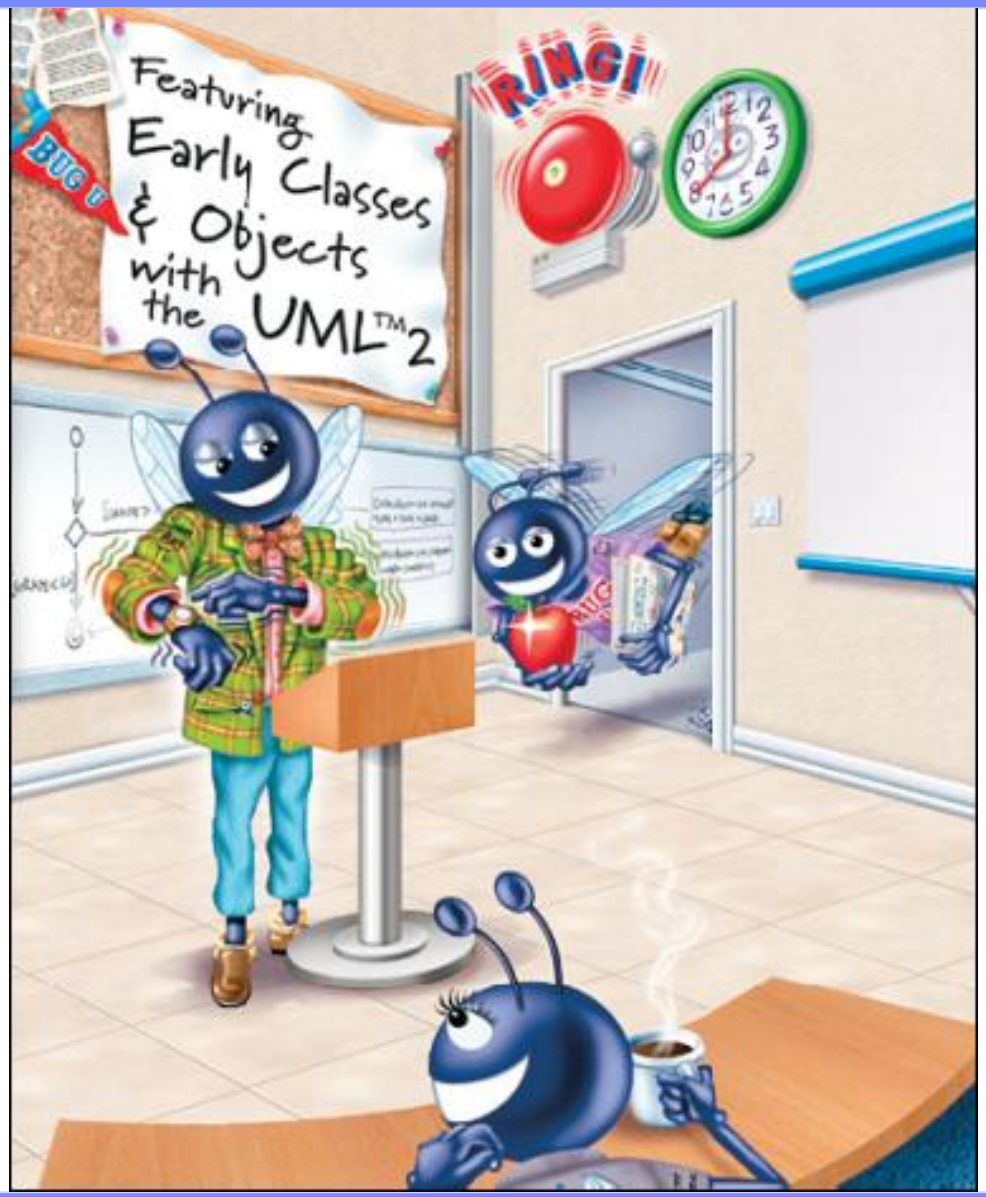


Lecture 13:
异常处理

第十六讲 异常处理

学习目标：

- 什么是异常
- 使用 try、catch 和 throw 来处理异常
- 处理 new 动态分配空间失败
- 利用 auto_ptr 阻止内存泄漏

1 Introduction

- 异常（Exceptions）
  - 程序运行过程中出现问题
  - 不经常出现

1 Introduction

- 异常处理（Exception handling）
  - 能够解析异常
    - ✓允许程序继续执行 或
    - ✓通知程序使用者 并
    - ✓以可控的方式终止程序
  - 使得程序健壮和容错

