hierarchy of Exceptions

Exception Handling

Exception Handling:

The process of converting system error messages into user friendly error message is known as Exception handling.

This is one of the powerful feature of C++ to handle run time error and maintain normal flow of C++ application.
Exceptions provide a way to transfer control from one part of a program to another.

Exception:

- An exception is an error or unexpected event.
- An exception is an event, which occurs during the execution of a program, that disrupts the normal flow of the program's Instructions.

Advantage of Exception Handling:

The core advantage of exception handling is to maintain the normal flow of the application.

Exception Handling

Handling the Exception:

Handling the exception is nothing but converting system error message into user friendly error message. We use three keywords for Handling the Exception in C++ , these are:

- try
- catch
- throw

Exception Handling

1. try block:

try block is used to enclose the code that might throw an exception.It's followed by one or more catch blocks.
The try block itself is indicated by the keyword try, followed by a brace-enclosed block of code indicating the code for which exceptions will be noticed.

Syntax:

```
try
{
// code to protect
}
```

2. throw expression:

The throw statement actually throws the exception of the specified type.
A throw statement, in essence, is a jump; that is, it tells a program to jump to statements at another location. The throw keyword indicates the throwing of an exception. It’s followed by a value, such as a character string or an object, that indicates the nature of the exception.

Syntax:

```
throw argument;
```

where argument is sent to the corresponding catch handler.

Exception Handling

Different types of throw expression are given below:

throw "An error occurred";

This example specifies that error message to be displayed.

throw object;

This example specifies that object is to be passed on to a handler.

throw;

This example specifies that the last exception thrown is to be thrown again.

3. catch block:

The catch block receives the thrown exception.
catch block is used to handle the Exception. It must be used after the try block only.
We can use multiple catch block with a single try.

It is one of the block in which we write the block of statements which will generates user friendly error messages.

Syntax:

```
catch()
{
…
}
```

Exception Handling

Syntax of try-catch:

```
try
{
// causes executions code
}
catch( ExceptionName e1 )
 {
 // catch block
}
```

try with multiple catch statement:

We can use multiple catches after try block.The catch statements are evaluated in the order of their appearance.

Once a match is found ,the subsequent catch-handlers are not examined.

If a matching handler is not found,the predefined runtime function terminate() is called.

Exception Handling

The layout of exception Handling is as follows::

```
try
{
//code to protect
}
catch(int x)
{
//handler int errors
}
catch(const char *str)
{
//handler char* errors
}
catch(float dx)
{
//handler float errors
}
```

This try block has three handlers.The first one that catches integer exception,the second one for character pointer & third catches float exception.One can define as many catch handler as one wants.

Exception Handling

Example without Exception Handling:

```
#include<iostream>
using namespace std;
int main()
{
int a, ans;
a=10;
ans=a/0;
cout<<"Result: "<<ans;
}
```

Output:
Abnormally terminate program

Exception Handling

Example using Exception Handling:

```
#include<iostream>
using namespace std;
int main ()
{ int x = 50;
  int y = 0;
  double z = 0;
  try
  {
          if(y==0)
          {
                  throw "Division by zero condition!";
          }
          z=x/y;
   cout << z << endl;
  }
  catch (const char* msg)
  {
    cout << msg << endl;
  }
  cout<<"end of main";
}
```

Exception Handling

Example:

```
#include<iostream>
using namespace std;
int main()
{
        try
        {
                int age;
                cout<<"enter age";
                cin>>age;
                if(age>100||age<1)
                throw "Invalid age!";
                cout<<"After the throw statement"<<endl;

        }
        catch(const char *msg)
        {
                cout<<"Error!"<<msg<<endl;
        }
        cout<<"end of the program"<<endl;
}
```

Output:

```
enter age  120
Error!Invalid age!
end of the program
```

Exception Handling

Catch all exceptions :

If it encounters a "…" in the catch expression, it will handle all exceptions, regardless of their data type.This we need if we do not know what type of exception occurred.
If we are using multiple catches with a try block , then '…' handler must be the last handler for its try block.

Syntax:

```
catch(...)      // catches all exceptions
{
…
}
```

Exception Handling

Example:

```
#include<iostream>
using namespace std;
int main()
{ cout<<"before the throw statement"<<endl;
  try
  {
  throw 29.5f;   //throw a float value
  }
  catch(const char *msg)
  {
          cout<<"Error!"<<msg<<endl;
  }
  catch(int i)
  {
          cout<<"caught an integer"<<endl;
  }
  catch(...)
  {
          cout<<"unknown data type"<<endl;
  }
  cout<<"end of the program"<<endl;}
```

Exception Handling

Rethrowing Exceptions:

Rethrowing an expression from within an exception handler can be done by calling throw, by itself, with no exception.
This causes current exception to be passed on to an outer try/catch sequence.
An exception can only be rethrown from within a catch block.
When an exception is rethrown, it is propagated outward to the next catch block.

