Exception Handling

Sai-Keung Wong

National Yang Ming Chiao Tung University

Hsinchu, Taiwan

Intended Learning Outcomes

- Describe the process for exception handling
- Define a try-catch block
- Implement exception handling in a simple program

When to Use Exceptions

- An exception is a problem that arises during a program execution.

- Common exceptions that may occur in multiple classes are candidates for exception classes.

- Simple errors that may occur in individual functions are best handled locally without throwing exceptions.

Exception-Handling Overview

The following program reads two integers and displays the quotient.

```
int quotient( int n1, int n2 ) {

  return n1 / n2;
}

int main( ) {
  cout << "Input two integers: "; int n1, n2;
  cin >> n1>> n2; // Read two integers

  int result = quotient( n1, n2 );
  cout << n1 << " / " << n2 << " is “ << result << endl;



  return 0;
}
```

Exception-Handling Overview

The following program reads two integers and displays the quotient.

```
int quotient( int n1, int n2 ) {

  return n1 / n2;
}

int main( ) {
  cout << "Input two integers: "; int n1, n2;
  cin >> n1>> n2; // Read two integers

  int result = quotient( n1, n2 );
  cout << n1 << " / " << n2 << " is “ << result << endl;



  return 0;
}
```

- We will have a runtime error if n2 is zero.

- To fix the error, we can add an if statement to test seconds.

Exception-Handling Overview

The following program reads two integers and displays the quotient.

```
int quotient( int n1, int n2 ) {

  return n1 / n2;
}

int main( ) {
  cout << "Input two integers: "; int n1, n2;
  cin >> n1>> n2; // Read two integers
  if ( n2 == 0 ) { cout <<  “Division by zero" << endl; return 0; }
  int result = quotient( n1, n2 );
  cout << n1 << " / " << n2 << " is “ << result << endl;



  return 0;
}
```

- We will have a runtime error if n2 is zero.
- To fix the error, we can add an if statement to test seconds.

Exception-Handling Overview

The following program reads two integers and displays the quotient.

We can use exception handling.

```
int quotient( int n1, int n2 ) {

  return n1 / n2;
}

int main( ) {
  cout << "Input two integers: "; int n1, n2;
  cin >> n1>> n2; // Read two integers
  if ( n2 == 0 ) { cout <<  “Division by zero" << endl; return 0; }
  int result = quotient( n1, n2 );
  cout << n1 << " / " << n2 << " is “ << result << endl;



  return 0;
}
```

Exception-Handling Overview

The following program reads two integers and displays the quotient.

```
int quotient( int n1, int n2 ) {
  if ( n2== 0 )  throw number1;
  return n1 / n2;
}

int main( ) {
  cout << "Input two integers: "; int n1, n2;
  cin >> n1>> n2; // Read two integers
  {if ( n2 == 0 ) { cout <<  “Division by zero" << endl; return 0; }try
  int result = quotient( n1, n2 );
  cout << n1 << " / " << n2 << " is “ << result << endl;
  } catch (int ex) {
    cout << "Exception: Division by zero" << endl;
  }
  return 0;
}
```

We can use exception handling.

```
int quotient( int n1, int n2 ) {

  return n1 / n2;
}

int main( ) {
  cout << "Input two integers: "; int n1, n2;
  cin >> n1>> n2; // Read two integers

     int result = quotient( n1, n2 );
     cout << n1 << " / " << n2 << " is “ << result << endl;



  return 0;
}
```

Exception-Handling Overview

The following program reads two integers and displays the quotient.

```
int quotient( int n1, int n2 ) {
  if ( n2== 0 )  throw number1;
  return n1 / n2;
}

int main( ) {
  cout << "Input two integers: "; int n1, n2;
  cin >> n1>> n2; // Read two integers
  {if ( n2 == 0 ) { cout <<  “Division by zero" << endl; return 0; }try
  int result = quotient( n1, n2 );
  cout << n1 << " / " << n2 << " is “ << result << endl;
  } catch (int ex) {
    cout << "Exception: Division by zero" << endl;
  }
  return 0;
}
```

We can use exception handling.

```
int quotient( int n1, int n2 ) {
  if ( n2== 0 )  throw n2;
  return n1 / n2;
}

int main( ) {
  cout << "Input two integers: "; int n1, n2;
  cin >> n1>> n2; // Read two integers

    int result = quotient( n1, n2 );
    cout << n1 << " / " << n2 << " is “ << result << endl;



  return 0;
}
```

Exception-Handling Overview

```
int quotient( int n1, int n2 ) {
  if ( n2== 0 )  throw number1;
  return n1 / n2;
}

int main( ) {
  cout << "Input two integers: "; int n1, n2;
  cin >> n1>> n2; // Read two integers
  {if ( n2 == 0 ) { cout <<  “Division by zero" << endl; return 0; }try
  int result = quotient( n1, n2 );
  cout << n1 << " / " << n2 << " is “ << result << endl;
  } catch (int ex) {
    cout << "Exception: Division by zero" << endl;
  }
  return 0;
}
```

We can use exception handling.

```
int quotient( int n1, int n2 ) {
  if ( n2== 0 )  throw n2;
  return n1 / n2;
}

int main( ) {
  cout << "Input two integers: "; int n1, n2;
  cin >> n1>> n2; // Read two integers
  try {
    int result = quotient( n1, n2 );
    cout << n1 << " / " << n2 << " is “ << result << endl;
  }


  return 0;
}
```

Exception-Handling Overview

```
int quotient( int n1, int n2 ) {
  if ( n2== 0 )  throw number1;
  return n1 / n2;
}

int main( ) {
  cout << "Input two integers: "; int n1, n2;
  cin >> n1>> n2; // Read two integers
  {if ( n2 == 0 ) { cout <<  “Division by zero" << endl; return 0; }try
  int result = quotient( n1, n2 );
  cout << n1 << " / " << n2 << " is “ << result << endl;
  } catch (int ex) {
    cout << "Exception: Division by zero" << endl;
  }
  return 0;
}
```

We can use exception handling.

```
int quotient( int n1, int n2 ) {
  if ( n2== 0 )  throw n2;
  return n1 / n2;
}

int main( ) {
  cout << "Input two integers: "; int n1, n2;
  cin >> n1>> n2; // Read two integers
  try {
     int result = quotient( n1, n2 );
     cout << n1 << " / " << n2 << " is “ << result << endl;
  } catch (int ex) {
    cout << "Exception: Division by zero" << endl;
  }
  return 0;
}
```

Exception-Handling Overview

```
int quotient( int n1, int n2 ) {
  if ( n2== 0 )  throw number1;
  return n1 / n2;
}

int main( ) {
  cout << "Input two integers: "; int n1, n2;
  cin >> n1>> n2; // Read two integers
  {if ( n2 == 0 ) { cout <<  “Division by zero" << endl; return 0; }try
  int result = quotient( n1, n2 );
  cout << n1 << " / " << n2 << " is “ << result << endl;
  } catch (int ex) {
    cout << "Exception: Division by zero" << endl;
  }
  return 0;
}
```

We can use exception handling.

```
int quotient( int n1, int n2 ) {
  if ( n2== 0 )  throw n2;
  return n1 / n2;
}

int main( ) {
  cout << "Input two integers: "; int n1, n2;
  cin >> n1>> n2; // Read two integers
  try {
     int result = quotient( n1, n2 );
     cout << n1 << " / " << n2 << " is “ << result << endl;
  } catch (int ex) {
    cout << "Exception: Division by zero" << endl;
  }
  return 0;
}
```

