10 When things go wrong

In any non-trivial program, there will come a time when things go wrong. There are several ways of responding: 365

1. terminate the program
2. do nothing
3. report an error
4. set an error status flag
5. return a special value
6. trigger an assertion failure
7. throw an exception

In this chapter, we describe each of these mechanisms in more detail, and give criteria for choosing the most appropriate one to use in a given situation.

We begin by characterizing the kind of problems we are considering.

- If the problem is caused by a ***logical inconsistency in the program***, the best way to deal with it is to use assertions. It is not appropriate to use `if` statements, error messages, and the like to deal with simple incorrectness.

  > If the program needs the value of $\sqrt{X}$, and the laws of physics ensure that $X \geq 0$, the program should not have to check X before taking the square root. If X is negative, the program probably has a logical error.

- If the problem is something that can ***reasonably be expected to happen***, the best solution is normal code. For example, an `if` statement to detect the condition followed by an appropriate response.

  > Users cannot be relied upon to enter valid data. Any system component that is reading data from the keyboard should validate the input and complain immediately if anything is wrong. This does not require fancy coding techniques.

Problems that do not fit into the above categories are probably ***serious***, ***rare***, and ***unavoidable***. The following subsections discuss suitable responses to problems of this kind.

10.1 Simple Mechanisms

10.1.1 Error messages

cerr

The easiest response to a problem is an error message, sent to `cout`, or (better) `cerr`, or, in a windowing environment, a dialog or message box.

The streams `cout` and `cerr` behave in essentially the same way. Both send messages to a stream that C++ calls ***standard output***. The difference is that `cout` is buffered and `cerr` is not. When the program crashes, it is more likely to have displayed `cerr` output than `cout` output.

If this method works for your application, use it. However, it is inappropriate for many 366 applications:

- The software may be ***embedded*** in a car, pacemaker, or electric toaster. There is nowhere to send messages to and perhaps no one to respond to them anyway.
- The software is part of a system that must do its job as well as possible without giving up or complaining. For example, an internet router or an airplane fly-by-wire system.
- The user is not someone who can handle the problem. For example, a web browser should not display error messages – although, of course, they sometimes do – because there isn't much typical users can do.

10.1.2 Error codes

An effective system used by many software packages, such as UNIX and OpenGL, is to set an error flag and continue in the best feasible way.

- A UNIX library function may set the global variable `errno`. Most library functions also do something else to indicate failure, such as returning `-1` instead of a `char`.

error handling in OpenGL

- Many OpenGL functions perform according to the following specification: either the function behaves as expected, or an error occurred and ***there is no other effect***. (The user calls `glGetError` to find out whether an error occurred and what went wrong.)

This system requires some discipline from the programmers using it. They should look at the error status from time to time to make sure that nothing has gone wrong. A common strategy is to check the error status just after doing something that is likely to fail, and otherwise ignore it.

Error codes are ideal for an application in which the consequences of failures are usually not serious, such as a graphics library. They are not secure enough for safety-critical applications.

10.1.3 Special return values

A function can return a funny value to indicate that it has not completed its task successfully. We will call these values ***special return values***. The classic example of this behaviour is the C function `getchar`: you would expect its type to be `char` or even `unsigned char`, but in fact it returns an `int` (thus causing much grief to C beginners). It does this so that it can report end-of-file by returning `-1`.

Special return values are easy to code and require no special language features. They have strong advocates (see http://joelonsoftware.com/items/2003/10/13.html). They also have disadvantages: 367

- There may be no suitable value to return. This is not a problem for `exp` and `log`, whose values are always positive, but how does `tan` report an error (its range is all of the reals from $-\infty$ to ∞)?
- In some cases, it is not feasible to pack all of the information about errors and returned data into a single value. Then we have to add reference parameters to the function to enable it to return all of the information.
- Using the return value for multiple purposes leads to awkward code. In particular, conditions with side-effects are often needed, as in the C idiom of Figure 121.
- Code can become verbose. Instead of writing `y = f(g(h(x)))`, we have to write something like the code in Figure 122 on the following page. 368
- Programmers have a tendency not to check for special return values. This problem is notorious in Unix. 369
- The caller may not know how to handle the problem. There will be multiple levels of functions, all specially coded to return funny results until someone can deal with the situation appropriately.

