10: EXCEPTIONS AND TEMPLATES

Programming Technique II
(SCSJ1023)

Adapted from Tony Gaddis and Barret Krupnow (2016), Starting out with C++: From Control Structures through Objects

Exceptions

Introduction to Exceptions

- Indicate that something unexpected has occurred or been detected.

- Allow program to deal with the problem in a controlled manner.

- Can be as simple or complex as program design requires.

Terminology

- **Exception**: object or value that signals an error ⇨ exceptional circumstance ⇨ run-time errors.

- **Throw an exception**: send a signal that an error has occurred.

- **Catch/ Handle an exception**: process the exception; interpret the signal.

Keywords

- `throw`: send a signal that an error has occurred.

- `try`: followed by a block { }, is used to invoke code that throws an exception.

- `catch`: followed by a block { }, is used to detect and process exceptions thrown in preceding `try` block. Takes a parameter that matches the type thrown.

Flow of Control

- A function that throws an exception is called from within a `try` block.

- If the function **throws an exception**:
  - The function terminates and the `try` block is immediately exited.
  - A `catch` block to process the exception is searched for in the source code immediately following the `try` block.

- If a `catch` block is found that matches the exception thrown, it is executed. If no `catch` block that matches the exception is found, the program terminates.

Example 1a: Using throw

```
//Function that throws an exception
int totalDays(int days, int weeks)
{
    if ((days < 0) || (days > 7))
        throw “Invalid number of days!";
        //the argument to throw is a literal c-string
    else
        return (7 * weeks + days);
}
```

Example 1b: Using try…catch

```
int main(){
      int totDays,days, weeks;

      cout << "Enter no. of. days and no. of. weeks =>";
      cin >> days >> weeks;

      try{
            totDays = totalDays(days, weeks);
            cout << "Total days: " << totDays;
      }
      catch (const char *msg){
            cout << "Error: " << msg;
      }

      return 0;
}
//code in the try-block is called protected code
//code in the catch-block is called exception handler
```

Correction:

catch (**const** char *msg)

Example 1: What Happens?

- `try` block is entered. `totalDays` function is called to.

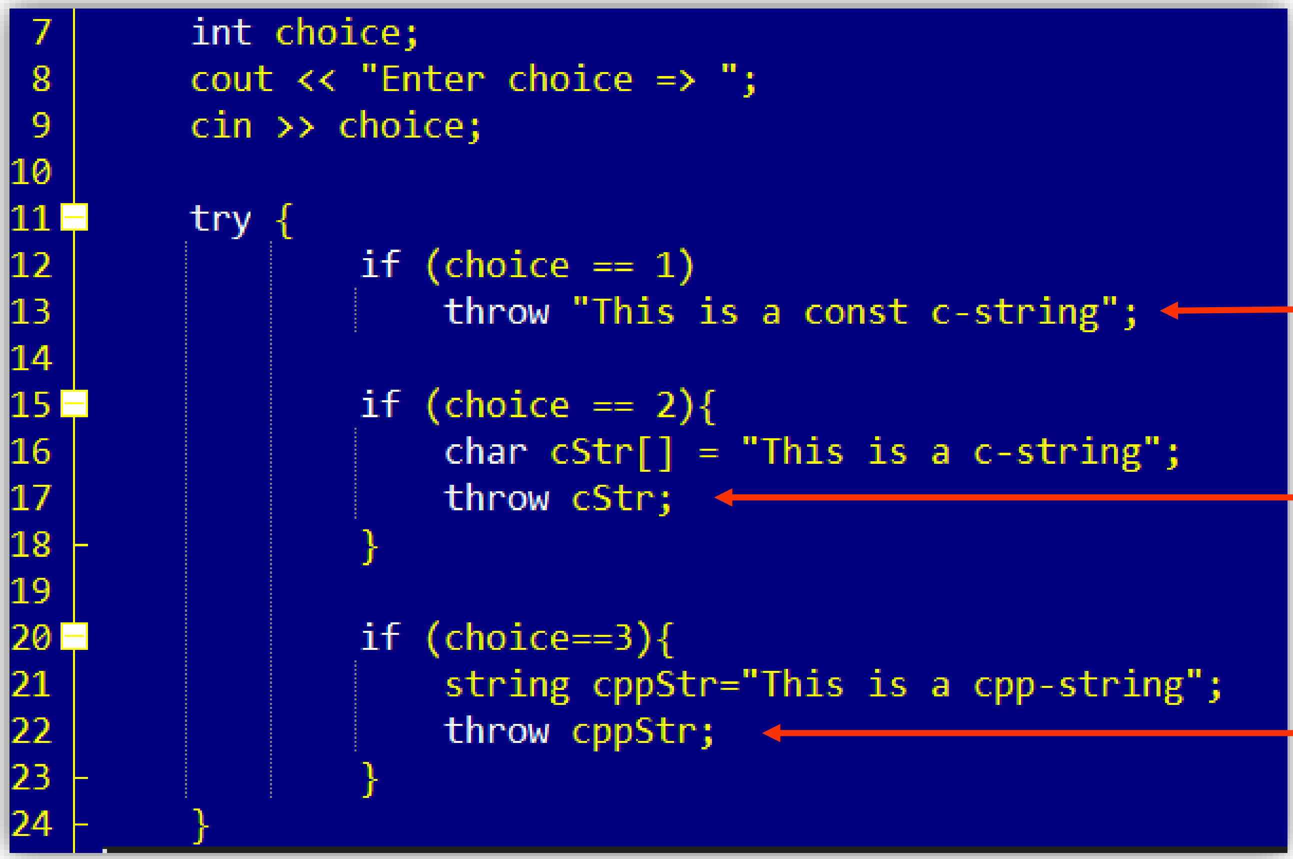
- If 1st parameter is between 0 and 7, total number of days is returned and `catch` block is skipped over (**no exception thrown**).

- If **exception is thrown**, function and `try` block are exited, `catch` blocks are scanned for 1st one that matches the data type of the thrown exception. `catch` block executes.

Additional Notes:

Dealing with string exceptions

- You can throw different types of string exceptions.
  - A literal string
  - C string
  - C++ string



```
    int choice;
    cout << "Enter choice => ";
    cin >> choice;

    try {
            if (choice == 1)
                throw "This is a const c-string";

            if (choice == 2){
                char cStr[] = "This is a c-string";
                throw cStr;
            }

            if (choice==3){
                string cppStr="This is a cpp-string";
                throw cppStr;
            }
    }
```

Throwing a literal c-string

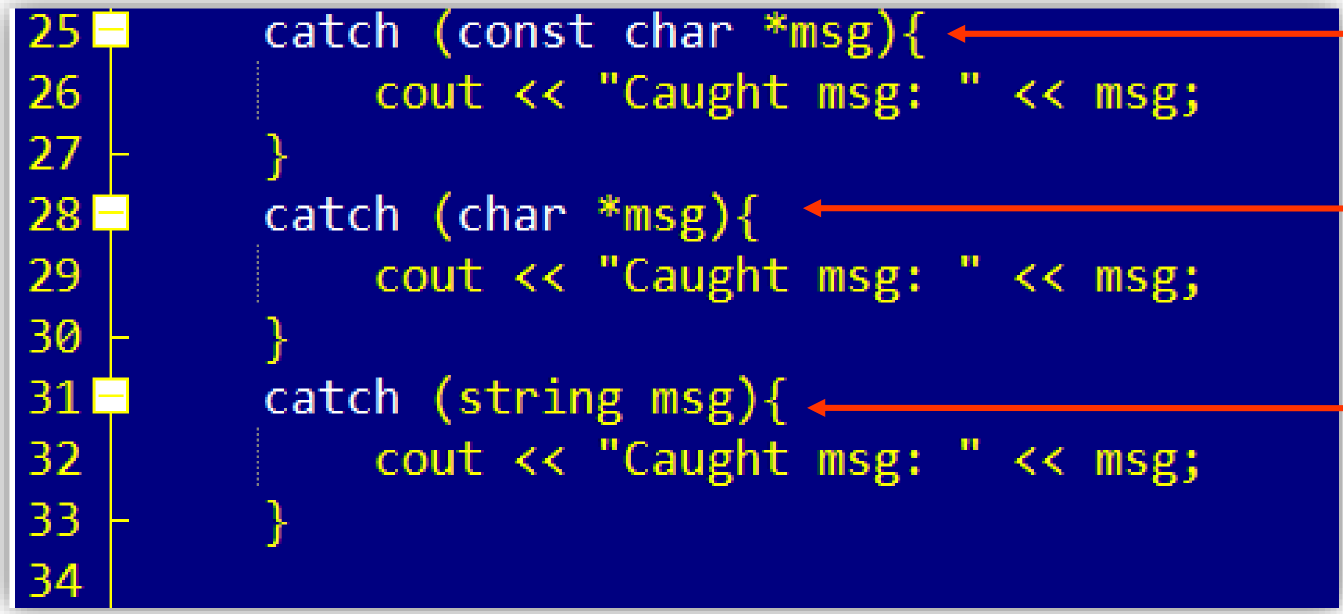
Throwing a c-string variable

Throwing a string object (c++ string)

