Exception Handling for C++

Andrew Koenig
Bjarne Stroustrup

AT&T Bell Laboratories
Murray Hill, New Jersey 07974
ark@europa.att.com
bs@research.att.com

ABSTRACT

This paper outlines a design for an exception handling mechanism for C++. It presents the reasoning behind the major design decisions and considers their implications for implementation alternatives. The mechanism is flexible, comparatively safe and easy to use, works in a mixed language execution environment, and can be implemented to run efficiently. Two implementation strategies are described in some detail.

1 Introduction

The author of a library can detect 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 primary purpose of the exception handling mechanism described here is to cope with this problem for C++ programs; other uses of what has been called exception handling in the literature are considered secondary. See the references for discussions of exception handling techniques and mechanisms.

The mechanism described is designed to handle only synchronous exceptions, such as array range checks. Asynchronous exceptions, such as keyboard interrupts, are not handled directly by this mechanism.

A guiding principle is that exceptions are rare compared to function calls and that exception handlers are rare compared to function definitions. We do not wish to use exception handling as a substitute for more conventional control structures. Further, we do not want to require the vast majority of functions to include code specifically relating to exception handling for a program to benefit from exception handling.

The mechanism described here can be implemented in several different ways. In particular, we outline a portable implementation based on C's `setjmp/longjmp` mechanism and an implementation that has no run-time costs when exceptions do *not* occur.

C++ is designed to coexist with other languages that do not support exception handling. Consequently, we rejected any ideas for exception handling that would have required all functions in a program to be written in C++.

After the presentation of the exception handling scheme we discuss its use compared to 'traditional' error handling techniques. Many details that we felt to be important, yet not essential to appreciate the fundamentals of the exception handling scheme, are placed in appendices.

This paper is written to stimulate discussion of exception handling in C++. The mechanisms described here are not part of C++ and might never become part of C++. Previous versions of these mechanisms were presented at the *C++ at Work* conference in November 1989[8] and the Usenix C++ Conference in San Francisco in April 1990[9]. The scheme described here owes much to the discussion generated by those presentations. In particular, the present scheme provides a greater degree of protection against mistakes by transferring more obligations from the programmer to the language implementation. A more formal description of the scheme can be found in reference 3.

2 An Example

Suppose that an exception called `xxii` can occur in a function `g()` called by a function `f()`. How can the programmer of `f()` gain control in that case? The wish to 'catch' the exception `xxii` when it occurs and `g()` doesn't handle it can be expressed like this:

```
int f()
{
    try {
        return g();
    }
    catch (xxii) {
        // we get here only if 'xxii' occurs
        error("g() goofed: xxii");
        return 22;
    }
}
```

The text from `catch` to the next close brace is called a *handler*† for the kind of exception named `xxii`. A single *try-block* can have handlers for several distinct exceptions; a handler marked by the ellipsis, `...`, picks up every exception not previously mentioned. For example:

```
int f()
{
    try {
        return g();
    }
    catch (xx) {
        // we get here only if 'xx' occurs
        error("g() goofed: xx");
        return 20;
    }
    catch (xxii) {
        // we get here only if 'xxii' occurs
        error("g() goofed: xxii");
        return 22;
    }
    catch (...) {
        // we get here only if an exception
        // that isn't 'xxii' or 'xx' occurs
        error("g() goofed");
        return 0;
    }
}
```

The series of handlers is rather like a `switch` statement. The handler marked `(...)` is rather like a `default`. Note, however, that there is no 'fall through' from one *handler* to another as there is from one `case` to another. An alternative and more accurate analogy is that the set of handlers looks very much like a set of overloaded functions. However, unlike a set of overloaded functions, the try clauses are checked in the sequence in which they appear.