Rethrow exception with the statement:

```
throw;                  //without arguments
```

Exception Handling

Example of Rethrowing the Exception:

```
#include <iostream>
using namespace std;
void MyHandler()
{
  try
  {
                    throw "hello";
  }
  catch (const char*)
  {
                    cout <<"Caught exception inside MyHandler\n";
                    throw;      //rethrow char* out of function
  }
}
int main()
{
  cout<<"Main start"<<endl;
  try
  {
          MyHandler();
  }
  catch(const char*)
  {
          cout <<"Caught exception inside Main\n";
  }
```

Exception Handling

```
 cout <<"Main end"<<endl;
}
```

Output:

```
Main start
Caught exception inside MyHandler
Caught exception inside Main
Main end
```

Exception Handling

Exception Specification:
We can qualify a function definition with an exception specification to indicate which kinds of exceptions it throws. To do so, we append the exception specification, which consists of the keyword throw followed by exception types enclosed in parentheses:

Example:

```
void f1() throw(int)
```

This accomplishes two things:
First, it tells the compiler what sort of exception or exceptions a function throws.
Second, using an exception specification alerts anyone who reads the prototype that this particular function throws an exception, reminding the reader that he or she may want to provide a try block and a handler. A function that throws more than one kind of exception can provide a comma-separated list of exception types:

```
void f1() throw(int,float)
```

Using empty parentheses in the exception specification indicates that the function does not throw exceptions:

```
double f1() throw();    // DOESN'T throw an exception
```

Exception Handling

Example:

```
#include <iostream>
using namespace std;
double division(int a, int b) throw(const char*)
{
    if( b == 0 )
    throw "Division by zero condition!";
    return (a/b);
}
int main ()
{ int x = 50;
  int y = 0;
  double z = 0;
  try
  {  z = division(x, y);
    cout << z << endl;
  }
  catch (const char* msg)
  {
    cerr << msg << endl;
  }
}
```

Exception Handling

Exceptions & Constructor/Destructor:

Exception handling can be used in conjunction with constructors and destructors to provide resource management .
Exceptions thrown by constructors cause any already-constructed component objects to call their destructors.Only those objects that have already been constructed will be destructed.

When an exception is thrown and control passes to a catch block following a try block, destructors are called for all automatic objects constructed since the beginning of the try block directly associated with that catch block.
If an exception is thrown during construction of an object consisting of subobjects or array elements, destructors are only called for those subobjects or array elements successfully constructed before the exception was thrown.

Exception Handling

Example:

```
#include<iostream>
using namespace std;
class A
{
  public:
   A(){
        cout<<"Hi Constructor of A\n";
   try
   {
        throw 10;
   }
   catch(...)
   {
        cout<<"the String is unexpected one in constructor\n";
   }
                        cout<<"Hi Constructor of A\n";
 }
   ~A(){
   cout<<"Hi destructor of A\n";
 }
};
```

Exception Handling

```
class B
{public:
  B()
  {
          cout<<"B's constructor";
  }
  ~B()
  {
          cout<<"B's destructor";
  }
};
int main()
{
try
{
   A obj ;
   throw "Bad allocation";
   B b1;
}
catch(int i)
{    printf("the Exception if Integer is = %d\n", i);  }
```

Exception Handling

```
catch(double i)
{
   printf("the Exception if double is = %f\n", i);
}
 catch(A *objE)
{
   printf("the Exception if Object \n");
}
 catch(...)
{
   printf("the Exception if character/string \n");
}
printf("Code ends\n");
return 0;
}
```

Exception Handling

Stack Unwinding:

The process of removing function entries from function call stack at run time is called Stack Unwinding. Stack Unwinding is generally related to Exception Handling. In C++, when an exception occurs, the function call stack is linearly searched for the exception handler, and all the entries before the function with exception handler are removed from the function call stack. So exception handling involves Stack Unwinding if exception is not handled in same function (where it is thrown).

Also if there were some local class objects inside f1() and f2(), destructors for those local objects would have been called in Stack Unwinding process.