Additional Notes:

Dealing with string exceptions

- Then, catch the exceptions accordingly



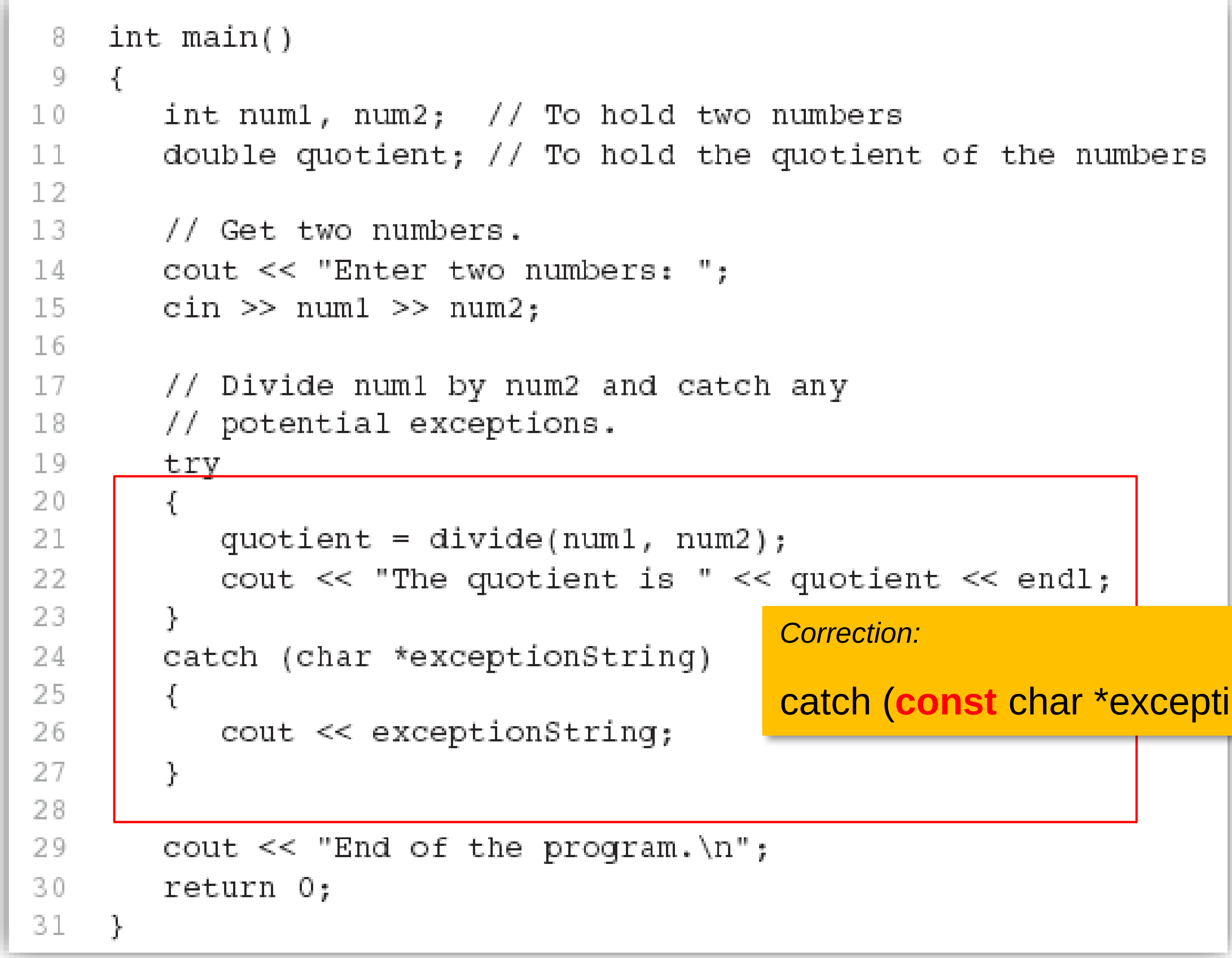
```
    catch (const char *msg){
        cout << "Caught msg: " << msg;
    }
    catch (char *msg){
        cout << "Caught msg: " << msg;
    }
    catch (string msg){
        cout << "Caught msg: " << msg;
    }

```

Catching a literal c-string exception

Catching a c-string exception

Catching a c++ string exception

Example 2a: Using `try…catch`

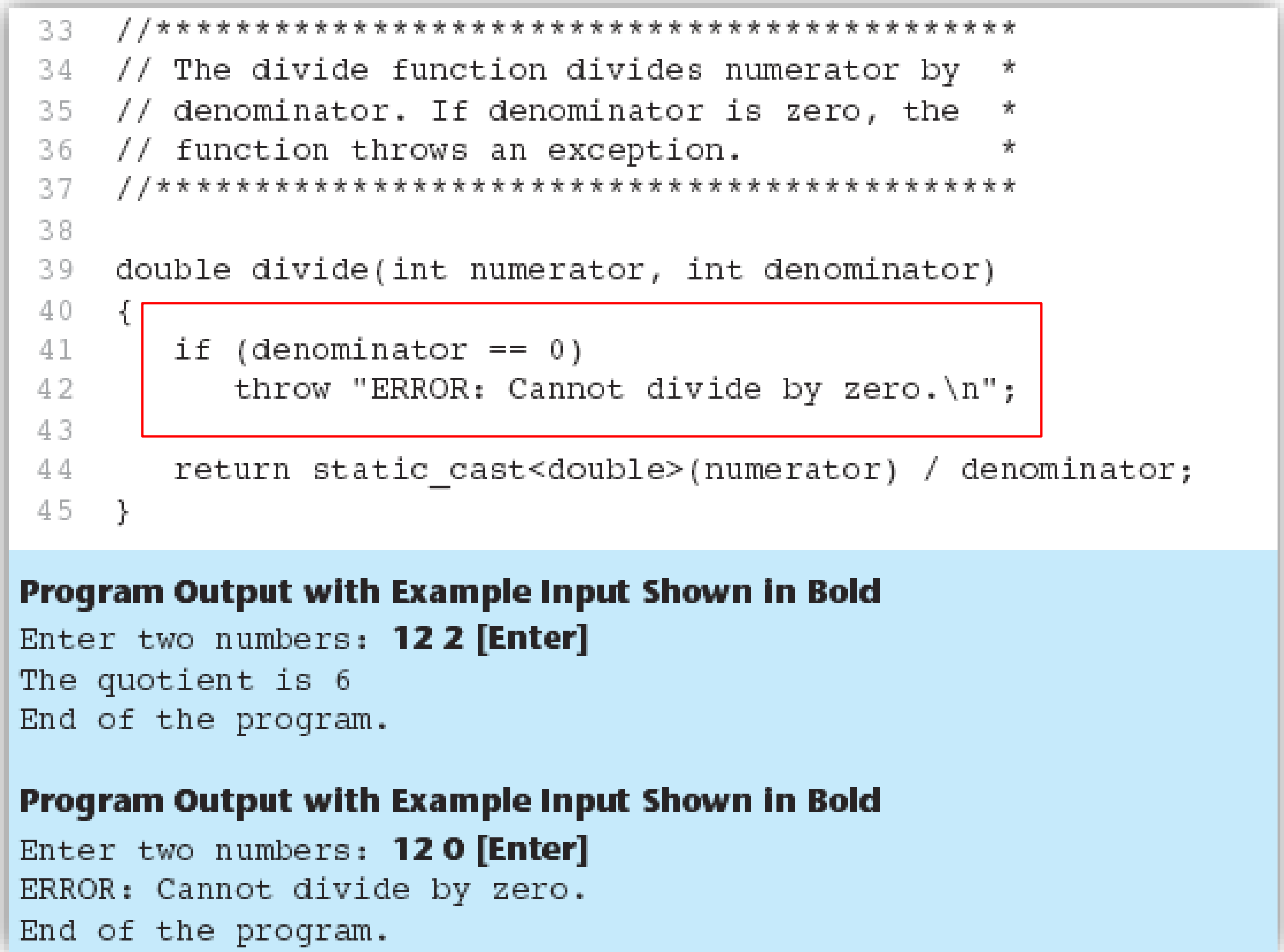


```
int main()
{
   int num1, num2;   // To hold two numbers
   double quotient; // To hold the quotient of the numbers

   // Get two numbers.
   cout << "Enter two numbers: ";
   cin >> num1 >> num2;

   // Divide num1 by num2 and catch any
   // potential exceptions.
   try
   {
      quotient = divide(num1, num2);
      cout << "The quotient is " << quotient << endl;
   }
   catch (char *exceptionString)
   {
      cout << exceptionString;
   }

   cout << "End of the program.\n";
   return 0;
}
```