Exception-Handling Overview

The following program reads two integers and displays the quotient.

```
int quotient( int n1, int n2 ) {

  return n1 / n2;
}

int main( ) {
  cout << "Input two integers: "; int n1, n2;
  cin >> n1>> n2; // Read two integers
  if ( n2 == 0 ) { cout <<  “Division by zero" << endl; return 0; }
  int result = quotient( n1, n2 );
  cout << n1 << " / " << n2 << " is “ << result << endl;



  return 0;
}
```

We can use exception handling.

```
int quotient( int n1, int n2 ) {
  if ( n2== 0 )  throw n2;
  return n1 / n2;
}

int main( ) {
  cout << "Input two integers: "; int n1, n2;
  cin >> n1>> n2; // Read two integers
  try {
     int result = quotient( n1, n2 );
     cout << n1 << " / " << n2 << " is “ << result << endl;
  } catch (int ex) {
    cout << "Exception: Division by zero" << endl;
  }
  return 0;
}
```

Exception-Handling Overview

The following program reads two integers and displays the quotient.

```
int quotient( int n1, int n2 ) {

  return n1 / n2;
}

int main( ) {
  cout << "Input two integers: "; int n1, n2;
  cin >> n1>> n2; // Read two integers
  if ( n2 == 0 ) { cout <<  “Division by zero" << endl; return 0; }
  int result = quotient( n1, n2 );
  cout << n1 << " / " << n2 << " is “ << result << endl;



  return 0;
}
```

We can use exception handling.

```
int quotient( int n1, int n2 ) {
  if ( n2== 0 )  throw n2;
  return n1 / n2;
}

int main( ) {
  cout << "Input two integers: "; int n1, n2;
  cin >> n1>> n2; // Read two integers
  try {
     int result = quotient( n1, n2 );
     cout << n1 << " / " << n2 << " is “ << result << endl;
  } catch (int ex) {
    cout << "Exception: Division by zero" << endl;
  }
  return 0;
}
```

Exception-Handling Overview

The **try-catch block** syntax

```
try {





}
catch (type e) {


}
```

```
int quotient( int n1, int n2 ) {
  if ( n2== 0 )  throw n2;
  return n1 / n2;
}

int main( ) {
  cout << "Input two integers: "; int n1, n2;
  cin >> n1>> n2; // Read two integers
  try {
     int result = quotient( n1, n2 );
     cout << n1 << " / " << n2 << " is " << result << endl;
  } catch (int ex) {
    cout << "Exception: Division by zero" << endl;
  }
  return 0;
}
```

Exception-Handling Overview

The **try-catch block** syntax

```
try {
   Execute instructions if possible;
   Throw an exception or from a function if necessary;
   Execute instructions if possible;
}
catch (type e) {
   Process the exception;
}
```

```
int quotient( int n1, int n2 ) {
  if ( n2== 0 )  throw n2;
  return n1 / n2;
}

int main( ) {
  cout << "Input two integers: "; int n1, n2;
  cin >> n1>> n2; // Read two integers
  try {
    int result = quotient( n1, n2 );
    cout << n1 << " / " << n2 << " is “ << result << endl;
  } catch (int ex) {
   cout << "Exception: Division by zero" << endl;
  }
  return 0;
}
```

Exception-Handling Overview

The **try-catch block** syntax

```
try {

   Execute instructions if possible;

   Throw an exception or from a function if necessary;

   Execute instructions if possible;

}
catch (type e) {

   Process the exception;

}
```

```
try
{
  // ...
}
catch ( type e )
{
  cout << "Error occurred “ << endl;
}
```

catch block parameter

```
try
{
  // ...
}
catch ( type )
{
  cout << "Error occurred “ << endl;
}
```

```
try
{
  // ...
}
catch ( type e )
{
  cout << "Error occurred “ << endl;
}
```

catch block parameter

```
try
{
  // ...
}
catch ( type )
{
  cout << "Error occurred " << endl;
}
```

```
try
{
  // ...
}
catch ( type e )
{
  cout << "Error occurred " << endl;
}
```

- If we do not care of the parameter, we can A1 it.
- An exception must be caught A2 Otherwise, A3: What kind of error? error occurs.

catch block parameter

```
try
{
  // ...
}
catch ( type )
{
  cout << "Error occurred “ << endl;
}
```

```
try
{
  // ...
}
catch ( type e )
{
  cout << "Error occurred “ << endl;
}
```

- ➢If we do not care of the parameter, we can omit it.
- ➢An exception must be caught some where. Otherwise, a runtime error occurs.

catch block parameter

```
try
{
  throw (DataType( ) );
}
catch (DataType)
{
  cout << "Error occurred “ << endl;
}
```

```
try
{
  throw (DataType( ) );
}

// no catch block to catch type DataType
// lead to runtime error
```

- If we do not care of the parameter, we can omit it.
- An exception must be caught some where. Otherwise, a runtime error occurs.

Exception-Handling Advantages

```
int quotient( int n1, int n2 ) {

  return n1 / n2;
}

int main( ) {
  cout << "Input two integers: "; int n1, n2;
  cin >> n1>> n2; // Read two integers
  if ( n2 == 0 ) { cout <<  “Division by zero" << endl; return 0; }
  int result = quotient( n1, n2 );
  cout << n1 << " / " << n2 << " is “ << result << endl;




  return 0;
}
```

```
int quotient( int n1, int n2 ) {
  if ( n2== 0 )  throw n2;
  return n1 / n2;
}

int main( ) {
  cout << "Input two integers: "; int n1, n2;
  cin >> n1>> n2; // Read two integers
  try {
     int result = quotient( n1, n2 );
     cout << n1 << " / " << n2 << " is “ << result << endl;
  } catch (int ex) {
    cout << "Exception: Division by zero" << endl;
  }
  return 0;
}
```

Exception-Handling Advantages

- Remove error-handling code from the main procedure.

```
int quotient( int n1, int n2 ) {

  return n1 / n2;
}

int main( ) {
  cout << "Input two integers: "; int n1, n2;
  cin >> n1>> n2; // Read two integers
  if ( n2 == 0 ) { cout <<  “Division by zero" << endl; return 0; }
  int result = quotient( n1, n2 );
  cout << n1 << " / " << n2 << " is “ << result << endl;



  return 0;
}
```

```
int quotient( int n1, int n2 ) {
  if ( n2== 0 )  throw n2;
  return n1 / n2;
}

int main( ) {
  cout << "Input two integers: "; int n1, n2;
  cin >> n1>> n2; // Read two integers
  try {
     int result = quotient( n1, n2 );
     cout << n1 << " / " << n2 << " is “ << result << endl;
  } catch (int ex) {
    cout << "Exception: Division by zero" << endl;
  }
  return 0;
}
```

Exception-Handling Advantages

- Remove error-handling code from the main procedure.

```
int quotient( int n1, int n2 ) {
  if ( n2== 0 )  throw n2;
  return n1 / n2;
}
void test( int n1, int n2 ) {
   try {
         cout << “quotient:” << quotient( n1, n2);
   } catch (int ex) {
   …
   }
}
int main( ) {
  …

   test( n1, n2 );

   return 0;
}
```

Exception-Handling Advantages

- Remove error-handling code from the main procedure.
- Can decide to handle certain exceptions and delegate others to the caller.

```
int quotient( int n1, int n2 ) {
  if ( n2== 0 )  throw n2;
  return n1 / n2;
}
void test( int n1, int n2 ) {
   try {
        cout << “quotient:” << quotient( n1, n2);
   } catch (int ex) {
   …
   }
}
int main( ) {
  …


   test( n1, n2 );

   return 0;
}
```

```
int quotient( int n1, int n2 ) {
  if ( n2== 0 )  throw n2;
  return n1 / n2;
}
void test( int n1, int n2 ) {
   try {
        cout << “quotient:” << quotient( n1, n2);
   }
}
int main( ) {
  …
  try {
     test( n1, n2 );
   } catch (int ex) {
   …
   }
  return 0;
}
```