10.2 Exceptions

Exceptions provide a form of communication between distant parts of a program. The problem that exceptions solve is: something has gone wrong but, at this level or in this context, there is no way to fix it. Examples: 370

- In the middle of complex fluid-flow calculations, the control system software of a nuclear reactor detects that a pressure sensor is sending improbable data.

```
int ch;
while ( (ch = getchar()) >= 0 )
{
    // process characters
}
// end of file
```

Figure 121: The consequence of ambiguous return values

```
double t1 = h(x);
if (t1 == h_error_value)
    // handle h_error
else
{
    double t2 = g(t1);
    if (t2 == g_error_value)
        // handle g_error
    else
    {
        double y = f(t2);
        if (y == f_error_value)
            // handle f_error
        else
        {
            // carry on
        }
    }
}
```

Figure 122: Catching errors in `y = f(g(h(x)))`

- A library function that has a simple, natural interface (e.g., `Matrix::invert`) but may nevertheless fail for some inputs (e.g., a singular matrix).

The main advantage of exceptions is that they can transfer control over many levels without
371 cluttering the code. Unfortunately, this is also their main disadvantage. The sequence

```
allocate_my_memory();
delete_my_memory();
```

looks perfectly fine until you realize that `allocate_my_memory` might throw an exception.

372 In more detail, the advantages of exceptions include:

- The thrower does not have to know where the catcher is.
- The catcher does not have to know where the thrower is.
- A catcher can be selective, by only catching exceptions of particular types.
- If no exceptions are thrown, the overhead of the exception handling system is essentially zero.
- The `try` and `throw` statements make it obvious in the code that a problem is being detected and handled.

373 But exceptions also have some disadvantages:

- A programmer who calls a function that might throw an exception, either directly or indirectly (i.e., at some deeply nested place) cannot rely on that function to return. (This is sometimes called "the invisible `goto` problem".)
- Exception handling in a complex system can quickly become unmanageable.
- Unhandled exceptions cause the program to terminate unexpectedly.
- Exceptions violate the "one flow in, one flow out" rule.
- It may be difficult for maintenance programmers to match corresponding `try` and `throw` statements. (The first problem is "who threw this?" and the second problem is "where is it going to end up?".)

The cure for the disadvantages of exceptions is ***disciplined use***: 374

- Use exceptions only when there is no alternative that is better.
- Document exceptions wherever they might affect behaviour in significant ways.
- Learn how to write "exception-safe code".
- For a large project:
  - plan an exception strategy during the initial design phase
  - define and use conventions for throwing and handling exceptions
  - make use of ***exception domains***, which are regions of the program from which no exception can escape

> ***An application should execute correctly in all normal situations with the exception handlers removed.***

Exception handling systems are required for large programs in which components are developed independently. If a component cannot perform the task assigned to it, it throws an exception.

> ***When no meaningful action can be performed locally, throw an exception.***

10.2.1 Exception handling syntax

Exception handling has two parts: 375

- The statement throw

  `throw` ⟨*expression*⟩

  raises, or throws, an exception. The exception is the object obtained by evaluating ⟨*expression*⟩. It can be anything that could be passed to a C++ function – a simple value (`char`, `string`, `int`, etc.), an object, or a reference or pointer to any of these things.

try...catch

- The statement

```
try
{
    ⟨try-sequence⟩
}
catch ( ⟨exc-type⟩ ⟨exc-name⟩ )
{
    ⟨catch-sequence⟩
}
```

handles exceptions thrown in ⟨*try-sequence*⟩. More precisely:

> If a statement in ⟨*try-sequence*⟩ throws an exception `e` whose class is ⟨*exc-type*⟩ or a class derived from ⟨*exc-type*⟩, then the statements ⟨*catch-sequence*⟩ are executed with ⟨*exc-name*⟩ bound to `e`.

10.2.2 Things to know about exceptions

There are several things worth knowing about `try/catch` statements:

- The `throw` statement may not be visible in ⟨*try-sequence*⟩: it may be (and often is) nested inside one or more levels of function call.

376

- The pair ⟨*exc-type*⟩ ⟨*exc-name*⟩ is analogous to the formal parameter of a function. It is as if the `catch` clause is "called" with the exception object as an argument. The semantics of catching an exception with a `catch` clause are exactly the same as receiving an argument with a function.

