Escola de Engenharia
Universidade do Minho

Complementos de Programação de Computadores – Aula 6b

Tratamento de Exceções

Mestrado Integrado em Electrónica Industrial e Computadores

Luís Paulo Reis

lpreis@dsi.uminho.pt

Professor Associado do Departamento de Sistemas de Informação, Escola de Engenharia, Universidade do Minho, Portugal

(Slides Baseados em P.Deitel e H.Deitel 2010)

Escola de Engenharia
Universidade do Minho

Outline

Escola de Engenharia
Universidade do Minho

Objectives

- To use `try`, `throw` and `catch` to watch for, indicate and handle exceptions, respectively
- To process uncaught and unexpected exceptions
- To be able to process `new` failures
- To use `auto_ptr` to prevent memory leaks
- To understand the standard exception hierarchy.

Escola de Engenharia
Universidade do Minho

23.1 Introduction

- **Errors can be dealt with at place error occurs**
  - Easy to see if proper error checking implemented
  - Harder to read application itself and see how code works
- **Exception handling**
  - Makes clear, robust, fault-tolerant programs
  - C++ removes error handling code from "main line" of program
- **Common failures**
  - `new` not allocating memory
  - Out of bounds array subscript
  - Division by zero
  - Invalid function parameters

Escola de Engenharia
Universidade do Minho

23.1 Introduction (II)

- **Exception handling - catch errors before they occur**
  - Deals with synchronous errors (i.e., Divide by zero)
  - Does not deal with asynchronous errors - disk I/O completions, mouse clicks - use interrupt processing
  - Used when system can recover from error
    - Exception handler - recovery procedure
  - Typically used when error dealt with in different place than where it occurred
  - Useful when program cannot recover but must shut down cleanly
- **Exception handling should not be used for program control**
  - Not optimized, can harm program performance

Escola de Engenharia
Universidade do Minho

23.1 Introduction (III)

- **Exception handling improves fault-tolerance**
  - Easier to write error-processing code
  - Specify what type of exceptions are to be caught
- **Most programs support only single threads**
  - Techniques in this chapter apply for multithreaded OS as well (windows NT, OS/2, some UNIX)
- **Exception handling another way to return control from a function or block of code**

23.2 When Exception Handling Should Be Used

- **Error handling should be used for**
  - Processing exceptional situations
  - Processing exceptions for components that cannot handle them directly
  - Processing exceptions for widely used components (libraries, classes, functions) that should not process their own exceptions
  - Large projects that require uniform error processing

23.3 Other Error-Handling Techniques

- **Use `assert`**
  - If assertion `false`, the program terminates
- **Ignore exceptions**
  - Use this "technique" on casual, personal programs - not commercial!
- **Abort the program**
  - Appropriate for nonfatal errors give appearance that program functioned correctly
  - Inappropriate for mission-critical programs, can cause resource leaks
- **Set some error indicator**
  - Program may not check indicator at all points where error could occur

23.3 Other Error-Handling Techniques (II)

- **Test for the error condition**
  - Issue an error message and call `exit`
  - Pass error code to environment
- **`setjump` and `longjump`**
  - In `<csetjmp>`
  - Jump out of deeply nested function calls back to an error handler.
  - Dangerous - unwinds the stack without calling destructors for automatic objects (more later)
- **Specific errors**
  - Some have dedicated capabilities for handling them
  - If new fails to allocate memory `new_handler` function executes to deal with problem

23.4 Basics of C++ Exception Handling: `try`, `throw`, `catch`

- **A function can `throw` an exception object if it detects an error**
  - Object typically a character string (error message) or class object
  - If exception handler exists, exception caught and handled
  - Otherwise, program terminates
- **Format**
  - Enclose code that may have an error in `try` block
  - Follow with one or more `catch` blocks
    - Each `catch` block has an exception handler
  - If exception occurs and matches parameter in `catch` block, code in catch block executed
  - If no exception thrown, exception handlers skipped and control resumes after catch blocks
  - `throw` point - place where exception occurred
    - Control cannot return to `throw` point