Exception-Handling Advantages

- Remove error-handling code from the main procedure.
- Can decide to handle certain exceptions and delegate others to the caller.
- An exception can be handled anywhere in the function call stack.

```
int quotient( int n1, int n2 ) {
  if ( n2== 0 )  throw n2;
  return n1 / n2;
}
void test( int n1, int n2 ) {
   try {
        cout << “quotient:” << quotient( n1, n2);
   } catch (int ex) {
   …
   }
}
int main( ) {
  …

   test( n1, n2 );

   return 0;
}
```

```
int quotient( int n1, int n2 ) {
  if ( n2== 0 )  throw n2;
  return n1 / n2;
}
void test( int n1, int n2 ) {
   try {
        cout << “quotient:” << quotient( n1, n2);
   }
}
int main( ) {
  …
  try {
     test( n1, n2 );
   } catch (int ex) {
   …
   }
  return 0;
}
```

Exception-Handling Advantages

- Remove error-handling code from the main procedure.
- Can decide to handle certain exceptions and delegate others to the caller.
- An exception can be handled anywhere in the function call stack.

```
int quotient( int n1, int n2 ) {
  if ( n2== 0 )  throw n2;
  return n1 / n2;
}
void test( int n1, int n2 ) {
   try {
        cout << “quotient:” << quotient( n1, n2);
   } catch (int ex) {
   …
   }
}
int main( ) {
  …

   test( n1, n2 );

   return 0;
}
```

```
int quotient( int n1, int n2 ) {
  if ( n2== 0 )  throw n2;
  return n1 / n2;
}
void test( int n1, int n2 ) {
   try {
        cout << “quotient:” << quotient( n1, n2);
   }
}
int main( ) {
  …
  try {
     test( n1, n2 );
   } catch (int ex) {
   …
   }
  return 0;
}
```

Exception-Handling Advantages

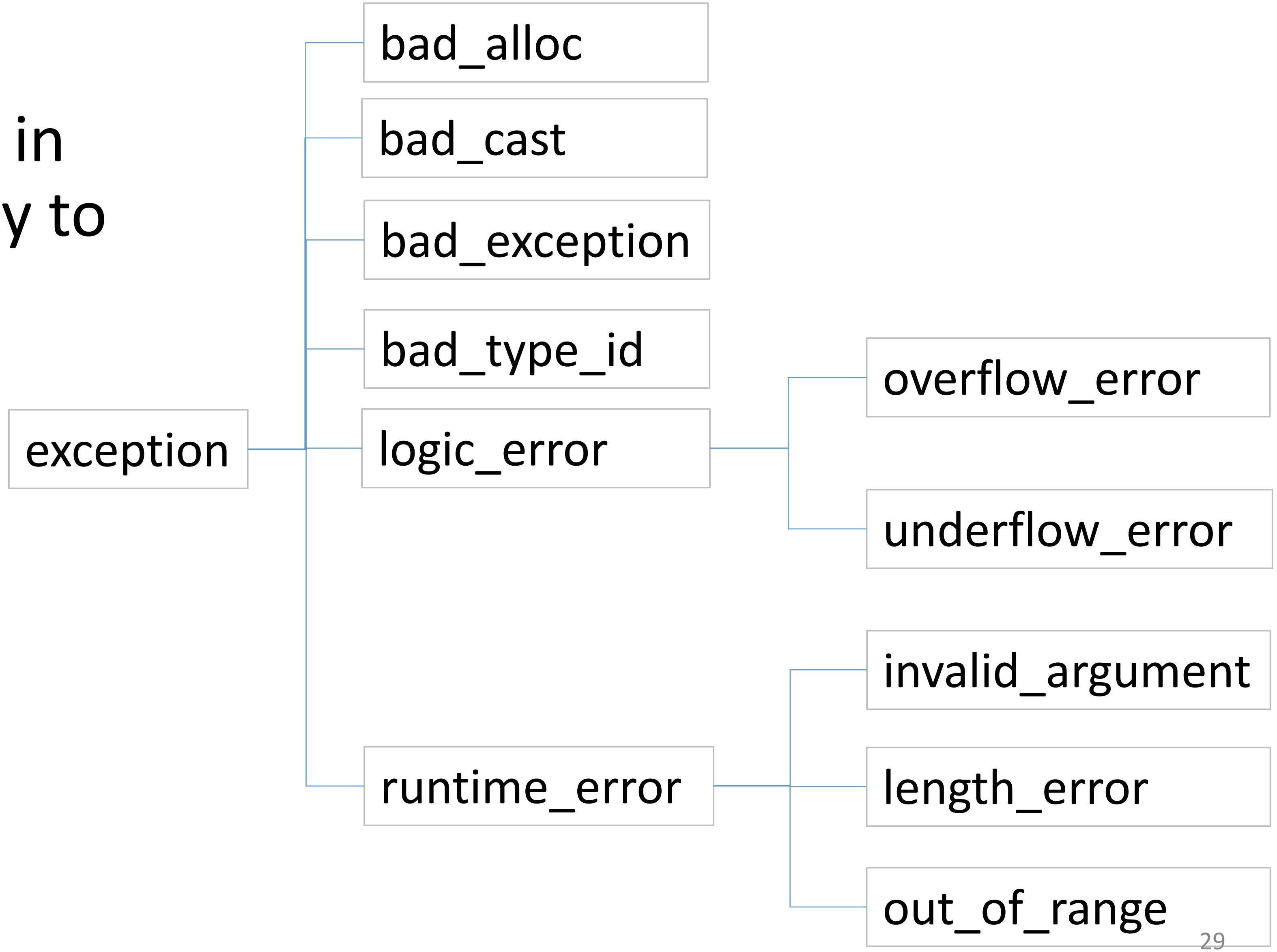
- Remove error-handling code from the main procedure.
- Can decide to handle certain exceptions and delegate others to the caller.
- An exception can be handled anywhere in the function call stack.

```
int quotient( int n1, int n2 ) {
  if ( n2== 0 )  throw n2;
  return n1 / n2;
}
void test( int n1, int n2 ) {
   try {
        cout << “quotient:” << quotient( n1, n2);
   } catch (int ex) {
    …
   }
}
int main( ) {
  …


   test( n1, n2 );

   return 0;
}
```

```
int quotient( int n1, int n2 ) {
  if ( n2== 0 )  throw n2;
  return n1 / n2;
}
void test( int n1, int n2 ) {
   try {
        cout << “quotient:” << quotient( n1, n2);
   }
}
int main( ) {
  …
  try {
     test( n1, n2 );
   } catch (int ex) {
    …
   }
  return 0;
}
```

Using Standard Classes

We can use the classes in the C++ standard library to throw exceptions.