2 Exception-Handling Overview

- 程序和错误处理逻辑相混合
  - 伪代码：

```
Perform a task
If the preceding task did not execute correctly
   Perform error processing
Perform next task
If the preceding task did not execute correctly
   Perform error processing
…
```

  - 使得程序难于阅读，修改和维护

2 Exception-Handling Overview

- 异常处理
  - 将错误处理代码从程序执行的“主线”中去除
  - 程序员可以有选择的处理任意异常
    - ✓所有异常,
    - ✓某一类型的异常
    - ✓一组相关类型的异常

3 Example: Handling an Attempt to Divide by Zero

- exception 类
  - 标准 C++ 中所有异常的基类
  - 虚拟函数 what
    - ✓返回异常存储的错误信息

```
1  // Fig. 16.1: DivideByZeroException.h
2  // Class DivideByZeroException definition.
3  #include <stdexcept> // stdexcept header file contains runtime_error
4  using std::runtime_error; // standard C++ library class runtime_error
5
6  // DivideByZeroException objects should be thrown by functions
7  // upon detecting division-by-zero exceptions
8  class DivideByZeroException : public runtime_error
9  {
10 public:
11    // constructor specifies default error message
12    DivideByZeroException::DivideByZeroException()
13       : runtime_error( "attempted to divide by zero" ) {}
14 }; // end class DivideByZeroException
```

```cpp
// Fig. 16.2: Fig16_02.cpp
// A simple exception-handling example that checks for
// divide-by-zero exceptions.
#include <iostream>
using std::cin;
using std::cout;
using std::endl;

#include "DivideByZeroException.h" // DivideByZeroException class

// perform division and throw DivideByZeroException object if
// divide-by-zero exception occurs
double quotient( int numerator, int denominator )
{
   // throw DivideByZeroException if trying to divide by zero
   if ( denominator == 0 )
      throw DivideByZeroException(); // terminate function

   // return division result
   return static_cast< double >( numerator ) / denominator;
} // end function quotient

int main()
{
   int number1; // user-specified numerator
   int number2; // user-specified denominator
   double result; // result of division

   cout << "Enter two integers (end-of-file to end): ";
```

```
30
31    // enable user to enter two integers to divide
32    while ( cin >> number1 >> number2 )
33    {
34       // try block contains code that might throw exception
35       // and code that should not execute if an exception occurs
36       try
37       {
38          result = quotient( number1, number2 );
39          cout << "The quotient is: " << result << endl;
40       } // end try
41
42       // exception handler handles a divide-by-zero exception
43       catch ( DivideByZeroException &divideByZeroException )
44       {
45          cout << "Exception occurred: "
46             << divideByZeroException.what() << endl;
47       } // end catch
48
49       cout << "\nEnter two integers (end-of-file to end): ";
50    } // end while
51
52    cout << endl;
53    return 0; // terminate normally
54 } // end main
```

```
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): ^Z
```

3 Example: Handling an Attempt to Divide by Zero

- try 语句块
  - 关键字 try 后跟花括号 ({})
  - 语句块中应包含
    - ✓可能产生异常的语句
    - ✓异常发生后应跳过的语句

3 Example: Handling an Attempt to Divide by Zero

- catch 处理
  - 紧跟在 try 语句块后
    - ✓ 一个 try 语句块可以跟多个 catch 处理
  - 关键字 catch
  - 在括号内包含异常参数
    - ✓ 表示处理异常的类型
  - 当与 try 语句块中 throw 的异常类型匹配时执行
    - ✓ 应该是被抛出异常的基类

3 Example: Handling an Attempt to Divide by Zero

- 异常处理的终止模型
  - 当异常发生 try 语句块结束
    - ✓ try 语句块中的局部变量退出作用域
  - 与异常相匹配的 catch 处理程序被执行
  - 最后一个 catch 处理程序后面的语句被继续执行
    - ✓ 控制权不再返回到异常抛出点