23.5 A Simple Exception-Handling Example: Divide by Zero

- Look at the format of `try` and `catch` blocks
- Afterwards, we will cover specifics


```
1  // Fig. 23.1: fig23_01.cpp
2  // A simple exception handling example.
3  // Checking for a divide-by-zero exception.
4  #include <iostream>
5
6  using std::cout;
7  using std::cin;
8  using std::endl;
9
10 // Class DivideByZeroException to be used in exception
11 // handling for throwing an exception on a division by zero.
12 class DivideByZeroException {
13 public:
14    DivideByZeroException()
15       : message( "attempted to divide by zero" ) { }
16    const char *what() const { return message; }
17 private:
18    const char *message;
19 }; // end class DivideByZeroException
20
21 // Definition of function quotient. Demonstrates throwing
22 // an exception when a divide-by-zero exception is encountered.
23 double quotient( int numerator, int denominator )
24 {
25    if ( denominator == 0 )
26       throw DivideByZeroException();
27
```

fig23_01.cpp (Part 1 of 3)

```
28    return static_cast< double > ( numerator ) / denominator;
29 } // end function quotient
30
31 // Driver program
32 int main()
33 {
34    int number1, number2;
35    double result;
36
37    cout << "Enter two integers (end-of-file to end): ";
38
39    while ( cin >> number1 >> number2 ) {
40
41       // the try block wraps the code that may throw an
42       // exception and the code that should not execute
43       // if an exception occurs
44       try {
45          result = quotient( number1, number2 );
46          cout << "The quotient is: " << result << endl;
47       } // end try
```

fig23_01.cpp (Part 2 of 3)


```
48       catch ( DivideByZeroException ex ) { // exception handler
49          cout << "Exception occurred: " << ex.what() << '\n';
50       } // end catch
51
52       cout << "\nEnter two integers (end-of-file to end): ";
53    } // end while
54
55    cout << endl;
56    return 0;      // terminate normally
57 } // end function main
```

fig23_01.cpp (Part 3 of 3)

Program Output

```
Enter two integers (end-of-file to end): 100 7
The quotient is: 14.2857

Enter two integers (end-of-file to end): 100 0
Exception occurred: attempted to divide by zero

Enter two integers (end-of-file to end): 33 9
The quotient is: 3.66667

Enter two integers (end-of-file to end):^Z
```

Escola de Engenharia
Universidade do Minho

23.6 Throwing an Exception

- **`throw` - indicates an exception has occurred**
  - Usually has one operand (sometimes zero) of any type
    - If operand an object, called an exception object
    - Conditional expression can be thrown
  - Code referenced in a `try` block can throw an exception
  - Exception caught by closest exception handler
  - Control exits current try block and goes to `catch` handler (if it exists)
  - Example (inside function definition)

```
if ( denominator == 0 )
   throw DivideByZeroException();
```

    - Throws a `dividebyzeroexception` object
- **Exception not required to terminate program**
  - However, terminates block where exception occurred

Escola de Engenharia
Universidade do Minho

23.7 Catching an Exception

- **Exception handlers are in `catch` blocks**
  - Format: `catch(` *exceptionType parameterName*`){`

```
        exception handling code
    }
```

  - Caught if argument type matches `throw` type
  - If not caught then `terminate` called which (by default) calls `abort`
  - Example:

```
catch ( DivideByZeroException ex) {
   cout << "Exception occurred: " << ex.what() <<'\n'
}
```

    - Catches exceptions of type `DivideByZeroException`
- **Catch all exceptions**

  `catch(...)` - catches all exceptions
    - You do not know what type of exception occurred
    - There is no parameter name - cannot reference the object

Escola de Engenharia
Universidade do Minho

23.7 Catching an Exception (II)

- **If no handler matches thrown object**
  - Searches next enclosing `try` block
    - If none found, `terminate` called
  - If found, control resumes after last `catch` block
  - If several handlers match thrown object, first one found is executed
- **`catch` parameter matches thrown object when**
  - They are of the same type
    - Exact match required - no promotions/conversions allowed
  - The `catch` parameter is a `public` base class of the thrown object
  - The `catch` parameter is a base-class pointer/ reference type and the thrown object is a derived-class pointer/ reference type
  - The `catch` handler is `catch( ... )`
  - Thrown `const` objects have `const` in the parameter type

Escola de Engenharia
Universidade do Minho

23.7 Catching an Exception (III)

- **Unreleased resources**
  - Resources may have been allocated when exception thrown
  - `catch` handler should `delete` space allocated by `new` and close any opened files
- **`catch` handlers can throw exceptions**
  - Exceptions can only be processed by outer `try` blocks