Bad allocation

```
int main() {
  try
  {
    for (int i = 1; i <= 100; i++)
    {
      new int[ 170000000 ];
      cout << i << " arrays have been created" << endl;
    }
  }
  catch (bad_alloc& ex)
  {
    cout << "Exception: " << ex.what() << endl;
  }

  return 0;
}
```

Bad allocation

```
int main() {
  try
  {
    for (int i = 1; i <= 100; i++)
    {
      new int[ 170000000 ];
      cout << i << " arrays have been created" << endl;
    }
  }
  catch (bad_alloc& ex)
  {
    cout << "Exception: " << ex.what() << endl;
  }

  return 0;
}
```

Bad cast exception

<typeinfo> defines types that are related to operators typeid and dynamic_cast.

what( ) :
returns an explanatory string

```
#include <typeinfo>
#include <iostream>
using namespace std;
……
int main() {
  try {
    Rectangle r(2, 7);
    Circle& c = dynamic_cast<Circle&>(r);
  }
  catch (bad_cast& ex)
  {
    cout << "Exception: " << ex.what() << endl;
  }

  return 0;
}
```

Bad cast exception

<typeinfo> defines types that are related to operators typeid and dynamic_cast.

what( ) :
returns an explanatory string

```
#include <typeinfo>
#include <iostream>
using namespace std;
......
int main() {
  try {
    Rectangle r(2, 7);
    Circle& c = dynamic_cast<Circle&>(r);
  }
  catch (bad_cast& ex)
  {
    cout << "Exception: " << ex.what() << endl;
  }

  return 0;
}
```

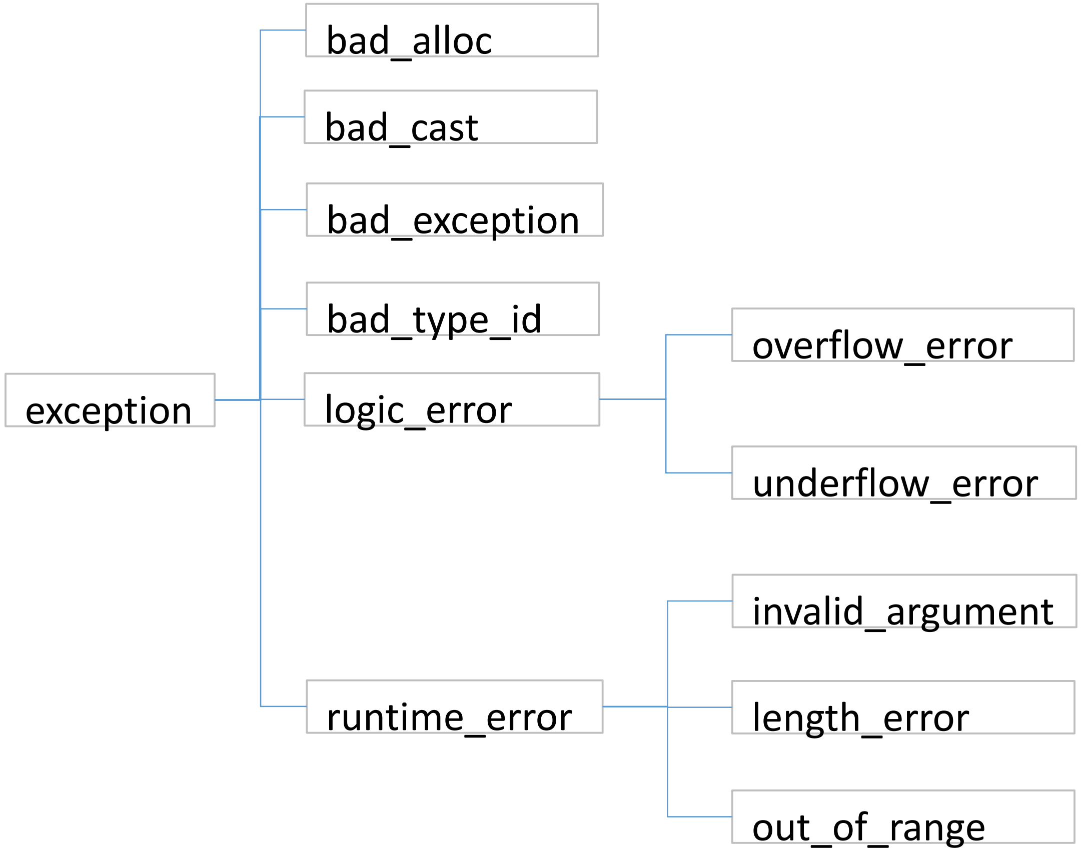
Invalid argument exception

```
#include <stdexcept>
using namespace std;
double getArea(double radius) {
  if (radius < 0)
    throw invalid_argument("Radius is negative");
  return radius * radius * 3.14159;
}
int main() { double radius;
  cout << "Enter radius: ";
  cin >> radius;
  try  {
    double result = getArea(radius);
    cout << "The area is " << result << endl;
  } catch (exception& ex) {
    cout << ex.what() << endl;
  }
  return 0;
}
```

Invalid argument exception

```
#include <stdexcept>
using namespace std;
double getArea(double radius) {
  if (radius < 0)
    throw invalid_argument("Radius is negative");
  return radius * radius * 3.14159;
}
int main() { double radius;
  cout << "Enter radius: ";
  cin >> radius;
  try  {
    double result = getArea(radius);
    cout << "The area is " << result << endl;
  } catch (exception& ex) {
    cout << ex.what() << endl;
  }
  return 0;
}
```

Invalid argument exception



```
#include <stdexcept>
using namespace std;
double getArea(double radius) {
  if (radius < 0)
    throw invalid_argument("Radius is negative");
  return radius * radius * 3.14159;
}
int main() { double radius;
  cout << "Enter radius: ";
  cin >> radius;
  try  {
    double result = getArea(radius);
    cout << "The area is " << result << endl;
  } catch (exception& ex) {
    cout << ex.what() << endl;
  }
  return 0;
}
```

Exception Classes

- We can **create our own exception class**.

- It is better for us to **derive our exception class from the exception classes in the standard library so that we can** utilize the common features, e.g., what( ).

An example: triangle

TriangleException.h

An example: triangle

TriangleException.h

```
#include <stdexcept>
using namespace std;
class TriangleException: public logic_error {
public:
  TriangleException(
     double side1
     , double side2
     , double side3)
    : A1 A2  {
    this->side1 = side1;
    this->side2 = side2;
    this->side3 = side3;
  }
  double getSide1() const {
    return side1;
  }
  double getSide2() const ……
  double getSide3() const ……
private:
  double side1, side2, side3;
}; // Semicolon required
```

An example: triangle

TriangleException.h

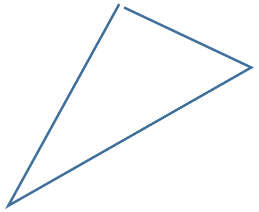
```
#include <stdexcept>
using namespace std;
class TriangleException: public logic_error {
public:
  TriangleException(
     double side1
     , double side2
     , double side3)
    : logic_error("Invalid triangle")  {
    this->side1 = side1;
    this->side2 = side2;
    this->side3 = side3;
  }
  double getSide1() const {
    return side1;
  }
  double getSide2() const ……
  double getSide3() const ……
private:
  double side1, side2, side3;
}; // Semicolon required
```

An example: triangle

TriangleException.h

```
#include <stdexcept>
using namespace std;
class TriangleException: public logic_error {
public:
  TriangleException(
     double side1
     , double side2
     , double side3)
    : logic_error("Invalid triangle")  {
    this->side1 = side1;
    this->side2 = side2;
    this->side3 = side3;
  }
  double getSide1() const {
    return side1;
  }
  double getSide2() const ……
  double getSide3() const ……
private:
  double side1, side2, side3;
}; // Semicolon required
```

An example: triangle



The sum of any two side lengths must be greater than the third side length.

