

Exceptions

Giuseppe Lipari
`http://retis.sssup.it`

Scuola Superiore Sant'Anna – Pisa

December 17, 2010

Outline

1 Handling error conditions

2 Exceptions

3 Hierarchies

4 Cleanup

Error conditions

- There can be an error at run-time for various problems:
 - We discover a bug that arises only on certain values of the variables
 - The user has done something wrong
 - A wrong input value
 - A wrong behavior of an external device
 - A failure of the hardware
- Errors have different levels of criticality
 - There are *recoverable* errors, which allow the program to continue or to re-try the last operation
 - There are unrecoverable errors, the program must exit in a clean way
- The problem is what to do when we discover an error at run-time

Error location

- An error can arise in a module, in a function very deep in the call stack
- Rarely we can handle the error at that level
 - It is much common to report the error to the upper layers
- The error conditions should be part of the interface of a module
 - The module reports (*raises*) the error
 - The user of the module receives the error and decides what to do

Error treatment

- How to signal that an error has happened?
 - 1 the function returns an error code
 - 2 the function returns a generic error condition, and set a global variable with the error code
 - 3 just exit the program
- In the first two cases we need to write a lot of special-purpose code for handling errors

Example of error treatment

- Suppose we decide to follow method 1 (return an error code)

```
int f() {...}

int main()
{
    int err;
    ...
    err = f();
    if (err < 0) { // error !
        if (err == ERR_CODE_A) // handle case A
        else if (err == ERR_CODE_B) // handle case B
        ...
    }
}
```

- The above code has to be repeated for every function call that can raise an error!

Error Forwarding

- Also, sometimes the error has to be forwarded to upper layers

```
int f() {...}

int g()
{
    int err;
    ...
    err = f();
    if (err < 0) { //error !
        if (err == ERR_CODE_A) {
            // handle case A locally
        }
        else if (err == ERR_CODE_B) {
            // forward case B
            return ERR_CODE_B;
        }
        ...
    }
}

int main()
{
    int err;
    ...
    err = g();
    if (err < 0) {
        //error !
    }
}
```

Other examples

- see `stack/` and `list/`
- `list` contains an operator `[]` for random access in the list
 - What if the user specifies an out-of-range index?
 - we can specify a special “error-return-value”
 - or we can print the error and call `exit()`;
- Neither of the two options is satisfactory!

Outline

- 1 Handling error conditions
- 2 Exceptions
- 3 Hierarchies
- 4 Cleanup

Exceptions

- An exception is an object of a class representing an exceptional occurrence
- This way, C++ uses the class mechanisms (like inheritance, etc.) to implement exceptions
 - The exception class has nothing to do with the other classes in the program
 - An exception can be thrown with the throw keyword
 - see `exc_stack/`

Try/catch

- An exception object is *thrown* by the programmer in case of an error condition
- An exception object can be caught inside a try/catch block

```
try {  
    //  
    // this code can generate exceptions  
    //  
} catch (ExcType1& e1) {  
    // all exceptions of ExcType1 are handled here  
}
```

Try/catch

- If the exception is not caught at the level where the function call has been performed, it is automatically forwarded to the upper layer
 - Until it finds a proper try/catch block that *catches* it
 - or until there is no upper layer (in which case, the program is aborted)

More catches

- It is possible to put more catch blocks in sequence
- they will be processed in order, the first one that catches the exception is the last one to execute

```
try {  
    //  
    // this code can generate exceptions  
    //  
} catch (ExcType1&e1) {  
    // all exceptions of ExcType1  
} catch (ExcType2 &e2) {  
    // all exceptions of ExcType2  
} catch (...) {  
    // every exception  
}
```

Re-throwing

- It is possible to re-throw the same exception that has been caught to the upper layers

```
catch(...) {  
    cout << "an exception was thrown" << endl;  
    // Deallocate your resource here, and then rethrow  
    throw;  
}
```

Terminate

- In case of abort, the C++ run-time will call the terminate(), which calls abort()
 - It is possible to change this behavior

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

void terminator() {
    cout << "I'll be back!" << endl;
    exit(0);
}
void (*old_terminate)() = set_terminate(terminator);

class Botch {
public:
    class Fruit {};
    void f() {
        cout << "Botch::f()" << endl;
        throw Fruit();
    }
    ~Botch() { throw 'c'; }
};

int main() {
    try {
        Botch b;
        b.f();
    } catch(...) {
        cout << "inside catch(...)" << endl;
    }
}
//::~~
```

Outline

- 1 Handling error conditions
- 2 Exceptions
- 3 Hierarchies**
- 4 Cleanup

Hierarchy of exceptions

- Exceptions can be organized in a hierarchy

```
class MathExc {
    string error;
    string where;
public:
    MathErr(const string &e, const string &w) :
        error(e), where (w)
    {}
    virtual string what() { return error + " " + where;}
};

class LogErr : public MathErr {
public:
    LogErr() :
        MathErr("Log of a negative number", "log module"),
        n(a)
    {}
}
```

Inheritance

```
double mylog(int a)
{
    if (a <= 0) throw LogErr();
    else return log(double(a));
}

void f(int i)
{
    mylog(i);
}

...

try {
    f(-5);
} catch(MathErr &e) {
    cout << e.what() << endl;
}
```

- This code will print “Log of a negative number - log module”
- you can also pass any parameter to LogErr, like the number that cause the error, or the name of the function which caused the error, etc.

Exception specification

- It is possible to specify which exceptions a function might throw, by listing them after the function prototype
- Exceptions are part of the interface!

