Chapter 17

EXCEPTION HANDLING

.

***Listing 17-1. Demonstrating Exception Handling***

***(with C’s setjmp() & longjmp())***

**//: C17:Nonlocal.cpp**

**// setjmp() & longjmp().**

**#include <iostream>**

**#include <csetjmp>**

**using namespace std;**

**class Rainbow {**

**public:**

**Rainbow() { cout << "Rainbow()" << endl; }**

**~Rainbow() { cout << "~Rainbow()" << endl; }**

**};**

**jmp\_buf kansas;**

**void oz() {**

**Rainbow rb;**

**for(int i = 0; i< 3; i++)**

**cout << "there's no place like home" << endl;**

**longjmp(kansas, 47);**

**}**

**int main() {**

**if(setjmp(kansas) == 0) {**

**cout << "tornado, witch, munchkins..." << endl;**

**oz();**

**} else {**

**cout << "Auntie Em! "**

**<< "I had the strangest dream..."**

**<< endl;**

**}**

**} ///:~*Listing 17-2. Throwing an Exception***

**//: C17:MyError.cpp {RunByHand}**

**classMyError {**

**const char\* const data;**

**public:**

**MyError(const char\* const msg = 0) : data(msg) {}**

**};**

**void f() {**

**// Here we "throw" an exception object:**

**tThrow MyError("something bad happened");**

**}**

**int main() {**

**// As you’ll see shortly, we’ll want a "try block" here:**

**f();**

**} ///:~*Listing 17-3. Illustrating Try & Catch Blocks***

**//: C17:Nonlocal2.cpp**

**// Illustrates exceptions.**

**#include <iostream>**

**using namespace std;**

**class Rainbow {**

**public:**

**Rainbow() { cout << "Rainbow()" << endl; }**

**~Rainbow() { cout << "~Rainbow()" << endl; }**

**};**

**vVoid oz() {**

**Rainbow rb;**

**for(int i = 0; i < 3; i++)**

**cout << "there's no place like home" << endl;**

**throw 47;**

**}**

**int main() {**

**try {**

**cout << "tornado, witch, munchkins..." << endl;**

**oz();**

**} catch(int) {**

**cout << "Auntie Em! I had the strangest dream..."**

**<< endl;**

**}**

**} ///:~*Listing 17-4. Illustrating Exception Matching***

**//: C17:Autoexcp.cpp**

**// No matching conversions**

**#include <iostream>**

**using namespace std;**

**class Except1 {};**

**class Except2 {**

**public:**

**Except2(Except1&) {}**

**};**

**void f() { throw Except1(); }**

**int main() {**

**try { f();**

**} catch (Except2) {**

**cout << "inside catch(Except2)" << endl;**

**} catch (Except1) {**

**cout << "inside catch(Except1)" << endl;**

**}**

**} ///:~*Listing 17-5. Illustrating Exception Hierarchies***

**//: C17:Basexcpt.cpp**

**// Exception hierarchies**

**#include <iostream>**

**using namespace std;**

**class X {**

**public:**

**class Trouble {};**

**class Small : public Trouble {};**

**class Big : public Trouble {};**

**void f() { throw Big(); }**

**};**

**int main() {**

**X x;**

**try {**

**x.f();**

**} catch(X::Trouble) {**

**cout << "caught Trouble" << endl;**

**// Hidden by previous handler:**

**} catch(X::Small) {**

**cout << "caught Small Trouble" << endl;**

**} catch(X::Big) {**

**cout << "caught Big Trouble" << endl;**

**}**

**} ///:~*Listing 17-6. Using set\_terminate(); also, Demonstrates Uncaught Exceptions***

**//: C17:Terminator.cpp**

**// Use of set\_terminate(); Also shows uncaught exceptions**

**#include <exception>**

**#include <iostream>**

**#include <cstdlib>**

**using namespace std;**

**void terminator() {**

**cout << "I'll be back!" << endl;**

**abort();**

**}**

**void (\*old\_terminate)()**

**= set\_terminate(terminator);**

**class Botch {**

**public:**

**class Fruit {};**

**void f() {**

**cout << "Botch::f()" << endl;**

**throw Fruit();**

**}**

**Chapter 16: Exception Handling**

**380**

**~Botch() { throw 'c'; }**

**};**

**int main() {**

**try{**

**Botch b;**

**b.f();**

**} catch(...) {**

**cout << "inside catch(...)" << endl;**

**}**

**} ///:~*Listing 17-7. Demonstrates that Exceptions don’t Clean up Incomplete Objects***

**//: C17:Cleanup.cpp**

**// Exceptions clean up complete objects only.