Triangle.h

```
#include "TriangleException.h"
#include <cmath>
class Triangle: public GeometricObject {
public: Triangle() {
       side1 = side2 = side3 = 1;
   }
  Triangle(double side1, double side2, double side3) {
   if ( !isValid( side1, side2, side3 ) )
     throw TriangleException(side1, side2, side3);
   this->side1 = side1;
   this->side2 = side2;
   this->side3 = side3;
  }
  bool isValid(double side1
     , double side2
     , double side3) const {
               return (side1 < side2 + side3)
                     && (side2 < side1 + side3)
                     && (side3 < side1 + side2);
  }
};
```

TriangleException.h

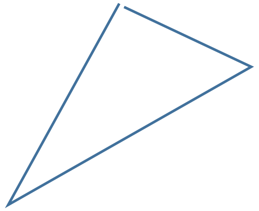
```
#include <stdexcept>
using namespace std;
class TriangleException: public logic_error {
public:
  TriangleException(
     double side1
     , double side2
     , double side3)
    : logic_error("Invalid triangle")  {
    this->side1 = side1;
    this->side2 = side2;
    this->side3 = side3;
  }
  double getSide1() const {
    return side1;
  }
  double getSide2() const ……
  double getSide3() const ……
private:
  double side1, side2, side3;
}; // Semicolon required
```

An example: triangle

Triangle.h

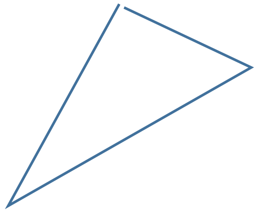
main.cpp

```
#include "TriangleException.h"
#include <cmath>
class Triangle: public GeometricObject {
public: Triangle() {
       side1 = side2 = side3 = 1;
   }
  Triangle(double side1, double side2, double side3) {
    if ( !isValid( side1, side2, side3 ) )
      throw TriangleException(side1, side2, side3);
    this->side1 = side1;
    this->side2 = side2;
    this->side3 = side3;
  }
  bool isValid(double side1
     , double side2
     , double side3) const {
               return (side1 < side2 + side3)
                     && (side2 < side1 + side3)
                     && (side3 < side1 + side2);
  }
};
```



The sum of any two side lengths must be greater than the third side length.

An example: triangle



The sum of any two side lengths must be greater than the third side length.

Triangle.h

```
#include "TriangleException.h"
#include <cmath>
class Triangle: public GeometricObject {
public: Triangle() {
      side1 = side2 = side3 = 1;
   }
  Triangle(double side1, double side2, double side3) {
   if ( !isValid( side1, side2, side3 ) )
     throw TriangleException(side1, side2, side3);
   this->side1 = side1;
   this->side2 = side2;
   this->side3 = side3;
  }
  bool isValid(double side1
     , double side2
     , double side3) const {
           return (side1 < side2 + side3)
                && (side2 < side1 + side3)
                && (side3 < side1 + side2);
  }
};
```

main.cpp

```
void main() {
  try  {
    Triangle triangle;
    cout << "Perimeter is "
            << triangle.getPerimeter() << endl;
    cout << "Area is " << triangle.getArea() << endl;
    triangle t(2, 5, 9);
    cout << "Perimeter is "
            << triangle.getPerimeter() << endl;
    cout << "Area is " << triangle.getArea() << endl;
  }
  catch (TriangleException& ex) {
    cout << ex.what();
    cout << " three sides are "
            << ex.getSide1() << " "
            << ex.getSide2() << " "
            << ex.getSide3() << endl;
  }
}
```

Multiple Catches

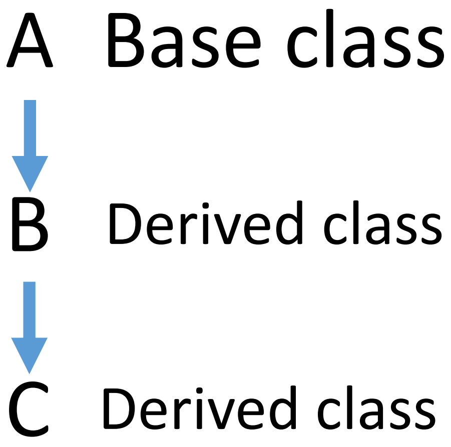
- ➢A try black may throw different exception types.
- ➢We need to add multiple catch blocks for catching multiple types of exceptions.

```
try {
   cin >> n1 >> n2;
   if ( n2 == 0 ) throw( string(“n2 is zero”) );
   if ( n1 > 0 ) throw(0);
   throw(1);
}
```

```
catch( const string &msg) {
  cout << msg << endl;
}
catch(int error_code) {
  switch(error_code) {
    case 0: cout << “n1 is positive” << endl;
    break;
    case 1: cout << “n1 is non-positive” << endl;
    break;
    default: cout << “Unexpected error” << endl;
  }
}
```

Catch block

A catch block, which catches exception objects of a base class, can catch all the exception objects of the derived classes of that base class.

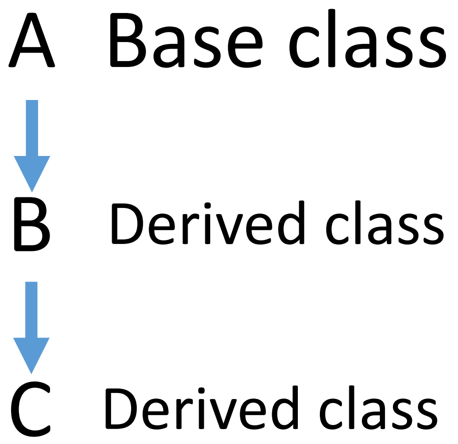


A Base class

B Derived class

C Derived class

Order of exception handlers

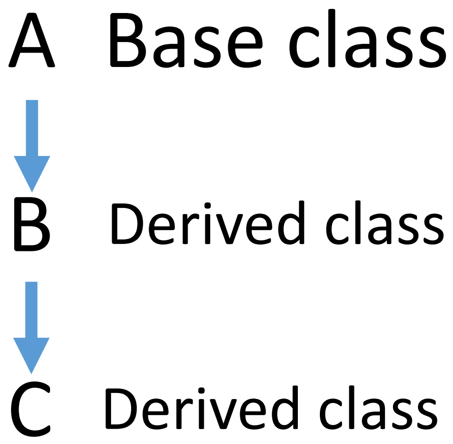


```
class EXCEPTION_A {
};
class EXCEPTION_B: EXCEPTION_A {
};
```

```
try {
  throw(EXCEPTION_B( ) );
}
catch( EXCEPTION_A &ex) {
  //it catches the exception B
}
catch( EXCEPTION_B &ex) {
}
```

```
try {
  ……
}
catch( EXCEPTION_B &ex) {
}
catch( EXCEPTION_A &ex) {
}
```

Order of exception handlers


A Base class

B Derived class

C Derived class

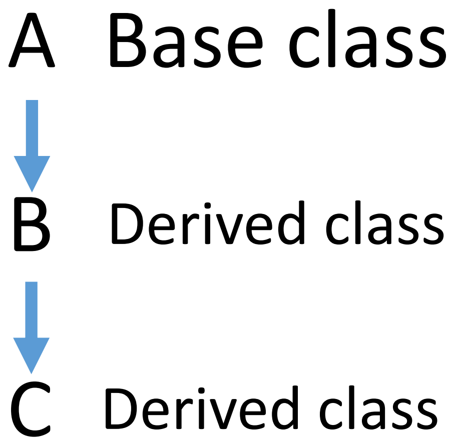
```
class EXCEPTION_A {
};
class EXCEPTION_B: EXCEPTION_A {
};
```

```
try {
  throw(EXCEPTION_B( ) );
}
catch( EXCEPTION_A &ex) {
  //it catches the exception B
}
catch( EXCEPTION_B &ex) {
}
```

```
try {
  ……
}
catch( EXCEPTION_B &ex) {
}
catch( EXCEPTION_A &ex) {
}
```

Order of exception handlers

- A catch block for a base class type should appear **after** a catch block for **a derived class type**.
- If not, the exception is always caught by the catch block for the base class.



```
class EXCEPTION_A {
};
class EXCEPTION_B: EXCEPTION_A {
};
```

```
try {
  throw(EXCEPTION_B( ) );
}
catch( EXCEPTION_A &ex) {
  //it catches the exception B
}
catch( EXCEPTION_B &ex) {
}
```

```
try {
  ……
}
catch( EXCEPTION_B &ex) {
}
catch( EXCEPTION_A &ex) {
}
```

Exception Propagation

```
class ERROR_CODE {
public: ERROR_CODE(int v) {
                  this->v = v;}
    int v;
};
void f3( ) {
    cout << "f3" << endl;
    throw("here");
}
void f2( ) {
    cout << "f2" << endl;
    try {
        f3( );
    }
    catch(double d) {
        cout << "f2:d" << endl;
    }
    cout << "end f2" << endl;
}
```

```
void f1( ) {
     try {
        f2( );
     }
     catch(string msg) {
         cout << "f1: msg:" << msg << endl;
         throw(ERROR_CODE(1));
     }
}
void main( ) {
    try {
        f1( );
    }
    catch(ERROR_CODE code) {
        cout << "main: " << code.v << endl;
    }
}
```

What are the output?

