14

Exception Handling

Don´t interrupt me
while I´m interrupting.
– Winston S. Churchill

Error handling — grouping of exceptions — catching exceptions — catch all — rethrow — resource management — ***auto_ptr*** — exceptions and ***new*** — resource exhaustion — exceptions in constructors — exceptions in destructors — exceptions that are not errors — exception specifications — unexpected exceptions — uncaught exceptions — exceptions and efficiency — error-handling alternatives — standard exceptions — advice — exercises.

14.1 Error Handling [except.error]

As pointed out in §8.3, the author of a library can detect run-time errors but does not in general have any idea what to do about them. The user of a library may know how to cope with such errors but cannot detect them – or else they would have been handled in the user's code and not left for the library to find. The notion of an *exception* is provided to help deal with such problems. The fundamental idea is that a function that finds a problem it cannot cope with *throws* an exception, hoping that its (direct or indirect) caller can handle the problem. A function that wants to handle that kind of problem can indicate that it is willing to *catch* that exception (§2.4.2, §8.3).

This style of error handling compares favorably with more traditional techniques. Consider the alternatives. Upon detecting a problem that cannot be handled locally, the program could:

[1] terminate the program,
[2] return a value representing ‘‘error,’’
[3] return a legal value and leave the program in an illegal state, or
[4] call a function supplied to be called in case of ‘‘error.’’

Case [1], ''terminate the program,'' is what happens by default when an exception isn't caught. For most errors, we can and must do better. In particular, a library that doesn't know about the purpose and general strategy of the program in which it is embedded cannot simply ***exit***( ) or ***abort***( ). A library that unconditionally terminates cannot be used in a program that cannot afford to crash. One way of viewing exceptions is as a way of giving control to a caller when no meaningful action can be taken locally.

Case [2], ''return an error value,'' isn't always feasible because there is often no acceptable ''error value.'' For example, if a function returns an ***int***, *every* ***int*** might be a plausible result. Even where this approach is feasible, it is often inconvenient because every call must be checked for the error value. This can easily double the size of a program (§14.8). Consequently, this approach is rarely used systematically enough to detect all errors.

Case [3], ''return a legal value and leave the program in an illegal state,'' has the problem that the calling function may not notice that the program has been put in an illegal state. For example, many standard C library functions set the global variable ***errno*** to indicate an error (§20.4.1, §22.3). However, programs typically fail to test ***errno*** consistently enough to avoid consequential errors caused by values returned from failed calls. Furthermore, the use of global variables for recording error conditions doesn't work well in the presence of concurrency.

Exception handling is not meant to handle problems for which case [4], ''call an error-handler function,'' is relevant. However, in the absence of exceptions, an error-handler function has exactly the three other cases as alternatives for how *it* handles the error. For a further discussion of error-handling functions and exceptions, see §14.4.5.

The exception-handling mechanism provides an alternative to the traditional techniques when they are insufficient, inelegant, and error-prone. It provides a way of explicitly separating error-handling code from ''ordinary'' code, thus making the program more readable and more amenable to tools. The exception-handling mechanism provides a more regular style of error handling, thus simplifying cooperation between separately written program fragments.

One aspect of the exception-handling scheme that will appear novel to C and Pascal programmers is that the default response to an error (especially to an error in a library) is to terminate the program. The traditional response has been to muddle through and hope for the best. Thus, exception handling makes programs more ''brittle'' in the sense that more care and effort must be taken to get a program to run acceptably. This seems preferable, though, to getting wrong results later in the development process – or after the development process is considered complete and the program is handed over to innocent users. Where termination is unacceptable, we can catch all exceptions (§14.3.2) or catch all exceptions of a specific kind (§14.6.2). Thus, an exception terminates a program only if a programmer allows it to terminate. This is preferable to the unconditional termination that happens when a traditional incomplete recovery leads to a catastrophic error.

Sometimes people have tried to alleviate the unattractive aspects of ''muddling through'' by writing out error messages, putting up dialog boxes asking the user for help, etc. Such approaches are primarily useful in debugging situations in which the user is a programmer familiar with the structure of the program. In the hands of nondevelopers, a library that asks the (possibly absent) user/operator for help is unacceptable. Also, in many cases error messages have no place to go (say, if the program runs in an environment in which ***cerr*** doesn't connect to anything a user notices); they would be incomprehensible to an end user anyway. At a minimum, the error message might be in the wrong natural language (say, in Finnish to a English user). Worse, the error

message would typically refer to library concepts completely unknown to a user (say, ''bad argument to atan2,'' caused by bad input to a graphics system). A good library doesn't ''blabber'' in this way. Exceptions provide a way for code that detects a problem from which it cannot recover to pass the problem on to some part of the system that might be able to recover. Only a part of the system that has some idea of the context in which the program runs has any chance of composing a meaningful error message.

The exception-handling mechanism can be seen as a run-time analog to the compile-time type checking and ambiguity control mechanisms. It makes the design process more important and can increase the work needed to get an initial and buggy version of a program running. However, the result is code that has a much better chance to run as expected, to run as an acceptable part of a larger program, to be comprehensible to other programmers, and to be amenable to manipulation by tools. Similarly, exception handling provides specific language features to support ''good style'' in the same way other C++ features support ''good style'' that can be practiced only informally and incompletely in languages such as C and Pascal.

It should be recognized that error handling will remain a difficult task and that the exception-handling mechanism – although more formalized than the techniques it replaces – is still relatively unstructured compared with language features involving only local control flow. The C++ exception-handling mechanism provides the programmer with a way of handling errors where they are most naturally handled, given the structure of a system. Exceptions make the complexity of error handling visible. However, exceptions are not the cause of that complexity. Be careful not to blame the messenger for bad news.

This may be a good time to review §8.3, where the basic syntax, semantics, and style-of-use aspects of exception handling are presented.

14.1.1 Alternative Views on Exceptions [except.views]

''Exception'' is one of those words that means different things to different people. The C++ exception-handling mechanism is designed to support handling of errors and other exceptional conditions (hence the name). In particular, it is intended to support error handling in programs composed of independently developed components.

The mechanism is designed to handle only synchronous exceptions, such as array range checks and I/O errors. Asynchronous events, such as keyboard interrupts and certain arithmetic errors, are not necessarily exceptional and are not handled directly by this mechanism. Asynchronous events require mechanisms fundamentally different from exceptions (as defined here) to handle them cleanly and efficiently. Many systems offer mechanisms, such as signals, to deal with asynchrony, but because these tend to be system-dependent, they are not described here.

The exception-handling mechanism is a nonlocal control structure based on stack unwinding (§14.4) that can be seen as an alternative return mechanism. There are therefore legitimate uses of exceptions that have nothing to do with errors (§14.5). However, the primary aim of the exception-handling mechanism and the focus of this chapter is error handling and the support of fault tolerance.

Standard C++ doesn't have the notion of a thread or a process. Consequently, exceptional circumstances relating to concurrency are not discussed here. The concurrency facilities available on your system are described in its documentation. Here, I'll just note that the C++ exception-

handling mechanism was designed to be effective in a concurrent program as long as the programmer (or system) enforces basic concurrency rules, such as properly locking a shared data structure while using it.

The C++ exception-handling mechanisms are provided to report and handle errors and exceptional events. However, the programmer must decide what it means to be exceptional in a given program. This is not always easy (§14.5). Can an event that happens most times a program is run be considered exceptional? Can an event that is planned for and handled be considered an error? The answer to both questions is yes. ''Exceptional'' does not mean ''almost never happens'' or ''disastrous.'' It is better to think of an exception as meaning ''some part of the system couldn't do what it was asked to do.'' Usually, we can then try something else. Exception ***throw***s should be infrequent compared to function calls or the structure of the system has been obscured. However, we should expect most large programs to ***throw*** and ***catch*** at least some exceptions in the course of a normal and successful run.