Correction:

catch (**const** char *exceptionString)

Example 2b: Using throw



```
//*******************************************
// The divide function divides numerator by  *
// denominator. If denominator is zero, the  *
// function throws an exception.             *
//*******************************************

double divide(int numerator, int denominator)
{
   if (denominator == 0)
      throw "ERROR: Cannot divide by zero.\n";

   return static_cast<double>(numerator) / denominator;
}
```

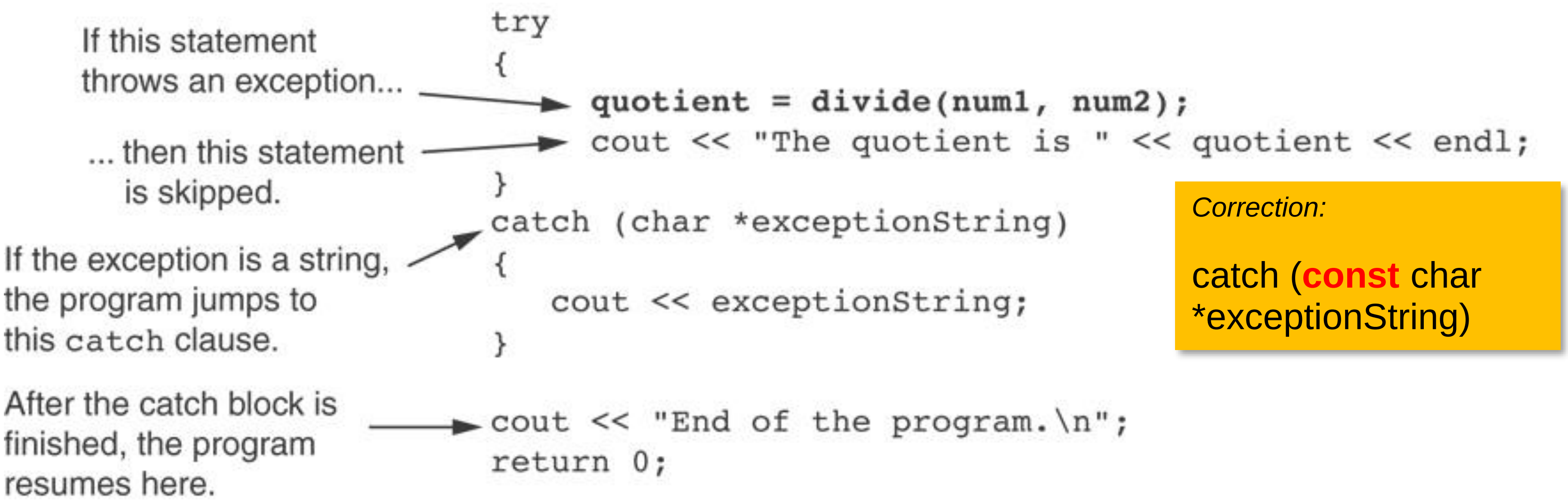
Program Output with Example Input Shown in Bold

```
Enter two numbers: 12 2 [Enter]
The quotient is 6
End of the program.
```

Program Output with Example Input Shown in Bold

```
Enter two numbers: 12 0 [Enter]
ERROR: Cannot divide by zero.
End of the program.
```

Example 2: What Happens in the `try…catch` Construct?



If this statement throws an exception...

... then this statement is skipped.

If the exception is a string, the program jumps to this `catch` clause.

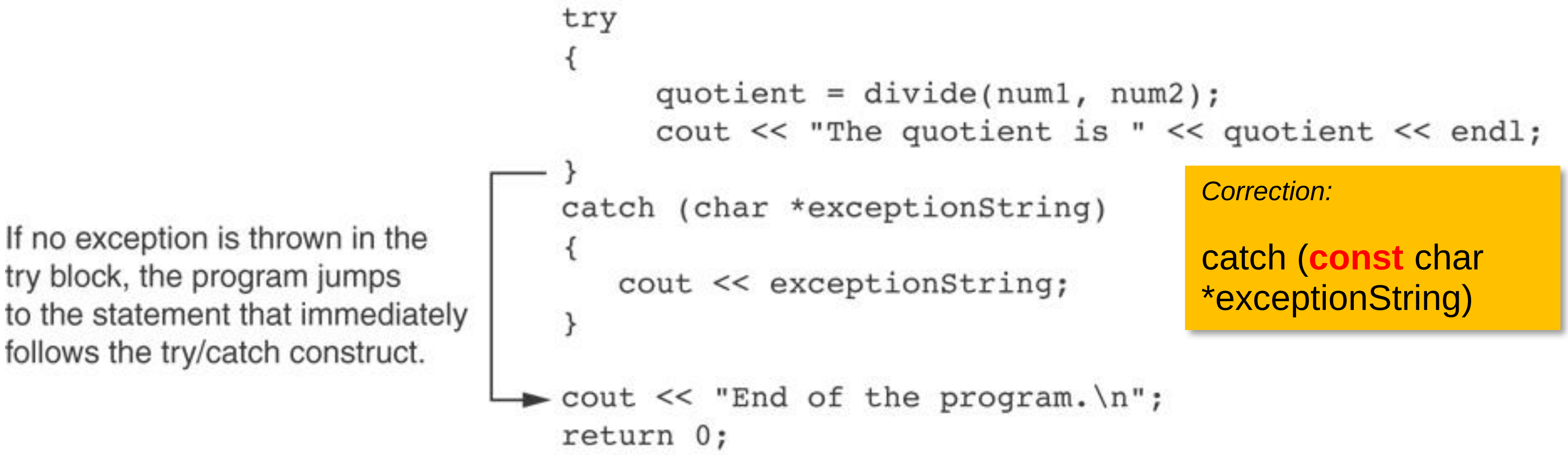
After the catch block is finished, the program resumes here.

```
try
{
    quotient = divide(num1, num2);
    cout << "The quotient is " << quotient << endl;
}
catch (char *exceptionString)
{
   cout << exceptionString;
}

cout << "End of the program.\n";
return 0;
```

Correction:

catch (**const** char *exceptionString)

Example 2: What Happens if No Exception is Thrown?



If no exception is thrown in the try block, the program jumps to the statement that immediately follows the try/catch construct.

```
try
{
     quotient = divide(num1, num2);
     cout << "The quotient is " << quotient << endl;
}
catch (char *exceptionString)
{
   cout << exceptionString;
}

cout << "End of the program.\n";
return 0;
```

Correction:

catch (**const** char *exceptionString)

Exception Not Caught?

- An exception will not be caught if
  - it is thrown from outside of a `try` block
  - there is no `catch` block that matches the data type of the thrown exception

- If an exception is not caught, the program will terminate.

