Secure Coding

6EXCEPTION HANDLING

- 12.1 Exception Handling Fundamentals
- 12.2 Handing Derived-Class Exceptions
- 12.3 Exception Handling Options
- 12.4 Understanding terminate() and unexpected()
- 12.5 Constructors and Destructors in Exception Handling

12.1 Exception Handling Fundamentals

An **exception** is a run-time error caused by some abnormal condition.

In C++, a function f can recognize conditions that identify exceptions and then signal that an exception has occurred.

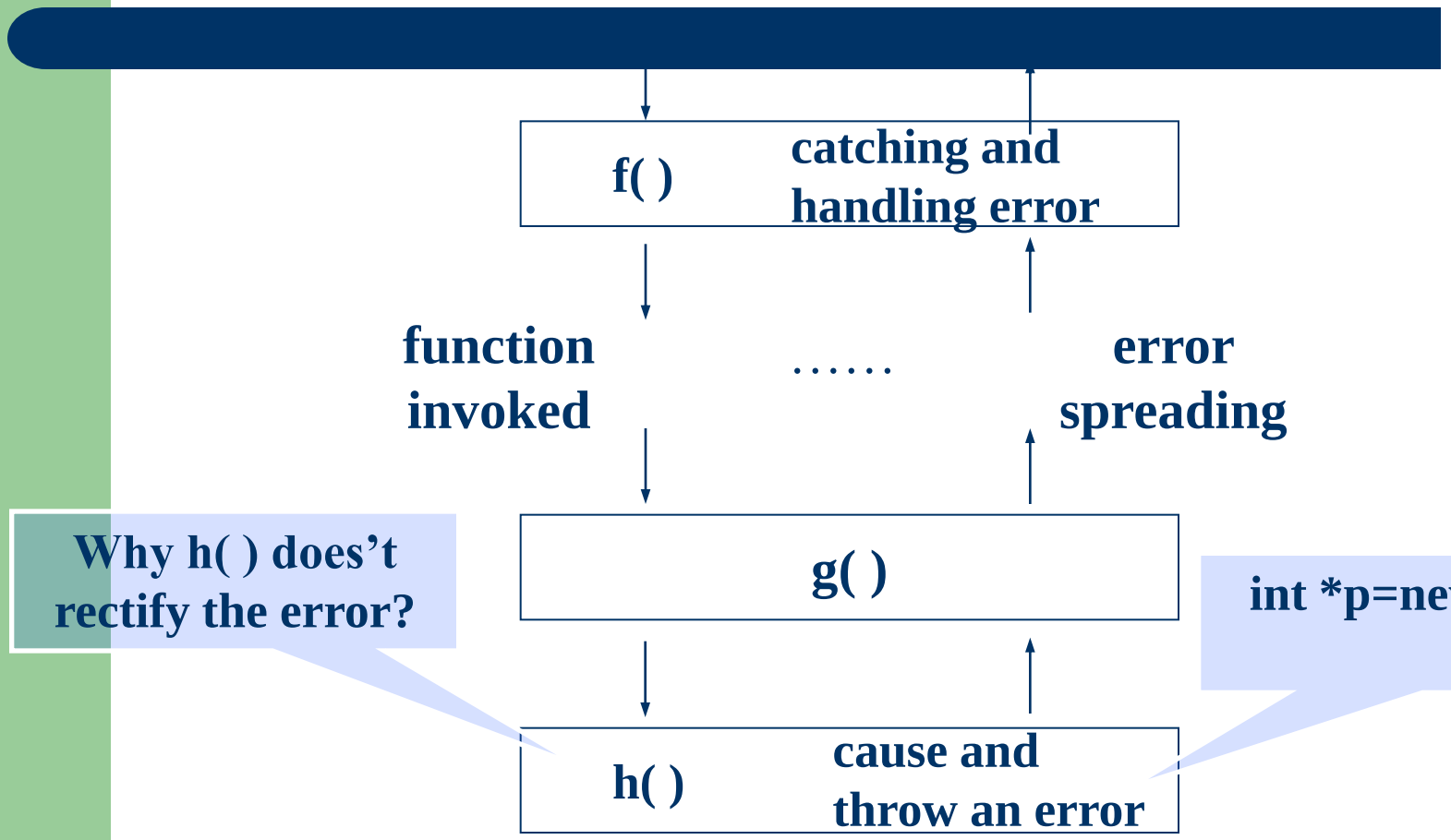
This signaling process is called **throwing an exception**.

12.1 Exception Handling Fundamentals

Once thrown, an exception can be caught or handled by a function that invokes f by using a **catch** block.

A catch block is an **exception handler** that occurs after a try block, which is used to indicate interest in exceptions.

12.1 Exception Handling Fundamentals



12.1 Exception Handling Fundamentals

The "perfect" objective was to map exceptions to some other form of error propagation should a designer choose to do so.

Not that it was always best to do so, but that it could be done.

12.1 Exception Handling Fundamentals

C++ exception handing is built upon three keywords: try, catch, and throw.

12.1 Exception Handling Fundamentals

throwing exception

```
.....
throw    expression;
.....
```

catching and handling exception

```
try
  expression
catch(exception type)
  expression
catch(exception type)
  expression
  .....
```

12.1 Exception Handling Fundamentals

The keyword **throw** creates an object that isn't there under normal program execution.

Then the object is, in effect, "returned" from the function, even though that object type isn't normally what the function is designed to return.

12.1 Exception Handling Fundamentals

Each **catch** clause (exception handler) is like a little function that takes a single argument of one particular type.

12.1 Exception Handling Fundamentals

Remarks:

A) Exception is made and thrown by throw block.

B) The code segment that could cause an exception will be in the try block.

C) The catch block will not invoked if there is not any exception in try block, and the code next to the last catch block will be invoked.

12.1 Exception Handling Fundamentals

D) If an exception is thrown, type-matched **catch block** will catch and handle it.