An exception handler is associated with a try-block and is invoked whenever its exception occurs in that block or in any function called directly or indirectly from it. For example, say in the example above that `xxii` didn't actually occur in `g()` but in a function `h()` called by `g()`:

---

† The grammar for the exception handling mechanism can be found in Appendix B.

```
int g() { return h(); }

int h()
{
    throw xxii();       // make exception 'xxii' occur
}
```

The handler in `f()` would still handle the exception `xxii`.

We will use the phrase 'throwing an exception' to denote the operation of causing an exception to occur. The reason we don't use the more common phrase 'raising an exception' is that `raise()` is a C standard library function and therefore not available for our purpose. The word `signal` is similarly unavailable. Similarly, we chose `catch` in preference to `handle` because `handle` is a commonly used C identifier.

A *handler* looks a lot like a function definition. A *throw-expression* looks somewhat like both a function call and a `return` statement. We will see below that neither similarity is superficial.

Because `g()` might be written in C or some other language that does not know about C++ exceptions, a fully general implementation of the C++ exception mechanism cannot rely on decorations of the stack frame, passing of hidden arguments to functions not in C++, or other techniques that require compiler cooperation for every function called.

Once a handler has caught an exception, that exception has been dealt with and other handlers that might exist for it become irrelevant. In other words, only the active handler most recently encountered by the thread of control will be invoked. For example, here `xxii` will still be caught by the handler in `f()`:

```
int e()
{
    try {
        return f();     // f() handles xxii
    }
    catch (xxii) {          // so we will not get here
        // ...
    }
}
```

Another way to look at it is that if a statement or function handles a particular exception, then the fact that the exception has been thrown and caught is invisible in the surrounding context − unless, of course, the exception handler itself notifies something in the surrounding context that the exception occurred.

3 Naming of Exceptions

What is an exception? In other words, how does one declare an exception? What do exception names, such as `xxii` above, really identify?

Perhaps the most important characteristic of exceptions is that the code that throws them is often written independently from the code that catches them. Whatever an exception is must therefore be something that allows separately written programs to communicate.

Moreover, it should be possible to define 'groups' of exceptions and provide handlers for such groups. For example, it should be possible for a single handler to cope with all exceptions coming from a major sub-system such as the file system, the stream I/O system, the network software, and so on. In the absence of such a mechanism, systems relying of large numbers of exceptions (say, hundreds) become unmanageable.

Finally, we want to be able to pass arbitrary, user-defined information from the point where an exception is thrown to the point where it is caught.

Two suggestions have been made for C++:

[1] that an exception be a class, and

[2] that an exception be an object.

We propose a combination of both.

3.1 Exceptions as Classes

Suppose an exception is a class. That would allow us to use inheritance to group exceptions. For instance, one can imagine an `Overflow` exception that is a kind of `Matherr` exception. Inheritance is a natural way to express such a relationship and this seems to support the notion that exceptions should be classes.

Closer inspection, however, raises a question. Throwing an exception passes information from where it is thrown to where it is caught. That information must be stored somewhere. This implies that we need a run-time representation of an exception, that is, an object representing the exception. In C++, a class is a type − *not* an object. Classes do not carry data; objects do. Also, if an exception is a class, it seems natural to declare objects of that class. If the classes themselves are used to express throwing and catching of exceptions, what good are the objects?

3.2 Exceptions as Objects

If exceptions are objects, what kind of objects? The simplest possibility is to make them integers whose values identify the kind of exception. The trouble with that is that it is difficult for people working apart to choose such integer values without clashing. I may choose the unlikely number `299793` to represent my exception, but so may you and when some unlucky third person tries to use our programs together trouble starts. It would seem more sensible to use the *identity* (that is, the address) of the exception object, rather than its *value*, to denote the particular exception being thrown.

But this raises problems as well. If we use the identity of an object to 'name' an exception, which particular object do we want to catch? The object presumably identifies the kind of exception that interests us, but that exception hasn't been thrown yet, so how can the object exist? Alternatively, if what we catch is a pre-existing object, do we throw that object as well? If so, the object probably needs to be external, but how do we throw an external object without guaranteeing trouble in an environment where multiple processes share a single address space? See reference 8 for a C++ exception handling scheme based on object identities.