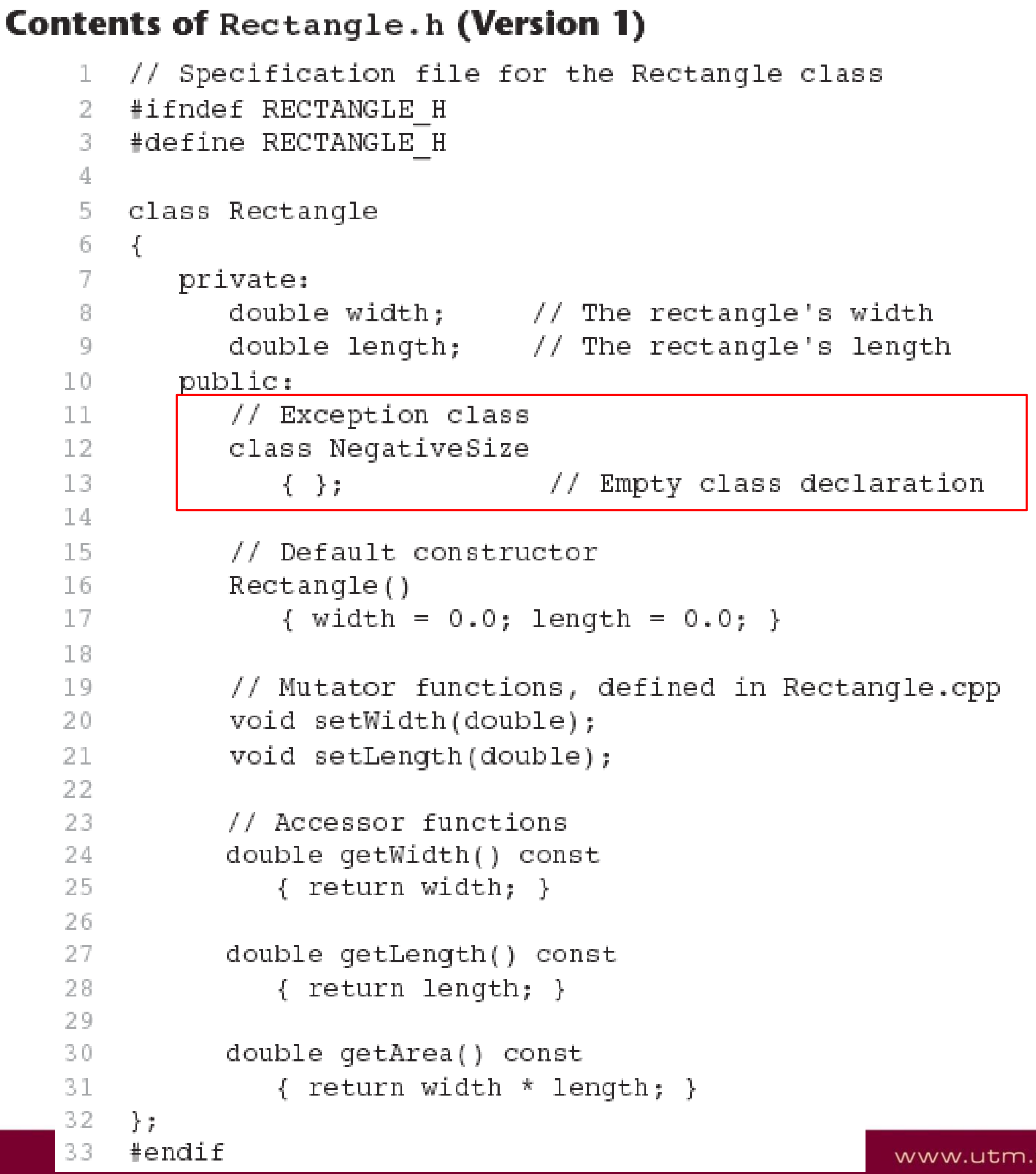
Exceptions and Objects

- An **exception class** can be defined in a class and thrown as an exception by a member function.

- An exception class may have:
  - no members: used only to signal an error
  - members: pass error data to `catch` block.

- A class can have more than one exception class.

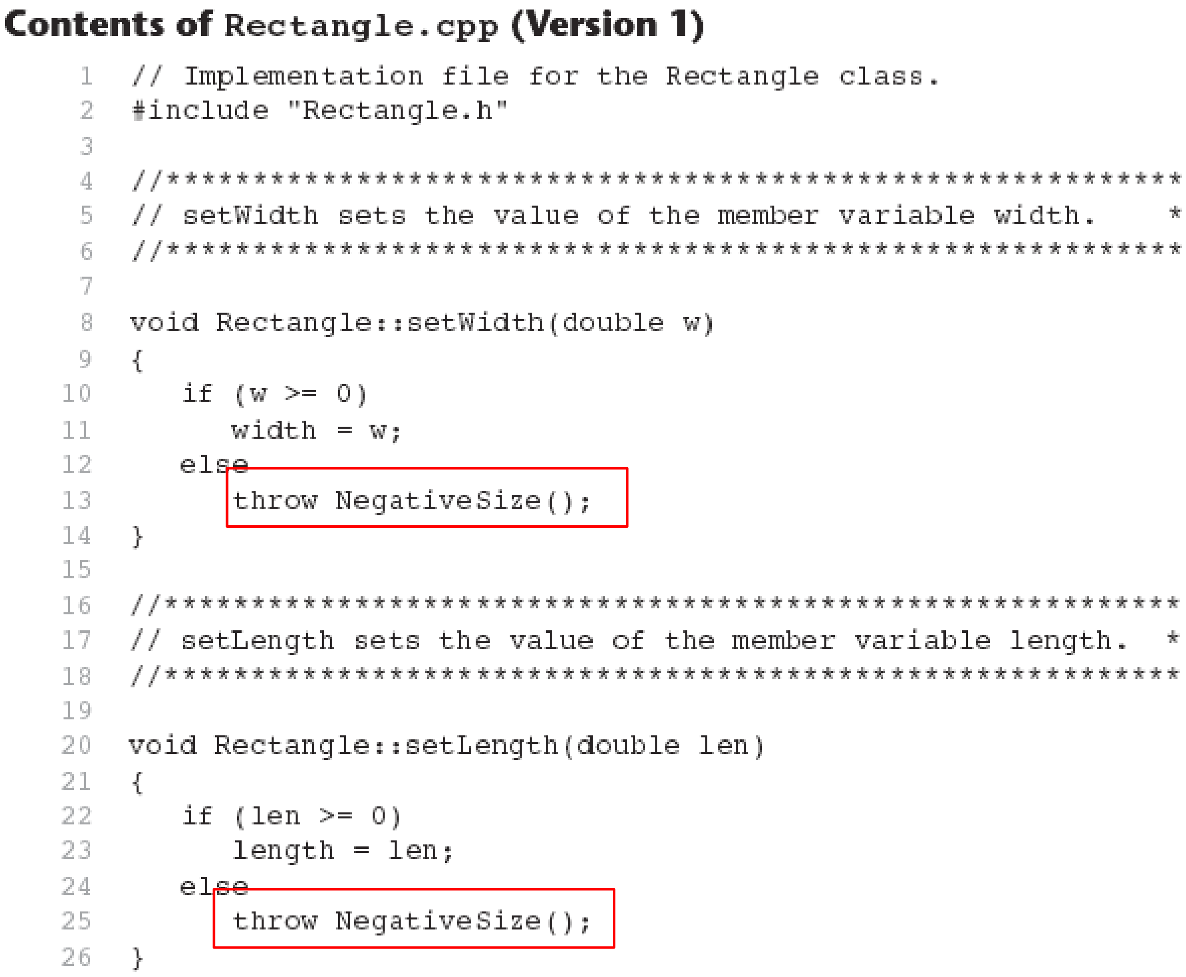
Example 3a



Contents of Rectangle.h (Version 1)

```
// Specification file for the Rectangle class
#ifndef RECTANGLE_H
#define RECTANGLE_H

class Rectangle
{
   private:
      double width;          // The rectangle's width
      double length;         // The rectangle's length
   public:
      // Exception class
      class NegativeSize
         { };                // Empty class declaration

      // Default constructor
      Rectangle()
         { width = 0.0; length = 0.0; }

      // Mutator functions, defined in Rectangle.cpp
      void setWidth(double);
      void setLength(double);

      // Accessor functions
      double getWidth() const
         { return width; }

      double getLength() const
         { return length; }

      double getArea() const
         { return width * length; }
};
#endif
```