- Although we mentioned the "class" of the exception object, basic objects such as `int`s and `char`s can be thrown as exceptions.

377

- If an instance of a derived class is thrown, a handler for a base class will catch it.

  For example, if there is a class hierarchy

```
class Animal { ... };
   class Carnivore : public Animal { ... };
      class Dog : public Carnivore { ... };
```

  then the exception in the following sequence is caught:

```
try
{
   throw Dog(...);
}
catch (Animal beast)
{
   ...
}
```

- There may be more than one `catch` clause. The run-time systems matches the exception against each `catch` clause in turn, executing the first one that fits. 378

  Using the same hierarchy as above and a `try` statement

  ```
  try { monster() }
  catch (Dog d)  { .... }
  catch (Carnivore c) { .... }
  catch (Animal a) { .... }
  ```

  then if `monster()` `throw`s a `Dog`, a `Carnivore`, or an `Animal`, the appropriate handler will catch it. But writing

  ```
  try { monster() }
  catch (Animal a) { .... }
  catch (Dog d)  { .... }
  catch (Carnivore c) { .... }
  ```

  is pointless because `Dog`s and `Carnivore`s will never be caught.

- The classes of multiple `catch` clauses do not have to be related by inheritance. 379

- The clause `catch(...)` (that is, you actually write three dots between parentheses) catches all exceptions. If you use this in a multiple `catch` sequence, it should be the last `catch` clause. `catch(...)`

- If there is no `catch` clause that matches the exception, the exception is propagated to the next enclosing `catch` clause.

- In particular, a C++ program behaves ***as if*** it is inside a `try` block: 380

  ```
  try
  {
      main();
  }
  catch (...)
  {
      fail();
  }
  ```

  Thus unhandled exceptions terminate the program, usually with some more or less helpful error message.

- You can throw exceptions by value, or you can create an exception object dynamically (using `new`) and throw a pointer to it. If a pointer is thrown, the handler must delete the exception. 381

```
try
{
   throw X(...);
}
catch (X exc)
{
   ...
}
```

```
try
{
   throw new X(...);
}
catch (X *pexc)
{
   ...
   delete pexc;
}
```

- If a `catch`-block discovers that it cannot handle the exception, it can execute `throw` to pass the same exception to the next level. Note that it is not necessary to mention the exception object again: see Figure 123.

382

383

- A function can declare the exceptions it throws, like this:

```
void f1() throw();              // doesn't throw anything
void f2() throw(T1);            // may throw a T1
void f3() throw(T2, T3);        // may throw a T2 or a T3
void f4();                      // may throw anything
```

  If a function declaration specifies the exceptions that it throws, the corresponding definition of the function must specify the same exceptions.

- When a virtual function is redefined in a derived class, its throw declarations (if any) must be ***more restricted*** than the base class function. In Figure 124 on the facing page, the declarations of `f` and `h` in class `Derived` are legal, because the exceptions that can be thrown by the derived functions are a subset of the base class exceptions. The declaration of `g` in class `Derived` is not allowed.

384

- It is possible to throw an exception instead of executing a `return` statement. If you are searching a tree, for example, the eventual `return` must back out through all the recursive instantiations of the function. Knowing this, a smart (?) programmer might write `throw` rather than `return`. This is not usually a good idea:

385

  – The caller must write `throw`/`catch` instead of a simple call.

  – Since the exception handler must unwind the stack (see Section 10.2.4 on page 210 below), the time saved is likely to be small.

```
try { .... }
catch (Exception e)
{
    if (understand(e))
        // process e
    else
        throw;
}
```

Figure 123: Rethrowing

```
class Base
{
public:
   virtual void f();
   virtual void g() throw(X);
   virtual void h() throw(X, Y);
};

class Derived : public Base
{
   void f() throw(X);
   void g() throw(X, Y); // Illegal!
   void h() throw(X);
};
```

Figure 124: Inheriting exception specifications

– A mechanism designed for one purpose is being used for another. That C++ programmers love to do this does not make it good software engineering practice.