3.3 Exceptions as Classes and Objects

It appears, therefore that the most natural way to describe exceptions is to *throw* objects and *catch* classes. A *copy* of the object thrown is passed to the handler.

Given that, it is possible to use any type in a *handler* and any object as the argument in a *throw-expression*. For example:

```
// add with overflow checking

struct Int_overflow {   // a class for integer overflow exception handling
    const char* op;
    int operand1, operand2;
    Int_overflow(const char* p, int q, int r):
        op(p), operand1(q), operand2(r) { }
};

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

    // If we get here, either overflow has
    // been checked and will not occur, or
    // overflow is impossible because
    // x and y have opposite sign

    return x + y;
}
```

If the arguments to this `add` function are out of range, it creates an `Int_overflow` object to describe what went wrong and makes (a copy of) that object available to a corresponding *handler*. The exception

thus thrown can be caught like this:

```
main()
{
    int a, b, c;

    cin >> a >> b;
    try {
        c = add(a,b);
    }
    catch (Int_overflow e) {
        cout << "overflow " << e.op << '('
             << e.operand1 << ',' << e.operand2
             << ")\n";
    }

    // and so on
}
```

When the *handler* is entered, `e` will be a copy of the `Int_overflow` object that was created inside the `add` function to describe what went wrong.

The parallel to function calls is now almost exact: The *throw-expression* passes an `Int_overflow` object to the *handler* declared to accept objects of class `Int_overflow`. The usual initialization semantics are used to pass this argument. The type of the object thrown is used to select the handler in approximately the way the arguments of a function call is used to select an overloaded function. Again, a slightly different way of looking at the *throw-expression* is as a `return` statement where the argument is not only passed back to the caller, but is also used to select which caller to return to.

C++ supports the definition of classes and the use of objects of classes. Using a class to describe the kind of exception expected simplifies type checking between a *throw-expression* and a *handler*. Using an object to describe the exception itself makes it easy to pass information safely and efficiently between those two points. Relying of such fundamental and well-supported language concepts rather than inventing some special ‘‘exception type’’ ensures that the full power of C++ is available for the definition and use of exceptions.

4 Grouping of Exceptions

Exceptions often fall naturally into families. For example, one could imagine a `Matherr` exception that includes `Overflow`, `Underflow`, and other possible exceptions.

One way of doing this might be simply to define `Matherr` as a class whose possible values include `Overflow` and the others:

```
enum Matherr { Overflow, Underflow, Zerodivide, /* ... */ };
```

It is then possible to say

```
try {
    f();
}
catch (Matherr m) {
    switch (m) {
    case Overflow:
        // ...
    case Underflow:
        // ...
    // ...
    }
    // ...
}
```

In other contexts, C++ uses inheritance and virtual functions to avoid this kind of switch on a type field. It is possible to use inheritance similarly to describe collections of exceptions too. For example:

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

Such classification is particularly convenient because of the desire to handle similarly any member of a collection of exceptions: it is easy to imagine many reasons why one might want to recover from any kind of `Matherr` without caring precisely which kind it was. Using inheritance in this way makes it possible to say

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

Here an `Overflow` is handled specifically, and all other `Matherr` exceptions will be handled by the general case. Of course, a program that says `catch(Matherr)` will not know what kind of `Matherr` it has caught; whatever it was, its copy will be a `Matherr` by the time the handler is entered. We will see later how a program can cater to a potentially unknown set of exceptions by catching a *reference* to an exception.

There are cases where an exception can reasonably be considered to belong to two or more groups. For example:

```
class network_file_err : public network_err , public file_system_err { };
```

Obviously, this too is easily handled without language extensions.