E) If no catch block is matched, the function **terminate** will be invoked automatically and the program is **aborted**.

12.1 Exception Handling Fundamentals

Example:

```
#include<iostream.h>
int Div(int x,int y);
void main( )
{
    try
    {
        cout<<"5/2="<<Div(5,2)<<endl;
        cout<<"8/0="<<Div(8,0)<<endl;
        cout<<"7/1="<<Div(7,1)<<endl;
    }
    catch(int)
    { cout<<"except of deviding zero.\n"; }
    cout<<"that is ok.\n";
}
```

Only the type of "y" is matter!

```
int Div(int x,int y)
{
    if(y==0)
        throw y;
    return x/y;
}
```

Output:

```
5/2=2
except of deviding zero.
that is ok.
```

12.1 Exception Handling Fundamentals

Usually, the code within a catch statement attempts to remedy an error by taking appropriate action.

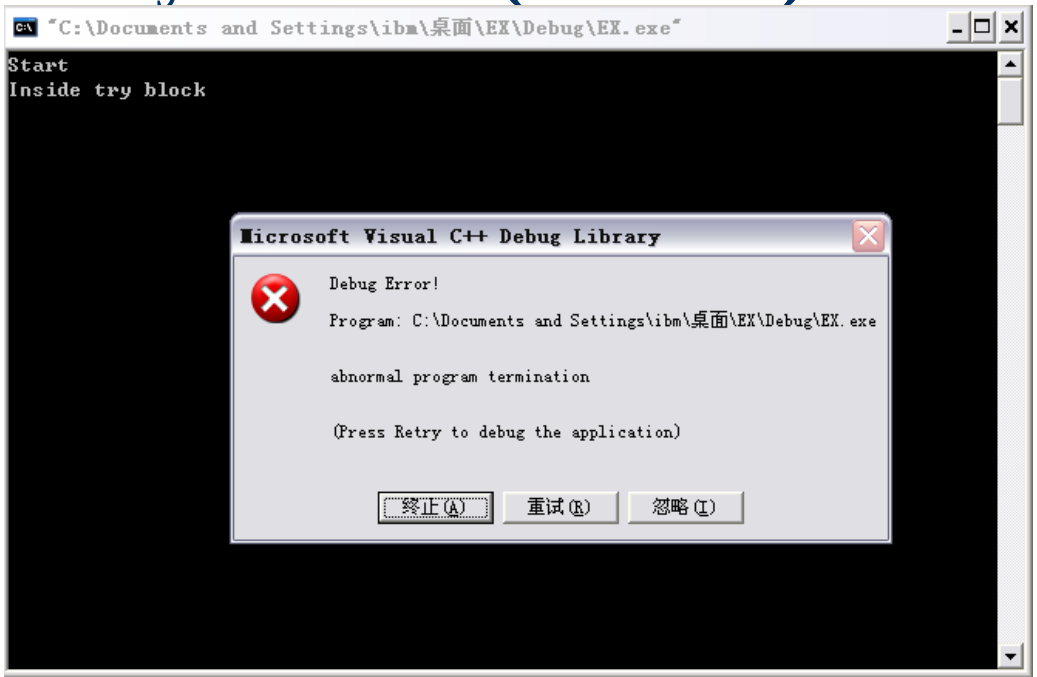
If the error can be fixed, execution will continue with the statements terminate the program with a call to exit( ) or abort( ).

12.1 Exception Handling Fundamentals

```
#include <iostream>
using namespace std;
int main( )
{
      cout<<"Start"<<endl;
      try      //start a try block
      {
            cout<<"Inside try block"<<endl;
            throw 100;
            cout<<"This will not execute";
      }
      catch(double i)    //won't work for an int exception
      {
            cout<<"Caught an exception—value is: ";
            cout<<i <<endl;
      }
      cout<<"End";
      return 0;
}
```

12.1 Exception Handling Fundamentals

This program produces the following output because the integer exception will not be caught by the catch(double i) statement.