14.2 Grouping of Exceptions [except.grouping]

An exception is an object of some class representing an exceptional occurrence. Code that detects an error (often a library) ***throw***s an object (§8.3). A piece of code expresses desire to handle an exception by a ***catch*** clause. The effect of a ***throw*** is to unwind the stack until a suitable ***catch*** is found (in a function that directly or indirectly invoked the function that threw the exception).

Often, exceptions fall naturally into families. This implies that inheritance can be useful to structure exceptions and to help exception handling. For example, the exceptions for a mathematical library might be organized like this:

```
class Matherr { };
class Overflow: public Matherr { };
class Underflow: public Matherr { };
class Zerodivide: public Matherr { };
// ...
```

This allows us to handle any ***Matherr*** without caring precisely which kind it is. For example:

```
void f()
{
    try {
        // ...
    }
    catch (Overflow) {
        // handle Overflow or anything derived from Overflow
    }
    catch (Matherr) {
        // handle any Matherr that is not Overflow
    }
}
```

Here, an ***Overflow*** is handled specifically. All other ***Matherr*** exceptions will be handled by the general case.

Organizing exceptions into hierarchies can be important for robustness of code. For example, consider how you would handle all exceptions from a library of mathematical functions without such a grouping mechanism. This would have to be done by exhaustively listing the exceptions:

```
void g()
{
    try {
        // ...
    }
    catch (Overflow) { /* ... */ }
    catch (Underflow) { /* ... */ }
    catch (Zerodivide) { /* ... */ }
}
```

This is not only tedious, but a programmer can easily forget to add an exception to the list. Consider what would be needed if we didn't group math exceptions. When we added a new exception to the math library, every piece of code that tried to handle every math exception would have to be modified. In general, such universal update is not feasible after the initial release of the library. Often, there is no way of finding every relevant piece of code. Even when there is, we cannot in general assume that every piece of source code is available or that we would be willing to make changes if it were. These recompilation and maintenance problems would lead to a policy that no new exceptions can be added to a library after its first release; *that* would be unacceptable for almost all libraries. This reasoning leads exceptions to be defined as per-library or per-subsystem class hierarchies (§14.6.2).

Please note that neither the built-in mathematical operations nor the basic math library (shared with C) reports arithmetic errors as exceptions. One reason for this is that detection of some arithmetic errors, such as divide-by-zero, are asynchronous on many pipelined machine architectures. The ***Matherr*** hierarchy described here is only an illustration. The standard library exceptions are described in §14.10.

14.2.1 Derived Exceptions [except.derived]

The use of class hierarchies for exception handling naturally leads to handlers that are interested only in a subset of the information carried by exceptions. In other words, an exception is typically caught by a handler for its base class rather than by a handler for its exact class. The semantics for catching and naming an exception are identical to those of a function accepting an argument. That is, the formal argument is initialized with the argument value (§7.2). This implies that the exception thrown is ''sliced'' to the exception caught (§12.2.3). For example:

```
class Matherr {
    // ...
    virtual void debug_print() const { cerr << "Math error"; }
};

class Int_overflow: public Matherr {
    const char* op;
    int a1, a2;
```

```
public:
    Int_overflow(const char* p, int a, int b) { op = p; a1 = a; a2 = b; }
    virtual void debug_print() const { cerr << op << '(' << a1 << ',' << a2 << ')'; }
    // ...
};

void f()
{
    try {
        g();
    }
    catch (Matherr m) {
        // ...
    }
}
```

When the *Matherr* handler is entered, *m* is a *Matherr* object – even if the call to *g*() threw *Int_overflow*. This implies that the extra information found in an *Int_overflow* is inaccessible.

As always, pointers or references can be used to avoid losing information permanently. For example, we might write:

```
int add(int x, int y)
{
    if ((x>0 && y>0 && x>INT_MAX-y) || (x<0 && y<0 && x<INT_MIN-y))
        throw Int_overflow("+",x,y);

    return x+y;      // x+y will not overflow
}

void f()
{
    try {
        int i1 = add(1,2);
        int i2 = add(INT_MAX,-2);
        int i3 = add(INT_MAX,2);      // here we go!
    }
    catch (Matherr& m) {
        // ...
        m.debug_print();
    }
}
```

The last call of *add*() triggers an exception that causes *Int_overflow*::*debug_print*() to be invoked. Had the exception been caught by value rather than by reference, *Matherr*::*debug_print*() would have been invoked instead.

14.2.2 Composite Exceptions [except.composite]

Not every grouping of exceptions is a tree structure. Often, an exception belongs to two groups. For example:

```
class Netfile_err : public Network_err, public File_system_err { /* ... */ };
```

Such a ***Netfile_err*** can be caught by functions dealing with network exceptions:

```
void f()
{
    try {
        // something
    }
    catch(Network_err& e) {
        // ...
    }
}
```

and also by functions dealing with file system exceptions:

```
void g()
{
    try {
        // something else
    }
    catch(File_system_err& e) {
        // ...
    }
}
```

This nonhierarchical organization of error handling is important where services, such as networking, are transparent to users. In this case, the writer of `g()` might not even be aware that a network is involved (see also §14.6).

14.3 Catching Exceptions [except.catch]

Consider:

```
void f()
{
    try {
        throw E();
    }
    catch(H) {
        // when do we get here?
    }
}
```

The handler is invoked:

[1] If ***H*** is the same type as ***E***.
[2] If ***H*** is an unambiguous public base of ***E***.
[3] If ***H*** and ***E*** are pointer types and [1] or [2] holds for the types to which they refer.
[4] If ***H*** is a reference and [1] or [2] holds for the type to which ***H*** refers.

In addition, we can add ***const*** to the type used to catch an exception in the same way that we can add it to a function parameter. This doesn't change the set of exceptions we can catch; it only restricts us from modifying the exception caught.

In principle, an exception is copied when it is thrown, so the handler gets hold of a copy of the original exception. In fact, an exception may be copied several times before it is caught. Consequently, we cannot throw an exception that cannot be copied. The implementation may apply a wide variety of strategies for storing and transmitting exceptions. It is guaranteed, however, that there is sufficient memory to allow ***new*** to throw the standard out-of-memory exception, ***bad_alloc*** (§14.4.5).

14.3.1 Re-Throw [except.rethrow]

Having caught an exception, it is common for a handler to decide that it can't completely handle the error. In that case, the handler typically does what can be done locally and then throws the exception again. Thus, an error can be handled where it is most appropriate. This is the case even when the information needed to best handle the error is not available in a single place, so that the recovery action is best distributed over several handlers. For example:

```
void h()
{
    try {
        // code that might throw Math errors
    }
    catch (Matherr) {
        if (can_handle_it_completely) {
            // handle the Matherr

            return;
        }
        else {
            // do what can be done here

            throw;      // re-throw the exception
        }
    }
}
```

A re-throw is indicated by a ***throw*** without an operand. If a re-throw is attempted when there is no exception to re-throw, ***terminate***() (§14.7) will be called. A compiler can detect and warn about some, but not all, such cases.

The exception re-thrown is the original exception caught and not just the part of it that was accessible as a ***Matherr***. In other words, had an ***Int_overflow*** been thrown, a caller of ***h***() could still catch an ***Int_overflow*** that ***h***() had caught as a ***Matherr*** and decided to re-throw.

14.3.2 Catch Every Exception [except.every]

A degenerate version of this catch-and-rethrow technique can be important. As for functions, the ellipsis ... indicates ''any argument'' (§7.6), so ***catch***(...) means ''catch any exception.'' For example:

```
void m()
{
    try {
        // something
    }
    catch (...) {        // handle every exception
        // cleanup
        throw;
    }
}
```

That is, if any exception occurs as the result of executing the main part of `m()`, the cleanup action in the handler is invoked. Once the local cleanup is done, the exception that caused the cleanup is re-thrown to trigger further error handling. See §14.6.3.2 for a technique to gain information about an exception caught by a `...` handler.

One important aspect of error handling in general and exception handling in particular is to maintain invariants assumed by the program (§24.3.7.1). For example, if `m()` is supposed to leave certain pointers in the state in which it found them, then we can write code in the handler to give them acceptable values. Thus, a ''catch every exception'' handler can be used to maintain arbitrary invariants. However, for many important cases such a handler is not the most elegant solution to this problem (see §14.4).