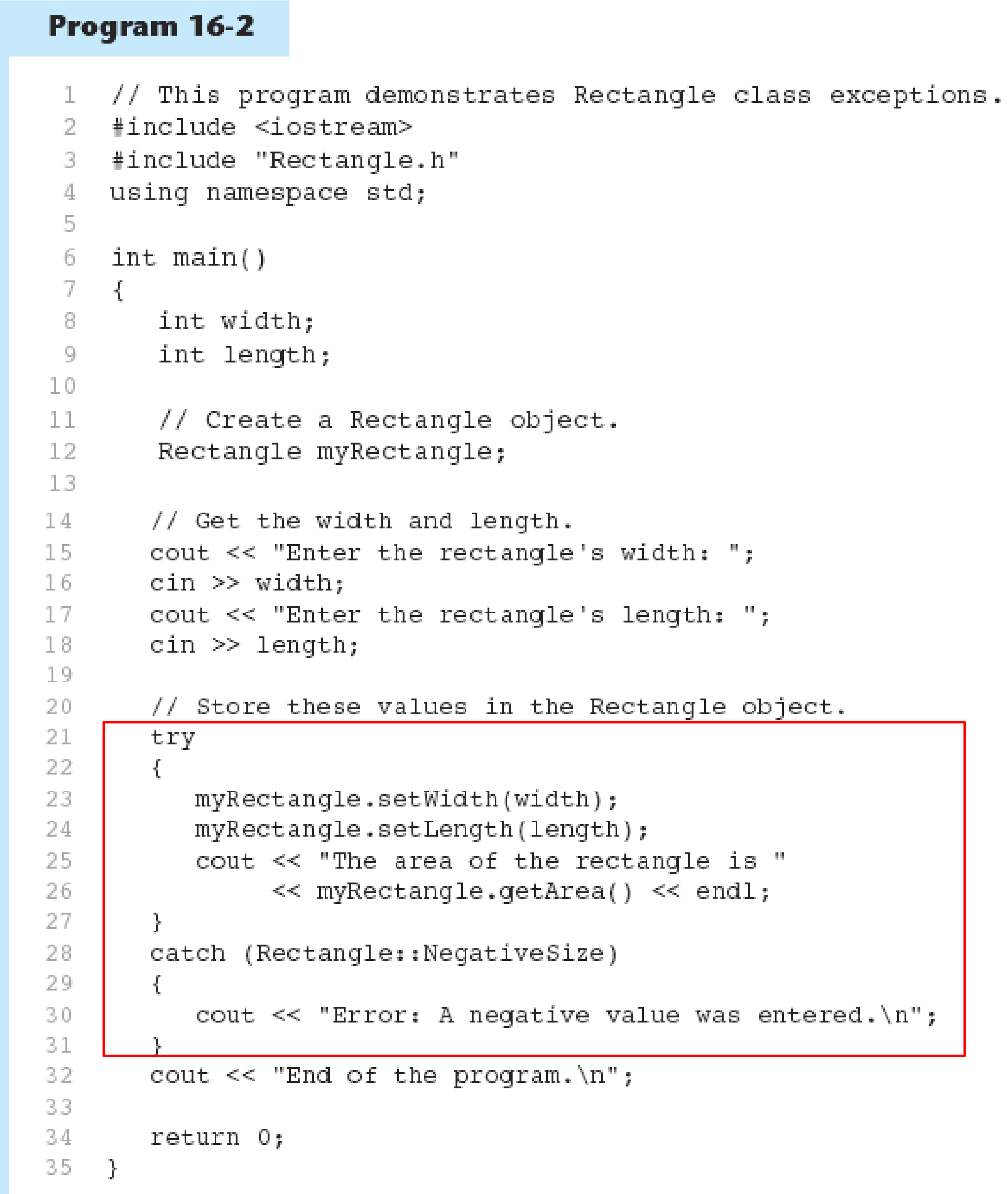
Example 3b



Contents of `Rectangle.cpp` (Version 1)

```
// Implementation file for the Rectangle class.
#include "Rectangle.h"

//***********************************************************
// setWidth sets the value of the member variable width.    *
//***********************************************************

void Rectangle::setWidth(double w)
{
   if (w >= 0)
      width = w;
   else
      throw NegativeSize();
}

//***********************************************************
// setLength sets the value of the member variable length.  *
//***********************************************************

void Rectangle::setLength(double len)
{
   if (len >= 0)
      length = len;
   else
      throw NegativeSize();
}
```

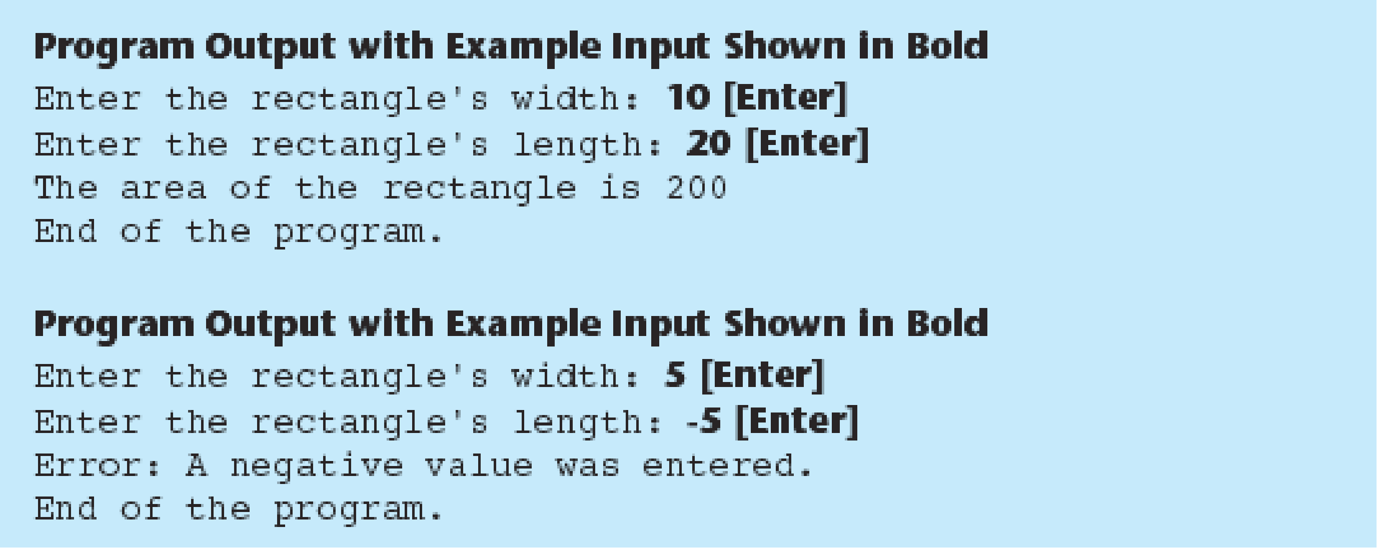
Example 3c



Program 16-2

```
// This program demonstrates Rectangle class exceptions.
#include <iostream>
#include "Rectangle.h"
using namespace std;

int main()
{
   int width;
   int length;

   // Create a Rectangle object.
   Rectangle myRectangle;

   // Get the width and length.
   cout << "Enter the rectangle's width: ";
   cin >> width;
   cout << "Enter the rectangle's length: ";
   cin >> length;

   // Store these values in the Rectangle object.
   try
   {
      myRectangle.setWidth(width);
      myRectangle.setLength(length);
      cout << "The area of the rectangle is "
           << myRectangle.getArea() << endl;
   }
   catch (Rectangle::NegativeSize)
   {
      cout << "Error: A negative value was entered.\n";
   }
   cout << "End of the program.\n";

   return 0;
}
```

Example 3d: Sample Output



Program Output with Example Input Shown in Bold

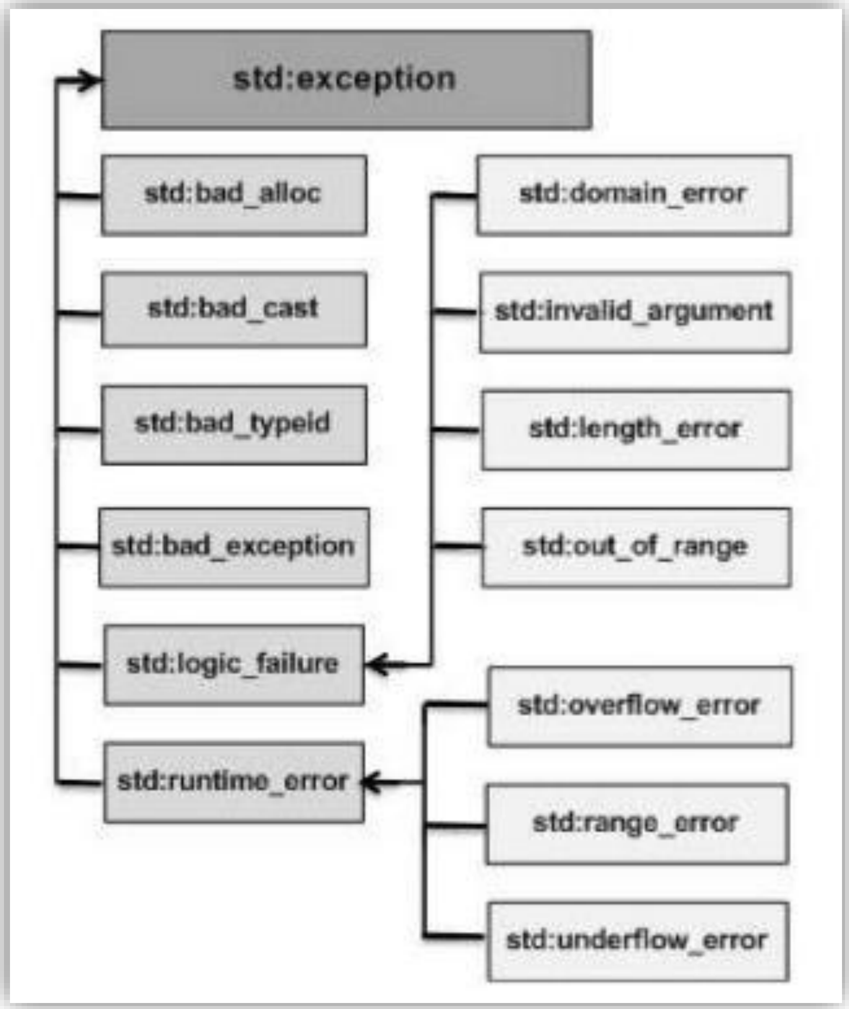
```
Enter the rectangle's width: 10 [Enter]
Enter the rectangle's length: 20 [Enter]
The area of the rectangle is 200
End of the program.
```