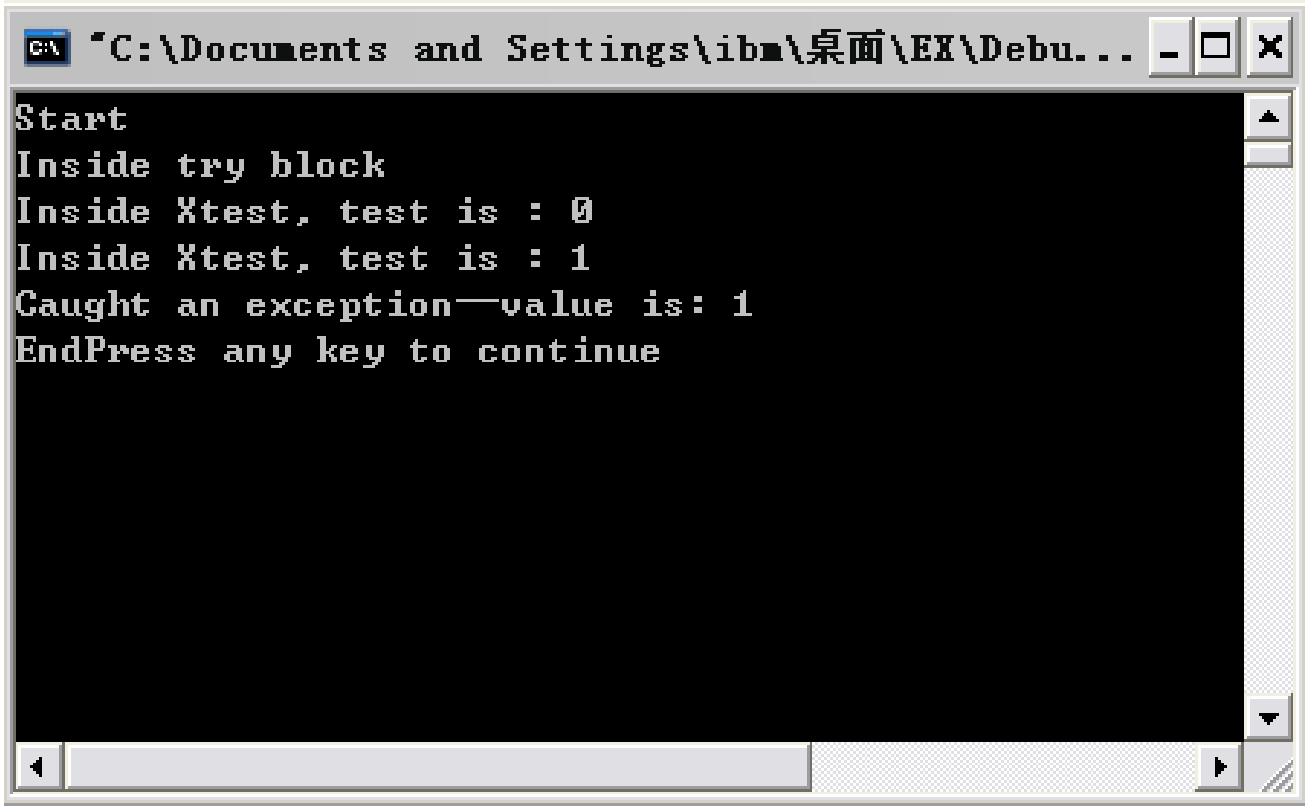
12.1 Exception Handling Fundamentals

An exception can be thrown from outside the try block as long as it is thrown by a function that is called from within try block.

```
#include <iostream>
using namespace std;
void Xtest(int test)
{
    cout<<"Inside Xtest, test is : "
        <<test<<endl;
    if(test)
        throw test;
}

int main( )
{
    cout<<"Start"<<endl;
    try    //start a try block
    {
        cout<<"Inside try block\n";
        Xtest(0);
        Xtest(1);
        Xtest(2);
    }
    catch(int i)      //catch an error
    {
        cout<<"Caught an exception—value is: ";
        cout<<i<<endl;
    }
    cout<<"End";
    return 0;
}
```

12.1 Exception Handling Fundamentals

This program produces the following output:



12.1 Exception Handling Fundamentals

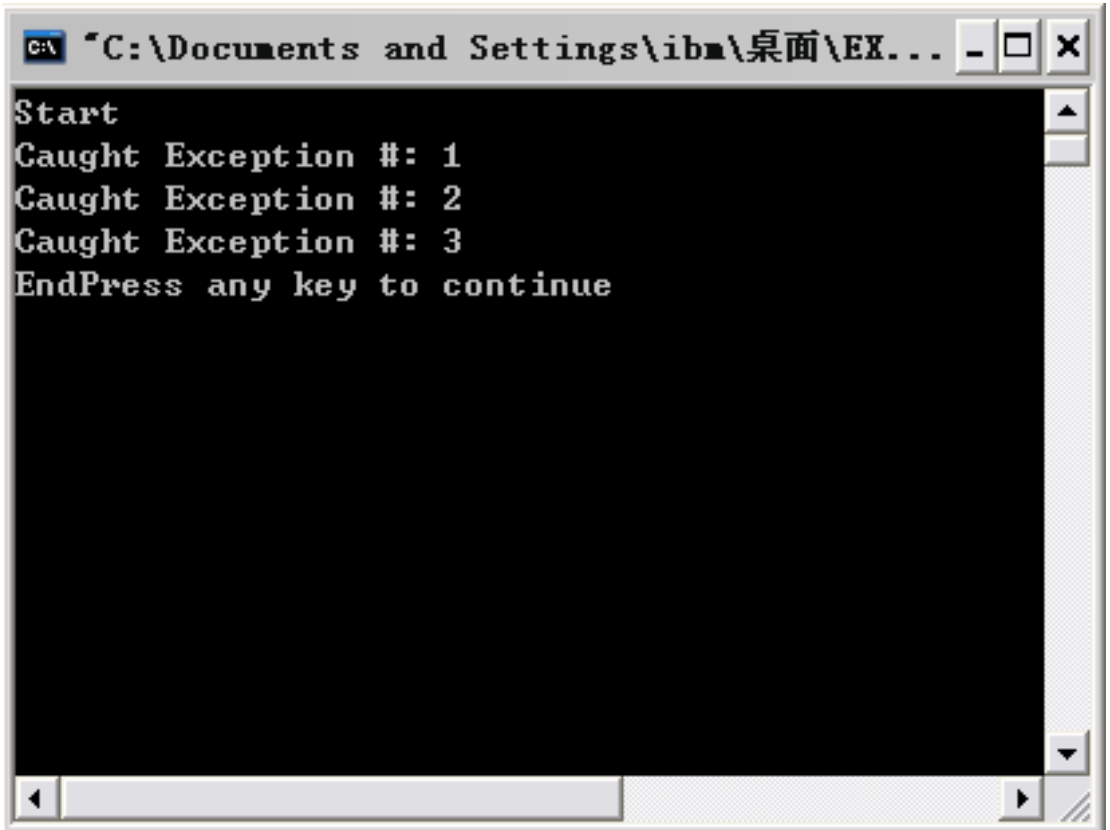
A try block can be localized to a function.

When this is the case, each time the function is entered, the exception handing relative to that function is reset.

```
#include <iostream>
using namespace std;
void Xhander(int test)
{
    try
    {
        if(test)
            throw test;
    }
    catch (int i)
    {
        cout<<"Caught Exception #:"
            <<i<<endl;
    }
}

int main( )
{
    cout<<"Start"<<endl;

    Xhander(1);
    Xhander(2);
    Xhander(0);
    Xhander(3);

    cout<<"End";
    return 0;
}
```

12.1 Exception Handling Fundamentals

This program display this output:



12.1 Exception Handling Fundamentals

As you see, three exceptions are thrown.

After each exception, the function returns. When the function is called again, the exception handling is reset.