10.2.3 You didn't expect that

When you declare the exceptions a function can throw, this is ***not*** checked at compile-time: the code in Figure 125 will compile without errors. What happens in this case? When encoun- 386

```
void paintball throw(green, blue)
{
   ...
   throw red;
}
```

Figure 125: Throwing unexpected exceptions

tering an unexpected exception at run-time, the program will invoke the pre-defined function `unexpected()`.[40] `unexpected()`

By default, `unexpected()` will simply call `terminate()`, which will end your program by calling `abort()`. But you can re-define `unexpected` by defining your own function, e.g., `Ununexpected()`, and then setting this function in your code as the one to call through `set_unexpected()` – see Figure 126 on the next page. 387

In this code, `throw` will cause the original exception to be rethrown; but you can also throw a different exception, e.g., `std::bad_exception`.

[40] No, I'm not making this up. Sorry, it's too late to quit this course!

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

void Ununexpected()
{
    cerr << "Can't fool me!";
    throw;
}

main ()
{
    set_unexpected( Ununexpected );
    ...
}
```

Figure 126: Expecting the unexpected

It's perhaps not a surprise that C++11 deprecates exception specifications, so when writing new code, you should not bother with them. However, you might encounter them in existing code, so always be prepared for the unexpected!

388

Exception Specifications in C++11

Exception specifications didn't work out as, well, expected (one area where Java learned from language design mistakes). Since C++11 they are now deprecated, which means they should not be used for new code.
However, C++11 adds one new feature, the `noexcept` keyword, which indicates that a function is not allowed to throw an exception:

```
int foo() noexcept;
```

There is also a `noexcept` operator that you can use to check if a function is not throwing exceptions, like in `noexcept(foo())`, returning a `bool`.

10.2.4 How exceptions work

It is useful to have an approximate understanding of exceptions. They work roughly as follows:

exception frame

- When the run-time system reaches `try`, it pushes an ***exception frame*** onto the stack.[41] This frame contains descriptors for each `catch` clause.
- If the `try`-block executes normally, the exception frame is popped from the stack.
- If code in the `try`-block throws an exception, there will generally be a number of stack frames on top of the exception frame. The run-time system moves down the stack, popping these frames and performing any clean-up actions required (e.g., calling destructors), until it reaches the exception frame. This is called ***unwinding the stack***.

stack unwinding

[41] An implementation may use a separate stack for exceptions. This stack will contain pointers to the main stack.

- The run-time system inspects the `catch` descriptors until it finds a match. It then passes the exception to the `catch` block and gives control to the `catch` block. If no match is found, the run-time system will continue to unwind the stack as before until it finds the next exception frame.
- When the `catch` block terminates, it passes control to the statement following the entire `try-catch` sequence.

10.3 Exception-safe Coding

Exception-safe coding is an important topic for C++ programmers, but a rather large one for this course. Herb Sutter (Sutter 2005) devotes 44 pages to it. The basic rules are: 389

- Write each function to be ***exception safe***, meaning that it works properly when the functions it calls raise exceptions. exception safe
- Write each function to be ***exception neutral***, meaning that any exceptions that the function cannot safely handle are propagated to the caller. exception neutral

As an example, the constructor of the class `Vec`, discussed in Section 9.1 on page 177, is exception-safe: see Figure 127. The only action that may cause an exception to be thrown is `new`, because the default allocator throws an exception when there is no memory available.[42] The constructor does not handle this exception but, by default, passes it on to the caller. Therefore the constructor is ***exception neutral***. 390

The remaining danger is that the constructor does not work properly when the exception is raised. For example, it might happen that the pointers are set even though no data has been allocated. This doesn't matter in practice, because ***a constructor that raises an exception is assumed to have failed completely***. There is no object, and so values of the attributes of the object are irrelevant.

So far so good. Now consider a case where there ***is*** a problem. The code in Figure 128 on the following page is supposed to implement "popping" a stack (Sutter 2005, page 34). The subtle problem with this code is that `return result` requires a copy operation that may fail and throw an exception. If it does, the state of the stack has been changed (`--vused_`) but the popped value has been lost. The code in Figure 129 on the next page avoids this problem. 391 392

```
template <typename T>
Vec<T>::Vec<T>(size_t n, const T & val)
    : data(new T[n]), avail(data + n), limit(data + n)
{
    for (iterator p = data; p != avail; ++p)
        *p = val;
}
```

Figure 127: Constructor for class `Vec`

[42] Some older implementations still return a `null` pointer in this case, which was the default behaviour before exceptions were added to C++.