```
void f(int a) throw(Exc1, Exc2, Exc3);  
void g();  
void h() throw();
```

- f() can **only** throw exception Exc1, Exc2 or Exc3
- g() can throw **any** exception
- h() **does not** throw any exception

Listing exceptions

- Pay attention: a function must list in the exception list all exception that it may throw, and all exception that all called functions may throw

```
int f() throw(E1) {...}  
  
int g() throw(E2)  
{  
    ...  
    if (cond) throw E2;  
    ...  
    f();  
}
```

Listing exceptions

- Pay attention: a function must list in the exception list all exception that it may throw, and all exception that all called functions may throw

```
int f() throw(E1) {...}  
  
int g() throw(E2)   
{  
    ...  
    if (cond) throw E2;  
    ...  
    f();  
}
```

It should contain E1 in the list, because g() calls f()

Exception list and inheritance

- if a member function in a base class says it will only throw an exception of type A,
- an override of that function in a derived class must not add any other exception types to the specification list
 - because that would break any programs that adhere to the base class interface.
- You can, however, specify fewer exceptions or none at all, since that doesn't require the user to do anything differently.

Exception list and inheritance

- It is possible to change the specification of an exception with a derived exception

```
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();
    }
}; ///:~
```

- Which one is correct?

Outline

- 1 Handling error conditions
- 2 Exceptions
- 3 Hierarchies
- 4 Cleanup**

Stack unrolling

```
void f() {  
    A a;  
  
    if (cond) throw Exc();  
}  
  
void g() {  
    A *p = new A;  
  
    if (cond) throw Exc();  
}
```

At this point, a is destructed



Stack unrolling

```
void f() {  
    A a;  
  
    if (cond) throw Exc();  
}  
  
void g() {  
    A *p = new A;  
  
    if (cond) throw Exc();  
}
```

At this point, a is destructed

memory pointed by p is **not** automatically deallocated

Cleaning up

- C++ exception handling guarantees that as you leave a scope, all objects in that scope *whose constructors have been completed* will have their destructors called.
- see exceptions/trace.cpp

Resource management

- When writing code with exceptions, it's particularly important that you always ask, "If an exception occurs, will my resources be properly cleaned up?"
- Most of the time you're fairly safe,
- but in constructors there's a particular problem:
 - if an exception is thrown before a constructor is completed, the associated destructor will not be called for that object.
 - Thus, you must be especially diligent while writing your constructor.
- The difficulty is in allocating resources in constructors.
 - If an exception occurs in the constructor, the destructor doesn't get a chance to deallocate the resource.
 - see `exceptions/rawp.cpp`

How to avoid the problem

- To prevent such resource leaks, you must guard against these “raw” resource allocations in one of two ways:
 - You can catch exceptions inside the constructor and then release the resources
 - You can place the allocations inside an object’s constructor, and you can place the deallocations inside an object’s destructor.
- The last technique is called Resource Acquisition Is Initialization (RAII for short) because it equates resource control with object lifetime.
- Example: wrapped.cpp

PWrap

- The difference is the use of the template to wrap the pointers and make them into objects.
 - The constructors for these objects are called before the body of the UseResources constructor,
 - any of these constructors that complete before an exception is thrown will have their associated destructors called during stack unwinding.
- The PWrap template shows a more typical use of exceptions than you've seen so far:
 - A nested class called RangeError is created to use in operator[] if its argument is out of range.
 - Because operator[] returns a reference, it cannot return zero!
 - An exception mechanism was necessary

Auto ptr

- Dynamic memory is the most frequent resource used in a typical C++ program,
- the standard provides an RAII wrapper for pointers to heap memory that automatically frees the memory.
- The **auto_ptr** class template, defined in the `<memory>` header, has a constructor that takes a pointer to its generic type
- The **auto_ptr** class template also overloads the pointer operators `*` and `->` to forward these operations to the original pointer
- So you can use the **auto_ptr** object as if it were a raw pointer.
- Example in `exceptions/autoptr.cpp`

auto_ptr example

```
class TraceHeap {
    int i;
public:
    static void* operator new(size_t siz) {
        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
} ///:~
```