14.3.2.1 Order of Handlers [except.order]

Because a derived exception can be caught by handlers for more than one exception type, the order in which the handlers are written in a *try* statement is significant. The handlers are tried in order. For example:

```
void f()
{
    try {
        // ...
    }
    catch (std::ios_base::failure) {
        // handle any stream io error (§14.10)
    }
    catch (std::exception& e) {
        // handle any standard library exception (§14.10)
    }
    catch (...) {
        // handle any other exception (§14.3.2)
    }
}
```

Because the compiler knows the class hierarchy, it can catch many logical mistakes. For example:

```
void g()
{
    try {
        // ...
    }
    catch (...) {
        // handle every exception (§14.3.2)
    }
    catch (std::exception& e) {
        // handle any standard library exception (§14.10)
    }
    catch (std::bad_cast) {
        // handle dynamic_cast failure (§15.4.2)
    }
}
```

Here, the ***exception*** will never be considered. Even if we removed the ''catch-all'' handler, ***bad_cast*** wouldn't be considered because it is derived from ***exception***.

14.4 Resource Management [except.resource]

When a function acquires a resource – that is, it opens a file, allocates some memory from the free store, sets an access control lock, etc., – it is often essential for the future running of the system that the resource be properly released. Often that ''proper release'' is achieved by having the function that acquired it release it before returning to its caller. For example:

```
void use_file(const char* fn)
{
    FILE* f = fopen(fn,"w");
    // use f
    fclose(f);
}
```

This looks plausible until you realize that if something goes wrong after the call of ***fopen***() and before the call of ***fclose***(), an exception may cause ***use_file***() to be exited without ***fclose***() being called. Exactly the same problem can occur in languages that do not support exception handling. For example, the standard C library function ***longjmp***() can cause the same problem. Even an ordinary ***return***-statement could exit ***use_file*** without closing ***f***.

A first attempt to make ***use_file***() to be fault-tolerant looks like this:

```
void use_file(const char* fn)
{
    FILE* f = fopen(fn,"r");
    try {
        // use f
    }
```

```
        catch ( . . . ) {
                fclose(f);
                throw;
        }
        fclose(f);
}
```

The code using the file is enclosed in a ***try*** block that catches every exception, closes the file, and re-throws the exception.

The problem with this solution is that it is verbose, tedious, and potentially expensive. Furthermore, any verbose and tedious solution is error-prone because programmers get bored. Fortunately, there is a more elegant solution. The general form of the problem looks like this:

```
void acquire()
{
        // acquire resource 1
        // ...
        // acquire resource n

        // use resources

        // release resource n
        // ...
        // release resource 1
}
```

It is typically important that resources are released in the reverse order of their acquisition. This strongly resembles the behavior of local objects created by constructors and destroyed by destructors. Thus, we can handle such resource acquisition and release problems by a suitable use of objects of classes with constructors and destructors. For example, we can define a class ***File_ptr*** that acts like a ***FILE****:

```
class File_ptr {
        FILE* p;
public:
        File_ptr(const char* n, const char* a) { p = fopen(n,a); }
        File_ptr(FILE* pp) { p = pp; }
        ~File_ptr() { fclose(p); }

        operator FILE*() { return p; }
};
```

We can construct a ***File_ptr*** given either a ***FILE**** or the arguments required for ***fopen()***. In either case, a ***File_ptr*** will be destroyed at the end of its scope and its destructor will close the file. Our program now shrinks to this minimum:

```
void use_file(const char* fn)
{
        File_ptr f(fn,"r");
        // use f
}
```

The destructor will be called independently of whether the function is exited normally or exited because an exception is thrown. That is, the exception-handling mechanisms enable us to remove the error-handling code from the main algorithm. The resulting code is simpler and less error-prone than its traditional counterpart.

The process of searching ''up through the stack'' to find a handler for an exception is commonly called ''stack unwinding.'' As the call stack is unwound, the destructors for constructed local objects are invoked.

14.4.1 Using Constructors and Destructors [except.using]

The technique for managing resources using local objects is usually referred to as ''resource acquisition is initialization.'' This is a general technique that relies on the properties of constructors and destructors and their interaction with exception handling.

An object is not considered constructed until its constructor has completed. Then and only then will stack unwinding call the destructor for the object. An object composed of sub-objects is constructed to the extent that its sub-objects have been constructed. An array is constructed to the extent that its elements have been constructed (and only fully constructed elements are destroyed during unwinding).

A constructor tries to ensure that its object is completely and correctly constructed. When that cannot be achieved, a well-written constructor restores – as far as possible – the state of the system to what it was before creation. Ideally, naively written constructors always achieve one of these alternatives and don't leave their objects in some ''half-constructed'' state. This can be achieved by applying the ''resource acquisition is initialization'' technique to the members.

Consider a class ***X*** for which a constructor needs to acquire two resources: a file ***x*** and a lock ***y***. This acquisition might fail and throw an exception. Class ***X***'s constructor must never return having acquired the file but not the lock. Furthermore, this should be achieved without imposing a burden of complexity on the programmer. We use objects of two classes, ***File_ptr*** and ***Lock_ptr***, to represent the acquired resources. The acquisition of a resource is represented by the initialization of the local object that represents the resource:

```
class X {
    File_ptr aa;
    Lock_ptr bb;
public:
    X(const char* x, const char* y)
        : aa(x,"rw"), // acquire 'x'
          bb(y)       // acquire 'y'
    {}
    // ...
};
```

Now, as in the local object case, the implementation can take care of all of the bookkeeping. The user doesn't have to keep track at all. For example, if an exception occurs after ***aa*** has been constructed but before ***bb*** has been, then the destructor for ***aa*** but not for ***bb*** will be invoked.

This implies that where this simple model for acquisition of resources is adhered to, the author of the constructor need not write explicit exception-handling code.

The most common resource acquired in an ad-hoc manner is memory. For example:

```
class Y {
        int* p;
        void init();
public:
        Y(int s) { p = new int[s]; init(); }
        ~Y() { delete[] p; }
        // ...
};
```

This practice is common and can lead to ''memory leaks.'' If an exception is thrown by *init*(), then the store acquired will not be freed; the destructor will not be called because the object wasn't completely constructed. A safe variant is:

```
class Z {
        vector<int> p;
        void init();
public:
        Z(int s) : p(s) { init(); }
        // ...
};
```

The memory used by *p* is now managed by *vector*. If *init*() throws an exception, the memory acquired will be freed when the destructor for *p* is (implicitly) invoked.

14.4.2 Auto_ptr [except.autoptr]

The standard library provides the template class *auto_ptr*, which supports the ''resource acquisition is initialization'' technique. Basically, an *auto_ptr* is initialized by a pointer and can be dereferenced in the way that a pointer can. Also, the object pointed to will be implicitly deleted at the end of the *auto_ptr*'s scope. For example:

```
void f(Point p1, Point p2, auto_ptr<Circle> pc, Shape* pb) // remember to delete pb on exit
{
        auto_ptr<Shape> p(new Rectangle(p1,p2));    // p points to a rectangle
        auto_ptr<Shape> pbox(pb);

        p->rotate(45); // use auto_ptr<Shape> exactly as a Shape*
        // ...
        if (in_a_mess) throw Mess();
        // ...
}
```

Here the *Rectangle*, the *Shape* pointed to by *pb*, and the *Circle* pointed to by *pc* are deleted whether or not an exception is thrown.

To achieve this *ownership semantics* (also called *destructive copy semantics*), *auto_ptr*s have a copy semantics that differs radically from that of ordinary pointers: When one *auto_ptr* is copied into another, the source no longer points to anything. Because copying an *auto_ptr* modifies it, a *const auto_ptr* cannot be copied.