12.1 Exception Handling Fundamentals

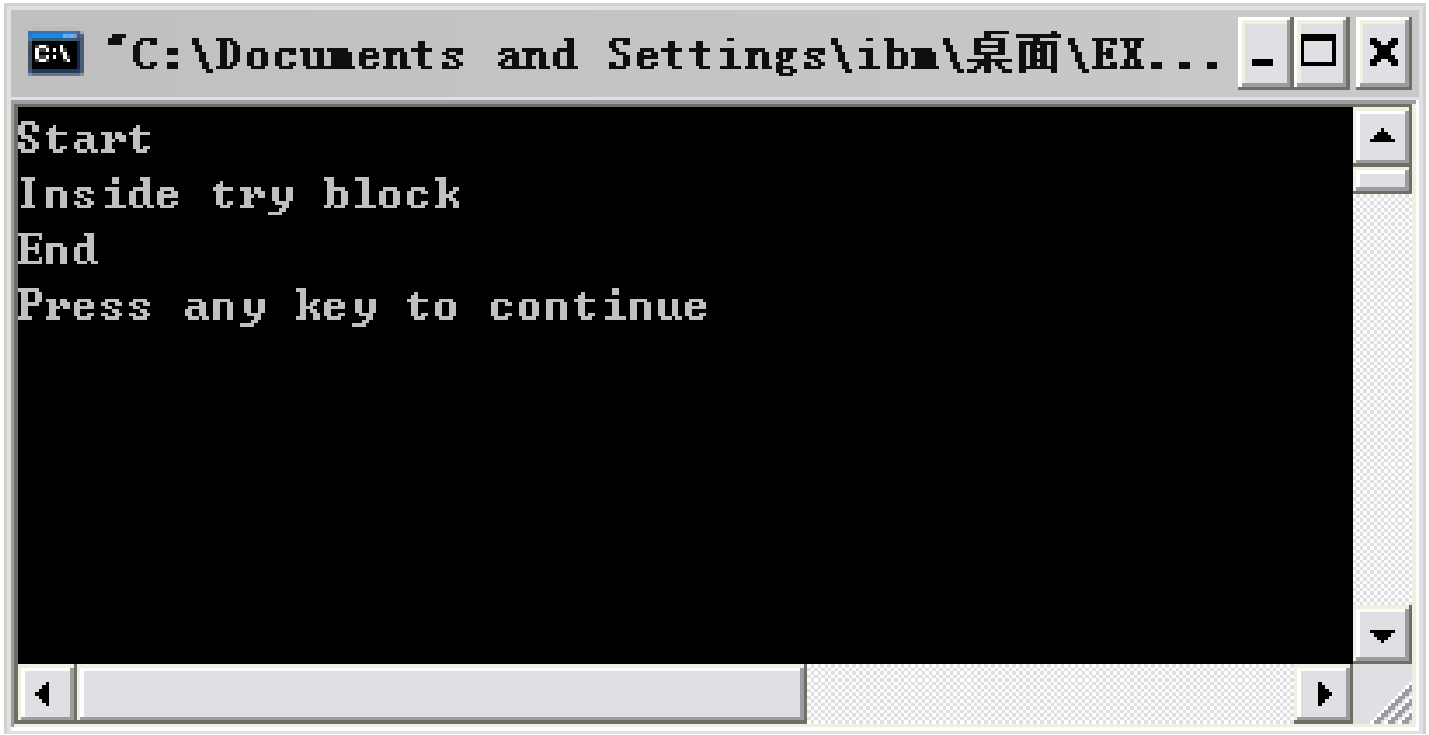
It is important to understand that the code associated with a catch statement will be executed only if it catches an exception.

Otherwise, execution simply bypasses the catch altogether.

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

int main( )
{
    cout<<"Start"<<endl;

    try
    {       //Start a try block
            cout<<"Inside try block"<<endl;
    }
    catch(int i) //catch an error
    {
            cout<<"Caught an exception--value is: ";
            cout<<i<<endl;
    }

    cout<<"End"<<endl;
    return 0;
}
```

12.1 Exception Handling Fundamentals

The preceding program produces the following output:



12.1 Exception Handling Fundamentals

- **Catching Class Types**

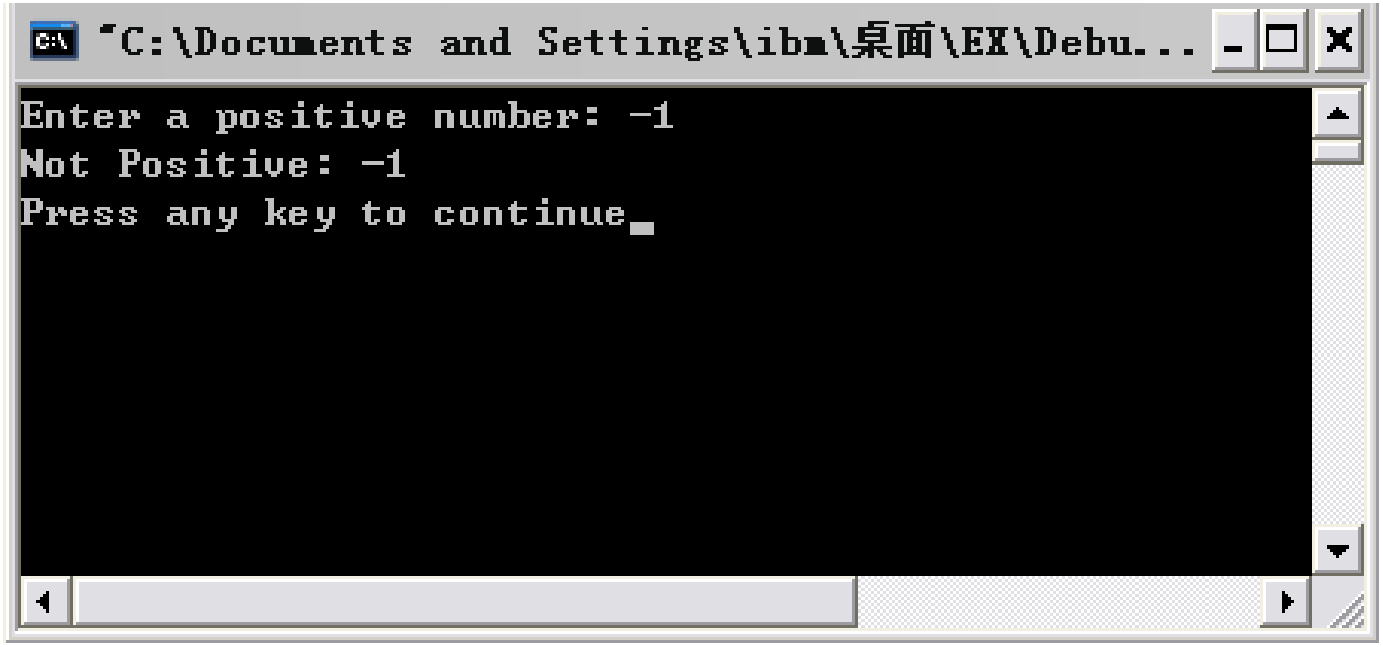
An exception can be of any types, including class types that you create.

Actually, in real-world programs, most exceptions will be class types than built-in types.