3 Example: Handling an Attempt to Divide by Zero

- 堆栈展开（Stack unwinding）
  - 如果没有发现匹配的 catch 处理程序时发生
  - 程序试图在调用函数中定位其他的 try 语句块

3 Example: Handling an Attempt to Divide by Zero

- 抛出异常
  - 使用关键字 throw 跟着表示异常类型的操作数
    - ✓可以抛出任何类型的异常
      - ◈ 如果抛出一个对象，称为异常对象
  - 抛出的异常初始化匹配的 catch 处理程序中的异常参数

4 When to Use Exception Handling

- 何时使用异常处理
  - 处理同步错误
    - ✓错误在语句执行时产生
  - 不处理异步错误
    - ✓错误的出现与程序执行并行产生

5 Rethrowing an Exception

- 重新抛出异常
  - 空 throw; 语句
  - 当一个 catch 处理程序无法处理一个异常
  - 下一个 try 语句块试图匹配这个异常，相应的 catch 处理程序将处理异常

```
1  // Fig. 16.3: Fig16_03.cpp
2  // Demonstrating exception rethrowing.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  #include <exception>
8  using std::exception;
9
10 // throw, catch and rethrow exception
11 void throwException()
12 {
13    // throw exception and catch it immediately
14    try
15    {
16       cout << "  Function throwException throws an exception\n";
17       throw exception(); // generate exception
18    } // end try
19    catch ( exception & ) // handle exception
20    {
21       cout << "  Exception handled in function throwException"
22          << "\n  Function throwException rethrows exception";
23       throw; // rethrow exception for further processing
24    } // end catch
25
26    cout << "This also should not print\n";
27 } // end function throwException
```

Rethrow the exception

```
28
29 int main()
30 {
31    // throw exception
32    try
33    {
34       cout << "\nmain invokes function throwException\n";
35       throwException();
36       cout << "This should not print\n";
37    } // end try
38    catch ( exception & ) // handle exception
39    {
40       cout << "\n\nException handled in main\n";
41    } // end catch
42
43    cout << "Program control continues after catch in main\n";
44    return 0;
45 } // end main
```

Catch rethrown exception

```
main invokes function throwException
  Function throwException throws an exception
  Exception handled in function throwException
  Function throwException rethrows exception

Exception handled in main
Program control continues after catch in main
```

6 Exception Specifications

- 异常说明（也称为抛出列表）
  - 关键字 throw，逗号分隔的参数
  - 例如：int someFunction(double value)
    throw (ExceptionA, ExceptionB,
    ExceptionC)
    {
    ...
    }
    - ✓ 表示 someFunction 可以抛出 ExceptionA, ExceptionB 和 ExceptionC 类型的异常

6 Exception Specifications

- 异常说明（也称为抛出列表）
  - 一个函数只能抛出其异常说明中的异常类型或其派生类
    - ✓ 如果函数抛出非异常说明中的异常，将会调用 unexpected 函数，这通常会终止程序
  - 没有异常说明表示这个函数可以抛出任意异常
  - 空异常说明 throw()，表示该函数不能抛出任何异常

7 Processing Unexpected Exceptions

- unexpected 函数
  - 当函数抛出其异常说明外的异常时被调用
  - 调用通过 set_unexpected 注册的函数
  - 默认的将调用 terminate 函数

7 Processing Unexpected Exceptions

- <exception> 中的 set_unexpected 函数
  - 将指向一个没有参数，返回 void 的函数指针作为参数
  - 返回 unexpected 调用的最后一个函数指针
    - ✓第一次返回 0

7 Processing Unexpected Exceptions

- terminate 函数
  - 何时调用
    - ✓ 没有发现与抛出异常相匹配的 catch 处理程序
    - ✓ 析构函数在堆栈展开时试图抛出异常
    - ✓ 试图在没有相应的异常处理时重抛异常
    - ✓ 在没有通过 set_unexpected 函数注册函数时调用 unexpected 函数