Exception Handling

Example:

```
#include<iostream>
using namespace std;
// A sample function f1() that throws an int exception
void f1() throw (int)
{
        cout<<"\n f1() Start ";
        throw 100;
        cout<<"\n f1() End ";
}
// Another sample function f2() that calls f1()
void f2() throw (int)
{
        cout<<"\n f2() Start ";
        f1();
        cout<<"\n f2() End ";
}
// Another sample function f3() that calls f2() and handles exception thrown by f1()
void f3()
{
        cout<<"\n f3() Start ";
```

Exception Handling

```
try
        {
        f2();
        }
        catch(int i)
        {
        cout<<"\n Caught Exception: "<<i;
        }
        cout<<"\n f3() End";
}
// A driver function to demonstrate Stack Unwinding  process
int main()
{
        f3();
}
```

Output:

```
f3()
Start f2()
Start f1()
Start Caught Exception: 100
f3() End
```

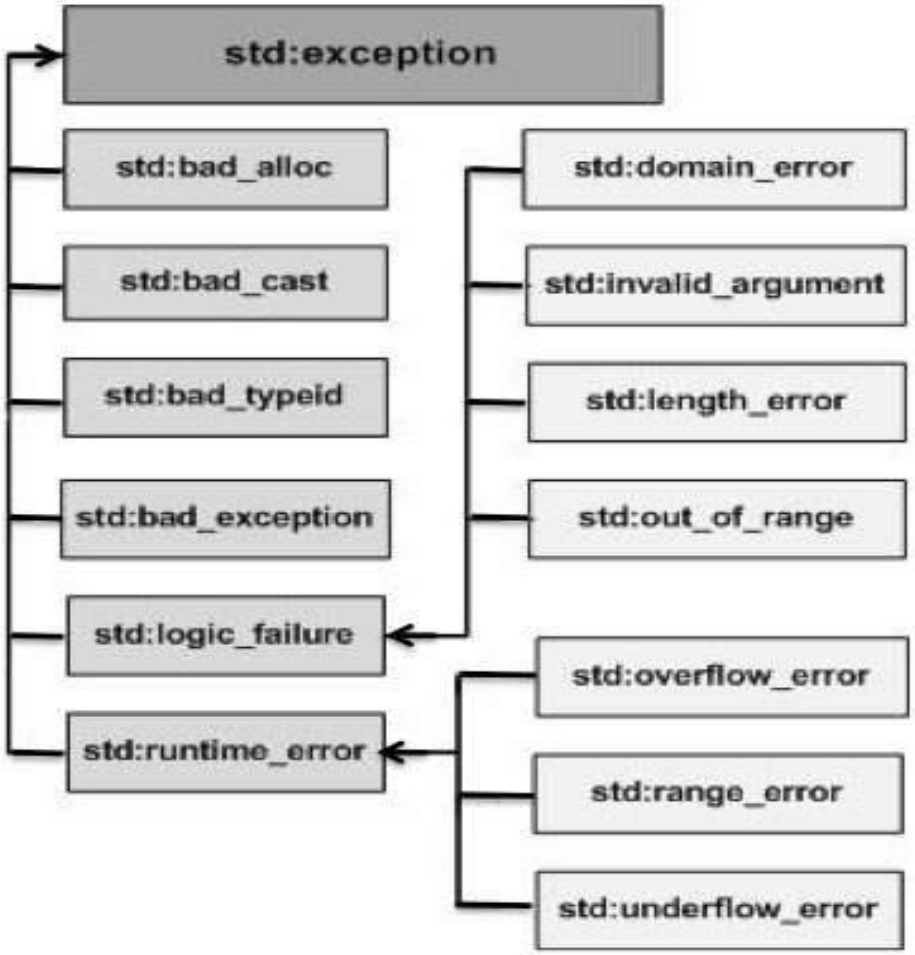
Exception Handling

In the above program, when f1() throws exception, its entry is removed from the function call stack (because it f1() doesn't contain exception handler for the thrown exception), then next entry in call stack is looked for exception handler. The next entry is f2(). Since f2() also doesn't have handler, its entry is also removed from function call stack. The next entry in function call stack is f3(). Since f3() contains exception handler, the catch block inside f3() is executed, and finally the code after catch block is executed. Note that the following lines inside f1() and f2() are not executed at all.

Exception Handling

C++ Standard Exception:

C++ provides a list of standard exceptions defined in <exception> which we can use in our programs. These are arranged in a parent-child class hierarchy shown below:



Exception Handling

Exception	Description
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.
std::underflow_error	This is thrown if a mathematical underflow occurs.

Exception Handling

These two new classes serve as bases for two families of derived classes.

- logic_error
- runtime-error

The logic_error family describes typical logic errors. In principle, sound programming could avoid such errors, but in practice, such errors might show up.

The **invalid_argument** exception alerts us that an unexpected value has been passed to a function. For example, if a function expects to receive a string for which each character is either a '0' or '1', it could throw the invalid_argument exception if some other character appeared in the string.
The **length_error** exception is used to indicate that not enough space is available for the desired action. For example, the string class has an append() method that throws a length_error exception if the resulting string would be larger than the maximum possible string length.
The **out_of_bounds** exception is typically used to indicate indexing errors. For example, we could define an array-like class for which operator()[] throws the out_of_bounds exception if the index used is invalid for that array.