23.8 Rethrowing an Exception

- **Rethrowing exceptions**
  - Used when an exception handler cannot process an exception
  - Rethrow exception with the statement:
    `throw;`
    - No arguments
    - If no exception thrown in first place, calls `terminate`
  - Handler can always rethrow exception, even if it performed some processing
  - Rethrown exception detected by next enclosing `try` block


```
1  // Fig. 23.2: fig23_02.cpp
2  // Demonstration of rethrowing an exception.
3  #include <iostream>
4
5  using std::cout;
6  using std::endl;
7
8  #include <exception>
9
10 using std::exception;
11
12 void throwException()
13 {
14    // Throw an exception and immediately catch it.
15    try {
16       cout << "Function throwException\n";
17       throw exception();  // generate exception
18    } // end try
19    catch( exception e )
20    {
21       cout << "Exception handled in function throwException\n";
22       throw;  // rethrow exception for further processing
23    } // end catch
24
25    cout << "This also should not print\n";
26 } // end function throwException
27
```

fig23_02.cpp (Part 1 of 2)

```
28 int main()
29 {
30    try {
31       throwException();
32       cout << "This should not print\n";
33    } // end try
34    catch ( exception e )
35    {
36       cout << "Exception handled in main\n";
37    } // end catch
38
39    cout << "Program control continues after catch in main"
40         << endl;
41    return 0;
42 } // end function main
```

fig23_02.cpp (Part 2 of 2)

Program Output

```
Function throwException
Exception handled in function throwException
Exception handled in main
Program control continues after catch in main
```


Escola de Engenharia
Universidade do Minho

23.9 Exception Specifications

- **Exception specification (`throw` list)**
  - Lists exceptions that can be thrown by a function

  Example:

```
int g( double h ) throw ( a, b, c )
   {
      // function body
   }
```

  - Function can throw listed exceptions or derived types
  - If other type thrown, function `unexpected` called
  - `throw()` (i.e., no `throw` list) states that function will not throw any exceptions
    - In reality, function can still throw exceptions, but calls `unexpected` (more later)
  - If no `throw` list specified, function can `throw` any exception

23.10 Processing Unexpected Exceptions

- **Function `unexpected`**
  - Calls the function specified with `set_unexpected`
    - Default: `terminate`
- **Function `terminate`**
  - Calls function specified with `set_terminate`
    - Default: `abort`
- **`set_terminate` and `set_unexpected`**
  - Prototypes in `<exception>`
  - Take pointers to functions (i.e., Function name)
    - Function must return `void` and take no arguments
  - Returns pointer to last function called by `terminate` or `unexpected`

23.11 Stack Unwinding

- **Function-call stack unwound when exception thrown and not caught in a particular scope**
  - Tries to catch exception in next outer `try/catch` block
  - Function in which exception was not caught terminates
    - Local variables destroyed
    - Control returns to place where function was called
  - If control returns to a `try` block, attempt made to `catch` exception
    - Otherwise, further unwinds stack
  - If exception not caught, `terminate` called

```
1  // Fig. 23.3: fig23_03.cpp
2  // Demonstrating stack unwinding.
3  #include <iostream>
4
5  using std::cout;
6  using std::endl;
7
8  #include <stdexcept>
9
10 using std::runtime_error;
11
12 void function3() throw ( runtime_error )
13 {
14    throw runtime_error( "runtime_error in function3" );
15 } // end function function3
16
17 void function2() throw ( runtime_error )
18 {
19    function3();
20 } // end function function2
21
22 void function1() throw ( runtime_error )
23 {
24    function2();
25 } // end function function1
26
```

fig23_03.cpp (Part 1 of 2)


```
27 int main()
28 {
29    try {
30       function1();
31    } // end try
32    catch ( runtime_error e )
33    {
34       cout << "Exception occurred: " << e.what() << endl;
35    } // end catch
36
37    return 0;
38 } // end function main
```

fig23_03.cpp (Part 2 of 2)

Program Output

```
Exception occurred: runtime_error in function3
```

23.12 Constructors, Destructors and Exception Handling

- **What to do with an error in a constructor?**
  - A constructor cannot return a value - how do we let the outside world know of an error?
    - Keep defective object and hope someone tests it
    - Set some variable outside constructor
  - A thrown exception can tell outside world about a failed constructor
  - `catch` handler must have a copy constructor for thrown object