```
#include <iostream>
#include <cstring>
using namespace std;
class MyException
{
public:
    char str_what[80];
    int what;
    MyException( )
    {
        *str_what=0;
        what=0;
    }
    MyException(char *s, int e)
    {
        strcpy(str_what,s);
        what=e;
    }
};

int main( )
{
    int i;

    try
    {
        cout<<"Enter a positive number: ";
        cin>>i;
        if(i<0)
            throw MyException("Not Positive", i);
    }
    catch(MyException e)//catch an error
    {
        cout<<e.str_what<<": ";
        cout<<e.what<<endl;
    }

    return 0;
}
```

12.1 Exception Handling Fundamentals

Here is a sample run:



12.1 Exception Handling Fundamentals

- **Using Multiple catch Statements**

As stated, you can have more than one catch associated with a try. In fact, it is common to do so.

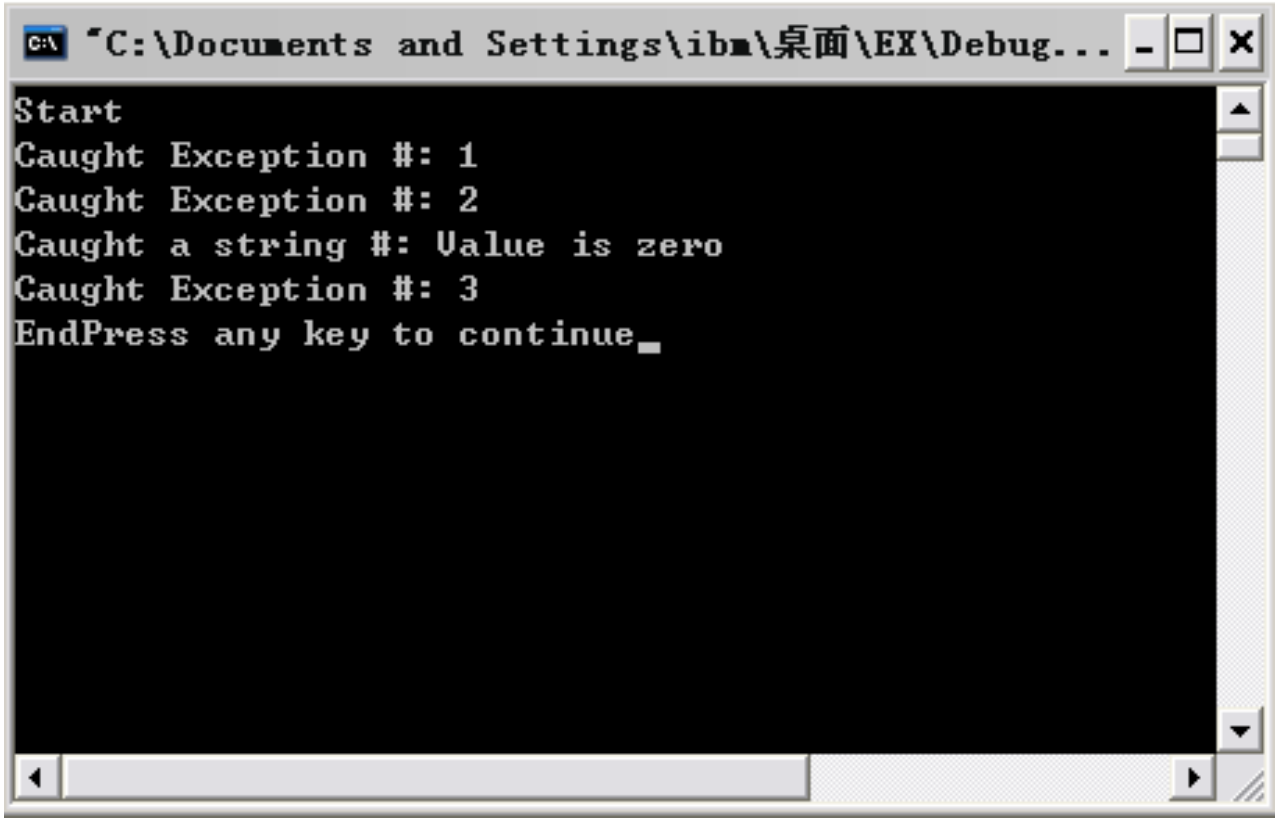
However, each catch must catch a different type of exception.