Program Output with Example Input Shown in Bold

```
Enter the rectangle's width: 5 [Enter]
Enter the rectangle's length: -5 [Enter]
Error: A negative value was entered.
End of the program.
```

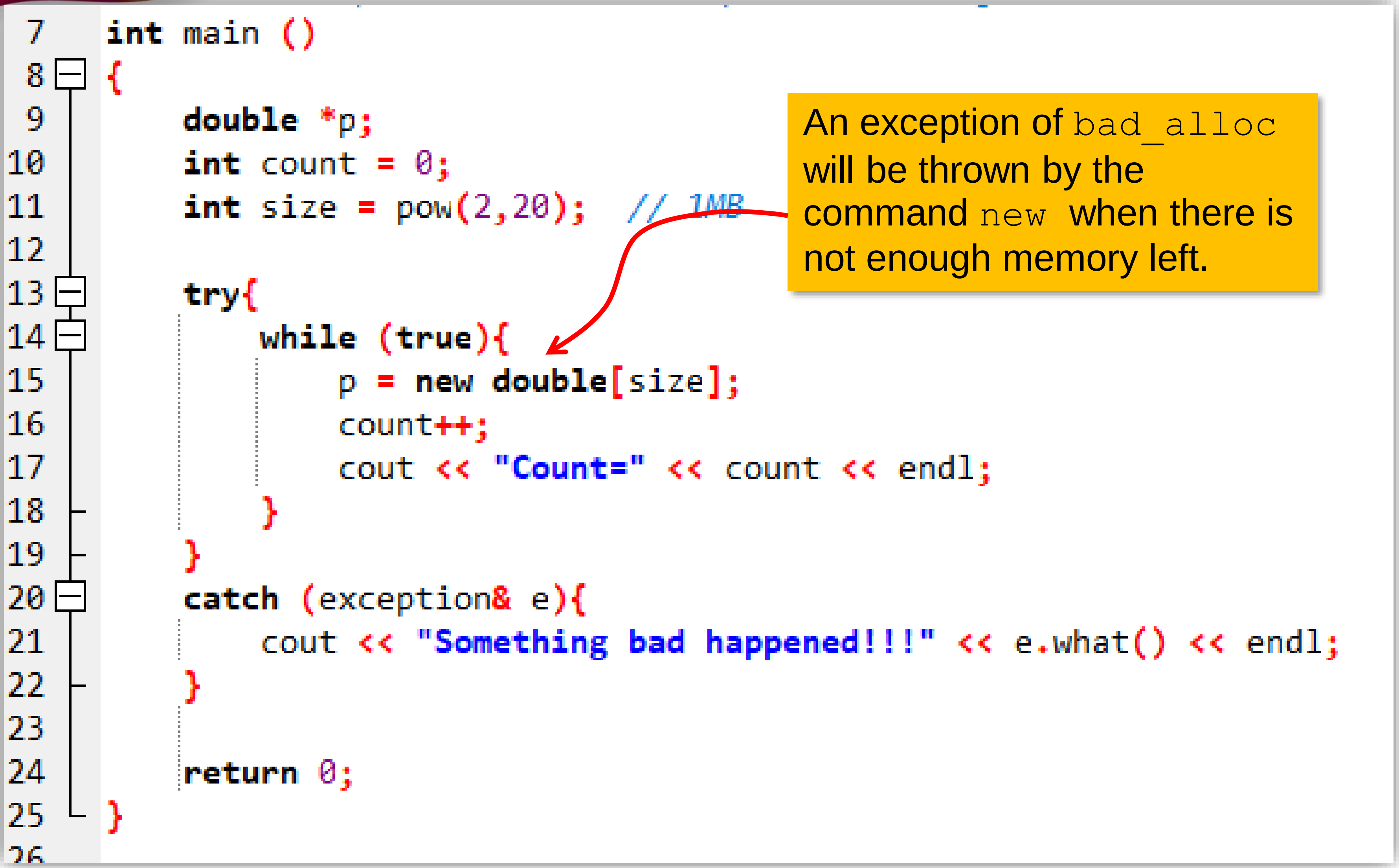
Additional Notes:

C++ Built-in Exception Classes



Source:
https://www.tutorialspoint.com/cplusplus/cpp_exceptions_handling.htm

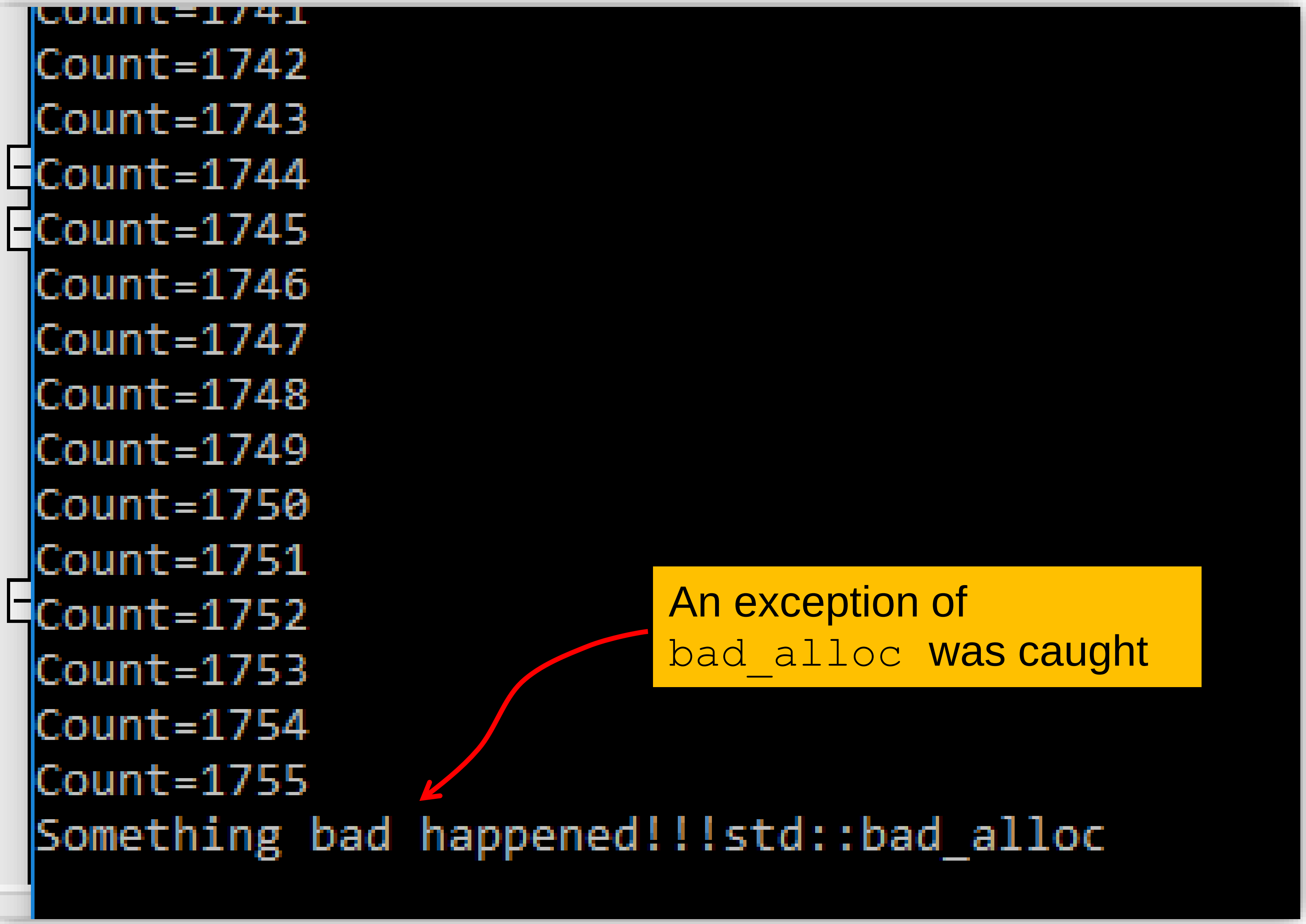
Example



```
int main ()
{
    double *p;
    int count = 0;
    int size = pow(2,20);   // 1MB

    try{
        while (true){
            p = new double[size];
            count++;
            cout << "Count=" << count << endl;
        }
    }
    catch (exception& e){
        cout << "Something bad happened!!!" << e.what() << endl;
    }

    return 0;
}
```

An exception of `bad_alloc` will be thrown by the command `new` when there is not enough memory left.