7 Processing Unexpected Exceptions

- terminate 函数
  - 调用通过 set_terminate 注册的函数
  - 默认的调用 abort 函数

7 Processing Unexpected Exceptions

- set_terminate 函数
  - 将指向不带参数，返回 void 的函数指针作为参数
  - 返回 terminate 调用的最后一个函数指针
    - ✓第一次返回 0

7 Processing Unexpected Exceptions

- abort 函数
  - 不调用自动存储或静态存储类对象的析构函数就终止程序
    - ✓可能导致资源泄漏

8 Stack Unwinding

- 堆栈展开（Stack unwinding）
  - 当抛出的异常没有在特定作用域内被捕捉时发生
  - 展开一个函数将终止该函数
    - ✓ 所有该函数的局部变量被销毁
    - ✓ 控制权返回调用该函数的语句
  - 试图在外层 try...catch 语句块中捕捉异常
  - 如果异常最终未被捕获，terminate 函数被调用

```cpp
// Fig. 16.4: Fig16_04.cpp
// Demonstrating stack unwinding.
#include <iostream>
using std::cout;
using std::endl;

#include <stdexcept>
using std::runtime_error;

// function3 throws run-time error
void function3() throw ( runtime_error )
{
   cout << "In function 3" << endl;

   // no try block, stack unwinding occur, return control to function2
   throw runtime_error( "runtime_error in function3" );
} // end function3

// function2 invokes function3
void function2() throw ( runtime_error )
{
   cout << "function3 is called inside function2" << endl;
   function3(); // stack unwinding occur, return control to function1
} // end function2
```

```
25
26 // function1 invokes function2
27 void function1() throw ( runtime_error )
28 {
29    cout << "function2 is called inside function1" << endl;
30    function2(); // stack unwinding occur, return control to main
31 } // end function1
32
33 // demonstrate stack unwinding
34 int main()
35 {
36    // invoke function1
37    try
38    {
39       cout << "function1 is called inside main" << endl;
40       function1(); // call function1 which throws runtime_error
41    } // end try
42    catch ( runtime_error &error ) // handle run-time error
43    {
44       cout << "Exception occurred: " << error.what() << endl;
45       cout << "Exception handled in main" << endl;
46    } // end catch
47
48    return 0;
49 } // end main
```

```
function1 is called inside main
function2 is called inside function1
function3 is called inside function2
In function 3
Exception occurred: runtime_error in function3
Exception handled in main
```

9 Constructors, Destructors and Exception Handling

- 异常和构造函数
  - 异常机制使得没有返回值的构造函数可以向程序报告错误
  - 构造函数抛出的异常使得任何已经构造好的对象组件调用它们的析构函数
    - ✓只有那些已经被构造的对象将被析构

9 Constructors, Destructors and Exception Handling

- 异常和析构函数
  - 当异常被抛出，try 语句块中所有的自动对象将调用其析构函数
  - 如果被堆栈展开调用的析构函数抛出异常，terminate 函数将被调用

10 Exceptions and Inheritance

- 从 exception 类继承
  - 新的异常类可以从已存在的异常类继承
  - 处理特定异常的 catch 处理程序也可以处理该异常的派生类

11 Processing new Failures

- new 失败
  - 一些编译器抛出 bad_alloc 异常
  - 一些编译器返回 0
    - ✓使用 new(nothrow)
  - 一些编译器抛出 bad_alloc，如果包含 <new>

```cpp
// Fig. 16.5: Fig16_05.cpp
// Demonstrating pre-standard new returning 0 when memory
// is not allocated.
#include <iostream>
using std::cerr;
using std::cout;

int main()
{
   double *ptr[ 50 ];

   // allocate memory for ptr
   for ( int i = 0; i < 50; i++ )
   {
      ptr[ i ] = new double[ 50000000 ];

      if ( ptr[ i ] == 0 ) // did new fail to allocate memo
      {
         cerr << "Memory allocation failed for ptr[ " << i << " ]\n";
         break;
      } // end if
      else // successful memory allocation
         cout << "Allocated 50000000 doubles in ptr[ " << i << " ]\n";
   } // end for

   return 0;
} // end main
```