23.12 Constructors, Destructors and Exception Handling (II)

- **Thrown exceptions in constructors**
  - Destructors called for all completed base-class objects and member objects before exception thrown
  - If the destructor that is originally called due to stack unwinding ends up throwing an exception, `terminate` called
  - If object has partially completed member objects when exception thrown, destructors called for completed objects
- **Resource leak**
  - Exception comes before code that releases a resource
  - One solution: initialize local object when resource acquired
    - Destructor will be called before exception occurs
- **`catch` exceptions from destructors**
  - Enclose code that calls them in `try` block followed by appropriate `catch` block

23.13 Exceptions and Inheritance

- Exception classes can be derived from base classes
- If `catch` can get a pointer/reference to a base class, can also `catch` pointers/references to derived classes

23.14 Processing new Failures

- **If new could not allocate memory**
  - Old method - use `assert` function
    - If new returns 0, `abort` (Does not allow program to recover)
  - Modern method (header <new>)
    - new throws `bad_alloc` exception
  - Method used depends on compiler
  - On some compilers: use `new(nothrow)` instead of new to have new return 0 when it fails
    - Function `set_new_handler(`*functionName*`)` - sets which function is called when new fails.
    - Function can return no value and take no arguments
    - new will not throw `bad_alloc`
- **new : Loop that tries to acquire memory**
- **A new handler function should either:**
  - Make more memory available by deleting other dynamically allocated memory and return to the loop in operator new
  - Throw an exception of type `bad_alloc`
  - Call function `abort` or `exit` (header <cstdlib>) to terminate program

fig23_04.cpp

```
1  // Fig. 23.4: fig23_04.cpp
2  // Demonstrating new returning 0
3  // when memory is not allocated
4  #include <iostream>
5
6  using std::cout;
7
8  int main()
9  {
10    double *ptr[ 50 ];
11
12    for ( int i = 0; i < 50; i++ ) {
13       ptr[ i ] = new double[ 5000000 ];
14
15       if ( ptr[ i ] == 0 ) { // new failed to allocate memory
16          cout << "Memory allocation failed for ptr[ "
17               << i << " ]\n";
18          break;
19       } // end if
20       else
21          cout << "Allocated 5000000 doubles in ptr[ "
22               << i << " ]\n";
23    } // end for
24
25    return 0;
26 } // end function main
```

Program Output

```
Allocated 5000000 doubles in ptr[ 0 ]
Allocated 5000000 doubles in ptr[ 1 ]
Allocated 5000000 doubles in ptr[ 2 ]
Allocated 5000000 doubles in ptr[ 3 ]
Memory allocation failed for ptr[ 4 ]
```


fig23_05.cpp (Part 1 of 2)

```
1  // Fig. 23.5: fig23_05.cpp
2  // Demonstrating new throwing bad_alloc
3  // when memory is not allocated
4  #include <iostream>
5
6  using std::cout;
7  using std::endl;
8
9  #include <new>
10
11 using std::bad_alloc;
12
13 int main()
14 {
15    double *ptr[ 50 ];
16
17    try {
18       for ( int i = 0; i < 50; i++ ) {
19          ptr[ i ] = new double[ 5000000 ];
20          cout << "Allocated 5000000 doubles in ptr[ "
21               << i << " ]\n";
22       } // end for
23    } // end try
```

```
24    catch ( bad_alloc exception ) {
25       cout << "Exception occurred: "
26            << exception.what() << endl;
27    } // end catch
28
29    return 0;
30 } // end function main
```

```
Allocated 5000000 doubles in ptr[ 0 ]
Allocated 5000000 doubles in ptr[ 1 ]
Allocated 5000000 doubles in ptr[ 2 ]
Allocated 5000000 doubles in ptr[ 3 ]
Exception occurred: Allocation Failure
```

fig23_05.cpp (Part 2 of 2)