Example (continued)



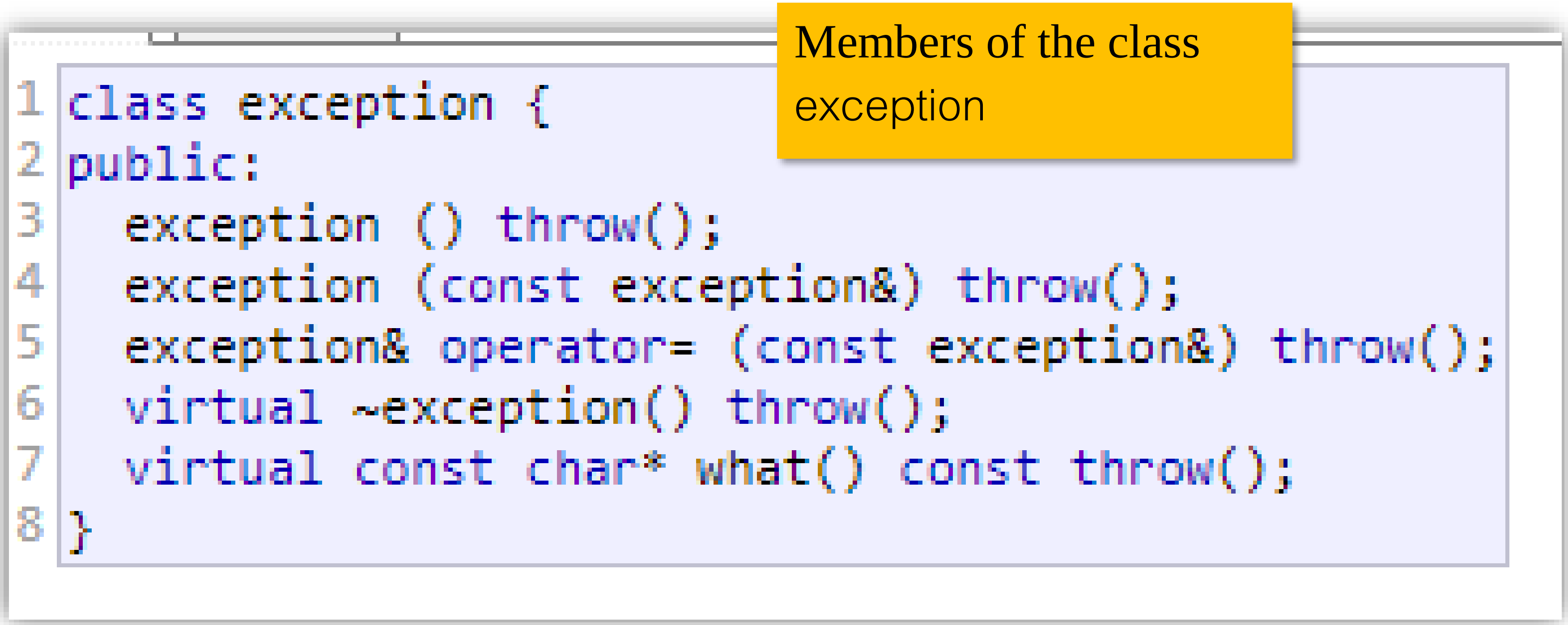
```
Count=1741
Count=1742
Count=1743
Count=1744
Count=1745
Count=1746
Count=1747
Count=1748
Count=1749
Count=1750
Count=1751
Count=1752
Count=1753
Count=1754
Count=1755
Something bad happened!!!std::bad_alloc
```

An exception of `bad_alloc` was caught

Additional Notes:

Creating a New Exception Class by Extending the class **exception**

- An exception class can also be defined outside of a class by **extending** the built-in classes e.g., the **class exception**

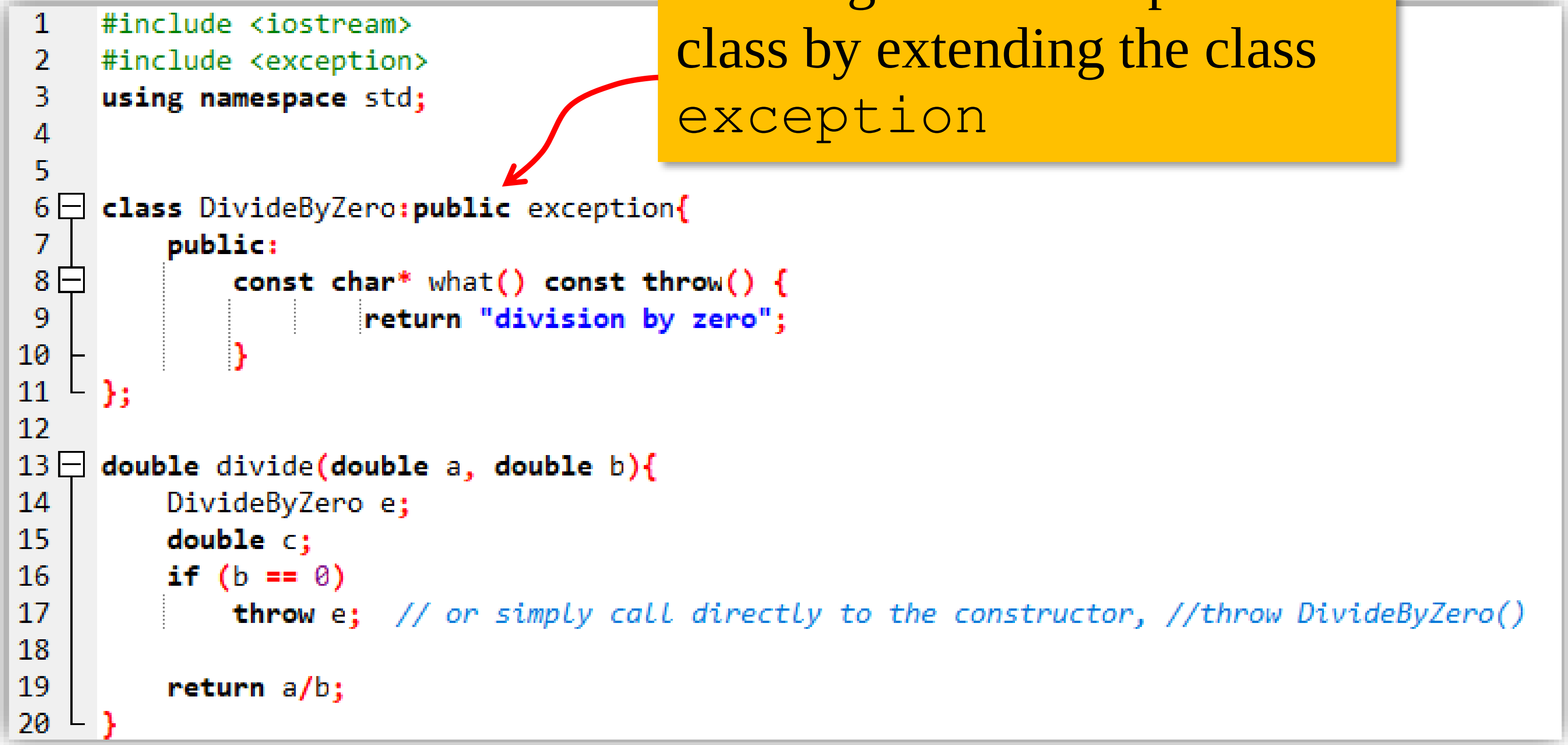


```
class exception {
public:
  exception () throw();
  exception (const exception&) throw();
  exception& operator= (const exception&) throw();
  virtual ~exception() throw();
  virtual const char* what() const throw();
}
```