**

**#include <iostream>**

**using namespace std;**

**class Trace {**

**static int counter;**

**int objid;**

**public:**

**Trace() {**

**objid = counter++;**

**cout << "constructing Trace #" << objid << endl;**

**if(objid == 3) throw 3;**

**}**

**~Trace() {**

**cout << "destructing Trace #" << objid << endl;**

**}**

**};**

**int Trace::counter = 0;**

**int main() {**

**try {**

**Trace n1;**

**// Throws exception:**

**Trace array[5];**

**Trace n2; // Won't get here.**

**} catch(int i) {**

**cout << "caught " << i << endl;**

**}**

**} ///:~*Listing 17-8. Demonstrates a Case of Naked Pointers***

**//: C17:Rawp.cpp**

**// Naked pointers.**

**#include <iostream>**

**#include <cstddef>**

**using namespace std;**

**class Cat {**

**public:**

**Cat() { cout << "Cat()" << endl; }**

**~Cat() { cout << "~Cat()" << endl; }**

**};**

**class Dog {**

**public:**

**void\* operator new(size\_tsz) {**

**cout << "allocating a Dog" << endl;**

**throw 47;**

**}**

**void operator delete(void\* p) {**

**cout << "deallocating a Dog" << endl;**

**::operator delete(p);**

**}**

**};**

**class UseResources {**

**Cat\* bp;**

**Dog\* op;**

**public:**

**UseResources(int count = 1) {**

**cout << "UseResources()" << endl;**

**bp = new Cat[count];**

**op = new Dog;**

**}**

**~UseResources() {**

**cout << "~UseResources()" << endl;**

**delete [] bp; // Array delete**

**delete op;**

**}**

**};**

**int main() {**

**try {**

**UseResources ur(3);**

**} catch(int) {**

**cout << "inside handler" << endl;**

**}**

**} ///:~**

***Listing 17-9. Illustrates Safe Atomic Pointers & Using RAII***

**//: C17:Wrapped.cpp**

**// Safe, atomic pointers**

**#include <fstream>**

**#include <cstdlib>**

**using namespace std;**

**ofstream out("wrapped.out");**

**// Simplified. Yours may have other arguments.**

**template<class T, int sz = 1> class PWrap {**

**T\* ptr;**

**public:**

**class RangeError {}; // Exception class**

**PWrap() {**

**ptr = new T[sz];**

**out << "PWrap constructor" << endl;**

**}**

**~PWrap() {**

**delete []ptr;**

**out << "PWrap destructor" << endl;**

**}**

**T& operator[](int i) throw(RangeError) {**

**if(i >= 0 && i < sz) return ptr[i];**

**throw RangeError();**

**}**

**};**

**class Cat {**

**public:**

**Cat() { out << "Cat()" << endl; }**

**~Cat() { out << "~Cat()" << endl; }**

**void g() {}**

**};**

**class Dog {**

**public:**

**void\* operator new[](size\_t sz) {**

**Chapter 16: Exception Handling**

**387**

**out << "allocating an Dog" << endl;**

**throw int(47);**

**}**

**void operator delete[](void\* p) {**

**out << "deallocating an Dog" << endl;**

**::delete p;**

**}**

**};**

**class UseResources {**

**PWrap<Cat, 3> Bonk;**

**PWrap<Dog> Og;**

**public:**

**UseResources() : Bonk(), Og() {**

**out << "UseResources()" << endl;**

**}**

**~UseResources() {**

**out << "~UseResources()" << endl;**

**}**

**void f() { Bonk[1].g(); }**

**};**

**int main() {**

**try {**

**UseResources ur;**

**} catch(int) {**

**out << "inside handler" << endl;**

**} catch(...) {**

**out << "inside catch(...)" << endl;**

**}**

**} ///:~**

***Listing 17-10. Demonstrates The RAII Nature of auto\_ptr***

**//: C17:Auto\_ptr.cpp**

**/// Illustrates the RAII nature of auto\_ptr.**

**#include <memory>**

**#include <iostream>**

**#include <cstddef>**

**using namespace std;**

**class TraceHeap {**

**int i;**

**public:**

**static void\* operator new(size\_tsiz) {**

**void\* p = ::operator new(siz);**

**cout << "Allocating TraceHeap object on the heap "**

**<< "at address " << p << endl;**

**return p;**

**}**

**static void operator delete(void\* p) {**

**cout << "Deleting TraceHeap object at address "**

**<< p << endl;**

**::operator delete(p);**

**}**

**TraceHeap(int i) : i(i) {}**

**int getVal() const { return i; }**

**};**

**int main() {**

**auto\_ptr<TraceHeap> pMyObject(new TraceHeap(5));**

**cout << pMyObject->getVal() << endl; // Prints 5**

**} ///:~*Listing 17-11. Illustrates Handling Exceptions from Subobjects***

**//: C17:InitExcept.cpp {-bor}**

**// Handles exceptions from subobjects.