```
template<class T>
T Stack<T>::pop()
{
    if (vused_ == 0)
    {
        throw "Popping empty stack";
    }
    else
    {
        T result = v_[used_ - 1];
        --vused_;
        return result;
    }
}
```

Figure 128: Popping a stack: version 1

```
template<class T>
void Stack<T>::pop(T & result)
{
    if (vused_ == 0)
    {
        throw "Popping empty stack";
    }
    else
    {
        result = v_[used_ - 1];
        --vused_;
    }
}
```

Figure 129: Popping a stack: version 2

The STL is exception-safe. In order to avoid problems like these, it provides ***two*** functions for
393 popping a stack:

`void stack::pop()` removes an element from the stack

`const T & stack::top() const` returns the top element of the stack

Another rule that is followed by and assumed by the STL is: ***destructors do not throw exceptions***.

10.4 Resource Acquisition Is Initialization (RAII)

Generalizing the ideas of the previous section leads to an important principle of C++ coding: ***Resource Acquisition Is Initialization*** – RAII (one of the more bizarre abbreviations, even by C++ standards).

The basic idea is this: Since we need to ensure proper management of resources (allocating/deallocating memory, opening/closing files, etc.), we use *objects* to manage these resources for us. In doing so, we can rely on the fact that constructors, destructors, and exceptions are carefully designed to work together.[43]

A properly-written constructor either ***succeeds*** and creates a complete object or it ***fails*** and leaves no wreckage in its wake. However control leaves a block (by "falling through", executing `return`, or by raising an exception), all objects constructed within that block will be destroyed. 394

Figure 130 on the following page provides a simple example. The function does not close the output file because there is nowhere to put the statement `ofs.close()` in such a way that it is guaranteed to work correctly. (Note, for example, that the exception might be raised because the `ofstream` constructor failed and the report could not be written for this reason.)

However, ***it doesn't matter*** because the run-time system will ensure that `ofs` is destroyed (and therefore closed) whether the exception is raised or not. 395

Figure 131 on the next page is a bit more complicated: it demonstrates the use of RAII in a constructor. The constructor must never return having acquired the `File` but not the `Lock`. The problem is that the constructors for `File` and `Lock` may fail, leaving no object constructed and throwing an exception. The implementation ensures that this code is safe. For example, if the `File` constructor succeeds but the `Lock` constructor fails, the `File` will be destroyed before `Process` terminates.

10.4.1 Autopointers

Consider the following code: 396

```
void f()
{
    Gizmo *pg(new Gizmo);
    .... // do some stuff
    delete pg;
}
```

There are two potential problems with the function `f`:

1. If we forget to include the `delete` statement, the program has a memory leak.
2. It would actually be pretty stupid to forget to write `delete`, but if "`do some stuff`" throws an exception, the program has a memory leak anyway.

The standard template class `auto_ptr` uses RAII to avoid problems of this kind. Previously, we have seen how `new` allocates and `delete` deallocates memory, and that these operations must be carefully paired. Autopointers simplify programming by hiding the memory management required for pointers. auto_ptr 397

Figure 132 on page 215 illustrates an attempt to buy a car with pointers. It is obviously wrong, because the `new`s do not match the `delete`s. Even if the caller remembers to `delete` the pointer

[43] The result is to achieve something like `finally` in Java.

```
void report()
{
   ofstream ofs("report.txt");
   try
   {
      // generate report
   }
   catch (string reason)
   {
      cerr << "Report generation failed because " << reason << ".\n";
   }
}
```

Figure 130: Generating a report

```
class Process
{
   public:
      Process(string a, string b) : fs(a), lk(b) {}
   private:
      File fs;
      Lock lk;
};
```

Figure 131: Acquiring a file and a lock

returned by the function, one of the objects allocated within the function will not be `deleted`,
398 and there will be a memory leak.

Figure 133 on page 216 shows the same function implemented with autopointers. It works with autopointers for the following reasons:

1. After an autopointer assignment `p = q`, `p` points to the object and `q` is (effectively) null. In general, autopointers do not allow a situation in which two pointers point to the same object.

   After the `if` statement in the function `buyCar`, `myCar` will be a non-null pointer and one of `bug` and `yug` will be null.

2. When the autopointer copy constructor is used, the same rule applies.

   After `buyCar` returns, `myCar` is null, and the pointer to the purchased is owned by the caller.

3. The destructor of an autopointer deletes a non-null pointer and does nothing for a null pointer.

   When `buyCar` returns, there is one non-null pointer, pointing to the car that was not sold; that object is deleted.