The *auto_ptr* template is declared in *<memory>*. It can be described by an implementation:

```
template<class X> class std::auto_ptr {
    template <class Y> struct auto_ptr_ref { /* ... */ };   // helper class
    X* ptr;
public:
    typedef X element_type;

    explicit auto_ptr(X* p =0) throw() { ptr=0; }
    auto_ptr(auto_ptr& a) throw() { ptr=a.ptr; a.ptr=0; }     // note: not const auto_ptr&
    template<class Y> auto_ptr(auto_ptr<Y>& a) throw() { ptr=a.ptr; a.ptr=0; }
    auto_ptr& operator=(auto_ptr& a) throw() { ptr=a.ptr; a.ptr=0; }
    template<class Y> auto_ptr& operator=(auto_ptr<Y>& a) throw() { ptr=a.ptr; a.ptr=0; }
    ~auto_ptr() throw() { delete ptr; }

    X& operator*() const throw() { return *ptr; }
    X* operator->() const throw() { return ptr; }
    X* get() const throw() { return ptr; }                    // extract pointer
    X* release() throw() { X* t = ptr; ptr=0; return t; }     // relinquish ownership
    void reset(X* p =0) throw() { if (p!=ptr) { delete ptr; ptr=p; } }

    auto_ptr(auto_ptr_ref<X>) throw();                  // copy from auto_ptr_ref
    template<class Y> operator auto_ptr_ref<Y>() throw();   // copy from auto_ptr_ref
    template<class Y> operator auto_ptr<Y>() throw();       // destructive copy from auto_ptr
};
```

The purpose of ***auto_ptr_ref*** is to implement the destructive copy semantics for ordinary ***auto_ptr***s while making it impossible to copy a ***const auto_ptr***. The template constructor and template assignment ensures that an ***auto_ptr<D>*** can be implicitly converted to a ***auto_ptr<B>*** if a ***D**** can be converted to a ***B****. For example:

```
void g(Circle* pc)
{
    auto_ptr<Circle> p2 = pc;  // now p2 is responsible for deletion
    auto_ptr<Circle> p3 = p2;  // now p3 is responsible for deletion (and p2 isn't)
    p2->m = 7;                 // programmer error: p2.get()==0
    Shape* ps = p3.get();      // extract the pointer from an auto_ptr
    auto_ptr<Shape> aps = p3; // transfer of ownership and convert type
    auto_ptr<Circle> p4 = p;   // programmer error: now p4 is also responsible for deletion
}
```

The effect of having more than one ***auto_ptr*** own an object is undefined; most likely the object will be deleted twice (with bad effects).

Note that ***auto_ptr***'s destructive copy semantics means that it does not meet the requirements for elements of a standard container or for standard algorithms such as ***sort()***. For example:

```
void h(vector< auto_ptr<Shape*> >& v)    // dangerous: use of auto_ptr in container
{
    sort(v.begin(),v.end());       // Don't do this: The sort will probably mess up v
}
```

Clearly, ***auto_ptr*** isn't a general smart pointer. However, it provides the service for which it was designed – exception safety for automatic pointers – with essentially no overhead.

14.4.3 Caveat [except.caveat]

Not all programs need to be resilient against all forms of failure, and not all resources are critical enough to warrant the effort to protect them using ''resource acquisition is initialization,'' *auto_ptr*, and *catch*(...). For example, for many programs that simply read an input and run to completion, the most suitable response to a serious run-time error is to abort the process (after producing a suitable diagnostic). That is, let the system release all acquired resources and let the user re-run the program with a more suitable input. The strategy discussed here is intended for applications for which such a simplistic response to a run-time error is unacceptable. In particular, a library designer usually cannot make assumptions about the fault tolerance requirements of a program using the library and is thus forced to avoid all unconditional run-time failures and to release all resources before a library function returns to the calling program. The ''resource acquisition is initialization'' strategy, together with the use of exceptions to signal failure, is suitable for many such libraries.

14.4.4 Exceptions and New [except.new]

Consider:

```
void f(Arena& a, X* buffer)
{
    X* p1 = new X;
    X* p2 = new X[10];

    X* p3 = new(buffer[10]) X;          // place X in buffer (no deallocation needed)
    X* p4 = new(buffer[11]) X[10];

    X* p5 = new(a) X;                   // allocation from Arena a (deallocate from a)
    X* p6 = new(a) X[10];
}
```

What happens if *X´s* constructor throws an exception? Is the memory allocated by the *operator new*() freed? For the ordinary case, the answer is yes, so the initializations of *p1* and *p2* don't cause memory leaks.

When the placement syntax (§10.4.11) is used, the answer cannot be that simple. Some uses of that syntax allocate memory, which then ought to be released; however, some don't. Furthermore, the point of using the placement syntax is to achieve nonstandard allocation, so nonstandard freeing is typically required. Consequently, the action taken depends on the allocator used. If an allocator *Z*::*operator new*() is used, *Z*::*operator delete*() is invoked if it exists; otherwise, no deallocation is attempted. Arrays are handled equivalently (§15.6.1). This strategy correctly handles the standard library placement *new* operator (§10.4.11), as well as any case in which the programmer has provided a matching pair of allocation and deallocation functions.

14.4.5 Resource Exhaustion [except.exhaust]

A recurring programming problem is what to do when an attempt to acquire a resource fails. For example, previously we blithely opened files (using *fopen*()) and requested memory from the free store (using operator *new*) without worrying about what happened if the file wasn't there or if we

had run out of free store. When confronted with such problems, programmers come up with two styles of solutions:

Resumption: Ask some caller to fix the problem and carry on.

Termination: Abandon the computation and return to some caller.

In the former case, a caller must be prepared to help out with resource acquisition problems in unknown pieces of code. In the latter, a caller must be prepared to cope with failure of the attempt to acquire the resource. The latter is in most cases far simpler and allows a system to maintain a better separation of levels of abstraction. Note that it is not the program that terminates when one uses the termination strategy; only an individual computation terminates. ''Termination'' is the traditional term for a strategy that returns from a ''failed'' computation to an error handler associated with a caller (which may re-try the failed computation), rather than trying to repair a bad situation and resume from the point at which the problem was detected.

In C++, the resumption model is supported by the function-call mechanism and the termination model is supported by the exception-handling mechanism. Both can be illustrated by a simple implementation and use of the standard library ***operator new()***:

```
void* operator new(size_t size)
{
    for (;;) {
        if (void* p = malloc(size)) return p;          // try to find memory
        if (_new_handler == 0) throw bad_alloc();      // no handler: give up
        _new_handler();                                // ask for help
    }
}
```

Here, I use the standard C library ***malloc()*** to do the real search for memory; other implementations of ***operator new()*** may choose other ways. If memory is found, ***operator new()*** can return a pointer to it. Otherwise, ***operator new()*** calls the ***_new_handler***. If the ***_new_handler*** can find more memory for ***malloc()*** to allocate, all is fine. If it can't, the handler cannot return to ***operator new()*** without causing an infinite loop. The ***_new_handler()*** might then choose to throw an exception, thus leaving the mess for some caller to handle:

```
void my_new_handler()
{
    int no_of_bytes_found = find_some_memory();
    if (no_of_bytes_found < min_allocation) throw bad_alloc();      // give up
}
```

Somewhere, there ought to be a *try_block* with a suitable handler:

```
try {
    // ...
}
catch (bad_alloc) {
    // somehow respond to memory exhaustion
}
```

The ***_new_handler*** used in the implementation of ***operator new()*** is a pointer to a function maintained by the standard function ***set_new_handler()***. If I want ***my_new_handler()*** to be used as

the ***_new_handler***, I say:

```
set_new_handler(&my_new_handler);
```

If I also want to catch ***bad_alloc***, I might say:

```
void f()
{
    void(*oldnh)() = set_new_handler(&my_new_handler);

    try {
        // ...
    }
    catch (bad_alloc) {
        // ...
    }
    catch (...) {
        set_new_handler(oldnh);   // re-set handler
        throw;      // re-throw
    }

    set_new_handler(oldnh);          // re-set handler
}
```

Even better, avoid the ***catch(...)*** handler by applying the ''resource acquisition is initialization'' technique described in §14.4 to the ***_new_handler*** (§14.12[1]).