**

**#include <iostream>**

**using namespace std;**

**class Base {**

**int i;**

**public:**

**classBaseExcept {};**

**Base(int i) : i(i) { throw BaseExcept(); }**

**};**

**class Derived : public Base {**

**public:**

**class DerivedExcept {**

**const char\* msg;**

**public:**

**DerivedExcept(const char\* msg) : msg(msg) {}**

**const char\* what() const { return msg; }**

**};**

**Derived(int j) try : Base(j) {**

**// Constructor body**

**cout << "This won't print" << endl;**

**} catch(BaseExcept&) {**

**throw DerivedExcept("Base subobject threw");**

**}**

**};**

**int main() {**

**try {**

**Derived d(3);**

**} catch(Derived::DerivedExcept& d) {**

**cout << d.what() << endl; // "Base subobject threw"**

**}**

**} ///:~*Listing 17-12. Demonstrates Function-Level try Blocks***

**//: C17:FunctionTryBlock.cpp {-bor}**

**/ Function-level try blocks**

**#include <iostream>**

**using namespace std;**

**int main() try {**

**throw "main";**

**} catch(const char\* msg) {**

**cout << msg << endl;**

**} ///:~*Listing 17-13. Demonstrates Deriving an Exception Class***

**//: C17:StdExcept.cpp**

**// Derives an exception class from std::runtime\_error.**

**#include <stdexcept>**

**#include <iostream>**

**using namespace std;**

**class MyError : public runtime\_error {**

**public:**

**MyError(const string& msg = "") : runtime\_error(msg) {}**

**};**

**int main() {**

**try {**

**throw MyError("my message");**

**} catch(MyError& x) {**

**cCout << x.what() << endl;**

**}**

**} ///:~*Listing 17-14. Using Exception Specifications & the unexpected() Mechanism***

**//: C17:Unexpected.cpp**

**// Exception specifications & unexpected(),**

**//{-msc} (Doesn’t terminate properly)**

**#include <exception>**

**#include <iostream>**

**using namespace std;**

**class Up {};**

**class Fit {};**

**void g();**

**void f(int i) throw(Up, Fit) {**

**switch(i) {**

**case 1: throw Up();**

**case 2: throw Fit();**

**}**

**g();**

**}**

**// void g() {} // Version 1**

**void g() { throw 47; } // Version 2**

**void my\_unexpected() {**

**cout << "unexpected exception thrown" << endl;**

**exit(0);**

**}**

**int main() {**

**set\_unexpected(my\_unexpected); // (Ignores return value)**

**for(int i = 1; i <= 3; i++)**

**try {**

**f(i);**

**} catch(Up) {**

**cout << "Up caught" << endl;**

**} catch(Fit) {**

**cout << "Fit caught" << endl;**

**}**

**} ///:~*Listing 17-15. Ilustrating The Two Cases of Bad Exceptions***

**//: C17:BadException.cpp {-bor}**

**#include <exception>    // For std::bad\_exception**

**#include <iostream>**

**#include <cstdio>**

**using namespace std;**

**// Exception classes:**

**class A {};**

**class B {};**

**// terminate() handler**

**void my\_thandler() {**

**cout << "terminate called" << endl;**

**exit(0);**

**}**

**// unexpected() handlers**

**void my\_uhandler1() { throw A(); }**

**void my\_uhandler2() { throw; }**

**// If we embed this throw statement in f or g,**

**// the compiler detects the violation and reports**

**// an error, so we put it in its own function.**

**void t() { throw B(); }**

**void f() throw(A) { t(); }**

**void g() throw(A, bad\_exception) { t(); }**

**int main() {**

**set\_terminate(my\_thandler);**

**set\_unexpected(my\_uhandler1);**

**try {**

**f();**

**} catch(A&) {**

**cout << "caught an A from f" << endl;**

**}**

**set\_unexpected(my\_uhandler2);**

**try {**

**g();**

**} catch(bad\_exception&) {**

**cout << "caught a bad\_exception from g" << endl;**

**}**

**try {**

**f();**

**} catch(...) {**

**cout << "This will never print" << endl;**

**}**

**} ///:~**

***Listing 17-16. Illustrating Covariance (Exception Specifications & Inheritance)***

**//: C17:Covariance.cpp {-xo}**

**// Should cause compile error.