```
Car *buyCar(long balance)
{
   Car *bug(new Bugatti("Veyron"));
   Car *yug(new Yugo("Cabrio"));
   Car *myCar;
   if (balance > 1250000)
      myCar = bug;
   else
      myCar = yug;
   try
   {
      myCar->drive();
   }
   catch (string crash)
   {
      cout << crash << endl;
      myCar = 0;
   }
   return myCar;
}
```

Figure 132: Buying a car with pointers

The caller receives a pointer to a car and, when that pointer goes out scope, the car is destroyed.[44]

An autopointer behaves in a way that is very similar to a pointer: we can use `*` or `->` to dereference it, the increment (`++`) and decrement (`--`) operators work, and so on. The difference is that the pointer target is automatically deleted at the end of the scope or when the stack is unwound during exception handling.

Autopointers have some special properties that make them different from ordinary pointers. These properties are necessary to avoid memory problems when autopointers are copied or passed around. The basic idea is that an autopointer has an ***owner*** who is responsible for deleting the object pointed to. In the following explanations, we assume that `pt` has type `auto_ptr<T>`.

- `*pt` is the object pointed to by the autopointer. 399
- `pt->m` provides access to members of the class of `*pt`, as usual.
- `pt.get()` returns the private variable `pt.myptr` which is an ordinary pointer of type `T*` used by the object `pt` to accomplish its magic.

  Note: The '`.`' indicates that we are calling a method of class `auto_ptr`, not class `T*`.
- `pt.release()` also returns `myptr` but then sets its value to null. The caller has ***taken ownership*** of the autopointer and is now responsible for releasing it.
- `pt.reset(newptr)` deletes `myptr` and replaces it by the new pointer `newptr`. 400

[44] And who would want to destroy a Veyron?

```
#include <memory>

auto_ptr<Car> buyCar()
{
   auto_ptr<Car> bug(new Bugatti("Veyron"));
   auto_ptr<Car> yug(new Yugo("Cabrio"));
   auto_ptr<Car> myCar;
   if (balance > 12500000)
      myCar = bug;
   else
      myCar = yug;
   try
   {
      myCar->drive();
   }
   catch (string crash)
   {
      cout << crash << endl;
      myCar.reset(0);
   }
   return myCar;
}
```

Figure 133: Buying a car with autopointers

- `pt.reset()` deletes `myptr` and sets it to 0.
- The assignment operator transfers ownership from one autopointer to another. Specifically, the assignment `pt1 = pt2` has an effect equivalent to

```
delete pt1;
pt1 = pt2.release();
```

The ownership property of autopointers has some interesting consequences. When the program in Figure 134 on the facing page is executed, it displays:

```
Woof!
Walking around the park ...
Aaargh!
Now we're back at home
```

Note the order of events carefully. The autopointer `pd` is set pointing to a newly constructed `Dog`. The autopointer is then passed by value to the function `walk`, which thereby assumes ownership of the `Dog`. The `Dog` walks around the park until the end of the scope. When `walk` returns, `pd` no longer points to the `Dog`, which expired at the end of its walk.

```
#include <memory>
#include <iostream>

using namespace std;

class Dog
{
public:
    Dog() { cout << "Woof!" << endl; }
    ~Dog() { cout << "Aaargh!" << endl; }
};

typedef auto_ptr<Dog> PD;

void walk(PD pd)
{
    cout << "Walking around the park ..." << endl;
}

void main()
{
    PD pd(new Dog);
    walk(pd);
    cout << "Now we're back at home" << endl;
}
```

Figure 134: Copying an autopointer

Warning! The ownership behaviour of autopointers is incompatible with the expectations of STL containers. Declarations such as this should never be used: 404