With the ***_new_handler***, no extra information is passed along from where the error is detected to the helper function. It is easy to pass more information. However, the more information that is passed between the code detecting a run-time error and a function helping correct that error, the more the two pieces of code become dependent on each other. This implies that changes to the one piece of code require understanding of and maybe even changes to the other. To keep separate pieces of software separate, it is usually a good idea to minimize such dependencies. The exception-handling mechanism supports such separation better than do function calls to helper routines provided by a caller.

In general, it is wise to organize resource allocation in layers (levels of abstraction) and avoid having one layer depend on help from the layer that called it. Experience with larger systems shows that successful systems evolve in this direction.

Throwing an exception requires an object to throw. A C++ implementation is required to have enough spare memory to be able to throw ***bad_alloc*** in case of memory exhaustion. However, it is possible that throwing some other exception will cause memory exhaustion.

14.4.6 Exceptions in Constructors [except.ctor]

Exceptions provide a solution to the problem of how to report errors from a constructor. Because a constructor does not return a separate value for a caller to test, the traditional (that is, non-exception-handling) alternatives are:

[1] Return an object in a bad state, and trust the user to test the state.

[2] Set a nonlocal variable (e.g., ***errno***) to indicate that the creation failed, and trust the user to test that variable.

[3] Don't do any initialization in the constructor, and rely on the user to call an initialization function before the first use.

[4] Mark the object ''uninitialized'' and have the first member function called for the object do the real initialization, and that function can then report an error if initialization fails.

Exception handling allows the information that a construction failed to be transmitted out of the constructor. For example, a simple *Vector* class might protect itself from excessive demands on memory like this:

```
class Vector {
public:
        class Size { };

        enum { max = 32000 };

        Vector::Vector(int sz)
        {
                if (sz<0 || max<sz) throw Size();
                // ...
        }
        // ...
};
```

Code creating *Vector*s can now catch *Vector::Size* errors, and we can try to do something sensible with them:

```
Vector* f(int i)
{
        try {
                Vector* p = new Vector(i);
                // ...
                return p;
        }
        catch(Vector::Size) {
                // deal with size error
        }
}
```

As always, the error handler itself can use the standard set of fundamental techniques for error reporting and recovery. Each time an exception is passed along to a caller, the view of what went wrong changes. If suitable information is passed along in the exception, the amount of information available to deal with the problem could increase. In other words, the fundamental aim of the error-handling techniques is to pass information about an error from the original point of detection to a point where there is sufficient information available to recover from the problem, and to do so reliably and conveniently.

The ''resource acquisition is initialization'' technique is the safest and most elegant way of handling constructors that acquire more than one resource (§14.4). In essence, the technique reduces the problem of handling many resources to repeated application of the (simple) technique for handling one resource.

14.4.6.1 Exceptions and Member Initialization [except.member]

What happens if a member initializer (directly or indirectly) throws an exception? By default, the exception is passed on to whatever invoked the constructor for the member's class. However, the constructor itself can catch such exceptions by enclosing the complete function body – including the member initializer list – in a *try-block*. For example:

```
class X {
    Vector v;
    // ...
public:
    X(int);
    // ...
};

X::X(int s)
try
    :v(s)       // initialize v by s
{
    // ...
}
catch (Vector::Size) { // exceptions thrown for v are caught here
    // ...
}
```

Copy constructors (§10.4.4.1) are special in that they are invoked implicitly and because they often both acquire and release resources. In particular, the standard library assumes proper – non-exception-throwing – behavior of copy constructors. For these reasons, care should be taken that a copy constructor throws an exception only in truly disastrous circumstances. Complete recovery from an exception in a copy constructor is unlikely to be feasible in every context of its use. To be even potentially safe, a copy constructor must leave behind two objects, each of which fulfills the invariant of its class (§24.3.7.1).

Naturally, copy assignment operators should be treated with as much care as copy constructors.

14.4.7 Exceptions in Destructors [except.dtor]

From the point of view of exception handling, a destructor can be called in one of two ways:

[1] *Normal call*: As the result of a normal exit from a scope (§10.4.3), a **delete** (§10.4.5), etc.

[2] *Call during exception handling*: During stack unwinding (§14.4), the exception-handling mechanism exits a scope containing an object with a destructor.

In the latter case, an exception may not escape from the destructor itself. If it does, it is considered a failure of the exception-handling mechanism and **std::terminate()** (§14.7) is called. After all, there is no general way for the exception-handling mechanism or the destructor to determine whether it is acceptable to ignore one of the exceptions in favor of handling the other.

If a destructor calls functions that may throw exceptions, it can protect itself. For example:

```
X::~X()
try {
        f(); // might throw
}
catch (...) {
        // do something
}
```

The standard library function ***uncaught_exception***() returns ***true*** if an exception has been thrown but hasn't yet been caught. This allows the programmer to specify different actions in a destructor depending on whether an object is destroyed normally or as part of stack unwinding.

14.5 Exceptions That Are Not Errors [except.not.error]

If an exception is expected and caught so that it has no bad effects on the behavior of the program, then how can it be an error? Only because the programmer thinks of it as an error and of the exception-handling mechanisms as tools for handling errors. Alternatively, one might think of the exception-handling mechanisms as simply another control structure. For example:

```
void f(Queue<X>& q)
{
        try {
                for (;;) {
                        X m = q.get();          // throws 'Empty' if queue is empty
                        // ...
                }
        }
        catch (Queue<X>::Empty) {
                return;
        }
}
```

This actually has some charm, so it is a case in which it is not entirely clear what should be considered an error and what should not.

Exception handling is a less structured mechanism than local control structures such as ***if*** and ***for*** and is often less efficient when an exception is actually thrown. Therefore, exceptions should be used only where the more traditional control structures are inelegant or impossible to use. Note that the standard library offers a ***queue*** of arbitrary elements without using exceptions (§17.3.2).

Using exceptions as alternate returns can be an elegant technique for terminating search functions – especially highly recursive search functions such as a lookup in a tree. For example:

```
void fnd(Tree* p, const string& s)
{
        if (s == p->str) throw p;       // found s
        if (p->left) fnd(p->left,s);
        if (p->right) fnd(p->right,s);
}
```

```
Tree* find(Tree* p, const string& s)
{
    try {
        fnd(p,s);
    }
    catch (Tree* q) {    // q->str==s
        return q;
    }
    return 0;
}
```

However, such use of exceptions can easily be overused and lead to obscure code. Whenever reasonable, one should stick to the ''exception handling is error handling'' view. When this is done, code is clearly separated into two categories: ordinary code and error-handling code. This makes code more comprehensible. Unfortunately, the real world isn't so clear cut. Program organization will (and to some extent should) reflect that.

Error handling is inherently difficult. Anything that helps preserve a clear model of what is an error and how it is handled should be treasured.

14.6 Exception Specifications [except.spec]

Throwing or catching an exception affects the way a function relates to other functions. It can therefore be worthwhile to specify the set of exceptions that might be thrown as part of the function declaration. For example:

```
void f(int a) throw (x2, x3);
```

This specifies that `f()` may throw only exceptions `x2`, `x3`, and exceptions derived from these types, but no others. When a function specifies what exceptions it might throw, it effectively offers a guarantee to its callers. If during execution that function does something that tries to abrogate the guarantee, the attempt will be transformed into a call of `std::unexpected()`. The default meaning of `unexpected()` is `std::terminate()`, which in turn normally calls `abort()`; see §9.4.1.1 for details.

In effect,

```
void f() throw (x2, x3)
{
    // stuff
}
```

is equivalent to:

```
void f()
try
{
    // stuff
}
```

```
catch (x2) { throw; }      // re-throw
catch (x3) { throw; }      // re-throw
catch ( . . . ) {
      std::unexpected();   // unexpected() will not return
}
```

The most important advantage is that the function *declaration* belongs to an interface that is visible to its callers. Function *definitions*, on the other hand, are not universally available. Even when we do have access to the source code of all our libraries, we strongly prefer not to have to look at it very often. In addition, a function with an *exception-specification* is shorter and clearer than the equivalent hand-written version.