**

**#include <iostream>**

**using namespace std;**

**class Base {**

**public:**

**class BaseException {};**

**class DerivedException : public BaseException {};**

**virtual void f() throw(DerivedException) {**

**throw DerivedException();**

**}**

**virtual void g() throw(BaseException) {**

**throw BaseException();**

**}**

**};**

**class Derived : public Base {**

**public:**

**void f() throw(BaseException) {**

**throw BaseException();**

**}**

**virtual void g() throw(DerivedException) {**

**throw DerivedException();**

**}**

**}; ///:~*Listing 17-17. Illustrating an Exception-Safe Operator (=)***

**//: C17:SafeAssign.cpp**

**// An Exception-safe operator=.**

**#include <iostream>**

**#include <new> // For std::bad\_alloc**

**#include <cstring>**

**#include <cstddef>**

**using namespace std;**

**// A class that has two pointer members using the heap**

**cClass HasPointers {**

**// A Handle class to hold the data**

**struct MyData {**

**const char\* theString;**

**const int\* theInts;**

**size\_t numInts;**

**MyData(const char\* pString, const int\* pInts,**

**size\_t nInts)**

**: theString(pString), theInts(pInts), numInts(nInts) {}**

**} \*theData; // The handle**

**// Clone and cleanup functions:**

**static MyData\* clone(const char\* otherString,**

**const int\* otherInts, size\_t nInts) {**

**char\* newChars = new char[strlen(otherString)+1];**

**int\* newInts;**

**try {**

**newInts = new int[nInts];**

**} catch(bad\_alloc&) {**

**delete [] newChars;**

**throw;**

**}**

**try {**

**// This example uses built-in types, so it won't**

**// throw, but for class types it could throw, so we**

**// use a try block for illustration. (This is the**

**// point of the example!)**

**strcpy(newChars, otherString);**

**for(size\_t i = 0; i < nInts; ++i)**

**newInts[i] = otherInts[i];**

**} catch(...) {**

**delete [] newInts;**

**delete [] newChars;**

**throw;**

**}**

**return new MyData(newChars, newInts, nInts);**

**}**

**static MyData\* clone(const MyData\* otherData) {**

**return clone(otherData->theString, otherData->theInts,**

**otherData->numInts);**

**}**

**static void cleanup(const MyData\* theData) {**

**delete [] theData->theString;**

**delete [] theData->theInts;**

**delete theData;**

**}**

**public:**

**HasPointers(const char\* someString, const int\* someInts,**

**size\_t numInts) {**

**theData = clone(someString, someInts, numInts);**

**}**

**HasPointers(const HasPointers& source) {**

**theData = clone(source.theData);**

**}**

**HasPointers& operator=(const HasPointers& rhs) {**

**if(this != &rhs) {**

**MyData\* newData = clone(rhs.theData->theString,**

**rhs.theData->theInts, rhs.theData->numInts);**

**cleanup(theData);**

**theData = newData;**

**}**

**return \*this;**

**}**

**~HasPointers() { cleanup(theData); }**

**friend ostream&**

**operator << (ostream& os, const HasPointers& obj) {**

**os << obj.theData->theString << ": ";**

**for(size\_t i = 0; i < obj.theData->numInts; ++i)**

**os << obj.theData->theInts[i] << ' ';**

**return os;**

**}**

**};**

**int main() {**

**int someNums[] = { 1, 2, 3, 4 };**

**size\_t someCount = sizeof someNums / sizeof someNums[0];**

**int someMoreNums[] = { 5, 6, 7 };**

**size\_t someMoreCount =**

**sizeof someMoreNums / sizeof someMoreNums[0];**

**HasPointers h1("Hello", someNums, someCount);**

**HasPointers h2("Goodbye", someMoreNums, someMoreCount);**

**cout << h1 << endl; // Hello: 1 2 3 4**

**h1 = h2;**

**cout << h1 << endl; // Goodbye: 5 6 7**

**} ///:~*Listing 17-18. Illustrating A Program With/Without Exception-Handling Support***

**//: C17:HasDestructor.cpp {O}**

***/\* shows that programs with exception-handling support are bigger than those without \*/***

***class HasDestructor {***

***public:***

***~HasDestructor() {}***

***};***

***void g(); // For all we know, g may throw.***

***void f() {***

***HasDestructor h;***

***g();***

***} ///:~***