4.1 Naming the Current Exception

We name the current exception analogously to naming the argument of a function:

```
try { /* ... */ } catch (zaq e) { /* ... */ }
```

This is a declaration of `e` with its scope identical to the body of the exception handler. The idea is that `e` is created using the copy constructor of class `zaq` before entering the handler.

Where there is no need to identify the particular exception, there is no need to name the exception object:

```
try { /* ... */ } catch (zaq) { /* ... */ }
```

This again is similar to the technique of not naming unused function arguments in a function definition.

As with functions, the argument declaration `(...)` accepts any call, so that

```
catch (...) { /* any exception will get here */ }
```

will catch every exception. There is no way of finding out what exception caused entry into such a handler. Were such a way provided, type safety would be compromised and the implementation of exception handling complicated. When a handler is done with an exception, it may sometimes want to re-throw that exception, whatever it was, without caring what kind of exception it was. To do this, say `throw` without specifying an exception:

```
catch (...) {
    // do something
    throw;      // re-throw current exception
}
```

Such a re-throw operation is valid only within a handler or within a function called directly or indirectly from a handler.

Because of things like `(...)` clauses, this kind of *throw-expression* offers a facility that cannot be simulated by other means. While this facility is sometimes essential, we will in section 6 below demonstrate how most problems typically solved by a re-throw operation can be solved more elegantly by careful use of destructors.

Note that

```
try { /* ... */ } catch (zaq e) { throw zaq; }
```

is *not* equivalent to

```
try { /* ... */ } catch (zaq e) { throw; }
```

In the first example, the usual rules of initialization ensures that if the exception originally thrown was an object of a class derived from `zaq`, then it will be ''cut down'' to a `zaq` when used to initialize `e`. In the second example, the original exception − possible of a class derived from `zaq` − is re-thrown.

4.2 Order of Matching

The *handler*s of a *try-block* are tried in order. For example:

```
try {
    // ...
}
catch (ibuf) {
    // handle input buffer overflow
}
catch (io) {
    // handle any io error
}
catch (stdlib) {
    // handle any library exception
}
catch (...) {
    // handle any other exception
}
```

The type in a catch clause matches if either it directly refers to the exception thrown or is of a base class of that exception, or if the exception thrown is a pointer and the exception caught is a pointer to a base class of that exception.

Since the compiler knows the class hierarchy it can warn about sillinesses such as a `(...)` clause that is not the last clause or a clause for a base class preceding a clause for a class derived from it. In both cases the later clause(s) could never be invoked because they were masked.

4.3 No Match

Where no match is found, the search for a handler continues to the calling function. When a handler is found and entered, the exception is no longer considered thrown. After dropping off the end of a *handler* the computation continues after the complete *try-block*. For example,

```
void f()
{   try {
        // ...
    }
    catch (x) {
        // handle exception 'x'
    }
    catch (...) {
        // do nothing
    }

    // whatever happens (short of hardware failure
    // terminate(), abort(), etc.) we'll get here

    h();
}
```

If the try-block raises exception `x`, the first handler will be executed. Any exception other than `x` will execute the second, empty handler. If execution continues at all, it will continue with the call to `h()`.

4.4 Exception Types

Any pointer type or type with a public copy constructor can be used to name an exception in a *handler*. When an exception of a derived class is thrown directly, the use of the copy constructor implies that the exception thrown is 'cut down' to the exception caught. For example:

```
class Matherr { /* .... */ };

class Int_overflow: public Matherr {
public:
    char* op;
    int operand1, operand2;
    // ...
};

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`.

For some applications, it may be useful to use virtual functions to deal with exceptions whose type is not statically known at the point where the exception is caught. One way to do that is to use references to exceptions:

```
void g()
{
    if ( /* ... */ ) throw Int_overflow( /* arguments */ );
}
```

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

In order for this to work, the `Int_overflow` object being thrown must remain in existence until after the corresponding handler has exited. This implies that throwing an exception to a handler that expects a reference might require allocation of memory on the heap. It is therefore unwise to use references to exceptions to deal with memory exhaustion.

The only conversions applied to exception objects are the derived class to base class conversions (for objects, references, and pointers) and conversions that add `const` or `volatile` to the subjects of pointers or references. Other standard conversions, even ones like `int` to `long`, are not applied.

4.5 Access to Local Variables

Since a *throw-expression* is an exit from a block and typically also a return from one or more functions, one cannot pass information stored in local variables in the block where the `throw` occurs − or in automatic (local) variables in functions returned from by the `throw`. For example:

```
void f()
{
    try {
        g();
    }
    catch (int* p) {
        if (*p == 0) // ...
    }
}

void g()
{
    int a = 0;
    throw &a;   // unwise
}
```

This is logically equivalent to returning a pointer to a local variable from a function using the normal function return mechanism.

5 Handling of Destructors

Consider what to do when an exception is thrown and we need to pass ‘up’ through a function or block to find a handler. This is commonly called ‘stack unwinding.’ If a local object has a destructor, that destructor must be called as part of the stack unwinding. This can be done by implicitly providing a handler that calls the destructor when an exception is thrown after construction but before destruction of the local object. Such destructor calls are done before entry into a handler. For example, if classes `A` and `B` have destructors,

```
void f()
{
    A a;
    B b;
    g();
}
```

might be implemented in the following equivalent way:

```
void f()    // pseudo code
{
    try {
        A a;
        try {
            B b;
            try {
                g();
            }
            catch (...) {
                        // destroy ‘b’
                        // destroy ‘a’

                throw;
            }

        }
        catch (...) {
            // exception occurred during
            // the construction of ‘b’

                    // destroy ‘a’

            throw;
        }

    }
    catch (...) {
        // exception occurred during
        // the construction of ‘a’

        throw;
    }
    // destroy ‘b’
    // destroy ‘a’
}
```

This looks horrendously complicated and expensive. Fortunately, a user will never have to write it, and in fact, there is an implementation technique that (in addition to the standard overhead of establishing a handler) involves only minimal overhead. Here is an example of C code that might be generated for the example above:

```
void f()    // generated code
{
    int __counter = 0;
    struct A a;
    struct B b;

    A::A(&a);
    __counter++;        /* one object created */

    B::B(&b);
    __counter++;        /* two objects created */

    g();

    __counter = 3;      /* no exception occurred */
```

```
__catch:
    switch (__counter) {
    case 3:
    case 2: b.B::~B();  /* fall through */
    case 1: a.A::~A();
    }

