



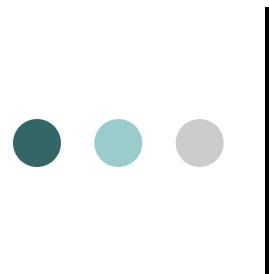
Chapter 16: Exceptions and Templates

Kafi Rahman
Assistant Professor
Truman State University

Multiple Exception Handling Structure

```
1 // exceptions
2 #include <iostream>
3 using namespace std;
4
5 int main () {
6     try {
7         // code here
8     }
9     catch (int param) { cout << "int exception"; }
10    catch (char param) { cout << "char exception"; }
11    catch (...) { cout << "default exception"; }
12
13    return 0;
14 }
```

- The program will sequentially try to match the thrown exception with the catch blocks
- If none of the catch blocks match then the **default catch block** will be executed to handle the error.



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 (exit)

Example: Handle and Correct Error

```
// exceptions
#include <iostream>
using namespace std;
const int NEGATIVE_VALUE_A= 10;
const int NEGATIVE_VALUE_B= 20;
// function declaration
int addTwoNums(int, int);

// main function
int main () {
    try {
        int s = addTwoNums(10, - 20);
        cout<<"\nSum is: "<<s;
    }
    catch (int param)
    { cout << "int exception";
    }
    // this will catch things
    catch (...)
    { cout << "default exception";
    }

    return 0;
}

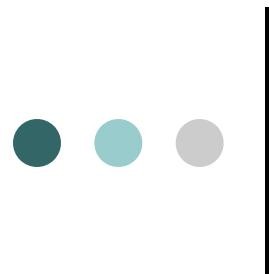
// definition
int addTwoNums(int a, int b)
{ int sum =0;
    try{
        if(b<0)
            throw NEGATIVE_VALUE_B;
        if(a<0)
            throw NEGATIVE_VALUE_A;
        // calculate the sum
        sum = a + b;
    }
    catch(int e)
    { if(e ==NEGATIVE_VALUE_A)
        { a = abs(a);
        }
        else if(e== NEGATIVE_VALUE_B)
        { b = abs(b);
        }
        // calculate the correct sum
        sum = a + b;
    }
    // return the sum
    return sum;
}
```



Nested try Blocks

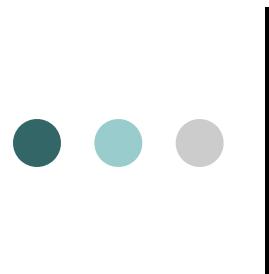
- try/catch blocks can occur within an enclosing try block
- Exceptions caught at an inner level can be passed up to a catch block at an outer level:

```
try
{
    try
    {
        ...      // one or more statements, and then
        throw 20;
    }
}
catch (int e)
{
    // handle the exception
}
```



Exceptions and Objects

- An exception class can be defined and thrown as an exception by a member function in a written program
- An exception class may have:
 - no members: used only to signal an error
 - members: pass error data to catch block
- A program can have more than one exception classes if required

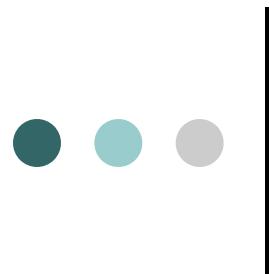


Exceptions and Objects

```
// exception class
class NegativeSize
{

};

// Rectangle class
class Rectangle
{
    private:
        int width, height;
    public:
        Rectangle()
        {
            this->width = 0;
            this->height = 0;
        }
        void set_width(int w);
        void set_height(int h);
        int get_area();
};
```

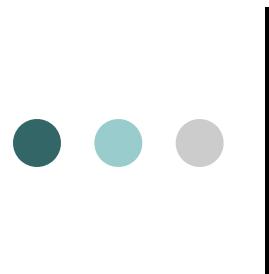


Exceptions and Objects

```
void Rectangle::set_width(int w)
{
    if(w<0)
        throw NegativeSize();
    this->width = w;
}

void Rectangle::set_height(int h)
{
    if(h<0)
        throw NegativeSize();
    this->height = h;
}

int Rectangle::get_area()
{
    return width * height;
}
```



Exceptions and Objects

```
int main()
{
    int w, h;
    Rectangle c_rect;
    cout<<"Enter height and width: ";
    cin>> w >>h;

    try
    {
        c_rect.set_width(w);
        c_rect.set_height(h);
        cout<<"The area of the rectangle is: "
            <<c_rect.get_area() << endl;
    }
    catch(const NegativeSize &e)
    {
        cout<<"Error: Negative value was entered .. " << endl;
    }
    catch(...)
    {
        cout<<"Unknown error occurred" << endl;
    }
    return 0;
}
```



Using Built-in Exception Classes

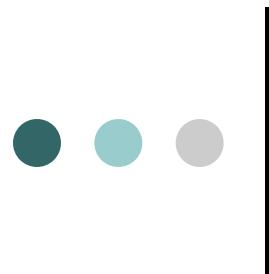
- All exceptions thrown by components of the C++ Standard library throw exceptions derived from the exception class.
- We have to include <exception> in our program to use these classes
- For example, bad_alloc is an exception class
 - The object of this class is thrown by new when it fails to allocate memory