Members of the class exception

Source:
http://www.cplusplus.com/doc/tutorial/exceptions/

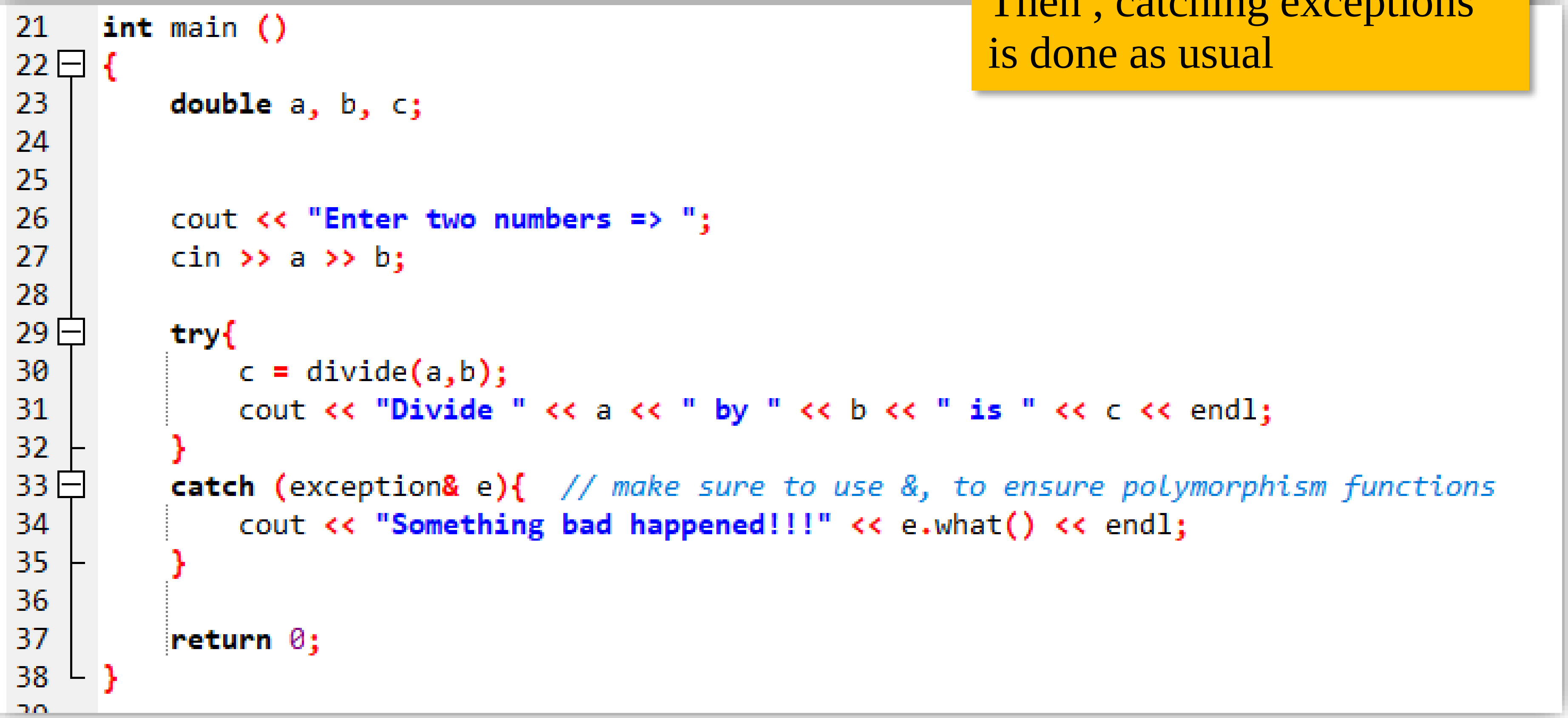
Example



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


class DivideByZero:public exception{
    public:
        const char* what() const throw() {
                return "division by zero";
        }
};

double divide(double a, double b){
    DivideByZero e;
    double c;
    if (b == 0)
        throw e;  // or simply call directly to the constructor, //throw DivideByZero()

    return a/b;
}
```

Creating a new exception class by extending the class `exception`

Then , catching exceptions is done as usual



```
int main ()
{
    double a, b, c;


    cout << "Enter two numbers => ";
    cin >> a >> b;

    try{
        c = divide(a,b);
        cout << "Divide " << a << " by " << b << " is " << c << endl;
    }
    catch (exception& e){  // make sure to use &, to ensure polymorphism functions
        cout << "Something bad happened!!!" << e.what() << endl;
    }

    return 0;
}
```

Function Templates

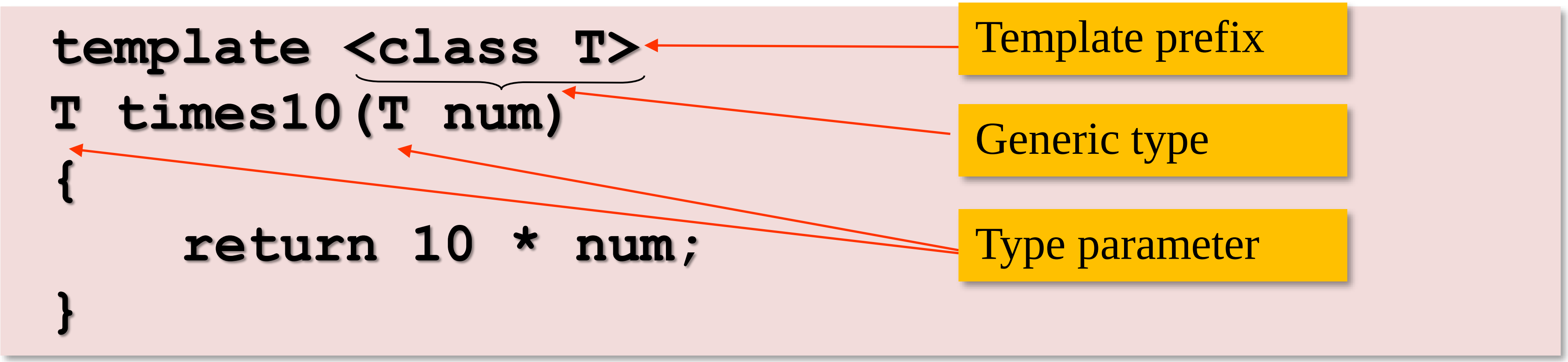
Introduction

- **Function template**: a pattern for a function that can work with many data types.

- When written, parameters are left for the data types.

- When called, compiler generates code for specific data types in function call.

Example 4a



```
template <class T>
T times10(T num)
{
     return 10 * num;
}
```

Template prefix

Generic type

Type parameter

What gets generated when **`times10`** is called with an **`int`**:	What gets generated when **`times10`** is called with a **`double`**:

`int times10(int num)`
`{`
`   return 10 * num;`
`}`
	`double times10(double num)`
	`{`
	`   return 10 * num;`
	`}`

Example 4b

```
template <class T>
T times10(T num)
{
   return 10 * num;
}
```

Call a template function in the usual manner:

```
int ival = 3;
double dval = 2.55;
cout << times10(ival); // displays 30
cout << times10(dval); // displays 25.5
```

Notes 1

- Function templates can be overloaded.

- Each template must have a unique parameter list.

```
template <class T>
T sumAll(T num) ...
template <class T1, class T2>
T1 sumall(T1 num1, T2 num2) ...
```

Notes 2

- All data types specified in template prefix **must be used in template definition**.

- Function calls **must pass parameters for all data types specified in the template prefix**.

- Like regular functions, function templates **must be defined before being called**.

Where to Start When Defining Templates

- Templates are often appropriate **for multiple functions** that perform the same task with **different parameter data types**.

- Develop function using usual data types first, then convert to a template:
  - add template prefix
  - convert data type names in the function to a type parameter (*i.e.*, a T type) in the template.

Class Templates

Introduction

- Classes can also be represented by templates.

- When a class object is created, type information is supplied to define the type of data members of the class.

- Unlike functions, classes are instantiated by supplying the type name (**`int`**, **`double`** , **`string`** , etc.) at object definition.

Example 5a

```
template <class T>
class Grade
{
    private:
         T score;
    public:
         Grade(T);
         void setGrade(T);
         T getGrade()
};
```

Example 5b

- Pass type information to class template when defining objects:

```
Grade<int> testList[20];
Grade<double> quizList[20];
```

- Use as ordinary objects once defined