```
#include <iostream>
using namespace std;
void Xhandler(int test)
{
    try
    {
        if(test)
            throw test;
        else
            throw "Value is zero";
    }
    catch (int i)
    {
        cout<<"Caught Exception #: "
            <<i<<endl;
    }
    catch (const char *str)
    {
        cout<<"Caught a string #: ";
        cout<<str<<endl;
    }
}
```

```
int main( )
{
    cout<<"Start"<<endl;

    Xhandler(1);
    Xhandler(2);
    Xhandler(0);
    Xhandler(3);

    cout<<"End";
    return 0;
}
```

12.1 Exception Handling Fundamentals

This program produces the following output:



12.2 Handing Derived-Class Exceptions

You need to be careful how you order catch statements when trying to catch exception types that involve base and derived class because a catch clause for a **base class will also match any class derived from that base**.

12.2 Handing Derived-Class Exceptions

Thus, if you want to catch exceptions of both a base class type and a derived class type, put the derived class first in the catch sequence.

If you don't do this, the base class catch will also catch all derived classes.

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

class B
{
};
class D: public B
{
};

int main( )
{
    D derived;

    try
    {
        throw derived;
    }
    catch(B b)
    {
        cout<<"Caught a base class."<<endl;
    }
    catch(D d)
    {
        cout<<"This won't execute."<<endl;
    }

    return 0;
}
```

12.2 Handing Derived-Class Exceptions

Here, because derived is an object that has B as a base class, it will be caught by the first catch clause and the second clause will never execute.

12.3 Exception Handling Options

- **Catching All Exceptions**

In some circumstances you will want an exception handler to catch all exceptions instead of just a certain type.

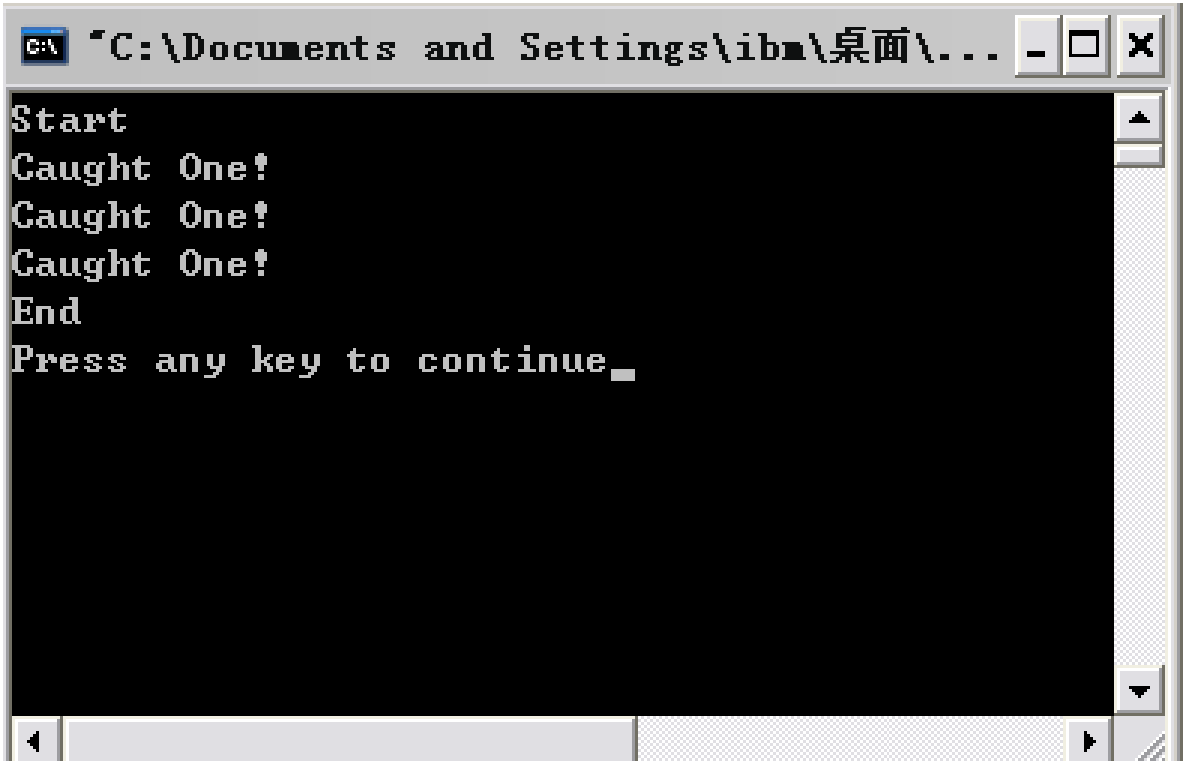
```
catch(...)
{
   //process all exceptions
}
```

```
#include <iostream>
using namespace std;
//use catch(…) as a default
void Xhandler(int test)
{
    try
    {
        if(test==0) throw test;        //throw int
        if(test==1) throw 'a';         //throw char
        if(test==2) throw 123.12;      //throw double
    }
    catch(...)        //catch all exception
    {
        cout<<"Caught One!"<<endl;
    }
}
```

```
int main( )
{
    cout<<"Start"<<endl;

    Xhandler(0);
    Xhandler(1);
    Xhandler(2);

    cout<<"End"<<endl;
    return 0;
}
```

12.3 Exception Handling Options

This program displays the following output:



12.3 Exception Handling Options

- **Restricting Exceptions**

You can restrict the type of exceptions that a function can throw out side of itself.

In fact, you can also prevent a function from throwing any exceptions whatsoever.

12.3 Exception Handling Options

To accomplish these restrictions, you must add a throw clause to a function definition.

The general form of this is shown here:

```
ret-type func-name (arg-list) throw (type-list)
{
   //…
}
```

12.3 Exception Handling Options

Here, only those data types contained in the type-list may be thrown by the function.

Throwing any other type of expression will cause abnormal program termination.

If you don’t want a function to be able to throw any exceptions, then use an empty list.

12.3 Exception Handling Options

You can list all types of exceptions in the definition of function.