```
vector<auto_ptr<double> > vec;
```

For example, a sorting algorithm may select one element of a vector as a pivot and make a local copy of it. The pivot is then used to partition the array. If the vector contains autopointers, copying an element will invalidate the element! Consequently, sorting will delete the pivot and the sorted array will contain a null pointer.

Don't store autopointers in STL containers.

More information on autopointers can be found in (Weiss 2004, Chapter 8). (Koenig and Moo 2000, Chapter 14) discusses the implementation of a generic handle class that behaves similar (but is not identical) to autopointers.

10.4.2 Autopointers in C++11

Realizing that a single "smart" pointer type is not suitable for all circumstances, C++11 deprecates `auto_ptr`, replacing it with a number of new smart pointers with different semantics: `unique_ptr`,
405 `shared_ptr`, and `weak_ptr`.

`shared_ptr` has a shared ownership semantics: Instead of transferring ownership on assignment, like `auto_ptr` does, it increments a *reference count*. Calling a destructor decrements this count, until it reaches zero, at which point the referenced object is destructed. The `shared_ptr` works correctly with STL containers.

`unique_ptr` is the replacement for `auto_ptr`. It works in almost the same way, but has a few more restrictions. In particular, you can only assign one `unique_ptr` to another if the rhs of the
406 assignment is a temporary variable:

```
unique_ptr<string> up1(new string("Hello"));
unique_ptr<string> up2(new string("World"));
up1 = up2; // not allowed!
up1 = move(up2);
```

`std::move` Here, `move` is `std::move`, which allows to move ownership from one resource to another. It is another new C++11 feature: for more details, look up ***move constructors*** and ***xvalues*** in a reference book.

`weak_ptr` is a shared, non-owning reference to an object and used in conjunction with `shared_ptr`.

For more details, check a C++ book that has C++11 coverage, such as (Prata 2012, Chapter 16).

10.5 Coding Policies

407 Suppose that the function

```
Matrix rotation(const Vector & u, const Vector & v);
```

returns a `Matrix` corresponding to a rotation that takes vector `u` to vector `v`. In other words, the function returns a matrix M such that $M\mathbf{u} = \mathbf{v}$.

The matrix computed by this function is correct only if `u` and `v` are unit vectors: $|\mathbf{u}| = |\mathbf{v}| = 1$. The problem is what to do if `u` or `v` is not a unit vector.

The following sections suggest some answers.

10.5.1 Cowboy coding

We could just do nothing and hope that the vectors we get are normalized. (Or say that callers who do not provide unit vectors deserve what they get.)

However, this might not be an acceptable alternative because the consequences could be serious. (Perhaps the rotation will be used to rotate an aircraft into the correct orientation for landing.)

10.5.2 Pessimistic coding

Assume the worst: normalize `u` and `v` before using them. This is quite an expensive (i.e., time consuming) operation and is a waste of time if callers provide unit vectors anyway.

More importantly, there is not usually an obvious way to correct invalid parameters, so this cannot be considered as a general solution. 408

10.5.3 Defensive coding

Inspect the vectors and take some action if they are not unit vectors: 409

```
Matrix rotation(const Vector & u, const Vector & v)
{
    if (fabs(u.length() - 1) > TOL || fabs(v.length() - 1) > TOL)
        // error handling
    else
        // compute rotation matrix
}
```

This approach is called ***don't trust the caller*** or, more politely, ***defensive coding***. We will discuss possible ways of handling errors later.

10.5.4 Contractual coding

Specify that the vectors passed to the function must be unit vectors and that the behaviour of the function is undefined if they are not. The usual way to do this is to write an assertion or comment called a ***precondition*** in the code: 410

```
/**
 * Construct the matrix that rotates one vector to another.
 * \param u is a vector representing an initial orientation.
 * \param v is a vector representing the final orientation.
 * The matrix, applied to \a u, will yield \a v.
 * \pre The vectors \a u and \a v must be unit vectors.
 */
Matrix rotation(const Vector & u, const Vector & v);
```

The precondition passes the responsibility to the caller. Effectively, the comments define a ***contract*** between the caller and the function: if you give me unit vectors, I will give you a rotation matrix; if you give me anything else, I guarantee nothing. This form of coding is called ***design by contract*** or DBC; it was pioneered by Bertrand Meyer (Meyer 1997) but has been recommended by other people as well (see, for example, (Hunt and Thomas 2000, pages 109–119)). The code for the function does not check the length of the vectors `u` and `v` except, perhaps, during development.

Most languages, including C++, do not provide direct support for DBC. However, we can use assertions (Section 2.7 on page 34) as an implementation. Using the matrix rotation example:

```
/** \pre The vectors \a u and \a v must be unit vectors. */
Matrix rotation(const Vector & u, const Vector & v)
{
    assert(fabs(u.length() - 1 <= TOL);
    assert(fabs(v.length() - 1 <= TOL);
    ...
}
```

There are advocates and opposers of both DBC and defensive coding. There is a general trend away from defensive coding (the "traditional" approach) and towards DBC. For example, the home page for the Java Modeling Language[45] says that JML is a "design by contract" language
411 for Java. Here are some arguments for and against each approach.

Efficiency: DBC has no run-time overhead. Defensive coding executes tests which almost always fail (e.g., most vectors passed to `rotation` are in fact unit vectors).

Safety: DBC relies on programmers to respect contracts. Defensive coding ensures that malingerers will be caught. (However, if DBC is implemented with assertions, contract violations will be caught whenever assertions are enabled.)

Clarity: DBC requires maintainers to read comments. With defensive coding, the checks are in the code.

Completeness: There are situations that DBC cannot handle, such as the validity of user input. In these situations, we can – and must – code defensively.

DBC and defensive programming are not mutually exclusive: you can use both. The best policy is to decide which fits the needs of the particular application better and then use it.

10.6 Static assertions

The `assert` macro is useful, but it doesn't do anything until run-time. It is sometimes useful to check a condition at compile-time, causing a compile-time error if the condition is false. This
static assertions is known as ***static assertion*** checking. It is even more useful if the compiler's error message
412 describes the problem. For example, the following program does not compile:

[45] `http://www.cs.iastate.edu/~leavens/JML/`

```
#define StaticAssert ....

int main()
{
    StaticAssert(sizeof(char) < sizeof(int),
        char_is_smaller_than_int);
    StaticAssert(sizeof(int) >= 8,
        int_has_at_least_eight_bytes);
}
```

The second argument of `StaticAssert` must be a valid C++ variable name but the variable does not have to be defined. The error messages from the compiler are: 413

```
assert.cpp(11): error C2466:
   cannot allocate an array of constant size 0
assert.cpp(11): error C2087:
   'ERROR_int_has_at_least_eight_bytes' : missing subscript
```

The second message tells the programmer that the assumption "`int`s have at least eight bytes" is incorrect.

The LOKI library[46] defines a `StaticAssert` in various ways for different compilers to ensure that the error message is useful. A simple way to achieve the effect can be seen here: 414

```
template<int> struct StaticAssertion;
template<> struct StaticAssertion<true> { };

#define STATIC_ASSERT(expression, message) \
    { StaticAssertion<((expression) != 0)> ASSERTION_##message; }

int main() {
  // static assertion using above macro
  STATIC_ASSERT( sizeof(int) >= 4,
                 size_of_int_less_than_4_bytes);

  // static assertion in C++11
  static_assert( sizeof(int) >= 4,
                 "Size of int is less than 4 bytes.");
  return 0;
}
```

The BOOST library also offers a static assertion check.

[46] LOKI, http://loki-lib.sourceforge.net/

`static_assert` in C++11

Since static assertions have become widely used, C++11 includes static assertion checking built-in via a `static_assert` command that you can use like this:

```
static_assert(sizeof(int) >= 4, "Size of int is less than 4 bytes.");
```

Static assertions are particularly useful for template programming, in order to check the computations performed at compile-time.

10.7 Summary

- Decide on a ***failure management policy*** for your application and follow it consistently.
- Use simple solutions for simple problems. Error codes might be good enough for a simple system.
- Distinguish between events that are ***possible*** (although they might be unwanted, unlikely, etc.) and events that are ***impossible*** (if the program is correct).

  Possible events should be handled with messages, codes, or exceptions. Impossible events should be "handled" with assertions or DBC.

References

Hunt, A. and D. Thomas (2000). *The Pragmatic Programmer*. Addison-Wesley.

Koenig, A. and B. E. Moo (2000). *Accelerated C++: Practical Programming by Example*. Addison-Wesley.

Meyer, B. (1997). *Object-Oriented Software Construction* (2nd ed.). Professional Technical Reference. Prentice Hall.

Prata, S. (2012). *C++ Primer Plus* (6th ed.). Addison-Wesley.

Sutter, H. (2005). *Exceptional C++ Style: 40 New Engineering Puzzles, Programming Problems, and Solutions*. C++ In-Depth Series. Addison-Wesley.

Weiss, M. A. (2004). *C++ for Java Programmers*. Pearson Prentice Hall.