Exception Propagation

```
class ERROR_CODE {
public: ERROR_CODE(int v) {
                  this->v = v;}
    int v;
};
void f3( ) {
    cout << "f3" << endl;
    throw("here");
}
void f2( ) {
    cout << "f2" << endl;
    try {
        f3( );
    }
    catch(double d) {
        cout << "f2:d" << d << endl;
    }
    cout << "end f2" << endl;
}
```

```
void f1( ) {
     try {
        f2( );
     }
     catch(string msg) {
         cout << "f1: msg:"
           << msg << endl;
         throw(ERROR_CODE(1));
     }
}
void main( ) {
    try {
        f1( );
    }
    catch(ERROR_CODE code) {
        cout << "main: "
        << code.v << endl;
    }
}
```

Runtime error. No catch block can handle it.

Exception Propagation

```
class ERROR_CODE {
public: ERROR_CODE(int v) {
                  this->v = v;}
    int v;
};
void f3( ) {
    cout << "f3" << endl;
    throw("here");
}
void f2( ) {
    cout << "f2" << endl;
    try {
        f3( );
    }
    catch(double d) {
        cout << "f2:d" << endl;
    }
    cout << "end f2" << endl;
}
```

This is a C-string, not C++ string.

```
void f1( ) {
     try {
        f2( );
     }
     catch(string msg) {
         cout << "f1: msg:"
           << msg << endl;
         throw(ERROR_CODE(1));
     }
}
void main( ) {
    try {
        f1( );
    }
    catch(ERROR_CODE code) {
        cout << "main: "
        << code.v << endl;
    }
}
```

Runtime error. No catch block can handle it.

Exception Propagation

```
class ERROR_CODE {
public: ERROR_CODE(int v) {
                  this->v = v;}
    int v;
};
void f3( ) {
    cout << "f3" << endl;
    throw(string("here"));
}
void f2( ) {
    cout << "f2" << endl;
    try {
        f3( );
    }
    catch(double d) {
        cout << "f2:d" << d << endl;
    }
    cout << "end f2" << endl;
}
```

```
void f1( ) {
     try {
        f2( );
     }
     catch(string msg) {
         cout << "f1: msg:"
           << msg << endl;
         throw(ERROR_CODE(1));
     }
}
void main( ) {
    try {
        f1( );
    }
    catch(ERROR_CODE code) {
        cout << "main: "
        << code.v << endl;
    }
}
```

Exception Propagation

```
class ERROR_CODE {
public: ERROR_CODE(int v) {
                  this->v = v;}
    int v;
};
void f3( ) {
    cout << "f3" << endl;
    throw(string("here"));
}
void f2( ) {
    cout << "f2" << endl;
    try {
        f3( );
    }
    catch(double d) {
        cout << "f2:d" << d << endl;
    }
    cout << "end f2" << endl;
}
```

```
void f1( ) {
     try {
        f2( );
     }
     catch(string msg) {
         cout << "f1: msg:"
           << msg << endl;
         throw(ERROR_CODE(1));
     }
}
void main( ) {
    try {
        f1( );
    }
    catch(ERROR_CODE code) {
        cout << "main: "
        << code.v << endl;
    }
}
```

What are the output?

Exception Propagation

```
class ERROR_CODE {
public: ERROR_CODE(int v) {
                  this->v = v;}
    int v;
};
void f3( ) {
    cout << "f3" << endl;
    throw(string("here"));
}
void f2( ) {
    cout << "f2" << endl;
    try {
        f3( );
    }
    catch(double d) {
        cout << "f2:d" << d << endl;
    }
    cout << "end f2" << endl;
}
```

```
void f1( ) {
     try {
        f2( );
     }
     catch(string msg) {
         cout << "f1: msg:"
           << msg << endl;
         throw(ERROR_CODE(1));
     }
}
void main( ) {
    try {
        f1( );
    }
    catch(ERROR_CODE code) {
        cout << "main: "
        << code.v << endl;
    }
}
```

What are the output?

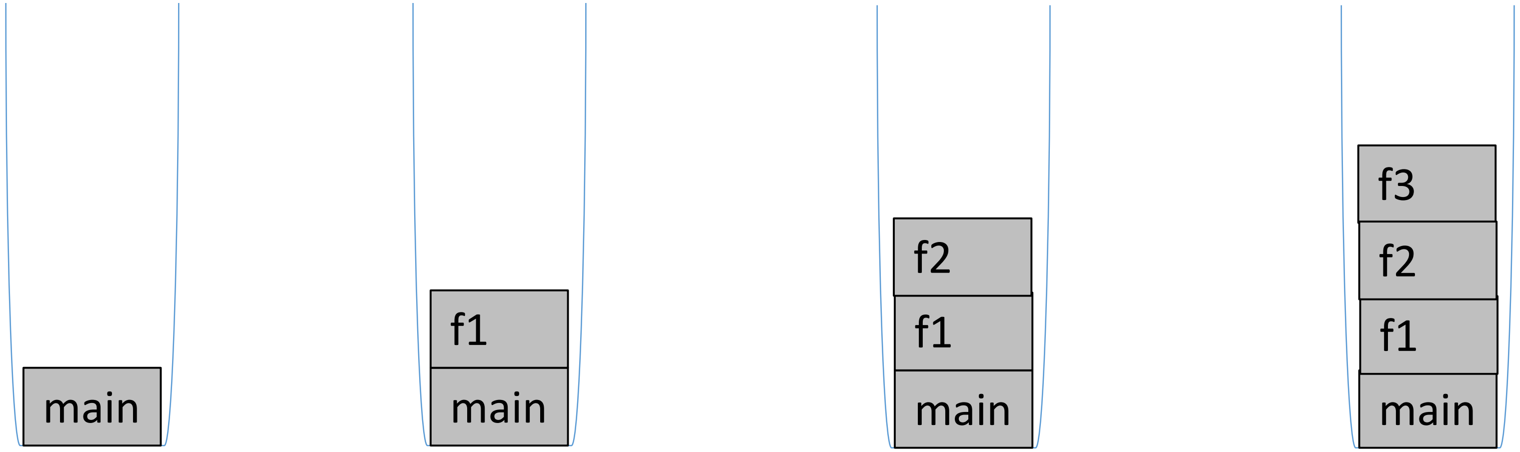
```
f2
f3
f1: msg:here
main: 1
```

Call Stack

```
void f3( ) {
    cout << "f3" << endl;
    throw(string("here"));
}
void f2( ) {
    cout << "f2" << endl;
    try {
        f3( );
    }
    catch(double d) {
      ......
    }
    cout << "end f2" << endl;
}
```

```
void f1( ) {
    try {
        f2( );
    }
    catch(string msg) {
        ......
    }
}
```

```
void main( ) {
    try {
        f1( );
    }
    catch(ERROR_CODE code) {
        ......
    }
}
```