Allocate **50000000 double** values

new will have returned **0** if the memory allocation operation failed

```
Allocated 50000000 doubles in ptr[ 0 ]
Allocated 50000000 doubles in ptr[ 1 ]
Allocated 50000000 doubles in ptr[ 2 ]
Memory allocation failed for ptr[ 3 ]
```

```cpp
// Fig. 16.6: Fig16_06.cpp
// Demonstrating standard new throwing bad_alloc when memory
// cannot be allocated.
#include <iostream>
using std::cerr;
using std::cout;
using std::endl;

#include <new> // standard operator new
using std::bad_alloc;

int main()
{
   double *ptr[ 50 ];

   // allocate memory for ptr
   try
   {
      // allocate memory for ptr[ i ]; new throws bad_alloc on failure
      for ( int i = 0; i < 50; i++ )
      {
         ptr[ i ] = new double[ 50000000 ]; // may throw exception
         cout << "Allocated 50000000 doubles in ptr[ " << i << " ]\n";
      } // end for
   } // end try
```

Allocate **50000000 double** values

```
26
27    // handle bad_alloc exception
28    catch ( bad_alloc &memoryAllocationException )
29    {
30       cerr << "Exception occurred: "
31          << memoryAllocationException.what() << endl;
32    } // end catch
33
34    return 0;
35 } // end main
```

```
Allocated 50000000 doubles in ptr[ 0 ]
Allocated 50000000 doubles in ptr[ 1 ]
Allocated 50000000 doubles in ptr[ 2 ]
Exception occurred: bad allocation
```

new throws a **bad_alloc** exception if the memory allocation operation failed

11 Processing new Failures

- new 失败
  - set_new_handler 函数
    - ✓注册一个函数来处理 new 失败
      - ◈ 当内存分配操作失败时，注册的函数被调用
    - ✓将指向没有参数，返回 void 的函数指针作为参数

11 Processing new Failures

- new 失败
  - set_new_handler 函数
    - ✓C++ 说明要求 new-handler 函数应该：
      - ◈ 使更多的内存可用，重新调用 new
      - ◈ 抛出 bad_alloc 异常或
      - ◈ 调用 abort or exit 函数来终止程序

```cpp
// Fig. 16.7: Fig16_07.cpp
// Demonstrating set_new_handler.
#include <iostream>
using std::cerr;
using std::cout;

#include <new> // standard operator new and set_new_handler
using std::set_new_handler;

#include <cstdlib> // abort function prototype
using std::abort;

// handle memory allocation failure
void customNewHandler()
{
   cerr << "customNewHandler was called";
   abort();
} // end function customNewHandler

// using set_new_handler to handle failed memory allocation
int main()
{
   double *ptr[ 50 ];

   // specify that customNewHandler should be called on
   // memory allocation failure
   set_new_handler( customNewHandler );
```

Create a user-defined **new**-handler function **customNewHandler**

Register **customNewHandler** with **set_new_handler**

```
28
29    // allocate memory for ptr[ i ]; customNewHandler will be
30    // called on failed memory allocation
31    for ( int i = 0; i < 50; i++ )
32    {
33       ptr[ i ] = new double[ 50000000 ]; // may throw exception
34       cout << "Allocated 50000000 doubles in ptr[ " << i << " ]\n";
35    } // end for
36
37    return 0;
38 }  // end main
```

Allocate **50000000 double** values

```
Allocated 50000000 doubles in ptr[ 0 ]
Allocated 50000000 doubles in ptr[ 1 ]
Allocated 50000000 doubles in ptr[ 2 ]
customNewHandler was called
```