Using Exception Classes

- The exception that may be caught by the exception handler in this example is a `bad_alloc`.
- Because `bad_alloc` is derived from the standard base class `exception`
- Exception description is found in the `e.what()` function

```
1 // bad_alloc standard exception
2 #include <iostream>
3 #include <exception>
4 using namespace std;
5
6 int main () {
7     try
8     {
9         int* myarray= new int[1000];
10    }
11    catch (exception& e)
12    {
13        cout << "Standard exception: "
14        << e.what() << endl;
15    }
16    return 0;
17 }
```



Using Exception Classes

exception	description
<code>logic_error</code>	error related to the internal logic of the program
<code>runtime_error</code>	error detected during runtime

- These classes are also derived from the exception class
- These are two popular generic exception classes that can be inherited by custom exceptions to report errors

Runtime Exception Classes

```
8 // Defining function Division
9 float Division(float num, float den)
10 {
11     // If denominator is Zero
12     // throw runtime_error
13     if (den == 0) {
14         throw runtime_error("Math error: Attempted to divide by Zero\n");
15     }
16     ..
17     // Otherwise return the result of division
18     return (num / den);
19     ..
20 } // end Division
21 ..
```

Runtime Exception Classes

```
22 int main()
23 {   float numerator, denominator, result;
24     numerator = 12.5;
25     denominator = 0;
26     *
27     // try block calls the Division function
28     try {
29         result = Division(numerator, denominator);
30     *
31         // this will not print in this example
32         cout << "The quotient is " << result << endl;
33     }
34     // catch block catches exception thrown by the Division function
35     catch (runtime_error& e) {
36         // prints that exception has occurred calls the what function
37         // using runtime_error object
38         cout << "Exception occurred" << endl
39             << e.what();
40     }
41 } // end main
```

Extending Exception Classes

: custom exception class

```
9 // User defined class for handling exception
10 // Class Exception publicly inherits
11 // the runtime_error class
12 ..
13 class Exception : public runtime_error {
14 public:
15     // Defining constructor of class Exception
16     // that passes a string message to the runtime_error class
17     Exception()
18         : runtime_error("Math error: Attempted to divide by Zero\n")
19     {
20     }
21 };
```

Extending Exception Classes

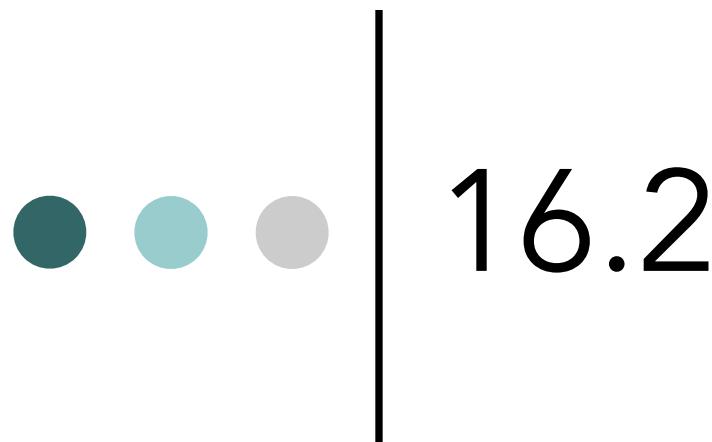
: Throwing the custom class

```
23 // defining Division function
24 float Division(float num, float den)
25 {   // If denominator is Zero
26     // throw user defined exception of type Exception
27     if (den == 0)
28         throw Exception();
29     .
30     // otherwise return the result of division
31     return (num / den);
32 } // end Division
```

Extending Exception Classes

: Catching the custom class

```
34 int main()
35 {   float numerator, denominator, result;
36     numerator = 12.5;
37     denominator = 0;
38     ..
39     // try block calls the Division function
40     try {
41         result = Division(numerator, denominator);
42         // this will not print in this example
43         cout << "The quotient is " << result << endl;
44     }
45     ..
46     // catch block catches exception if any of type Exception
47     catch (Exception& e) {
48         // prints that exception has occurred calls the what function
49         // using object of the user defined class called Exception
50         cout << "Exception occurred" << endl
51             << e.what();
52     }
53 } // end main
```



Function Templates



Function Templates

- 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

Function Template Example

```
// template prefix
template <class T> // class T is generic data type
T times10(T num) // T is type parameter
{    return 10 * num;
}
```

What gets generated when times10 is called with an int:

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

What gets generated when times10 is called with a double:

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

Function Template Example

```
// template prefix
template <class T>
// class T is generic data type
T times10(T num)
// T is type parameter
{    return 10 * num;
}
```

```
int main()
{
    int ival = 3;
    double dval = 2.55;
    // displays 30
    cout << times10(ival)<<endl;
    // displays 25.5
    cout << times10(dval)<<endl;
    // not an error: 320 % 256 = 64
    cout << times10(' ') << endl;
    // error: unsupported binary operation
    //cout << times10("A") << endl;
}
```