A function declared without an *exception-specification* is assumed to throw every exception. For example:

```
int f();          // can throw any exception
```

A function that will throw no exceptions can be declared with an empty list:

```
int g() throw ();     // no exception thrown
```

One might think that the default should be that a function throws no exceptions. However, that would require exception specifications for essentially every function, would be a significant cause for recompilation, and would inhibit cooperation with software written in other languages. This would encourage programmers to subvert the exception-handling mechanisms and to write spurious code to suppress exceptions. It would provide a false sense of security to people who failed to notice the subversion.

14.6.1 Checking Exception Specifications [except.check.spec]

It is not possible to catch every violation of an interface specification at compile time. However, much compile-time checking is done. The way to think about *exception-specification*s is to assume that a function *will* throw any exception it can. The rules for compile-time checking *exception-specification*s outlaw easily detected absurdities.

If any declaration of a function has an *exception-specification*, every declaration of that function (including the definition) must have an *exception-specification* with exactly the same set of exception types. For example:

```
int f() throw (std::bad_alloc);

int f()     // error: exception-specification missing
{
      // ...
}
```

Importantly, *exception-specification*s are not required to be checked exactly across compilation-unit boundaries. Naturally, an implementation can check. However, for many large and long-lived systems, it is important that the implementation does not – or, if it does, that it carefully gives hard errors only where violations will not be caught at run time.

The point is to ensure that adding an exception somewhere doesn't force a complete update of related exception specifications and a recompilation of all potentially affected code. A system can

then function in a partially updated state relying on the dynamic (run-time) detection of unexpected exceptions. This is essential for the maintenance of large systems in which major updates are expensive and not all source code is accessible.

A virtual function may be overridden only by a function that has an *exception-specification* at least as restrictive as its own (explicit or implicit) *exception-specification*. For example:

```
class B {
public:
    virtual void f();                   // can throw anything
    virtual void g() throw(X,Y);
    virtual void h() throw(X);
};

class D : public B {
public:
    void f() throw(X);                  // ok
    void g() throw(X);                  // ok: D::g() is more restrictive than B::g()
    void h() throw(X,Y);                // error: D::h() is less restrictive than B::h()
};
```

This rule is really only common sense. If a derived class threw an exception that the original function didn't advertise, a caller couldn't be expected to catch it. On the other hand, an overriding function that throws fewer exceptions clearly obeys the rule set out by the overridden function's *exception-specification*.

Similarly, you can assign a pointer to function that has a more restrictive *exception-specification* to a pointer to function that has a less restrictive *exception-specification*, but not vice versa. For example:

```
void f() throw(X);
void (*pf1)() throw(X,Y) = &f;      // ok
void (*pf2)() throw() = &f;         // error: f() is less restrictive than pf2
```

In particular, you cannot assign a pointer to a function without an *exception-specification* to a pointer to function that has one:

```
void g(); // might throw anything

void (*pf3)() throw(X) = &g;        // error: g() less restrictive than pf3
```

An *exception-specification* is not part of the type of a function and a ***typedef*** may not contain one. For example:

```
typedef void (*PF)() throw(X);      // error
```

14.6.2 Unexpected Exceptions [except.unexpected]

An *exception-specification* can lead to calls to ***unexpected***`()`. Such calls are typically undesirable except during testing. Such calls can be avoided through careful organization of exceptions and specification of interfaces. Alternatively, calls to ***unexpected***`()` can be intercepted and rendered harmless.

A well-defined subsystem Y will often have all its exceptions derived from a class ***Yerr***. For example, given

```
class Some_Yerr : public Yerr { /* ... */ };
```

a function declared

```
void f() throw (Xerr, Yerr, exception);
```

will pass any ***Yerr*** on to its caller. In particular, ***f()*** would handle a ***Some_Yerr*** by passing it on to its caller. Thus, no ***Yerr*** in ***f()*** will trigger ***unexpected()***.

All exceptions thrown by the standard library are derived from class ***exception*** (§14.10).

14.6.3 Mapping Exceptions [except.mapping]

Occasionally, the policy of terminating a program upon encountering an unexpected exception is too Draconian. In such cases, the behavior of ***unexpected()*** must be modified into something acceptable.

The simplest way of achieving that is to add the standard library exception ***std::bad_exception*** to an *exception-specification*. In that case, ***unexpected()*** will simply throw ***bad_exception*** instead of invoking a function to try to cope. For example:

```
class X { };
class Y { };

void f() throw(X, std::bad_exception)
{
    // ...
    throw Y();      // throw ''bad'' exception
}
```

The *exception-specification* will catch the unacceptable exception ***Y*** and throw an exception of type ***bad_exception*** instead.

There is actually nothing particularly bad about ***bad_exception***; it simply provides a mechanism that is less drastic than calling ***terminate()***. However, it is still rather crude. In particular, information about which exception caused the problem is lost.

14.6.3.1 User Mapping of Exceptions [except.user.mapping]

Consider a function ***g()*** written for a non-networked environment. Assume further that ***g()*** has been declared with an *exception-specification* so that it will throw only exceptions related to its ''subsystem Y:''

```
void g() throw(Yerr);
```

Now assume that we need to call ***g()*** in a networked environment.

Naturally, ***g()*** will not know about network exceptions and will invoke ***unexpected()*** when it encounters one. To use ***g()*** in a distributed environment, we must either provide code that handles network exceptions or rewrite ***g()***. Assuming a rewrite is infeasible or undesirable, we can handle the problem by redefining the meaning of ***unexpected()***.

Memory exhaustion is dealt with by the *_new_handler* determined by *set_new_handler*( ). Similarly, the response to an unexpected exception is determined by an *_unexpected_handler* set by *std*::*set_unexpected*( ) from <*exception*>:

```
typedef void(*unexpected_handler)();
unexpected_handler set_unexpected(unexpected_handler);
```

To handle unexpected exceptions well, we first define a class to allow us to use the ''resource acquisition is initialization'' technique for *unexpected*( ) functions:

```
class STC {      // store and reset class
      unexpected_handler old;
public:
      STC(unexpected_handler f) { old = set_unexpected(f); }
      ~STC( ) { set_unexpected(old); }
};
```

Then, we define a function with the meaning we want for *unexpected*( ) in this case:

```
class Yunexpected : Yerr { };

void throwY( ) throw(Yunexpected) { throw Yunexpected( ); }
```

Used as an *unexpected*( ) function, *throwY*( ) maps any unexpected exception into *Yunexpected*.

Finally, we provide a version of *g*( ) to be used in the networked environment:

```
void networked_g( ) throw(Yerr)
{
      STC xx(&throwY);    // now unexpected() throws Yunexpected
      g( );
}
```

Because *Yunexpected* is derived from *Yerr*, the *exception-specification* is not violated. Had *throwY*( ) thrown an exception that did violate the *exception-specification*, *terminate*( ) would have been called.

By saving and restoring the *_unexpected_handler*, we make it possible for several subsystems to control the handling of unexpected exceptions without interfering with each other. Basically, this technique for mapping an unexpected exception into an expected one is a more flexible variant of what the system offers in the form of *bad_exception*.

14.6.3.2 Recovering the Type of an Exception [except.recover]

Mapping unexpected exceptions to *Yunexpected* would allow a user of *networked_g*( ) to know that an unexpected exception had been mapped into *Yunexpected*. However, such a user wouldn't know which exception had been mapped. That information was lost in *throwY*( ). A simple technique allows that information to be recorded and passed on:

```
class Yunexpected : public Yerr {
public:
        Network_exception* pe;

        Yunexpected(Network_exception* p) :pe(p) { }
};

void throwY() throw(Yunexpected)
{
        try {
                throw; // re-throw to be caught immediately!
        }
        catch(Network_exception& p) {
                throw Yunexpected(&p);   // throw mapped exception
        }
        catch(...) {
                throw Yunexpected(0);
        }
}
```

Re-throwing an exception and catching it allows us to get a handle on any exception of a type we can name. The ***throwY()*** function is called from ***unexpected()***, which is conceptually called from a ***catch(...)*** handler. There therefore is definitely an exception to re-throw. It is not possible for an ***unexpected()*** function to ignore the exception and return. If it tries to, ***unexpected()*** itself will throw a ***bad_exception*** (§14.6.3).