12 Class auto_ptr and Dynamic Memory Allocation

- 类模板 auto_ptr
  - 在头文件 <memory> 中定义
  - 维护一个指针来动态分配内存
    - ✓ 它的析构函数执行删除指针数据成员
      - ◈即使在异常发生时也删除动态分配的内存，以防止内存泄漏
    - ✓ 提供重载的运算符 * 和 -> 就像一个常规的指针变量

```
1  // Fig. 16.8: Integer.h
2  // Integer class definition.
3
4  class Integer
5  {
6  public:
7     Integer( int i = 0 ); // Integer default constructor
8     ~Integer(); // Integer destructor
9     void setInteger( int i ); // functions to set Integer
10    int getInteger() const; // function to return Integer
11 private:
12    int value;
13 }; // end class Integer
```

```cpp
// Fig. 16.9: Integer.cpp
// Integer member function definition.
#include <iostream>
using std::cout;
using std::endl;

#include "Integer.h"

// Integer default constructor
Integer::Integer( int i )
   : value( i )
{
   cout << "Constructor for Integer " << value << endl;
} // end Integer constructor

// Integer destructor
Integer::~Integer()
{
   cout << "Destructor for Integer " << value << endl;
} // end Integer destructor
```

```
21
22 // set Integer value
23 void Integer::setInteger( int i )
24 {
25    value = i;
26 } // end function setInteger
27
28 // return Integer value
29 int Integer::getInteger() const
30 {
31    return value;
32 } // end function getInteger
```

```
1  // Fig. 16.10: Fig16_10.cpp
2  // Demonstrating auto_ptr.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  #include <memory>
8  using std::auto_ptr; // auto_ptr class definition
9
10 #include "Integer.h"
11
12 // use auto_ptr to manipulate Integer object
13 int main()
14 {
15    cout << "Creating an auto_ptr object that points to an Integer\n";
16
17    // "aim" auto_ptr at Integer object
18    auto_ptr< Integer > ptrToInteger( new Integer( 7 ) );
19
20    cout << "\nUsing the auto_ptr to manipulate the Integer\n";
21    ptrToInteger->setInteger( 99 ); // use auto_ptr to set Integer val
22
23    // use auto_ptr to get Integer value
24    cout << "Integer after setInteger: " << ( *ptrToInteger ).getInteger()
25    return 0;
26 } // end main
```

Create an **auto_ptr** to point to a dynamically allocated **Integer** object

Manipulate the **auto_ptr** as if it were a pointer to an **Integer**

The dynamically allocated memory is automatically deleted by the **auto_ptr** when it goes out of scope