Program Output


```
1  // Fig. 23.6: fig23_06.cpp
2  // Demonstrating set_new_handler
3  #include <iostream>
4
5  using std::cout;
6  using std::cerr;
7
8  #include <new>
9  #include <cstdlib>
10
11 using std::set_new_handler;
12
13 void customNewHandler()
14 {
15    cerr << "customNewHandler was called";
16    abort();
17 } // end function customNewHandler
18
19 int main()
20 {
21    double *ptr[ 50 ];
22    set_new_handler( customNewHandler );
23
24    for ( int i = 0; i < 50; i++ ) {
25       ptr[ i ] = new double[ 5000000 ];
26
```

fig23_06.cpp (Part 1 of 2)

```
27       cout << "Allocated 5000000 doubles in ptr[ "
28            << i << " ]\n";
29    } // end for
30
31    return 0;
32 } // end function main
```

```
Allocated 5000000 doubles in ptr[ 0 ]
Allocated 5000000 doubles in ptr[ 1 ]
Allocated 5000000 doubles in ptr[ 2 ]
Allocated 5000000 doubles in ptr[ 3 ]
customNewHandler was called
```

fig23_06.cpp (Part 2 of 2)

Program Output

Escola de Engenharia
Universidade do Minho

23.15 Class `auto_ptr` and Dynamic Memory Allocation

- **Pointers to dynamic memory**
  - Memory leak can occur if exceptions happens before `delete` command
  - Use class template `auto_ptr` (header `<memory>`) to resolve this
  - `auto_ptr` objects act just like pointers
    - Automatically deletes what it points to when it is destroyed (leaves scope)
    - Can use * and -> like normal pointers

```
1  // Fig. 23.7: fig23_07.cpp
2  // Demonstrating auto_ptr
3  #include <iostream>
4
5  using std::cout;
6  using std::endl;
7
8  #include <memory>
9
10 using std::auto_ptr;
11
12 class Integer {
13 public:
14    Integer( int i = 0 ) : value( i )
15       { cout << "Constructor for Integer " << value << endl; }
16    ~Integer()
17       { cout << "Destructor for Integer " << value << endl; }
18    void setInteger( int i ) { value = i; }
19    int getInteger() const { return value; }
20 private:
21    int value;
22 }; // end class Integer
23
```

fig23_07.cpp (Part 1 of 2)


```
24 int main()
25 {
26    cout << "Creating an auto_ptr object that points "
27         << "to an Integer\n";
28
29    auto_ptr< Integer > ptrToInteger( new Integer( 7 ) );
30
31    cout << "Using the auto_ptr to manipulate the Integer\n";
32    ptrToInteger->setInteger( 99 );
33    cout << "Integer after setInteger: "
34         << ( *ptrToInteger ).getInteger()
35         << "\nTerminating program" << endl;
36
37    return 0;
38 } // end function main
```

fig23_07.cpp (Part 2 of 2)

Program Output

```
Creating an auto_ptr object that points to an Integer
Constructor for Integer 7
Using the auto_ptr to manipulate the Integer
Integer after setInteger: 99
Terminating program
Destructor for Integer 99
```

23.16 Standard Library Exception Hierarchy

- **Exceptions fall into categories**
  - Hierarchy of exception classes
  - Base class `exception` (header `<exception>`)
    - Function `what()` issues appropriate error message
  - Derived classes: `runtime_error` and `logic_error` (header `<stdexcept>`)
- **Class `logic_error`**
  - Errors in program logic, can be prevented by writing proper code
  - Derived classes:
    - `invalid_argument` - invalid argument passed to function
    - `length_error` - length larger than maximum size allowed was used
    - `out_of_range` - out of range subscript

23.16 Standard Library Exception Hierarchy (II)

- **Class `runtime_error`**
  - Errors detected at execution time
  - Derived classes:
    - `overflow_error` - arithmetic overflow
    - `underflow_error` - arithmetic underflow
- **Other classes derived from `exception`**
  - Exceptions thrown by C++ language features
    - `new - bad_alloc`
    - `dynamic_cast - bad_cast`
    - `typeid - bad_typeid`
  - Put `std::bad_exception` in `throw` list
    - `unexpected()` will `throw bad_exception` instead of calling function set by `set_unexpected`

Escola de Engenharia
Universidade do Minho

Complementos de Programação de Computadores – Aula 6b

Tratamento de Exceções

Mestrado Integrado em Electrónica Industrial e Computadores

Luís Paulo Reis

lpreis1970@gmail.com / lpreis@dsi.uminho.pt

Professor Associado do Departamento de Sistemas de Informação, Escola de Engenharia, Universidade do Minho, Portugal

(Slides Baseados em P.Deitel e H.Deitel 2010)