14.7 Uncaught Exceptions [except.uncaught]

If an exception is thrown but not caught, the function ***std::terminate()*** will be called. The ***terminate()*** function will also be called when the exception-handling mechanism finds the stack corrupted and when a destructor called during stack unwinding caused by an exception tries to exit using an exception.

An unexpected exception is dealt with by the ***_unexpected_handler*** determined by ***set_unexpected()***. Similarly, the response to an uncaught exception is determined by an ***_uncaught_handler*** set by ***std::set_terminate()*** from <*exception*>:

```
typedef void(*terminate_handler)();
terminate_handler set_terminate(terminate_handler);
```

The return value is the previous function given to ***set_terminate()***.

The reason for ***terminate()*** is that exception handling must occasionally be abandoned for less subtle error-handling techniques. For example, ***terminate()*** could be used to abort a process or maybe to re-initialize a system. The intent is for ***terminate()*** to be a drastic measure to be applied when the error-recovery strategy implemented by the exception-handling mechanism has failed and it is time to go to another level of a fault tolerance strategy.

By default, ***terminate()*** will call ***abort()*** (§9.4.1.1). This default is the correct choice for most users – especially during debugging.

An *_uncaught_handler* is assumed not to return to its caller. If it tries to, *terminate*() will call *abort*().

Note that *abort*() indicates abnormal exit from the program. The function *exit*() can be used to exit a program with a return value that indicates to the surrounding system whether the exit is normal or abnormal (§9.4.1.1).

It is implementation-defined whether destructors are invoked when a program is terminated because of an uncaught exception. On some systems, it is essential that the destructors are not called so that the program can be resumed from the debugger. On other systems, it is architecturally close to impossible *not* to invoke the destructors while searching for a handler.

If you want to ensure cleanup when an uncaught exception happens, you can add a catch-all handler (§14.3.2) to *main*() in addition to handlers for exceptions you really care about. For example:

```
int main()
try {
        // ...
}
catch (std::range_error)
{
        cerr << "range error: Not again!\n";
}
catch (std::bad_alloc)
{
        cerr << "new ran out of memory\n";
}
catch (...) {
        // ...
}
```

This will catch every exception, except those thrown by construction and destruction of global variables. There is no way of catching exceptions thrown during initialization of global variables. The only way of gaining control in case of *throw* from an initializer of a nonlocal static object is *set_unexpected*() (§14.6.2). This is another reason to avoid global variables whenever possible.

When an exception is caught, the exact point where it was thrown is generally not known. This represents a loss of information compared to what a debugger might know about the state of a program. In some C++ development environments, for some programs, and for some people, it might therefore be preferable *not* to catch exceptions from which the program isn't designed to recover.

14.8 Exceptions and Efficiency [except.efficiency]

In principle, exception handling can be implemented so that there is no run-time overhead when no exception is thrown. In addition, this can be done so that throwing an exception isn't all that expensive compared to calling a function. Doing so without adding significant memory overhead while maintaining compatibility with C calling sequences, debugger conventions, etc., is possible, but hard. However, please remember that the alternatives to exceptions are not free either. It is not unusual to find traditional systems in which half of the code is devoted to error handling.

Consider a simple function *f*() that appears to have nothing to do with exception handling:

```
void g(int);

void f()
{
    string s;
    // ...
    g(1);
    g(2);
}
```

However, *g*() may throw an exception, so *f*() must contain code ensuring that *s* is destroyed correctly in case of an exception. However, had *g*() not thrown an exception it would have had to report its error some other way. Consequently, the comparable code using ordinary code to handle errors instead of exceptions isn't the plain code above, but something like:

```
bool g(int);

bool f()
{
    string s;
    // ...
    if (g(1))
        if (g(2))
            return true;
        else
            return false;
    else
        return false;
}
```

People don't usually handle errors this systematically, though, and it is not always critical to do so. However, when careful and systematic handling of errors is necessary, such housekeeping is best left to a computer, that is, to the exception-handling mechanisms.

Exception-specifications (§14.6) can be most helpful in improving generated code. Had we stated that *g*() didn't throw an exception:

```
void g(int) throw();
```

the code generation for *f*() could have been improved. It is worth observing that no traditional C function throws an exception, so in most programs every C function can be declared with the empty throw specification *throw*(). In particular, an implementation knows that only a few standard C library functions (such as *atexit*() and *qsort*()) can throw exceptions, and it can take advantage of that fact to generate better code.

Before giving a ''C function'' an empty *exception-specification*, *throw*(), take a minute to consider if it could possibly throw an exception. For example, it might have been converted to use the C++ operator *new*, which can throw *bad_alloc*, or it might call a C++ library that throws an exception.

14.9 Error-Handling Alternatives [except.alternatives]

The purpose of the exception-handling mechanisms is to provide a means for one part of a program to inform another part of a program that an ''exceptional circumstance'' has been detected. The assumption is that the two parts of the program are written independently and that the part of the program that handles the exception often can do something sensible about the error.

To use handlers effectively in a program, we need an overall strategy. That is, the various parts of the program must agree on how exceptions are used and where errors are dealt with. The exception-handling mechanisms are inherently nonlocal, so adherence to an overall strategy is essential. This implies that the error-handling strategy is best considered in the earliest phases of a design. It also implies that the strategy must be simple (relative to the complexity of the total program) and explicit. Something complicated would not be consistently adhered to in an area as inherently tricky as error recovery.

First of all, the idea that a single mechanism or technique can handle all errors must be dispelled; it would lead to complexity. Successful fault-tolerant systems are multilevel. Each level copes with as many errors as it can without getting too contorted and leaves the rest to higher levels. The notion of ***terminate***`()` is intended to support this view by providing an escape if the exception-handling mechanism itself is corrupted or if it has been incompletely used, thus leaving exceptions uncaught. Similarly, the notion of ***unexpected***`()` is intended to provide an escape when the strategy using *exception-specification*s to provide firewalls fails.

Not every function should be a firewall. In most systems, it is not feasible to write every function to do sufficient checking to ensure that it either completes successfully or fails in a well-defined manner. The reasons that this will not work varies from program to program and from programmer to programmer. However, for larger programs:

[1] The amount of work needed to ensure this notion of ''reliability'' is too great to be done consistently.

[2] The overheads in time and space are too great for the system to run acceptably (there will be a tendency to check for the same errors, such as invalid arguments, over and over again).

[3] Functions written in other languages won't obey the rules.

[4] This purely local notion of ''reliability'' leads to complexities that actually become a burden to overall system reliability.

However, separating the program into distinct subsystems that either complete successfully or fail in well-defined ways is essential, feasible, and economical. Thus, a major library, subsystem, or key function should be designed in this way. Exception specifications are intended for interfaces to such libraries and subsystems.

Usually, we don't have the luxury of designing all of the code of a system from scratch. Therefore, to impose a general error-handling strategy on all parts of a program, we must take into account program fragments implemented using strategies different from ours. To do this we must address a variety of concerns relating to the way a program fragment manages resources and the state in which it leaves the system after an error. The aim is to have the program fragment appear to follow the general error-handling strategy even if it internally follows a different strategy.

Occasionally, it is necessary to convert from one style of error reporting to another. For example, we might check ***errno*** and possibly throw an exception after a call to a C library or, conversely, catch an exception and set ***errno*** before returning to a C program from a C++ library:

```
void callC() throw(C_blewit)
{
    errno = 0;
    c_function();
    if (errno) {
        // cleanup, if possible and necessary
        throw C_blewit(errno);
    }
}

extern "C" void call_from_C() throw()
{
    try {
        c_plus_plus_function();
    }
    catch (...) {
        // cleanup, if possible and necessary
        errno = E_CPLPLFCTBLEWIT;
    }
}
```

In such cases, it is important to be systematic enough to ensure that the conversion of error reporting styles is complete.