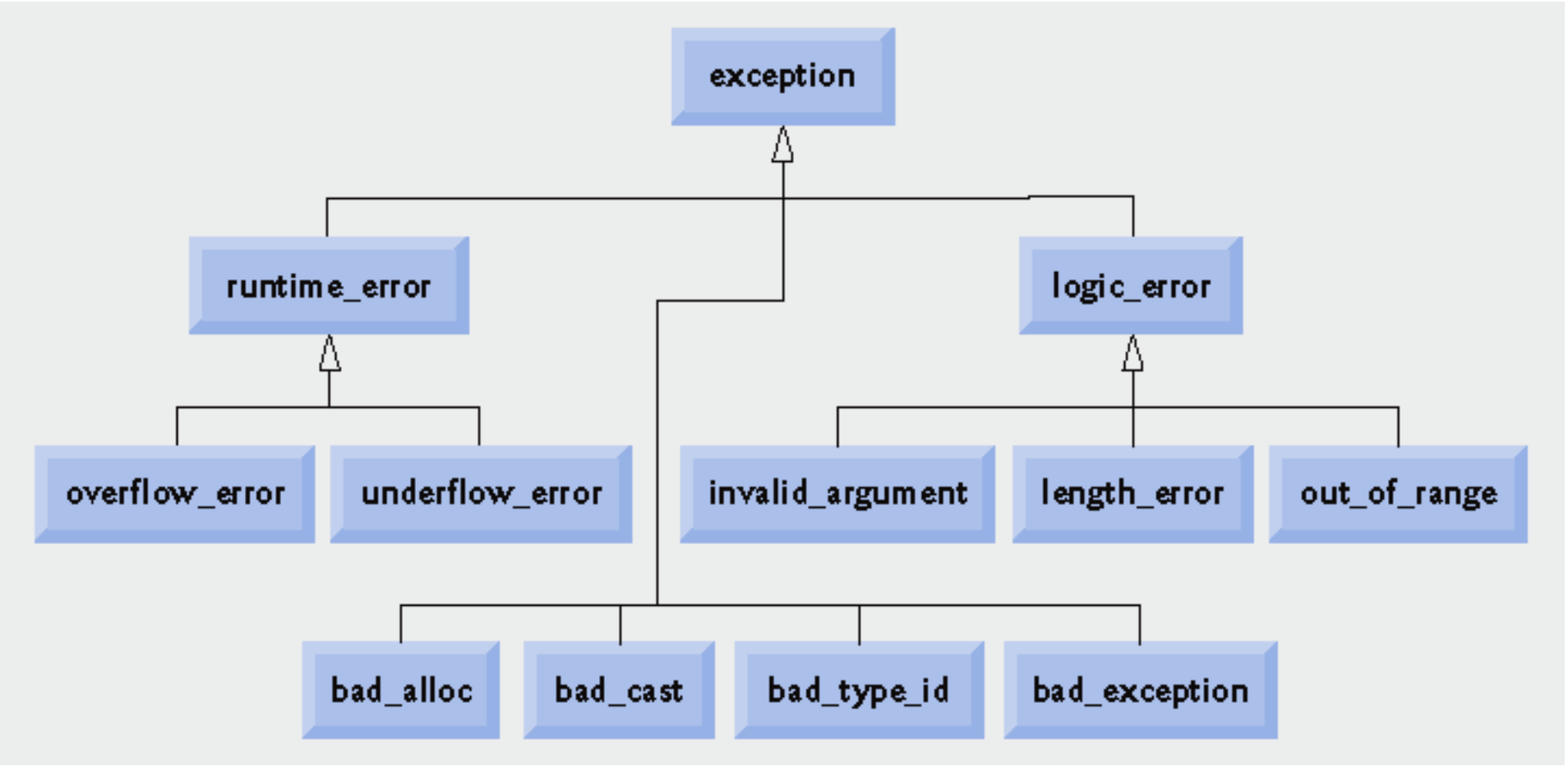
```
Creating an auto_ptr object that points to an Integer
Constructor for Integer 7

Using the auto_ptr to manipulate the Integer
Integer after setInteger: 99

Terminating program
Destructor for Integer 99
```

13 Standard Library Exception Hierarchy

- 异常类的层次
  - 基类：exception
    - ✓包含虚拟函数 what 来存储错误信息
    - ✓异常类继承自 exception
      - bad_alloc – thrown by new
      - bad_cast – thrown by dynamic_cast
      - bad_typeid – thrown by typeid
      - bad_exception – thrown by unexpected


exception
runtime_error
logic_error
overflow_error
underflow_error
invalid_argument
length_error
out_of_range
bad_alloc
bad_cast
bad_type_id
bad_exception

13 Standard Library Exception Hierarchy

- 异常类的层次
  - 类 logic_error, 继承自 exception
    - ✓指示程序的逻辑错误
    - ✓继承自 logic_error 的异常类
      - ◈ invalid_argument：函数的无效参数
      - ◈ length_error：长度超出对象的大小
      - ◈ out_of_range：例如，数组小标越界

13 Standard Library Exception Hierarchy

- 异常类的层次
  - 类 runtime_error 继承自 exception
    - ✓指示运行时错误
    - ✓继承自 runtime_error 的异常类
      - ◈ overflow_error：算术上溢错误
      - ◈ underflow_error：算术下溢错误

14 Other Error-Handling Techniques

- 其他错误处理技术
  - 忽略异常
    - ✓ 对于商业软件和关键业务软件是破坏性的
  - 退出程序
    - ✓ 阻止程序给用户错误结果
    - ✓ 对于关键业务程序是不合适的
    - ✓ 在退出时应释放已获得的资源