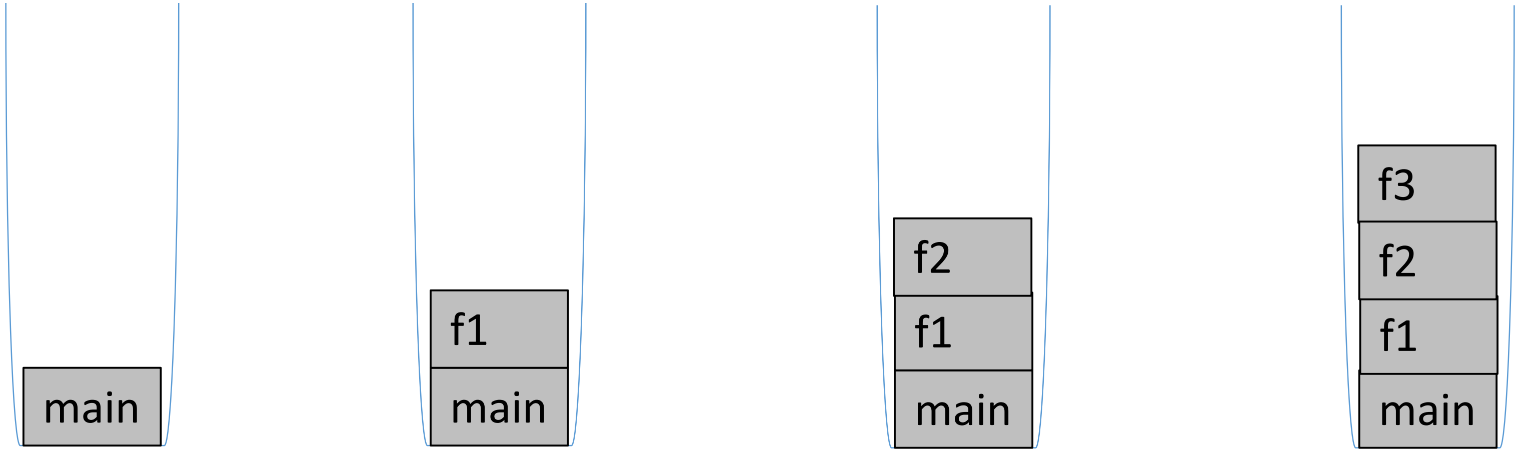
When a function throws an exception, there must be a catch block handling the exception and it is in a function in the call stack.

Call Stack

```
void f3( ) {
    cout << "f3" << endl;
    throw(ERROR_CODE(2));
}
void f2( ) {
    cout << "f2" << endl;
    try {
        f3( );
    }
    catch(double d) {
      ......
    }
    cout << "end f2" << endl;
}
```

```
void f1( ) {
     try {
        f2( );
     }
     catch(string msg) {
        ......
     }
}
```

```
void main( ) {
     try {
        f1( );
     }
     catch(ERROR_CODE code) {
        ......
     }
}
```



When a function throws an exception, there must be a catch block handling the exception and it is in a function in the call stack.

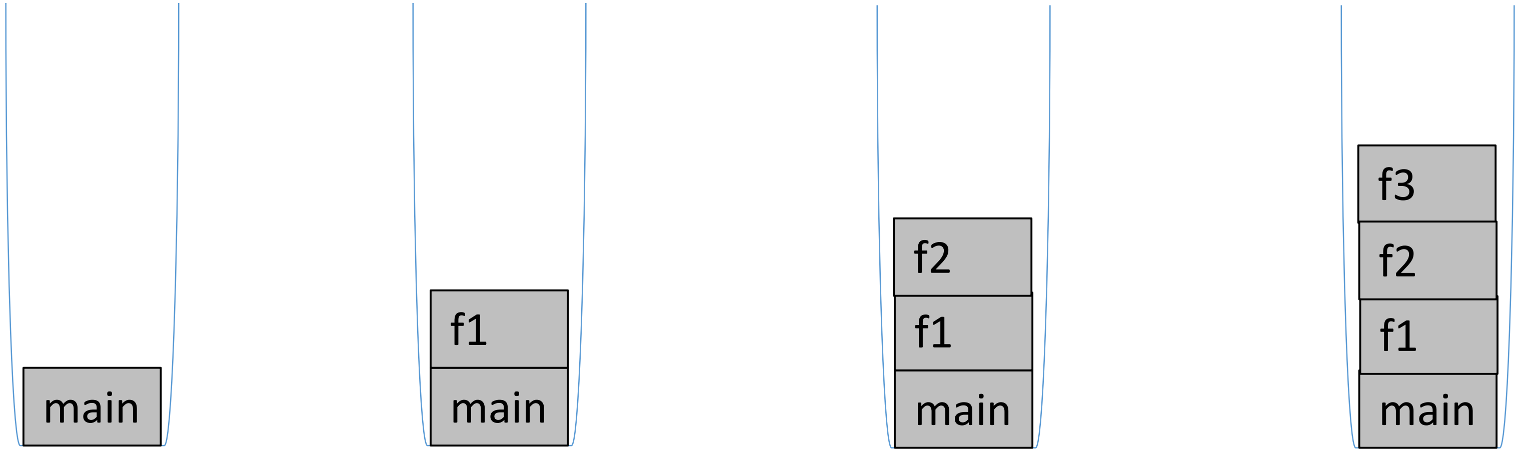
Call Stack

```
void f3( ) {
    cout << "f3" << endl;
    throw(ERROR_CODE(2));
}
void f2( ) {
    cout << "f2" << endl;
    try {
        f3( );
    }
    catch(double d) {
      ......
    }
    cout << "end f2" << endl;
}
```

```
void f1( ) {
     try {
        f2( );
     }
     catch(string msg) {
         ......
     }
}
```

```
void main( ) {
     try {
         f1( );
     }
     catch(ERROR_CODE code) {
         cout << code.v;
     }
}
```

What are the output?



Call Stack

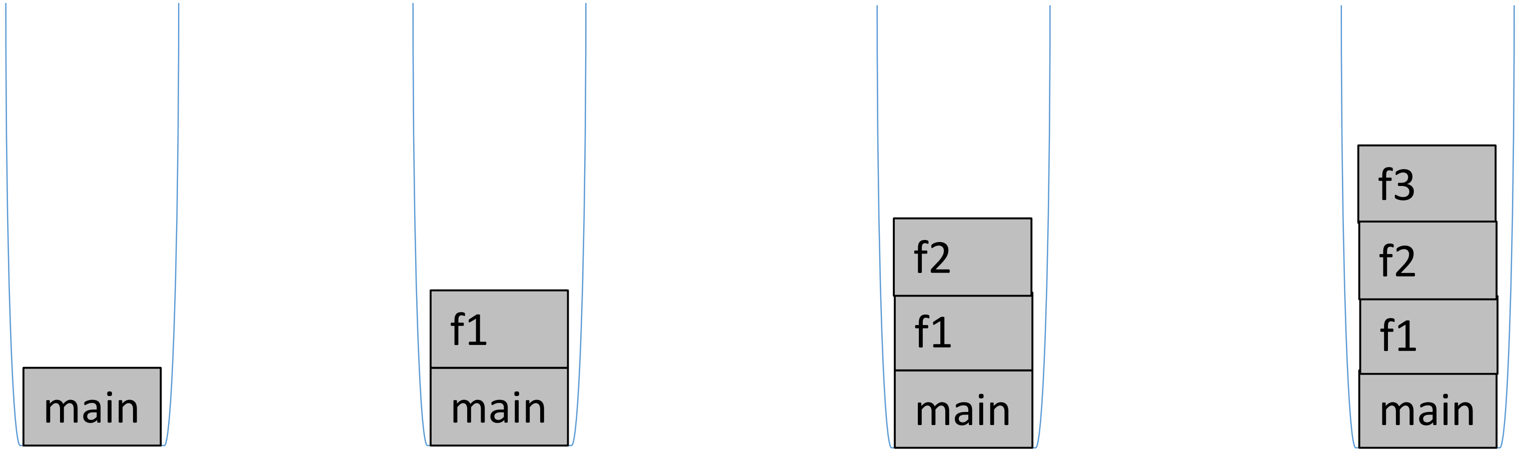
```
void f3( ) {
    cout << "f3" << endl;
    throw(ERROR_CODE(2));
}
void f2( ) {
    cout << "f2" << endl;
    try {
        f3( );
    }
    catch(double d) {
      ......
    }
    cout << "end f2" << endl;
}
```

```
void f1( ) {
     try {
        f2( );
     }
     catch(string msg) {
         ......
     }
}
```

```
void main( ) {
     try {
        f1( );
     }
     catch(ERROR_CODE code) {
        cout << code.v;
     }
}
```