Example:

```
void fun( ) throw (A，B，C，D);
```

You can also define a function that can not throw any type of exceptions.

Example:

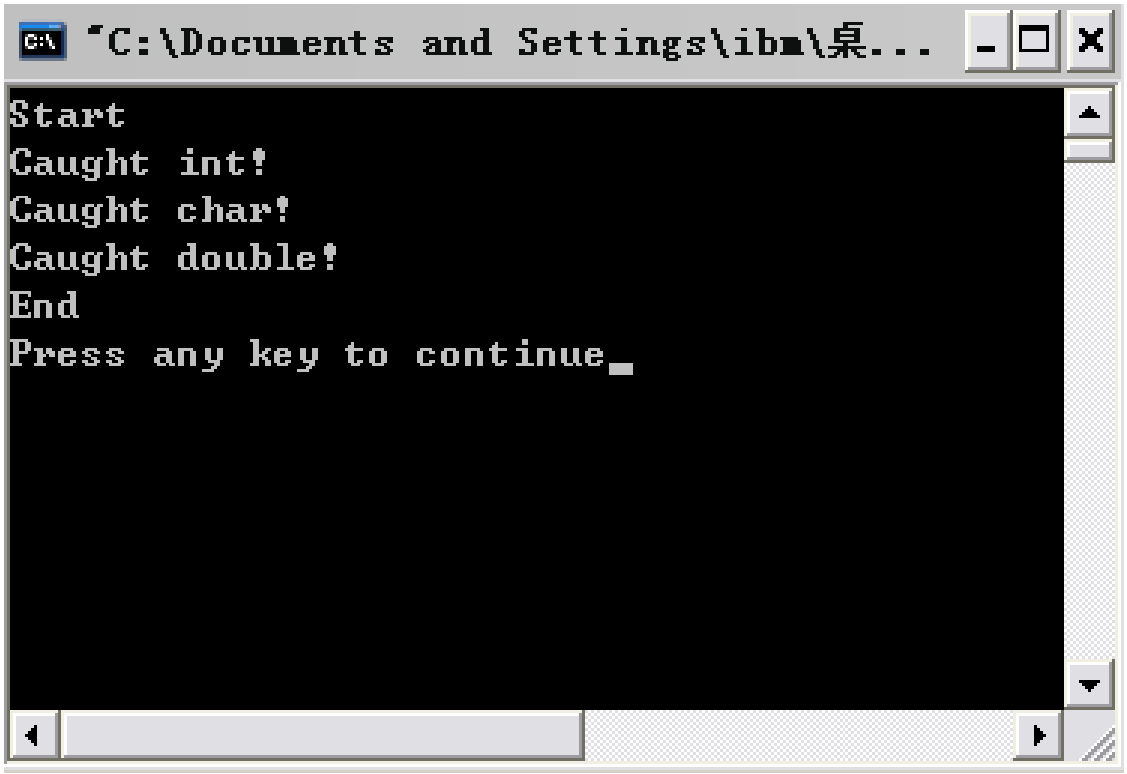
```
void fun( ) throw( );
```

```
#include <iostream>
using namespace std;
//This function can only throw ints, chars and doubles.
void Xhandler(int test) throw( int, char, double )
{
    try
    {
        if(test==0) throw test;        //throw int
        if(test==1) throw 'a';         //throw char
        if(test==2) throw 123.12;      //throw double
    }
    catch(int)
    {
        cout<<"Caught int!"<<endl;
    }
    catch(char)
    {
        cout<<"Caught char!"<<endl;
    }
    catch(double)
    {
        cout<<"Caught double!"<<endl;
    }
}
```

```
int main( )
{
    cout<<"Start"<<endl;

    Xhandler(0);
    Xhandler(1);
    Xhandler(2);

    cout<<"End"<<endl;
    return 0;
}
```

12.3 Exception Handling Options

Here is the result:



12.3 Exception Handling Options

In this program, if it attempts to throw any other type of exception, an abnormal program termination will occur.

12.3 Exception Handling Options

It is important to understand that a function can be restricted only in what types of exceptions it throws back to the try block that called it.

That is, a try block within a function may throw any type of exception so long as it is caught within that function.

12.4 Understanding terminate( ) and unexpected( )

As mentioned, terminate( ) and unexpected( ) are called when something goes wrong during the exception handling process.

```
#include <exception>
void terminate( );
void unexpected( );
```

By default, terminate( ) calls abort( ).

12.4 Understanding terminate( ) and unexpected( )

- **Setting the Terminate and Unexpected Handlers**

The terminate( ) and unexpected( ) functions simply call other functions to actually handle an error.

By default terminate( ) calls abort( ), and unexpected( ) calls terminate( ).

12.4 Understanding terminate( ) and unexpected( )

To change the terminate handler, use set_terminate( ) and set_unexpected( ).

```
#include <iostream>
#include <exception>
using namespace std;
void my_Thandler( )
{
    cout<<"Inside new terminate handler"<<endl;
    abort( );
}

int main( )
{
    //Set a new terminate handler
    set_terminate(my_Thandler);
    try
    {
        cout<<"Inside try block"<<endl;
        throw 100;
    }
    catch(double i)
    {
    }
    return 0;
}
```

12.4 Understanding terminate( ) and unexpected( )

The output from this program is shown here.

12.5 Constructors and Destructors in Exception Handling

The C++ Spirit:" Trust the programmer".

An object comes into being only after it as been constructed.

12.5 Constructors and Destructors in Exception Handling

If an exception is matched by the catch block, some thing will be done to handle it.

A) Initializing the arguments.

B) All the objects that have been constructed in the try block will be destructed automatically.

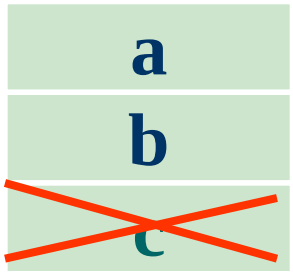
C) The code segment next to the last catch block will be invoked.