14 Other Error-Handling Techniques

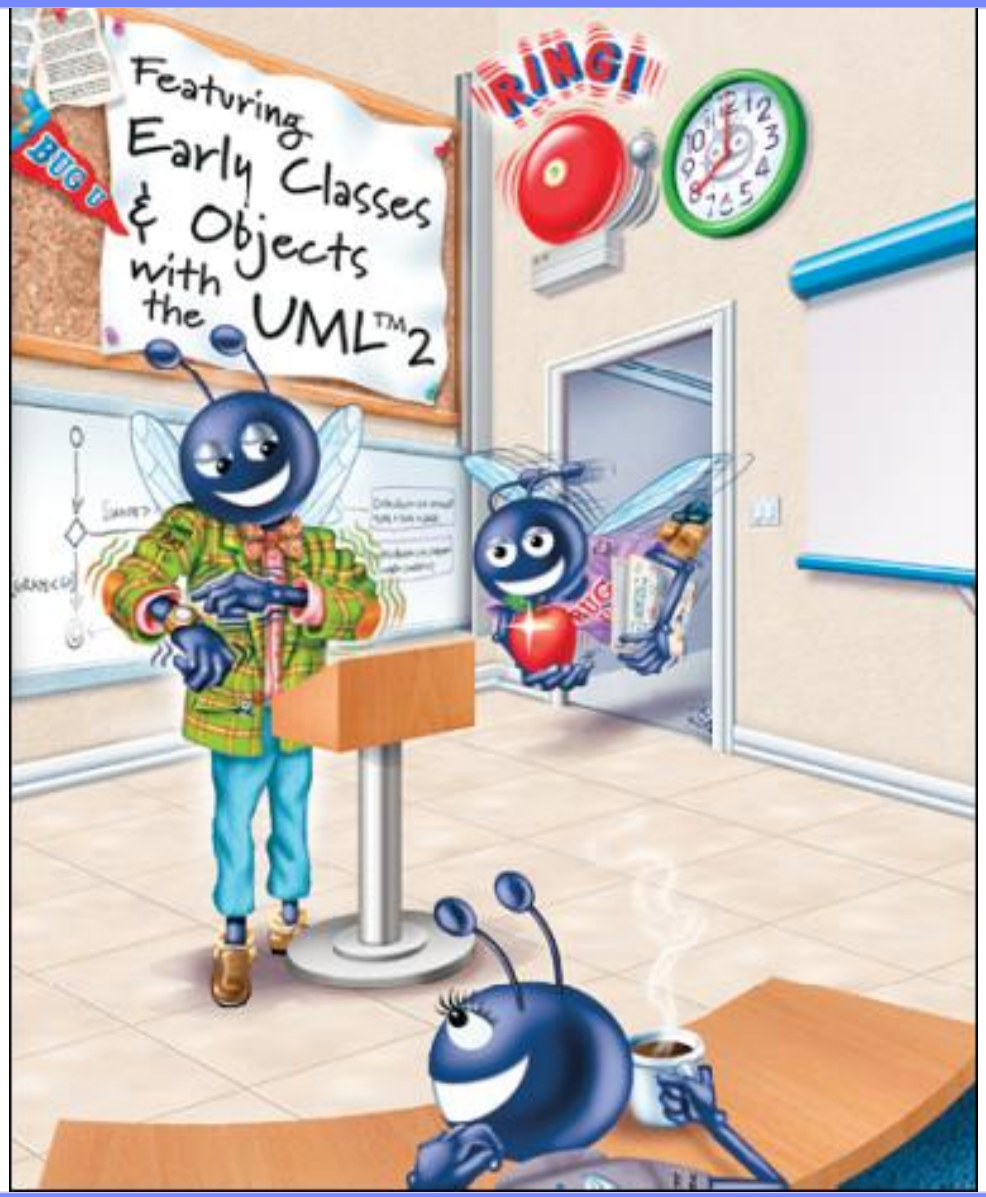
- 其他错误处理技术
  - 设置错误指示器
  - 发出错误信息，向 exit 传递适当的错误代码，返回程序运行环境

14 Other Error-Handling Techniques

- 其他错误处理技术
  - 使用 setjump 和 longjump 函数
    - ✓在 <csetjmp> 中定义
    - ✓用来立即从嵌套的函数中跳出，来调用错误处理程序
      - ◈ 在展开堆栈时，没有析构自动对象

14 Other Error-Handling Techniques

- 其他错误处理技术
  - 使用特定的错误处理方法
    - ✓例如：通过 set_new_handler 为运算符 new 注册一个错误处理程序



Thank you!