What are the output?

f2
f3
2



Intended Learning Outcomes

- Describe the process for exception handling
- Define a try-catch block
- Implement exception handling in a simple program

Supplemental Materials

When to Use Exceptions

- An exception is a problem that arises during a program execution.

- Common exceptions that may occur in multiple classes are candidates for exception classes.

- Simple errors that may occur in individual functions are best handled locally without throwing exceptions.

When to Use Exceptions

- **Exception handling** is for dealing with **unexpected error conditions**.
- Do not use a try-catch block to deal with simple, expected situations.
- Which situations are exceptional and which are expected is sometimes difficult to decide.
- The point is **not to abuse exception handling** as a way to deal with a simple logic test.

Exercise: What are the output?

```
void foo( ) throw( )  {
     int a = 10.0;
     return;
}
```

```
void h() {
   throw(28);
}
void k() {
   h();
   cout << "here" << endl;
   throw(string("call k()"));
};
```

```
void g() {
   try {
      k( );
   }
   catch(int num) {
      cout << "Num:" << num << endl;
   }
   catch( string &e ) {
      cout << "Error:" << e << endl;
   }
}
```

```
// Question
// What are the output?
int main( )
{
   try {
      foo( );
   }
   catch( int e ) {
      cout << "Error:" << e << endl;
   }
   g();

            system("pause");
            return 0;
}
```

Exercise: What are the output?

```
void foo( ) throw( )   {
      int a = 10.0;
      return;
}
```

```
void h() {
   //throw(28);
}
void k() {
   h();
   cout << "here" << endl;
   throw(string("call k()"));
};
```

```
void g() {
   try {
      k( );
   }
   catch(int num) {
      cout << "Num:" << num << endl;
   }
   catch( string &e ) {
      cout << "Error:" << e << endl;
   }
}
```

```
// Question
// What are the output?
int main( )
{
   try {
      g( );
   }
   catch( int e ) {
      cout << "Error:" << e << endl;
   }
   g();

             system("pause");
             return 0;
}
```

Avoid using many layers of if-structures

```
if ( ) {
        if ( ) {
                if ( ) {
                if ( ) {
                        if ( ) {
                                if (A == 0 ) throw A;

                        if ( ) {

                        }
                        }
                }

                }
        }
}
```

```
class NonPositiveSideException: public logic_error
{
public:
  NonPositiveSideException(double side)
    : logic_error("Non-positive side")
  {
    this->side = side;
  }

  double getSide()
  {
    return side;
  }

private:
  double side;
};
```

```
int main() {
  try
  {
    cout << "Enter three sides: ";
    double side1, side2, side3;
    cin >> side1 >> side2 >> side3;
    Triangle triangle(side1, side2, side3);
    cout << "Perimeter is “
                << triangle.getPerimeter() << endl;
    cout << "Area is "
               << triangle.getArea() << endl;
  }



  return 0;
}
```

```
catch (NonPositiveSideException& ex) {
   cout << ex.what();
   cout << " the side is "
               << ex.getSide()
               << endl;
 }
 catch (TriangleException& ex) {
   cout << ex.what();
   ……
 }
```

```
int main() {
  try
  {
    cout << "Enter three sides: ";
    double side1, side2, side3;
    cin >> side1 >> side2 >> side3;
    Triangle triangle(side1, side2, side3);
    cout << "Perimeter is “
             << triangle.getPerimeter() << endl;
    cout << "Area is "
            << triangle.getArea() << endl;
  }



  return 0;
}
```

```
catch (NonPositiveSideException& ex) {
   cout << ex.what();
   cout << " the side is " << ex.getSide() << endl;
 }
 catch (TriangleException& ex) {
   cout << ex.what();
   ……
 }
```

```
int main() {
  try
  {
    cout << "Enter three sides: ";
    double side1, side2, side3;
    cin >> side1 >> side2 >> side3;
    Triangle triangle(side1, side2, side3);
    cout << "Perimeter is “
              << triangle.getPerimeter() << endl;
    cout << "Area is "
             << triangle.getArea() << endl;
  }
 catch (NonPositiveSideException& ex) {
    cout << ex.what();
    cout << " the side is " << ex.getSide() << endl;
  }
  catch (TriangleException& ex) {
    cout << ex.what();
    ……
  }
  return 0;
}
```

```
catch (NonPositiveSideException& ex) {
   cout << ex.what();
   cout << " the side is " << ex.getSide() << endl;
 }
 catch (TriangleException& ex) {
   cout << ex.what();
   ……
 }
```

This topic may be out of dated. Please read the new update about C++.

Rethrowing Exceptions

- C++ allows an exception handler to rethrow the exception
- if the handler **cannot process the exception**
- or the handler simply wants to let its caller be notified of the exception.

```
try
{
  statements;
}
catch (TheException &ex)
{
  perform operations before exits;
  throw;
}
```

This topic may be out of dated.
Please read the new update
about C++.

Rethrowing Exceptions

```
int f1() {
  try {
    throw runtime_error("Exception in f1");
  }
  catch (exception& ex) {
    cout << "Exception caught in function f1" << endl;
    cout << ex.what() << endl;
    throw; // Rethrow the exception
  }
}
void main() {
  try {
    f1();
  }
  catch (exception& ex) {
    cout << "Exception caught in function main" << endl;
    cout << ex.what() << endl;
  }
}
```

```
try
{
  statements;
}
catch (TheException &ex)
{
  perform operations before exits;
  throw;
}
```

This topic may be out of dated. Please read the new update about C++.

Exception Specification

- An *exception specification*, also known as *throw list*, lists exceptions that a function can throw.
- A function without a throw list can throw any exception.
- A function should warn programmers what it might throw.

```
returnType functionName( parameterList ) throw ( exceptionList )
```

This topic may be out of dated.
Please read the new update
about C++.

Exception Specification

- An *exception specification*, also known as *throw list*, lists exceptions that a function can throw.
- A function without a throw list can throw any exception.
- A function should warn programmers what it might throw.

returnType functionName(parameterList) throw (exceptionList)

Empty exception specification

This topic may be out of dated. Please read the new update about C++.

```
void foo ( int a, double b) throw( )

{ // this function does not throw any exception.

// If something is thrown, error.

……

}
```

```
void foo( ) throw( ) {
    int a;
    ……
    throw a; //runtime error
            //should not throw any
    return;
}
```

Empty exception specification

This topic may be out of dated. Please read the new update about C++.

- **An *empty exception specification:*** Place throw() after a function header.

- If a function attempts to throw an exception, a runtime error occurs.

```
void foo ( int a, double b) throw( )

{ // this function does not throw any exception.

// If something is thrown, error.

……

}
```

```
void foo( ) throw( ) {
    int a;
    ……
    throw a; //runtime error
             //should not throw any
    return;
}
```