12.5 Constructors and Destructors in Exception Handling

Example:

```
class X
{
public:
    A   a;
    B   b;
    C   c;
};
```

```
void main( )
{
    X x;
}
```



12.5 Constructors and Destructors in Exception Handling

Example:

```
#include <iostream.h>
void MyFunc( void );
class Expt
{
public:
   Expt( ){};
   ~Expt( ){};
   const char *ShowReason( ) const
   {
     return "Expt";
   }
};
```

12.5 Constructors and Destructors in Exception Handling

```
class Demo
{
public:
     Demo( );
     ~Demo( );
};
Demo::Demo( )
{
    cout<<"Constructing Demo."<<endl;
}
Demo::~Demo( )
{
    cout<<"Destructing Demo."<<endl;
}
```

12.5 Constructors and Destructors in Exception Handling

```
void MyFunc( )
{
    Demo D;
    cout<<"Throw Expt in MyFunc() "<<endl;
    throw Expt( );
}
int main( )
{
    cout<<"In main function "<<endl;
    try
    {
        cout<<"In try block, MyFunc( ) is invoked" <<endl;
        MyFunc( );
    }
```

There will be some errors.

12.5 Constructors and Destructors in Exception Handling

```
catch( Expt E )
{
        cout<<"In catch block "<<endl;
        cout<<"Catching the Expt  ";
        cout<<E.ShowReason()<<endl;
}
catch( char *str )
{
        cout<<"Catching the other exception：  "<<str<<endl;
}
cout<<"Back to main function" <<endl;
return 0;
}
```

12.5 Constructors and Destructors in Exception Handling

Output:

```
In main function
In try block, MyFunc() is invoked
Constructing Demo.
Throw Expt in MyFunc()
Destructing Demo.
In catch block
Catching the Expt  Expt
Back to main function
```

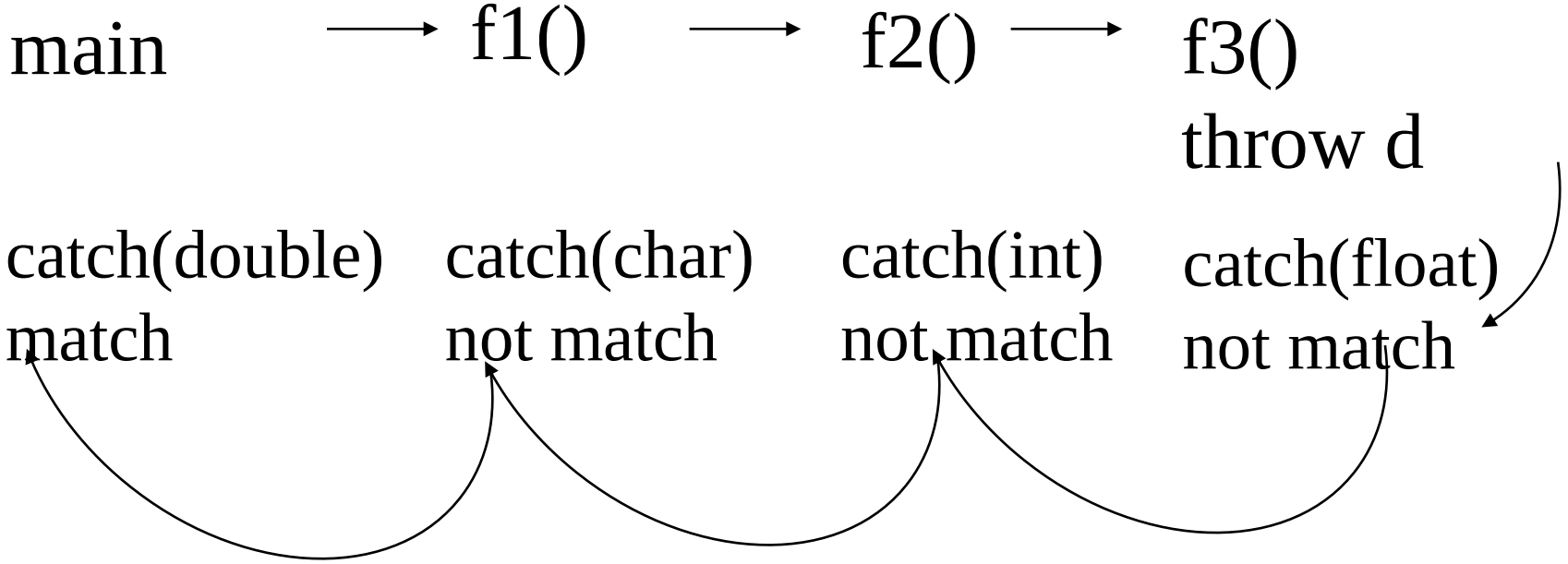
Exercise

```
#include <iostream.h>
int main( )
{
    void f1( );
    try
        {f1( );}
    catch(double)
        {cout<<“OK0!”<<endl;}
    cout<<“end0”<<endl;l
    return 0;
}
```

```
void f1( )
{
    void f2( );
    try
        {f2( );}
    catch(char)
        {cout<<“OK1!”<<endl;}
    cout<<“end1”<<endl;l
}
```

```
void f2( )
{
    void f3( );
    try
        {f3( );}
    catch(int)
        {cout<<“OK2!”<<endl;}
    cout<<“end2”<<endl;l
}
```

```
void f3( )
{
    double d=0;
    try
        {throw d;}
    catch(float)
        {cout<<“OK3!”<<endl;}
    cout<<“end3”<<endl;l
}
```



Output：

OK0!

end0

What will happen if function f3's argument type of catch turns to be double?

OK3!

end3

end2

end1

end0

Summarize

- 12.1 Exception Handling Fundamentals
- 12.2 Handing Derived-Class Exceptions
- 12.3 Exception Handling Options
- 12.4 Understanding terminate() and unexpected()
- 12.5 Constructors and Destructors in Exception Handling