Exception Handling

the runtime_error show up during runtime but that could not easily be predicted and prevented.

An **underflow_error** can occur in floating-point calculations. In general, there is a smallest nonzero magnitude that a floating-point type can represent. A calculation that would produce a smaller value would cause an underflow error.
An **overflow_error** can occur with either integer or floating-point types when the magnitude of the result of a calculation would exceed the largest representable value for that type.
The **range_error** exception can occur in the situations when computational result can lie outside the valid range of a function without being an underflow or overflow.

Exception Handling

Exmaple of bad_alloc standard exception:

```
#include <iostream>
using namespace std;
int main ()
{
        try
        {
                int* myarray= new int[100000000000];
        }
        catch (exception& e)
        {
                cout << "Standard exception: " << e.what() << endl;
        }
return 0;
}
```

Exception Handling

Example of length_error exception:

```
#include <iostream>
#include <stdexcept>      // std::length_error
#include <vector>         // std::vector
using namespace std;
int main ()
{
  try
  {
    // vector throws a length_error if resized above max_size
            vector<int> myvector;
    myvector.resize(myvector.max_size()+1);
  }
  catch (length_error& le)
  {
            cout << "Length error: " << le.what() << '\n';
  }
}
```

Exception Handling

User defined Exception:

We can define our own exceptions by inheriting and overriding exception class functionality. It is called std::exception and is defined in the <exception> header. This class has a virtual member function called what that returns a null-terminated character sequence (of type char *) and that can be overwritten in derived classes to contain some sort of description of the exception.

Following is the example, which shows how we can use std::exception class to implement our own exception in standard way:

Here, 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.

Example:

```
#include <iostream>
//#include <exception>
using namespace std;
class MyException : public exception
{
  public:
  const char * what ()
  {
    return "C++ Exception";
  }
};
```

Exception Handling

```
int main()
{
  try
  {
    throw MyException();
  }
  catch(MyException& e)
  {
    cout << "MyException caught" << std::endl;
    cout << e.what() << std::endl;
  }
  catch(exception& e)
  {
    //Other errors
  }
}
```

Output:

MyException caught

C++ Exception

Exception Handling

Hierarchy of Exceptions:
If we have an inheritance hierarchy of exception classes, we should arrange the order of the catch blocks so that the exception of the most-derived class (that is, the class furthest down the class hierarchy sequence) is caught first and the base-class exception is caught last.
Arranging catch blocks in the proper sequence allows us to be selective about how each type of exception is handled.

A base-class reference can catch all objects of a family, but a derived-class object can only catch that object and objects of classes derived from that class. This suggests arranging the catch blocks in inverse order of derivation.

Example:

```
class bad_1 {...};
class bad_2 : public bad_1 {...};
class bad_3 : public bad 2 {...};
...
void check()     // matches base- and derived-class objects
{
...
if (condition1)
throw bad_1();
if (condition2)
throw bad_2();
if (condition3)
throw bad_3();
}
```

Exception Handling

```
…
try
{
check();
}
catch(bad_3 &be)
{ // statements }
catch(bad_2 &be)
{ // statements }
catch(bad_1 &be)
{ // statements }
```

If the bad_1 & handler were first, it would catch the bad_1, bad_2, and bad_3 exceptions.
With the inverse order, a bad_3 exception would be caught by the bad_3 & handler.

Exception Handling

```
#include <iostream>
#include <exception>
using namespace std;
class A : public exception
{  public:
  const char * what ()
  {
    return "Invalid interest rate";
   } };
class B :public A
{  public:
  const char * what ()
  {
    return "We can't give this much interest";
   } };
class C :public B
{  public:
  const char * what ()
  {
    return "Interest rate can't be in -ve value";
   } };
```

Exception Handling

```
void check()
{ int rate;
   cout<<"enter interest rate";
   cin>>rate;
   if(rate>10)
   throw B();
   if(rate<0)
   throw C();
   if(rate<0 || rate>=10)
   throw A();}
int main()
{
  try
  {
   check();
  }
catch(C& e)
  {
    cout << "MyException caught1" <<endl;
    cout << e.what() <<endl;
  }
```

Exception Handling

```
catch(B& e)
  {
    cout << "MyException caught2" <<endl;
    cout << e.what() <<endl;
  }
  catch(A& e)
  {
    cout << "MyException caught3" <<endl;
    cout << e.what() <<endl;
  }
  catch(std::exception& e)
  {
    //Other errors
    cout<<"catching other types of errors";
  }
}
```

Discussions