Error handling should be – as far as possible – hierarchical. If a function detects a run-time error, it should not ask its caller for help with recovery or resource acquisition. Such requests set up cycles in the system dependencies. That in turn makes the program hard to understand and introduces the possibility of infinite loops in the error-handling and recovery code.

Simplifying techniques such as ''resource acquisition is initialization'' and simplifying assumptions such as ''exceptions represent errors'' should be used to make the error-handling code more regular. See also §24.3.7.1 for ideas about how to use invariants and assertions to make the triggering of exceptions more regular.

14.10 Standard Exceptions [except.std]

Here is a table of standard exceptions and the functions, operators, and general facilities that throw them:

Standard Exceptions (thrown by the language)			
Name	Thrown by	Reference	Header
bad_alloc	*new*	§6.2.6.2, §19.4.5	<new>
bad_cast	*dynamic_cast*	§15.4.1.1	<typeinfo>
bad_typeid	*typeid*	§15.4.4	<typeinfo>
bad_exception	*exception specification*	§14.6.3	<exception>

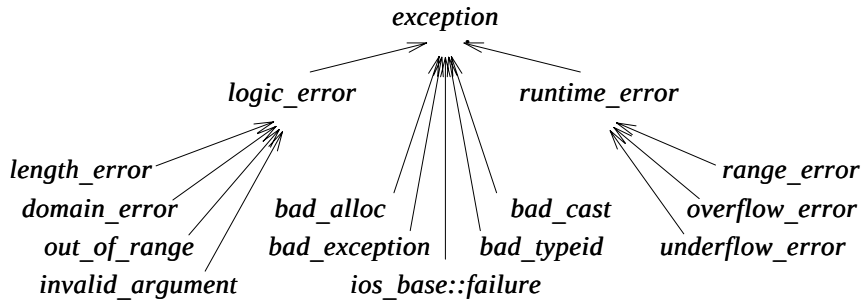
Standard Exceptions (thrown by the standard library)			
Name	Thrown by	Reference	Header
out_of_range	*at()*	§3.7.2, §16.3.3, §20.3.3	<stdexcept>
	bitset<>::operator[]()	§17.5.3	<stdexcept>
invalid_argument	*bitset constructor*	§17.5.3.1	<stdexcept>
overflow_error	*bitset<>::to_ulong()*	§17.5.3.3	<stdexcept>
ios_base::failure	*ios_base::clear()*	§21.3.6	<ios>

The library exceptions are part of a class hierarchy rooted in the standard library exception class ***exception*** presented in ***<exception>***:

```
class exception {
public:
        exception() throw();
        exception(const exception&) throw();
        exception& operator=(const exception&) throw();
        virtual ~exception() throw();

        virtual const char* what() const throw();
private:
        // ...
};
```

The hierarchy looks like this:



This seems rather elaborate for organizing the eight standard exceptions. This hierarchy attempts to provide a framework for exceptions beyond the ones defined by the standard library. Logic errors are errors that in principle could be caught either before the program starts executing or by tests of arguments to functions and constructors. Run-time errors are all other errors. Some people view this as a useful framework for all errors and exceptions; I don't.

The standard library exception classes don't add functions to the set provided by ***exception***; they simply define the required virtual functions appropriately. Thus, we can write:

```
void f()
try {
        // use standard library
}
```

```
catch (exception& e) {
        cout << "standard library exception " << e.what() << '\n';     // well, maybe
        // ...
}
catch (...) {
        cout << "other exception\n";
        // ...
}
```

The standard exceptions are derived from ***exception***. However, not every exception is, so it would be a mistake to try to catch every exception by catching ***exception***. Similarly, it would be a mistake to assume that every exception derived from ***exception*** is a standard library exception: programmers can add their own exceptions to the ***exception*** hierarchy .

Note that ***exception*** operations do not themselves throw exceptions. In particular, this implies that throwing a standard library exception doesn't cause a ***bad_alloc*** exception. The exception-handling mechanism keeps a bit of memory to itself for holding exceptions (possibly on the stack). Naturally, it is possible to write code that eventually consumes all memory in the system, thus forcing a failure.

Here is a function that – if called – tests whether the function call or the exception-handling mechanism runs out of memory first:

```
void perverted()
{
        try {
                throw exception();   // recursive exception throw
        }
        catch (exception& e) {
                perverted();         // recursive function call
                cout << e.what();
        }
}
```

The purpose of the output statement is simply to prevent the compiler from re-using the memory occupied by the exception named ***e***.

14.11 Advice [except.advice]

[1] Use exceptions for error handling; §14.1, §14.5, §14.9.
[2] Don't use exceptions where more local control structures will suffice; §14.1.
[3] Use the ''resource allocation is initialization'' technique to manage resources; §14.4.
[4] Not every program needs to be exception safe; §14.4.3.
[5] Use ''resource allocation is initialization'' and exception handlers to maintain invariants; §14.3.2.
[6] Minimize the use of *try-block*s. Use ''resource acquisition is initialization'' instead of explicit handler code; §14.4.
[7] Not every function needs to handle every possible error; §14.9.

[8] Throw an exception to indicate failure in a constructor; §14.4.6.
[9] Avoid throwing exceptions from copy constructors; §14.4.6.1.
[10] Avoid throwing exceptions from destructors; §14.4.7.
[11] Have ***main***`()` catch and report all exceptions; §14.7.
[12] Keep ordinary code and error-handling code separate; §14.4.5, §14.5.
[13] Be sure that every resource acquired in a constructor is released when throwing an exception in that constructor; §14.4.
[14] Keep resource management hierarchical; §14.4.
[15] Use *exception-specification*s for major interfaces; §14.9.
[16] Beware of memory leaks caused by memory allocated by ***new*** not being released in case of an exception; §14.4.1, §14.4.2, §14.4.4.
[17] Assume that every exception that can be thrown by a function will be thrown; §14.6.
[18] Don't assume that every exception is derived from class ***exception***; §14.10.
[19] A library shouldn't unilaterally terminate a program. Instead, throw an exception and let a caller decide; §14.1.
[20] A library shouldn't produce diagnostic output aimed at an end user. Instead, throw an exception and let a caller decide; §14.1.
[21] Develop an error-handling strategy early in a design; §14.9.

14.12 Exercises [except.exercises]

1. (∗2) Generalize the ***STC*** class (§14.6.3.1) to a template that can use the ''resource acquisition is initialization'' technique to store and reset functions of a variety of types.
2. (∗3) Complete the ***Ptr_to_T*** class from §11.11 as a template that uses exceptions to signal run-time errors.
3. (∗3) Write a function that searches a binary tree of nodes based on a ***char****** field for a match. If a node containing ***hello*** is found, ***find***`("`***hello***`")` will return a pointer to that node. Use an exception to indicate ''not found.''
4. (∗3) Define a class ***Int*** that acts exactly like the built-in type ***int***, except that it throws exceptions rather than overflowing or underflowing.
5. (∗2.5) Take the basic operations for opening, closing, reading, and writing from the C interface to your operating system and provide equivalent C++ functions that call the C functions but throw exceptions in case of errors.
6. (∗2.5) Write a complete ***Vector*** template with ***Range*** and ***Size*** exceptions.
7. (∗1) Write a loop that computes the sum of a ***Vector*** as defined in §14.12[6] without examining the size of the ***Vector***. Why is this a bad idea?
8. (∗2.5) Consider using a class ***Exception*** as the base of all classes used as exceptions. What should it look like? How should it be used? What good might it do? What disadvantages might result from a requirement to use such a class?
9. (∗1) Given a

```
int main() { /* ... */ }
```

   change it so that it catches all exceptions, turns them into error messages, and ***abort***`()`s. Hint:

***call_from_C*( )** in §14.9 doesn't quite handle all cases.

10. (∗2) Write a class or template suitable for implementing callbacks.
11. (∗2.5) Write a ***Lock*** class for some system supporting concurrency.