    if (__counter == 3) return;
    throw;          /* or whatever the corresponding C is */
}
```

For this to work we assume two things: a way of getting to the label `__catch` in case an exception is thrown in `g()`, and a way of re-throwing the exception at the end of `f()` if an exception caused transfer of control to `__catch`.

The problem becomes somewhat more complicated in the face of arrays of objects with destructors, but we do not believe that dealing with these complications is any more than tedious.

5.1 Suppressing Destructor Calls

It has been observed that for many objects destruction is irrelevant once an exception is thrown and that an exception may be thrown precisely under conditions where it would be dangerous to run the destructor for certain objects. One could therefore imagine facilities for specifying that objects of certain classes should not be destroyed during the stack unwinding caused by exception handling. Such facilities could lead to faster and safer exception handling and more compact code.

Nevertheless, lack of experience argues that we should not immediately include specific facilities for that. Where necessary, the user can simulate the effect of such facilities by inserting suitable checks in destructors. Should experience show that such code proliferates, and especially if experience shows such code to be baroque, we should revisit the question of a facility for selectively suppressing destructor calls.

5.2 Throwing Exceptions in Destructors

Naturally, a destructor can throw an exception. In fact, since destructors often call other functions that may throw exceptions it cannot be avoided. Throwing exceptions during a ‘‘normal’’ invocation of a destructor causes no special problems. Throwing an exception during the execution of a destructor that was called as part of the stack unwinding caused by an exception being thrown may be another matter. This should be allowed to avoid putting artificial constraints on the writing of destructors.

Only one restriction is needed: If a destructor invoked during stack unwinding throws an exception, directly or indirectly, that propagates uncaught back through that destructor itself, the exception will be turned into a call of `terminate()`. The default meaning of `terminate()` is `abort()`; see Appendix G. Without this restriction the implementation of exception handling would be needlessly complicated and it would not be at all clear what exception should be considered thrown: the original exception or the one thrown by the destructor? We see no general rule for resolving that question.

6 Handling of Constructors

An object is not considered constructed until its constructor has completed. 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.

A well written constructor should ensure that its object is completely and correctly constructed. Failing that, the constructor should restore the state of the system after failure to what it was before creation. It would be ideal for naively written constructors always to achieve one of these alternatives and not leave their objects in some ‘half-constructed’ state.

Consider a class `X` for which a constructor needs to acquire two resources `x` and `y`. This acquisition might fail and throw an exception. Without imposing a burden of complexity on the programmer, the class `X` constructor must never return having acquired resource `x` but not resource `y`.

We can use the technique we use to handle local objects. The trick is to use objects of two classes `A` and `B` to represent the acquired resources (naturally a single class would be sufficient if the resources `x` and `y` are of the same kind). The acquisition of a resource is represented by the initialization of the local object

that represents the resource:

```
class X {
    A a;
    B b;
    // ...
    X()
        : a(x), // acquire 'x'
        b(y)    // acquire 'y'
    {}
    // ...
};
```

Now, as in the local object case, the implementation can take care of all the bookkeeping. The user doesn't have to keep track at all.

This implies that where this simple model for acquisition of resources (a resource is acquired by initialization of a local object that represents it) is adhered to, all is well and − importantly − the constructor author need not write explicit exception handling code.

Here is a case that does not adhere to this simple model:

```
class X {
    static int no_of_Xs;
    widget* p;
    // ...
    X() { no_of_Xs++; p = grab(); }
    ~X() { no_of_Xs--; release(p); }
};
```

What happens if `grab()` throws some exception so that the assignment to `p` never happens for some object? The rule guaranteeing that an object is only destroyed automatically provided it has been completely constructed ensures that the destructor isn't called − thus saving us from the `release()` of a random pointer value − but the `no_of_Xs` count will be wrong. It will be the programmer's job to avoid such problems. In this case the fix is easy: simply don't increase `no_of_Xs` until the possibility of an exception has passed:

```
X::X() { p = grab(); no_of_Xs++; }
```

Alternatively, a more general and elaborate technique could be used:

```
X::X()
{
    no_of_Xs++;
    try {
        p = grab();
    }
    catch (...) {
        no_of_Xs--;
        throw;       // re-throw
    }
}
```

There is no doubt that writing code for constructors and destructors so that exception handling does not cause 'half constructed' objects to be destroyed is an error-prone aspect of C++ exception handling. The 'resource acquisition is initialization' technique can help making the task manageable. It should also be noted that the C++ default memory allocation strategy guarantees that the code for a constructor will never be called if `operator new` fails to provide memory for an object so that a user need not worry that constructor (or destructor) code might be executed for a non-existent object.

7 Termination vs. Resumption

Should it be possible for an exception handler to decide to resume execution from the point of the exception? Some languages, such as PL/I and Mesa, say yes and reference 11 contains a concise argument for resumption in the context of C++. However, we feel it is a bad idea and the designers of Clu, Modula2+, Modula3, and ML agree.

First, if an exception handler can return, that means that a program that throws an exception must assume that it will get control back. Thus, in a context like this:

```
if (something_wrong) throw zxc();
```

it would not be possible to be assured that `something_wrong` is false in the code following the test because the `zxc` handler might resume from the point of the exception. What such resumption provides is a contorted, non-obvious, and error-prone form of co-routines. With resumption possible, throwing an exception ceases to be a reliable way of escaping from a context.

This problem might be alleviated if the exception itself or the throw operation determined whether to resume or not. Leaving the decision to the handler, however, seems to make programming trickier than necessary because the handler knows nothing about the context of the throw-point. However, leaving the resumption/termination decision to the throw-point is equivalent to providing a separate mechanism for termination and resumption − and that is exactly what we propose.

Exception handling implies termination; resumption can be achieved through ordinary function calls. For example:

```
void problem_X_handler(arguments)   // pseudo code
{
    // ...
    if (we_cannot_recover) throw X("Oops!");
}
```

Here, a function simulates an exception that may or may not resume. We have our doubts about the wisdom of using *any* strategy that relies on conditional resumption, but it is achievable through ordinary language mechanisms provided there is cooperation between the program that throws the exception and the program that handles it. Such cooperation seems to be a prerequisite for resumption. Pointers to such ‘‘error handling’’ functions provide yet another degree of freedom in the design of error handling schemes based of the termination-oriented exception handling scheme presented here.

Resumption is of only limited use without some means of correcting the situation that led to the exception. However, correcting error conditions after they occur is generally much harder than detecting them beforehand. Consider, for example, an exception handler trying to correct an error condition by changing some state variable. Code executed in the function call chain that led from the block with the handler to the function that threw the exception might have made decisions that depended on that state variable. This would leave the program in a state that was impossible without exception handling; that is, we would have introduced a brand new kind of bug that is very nasty and hard to find. In general, it is much safer to re-try the operation that failed from the exception handler than to resume the operation at the point where the exception was thrown.

If it were possible to resume execution from an exception handler, that would force the unwinding of the stack to be deferred until the exception handler exits. If the handler had access to the local variables of its surrounding scope without unwinding the stack first we would have introduced an equivalent to nested functions. This would complicate either the implementation/semantics or the writing of handlers.

It was also observed that many of the ‘‘potential exceptions’’ that people wanted to handle using a resumption-style exception handling mechanism where asynchronous exceptions, such as a user hitting a break key. However, handling those would imply the complications of the exception handling mechanism as discussed in section 9.

Use of a debugger can be seen as constituting a special form of resumption: Where a debugger is activated because an exception has been thrown but not caught, the user expects to find the program in a state where the local variables at the throw-point are accessible. After inspecting and possibly modifying the state of the program, the programmer might want to resume. This special case of resumption can be supported with minor overhead in the case where a debugger is active (only). Modifications of the program’s behavior in this case are of the same kind as any other modifications of a program’s state using a debugger;

they are extra-linguistic and defined as part of a programming environment rather than as part of the language.

8 Safety and Interfaces

Throwing or catching an exception affects the way a function relates to other functions. It is therefore worth considering if such things should affect the type of a function. One could require the set of exceptions that might be thrown to be declared as part of the type of a function. For example:

```
void f(int a) throw (x2, x3, x4);
```

This would improve static analysis of a program and reduce the number of programmer errors[5,6]. In particular, one could ensure that no exception could be thrown unless there were a matching handler for it.

8.1 Static Checking

Compile and link time enforcement of such rules is ideal in the sense that important classes of nasty run-time errors are completely eliminated (turned into compile time errors). Such enforcement does not carry any run-time costs − one would expect a program using such techniques to be smaller and run faster than the less safe traditional alternative of relying on run-time checking. Unfortunately, such enforcement also has nasty consequences.

Suppose a function `f()` can throw an exception `e`:

```
void f() throw (e)
{
    // ...
}
```

For static checking to work, it must be an error for a function `g()` to call `f()` unless either `g()` is also declared to throw `e` or `g()` protects itself against `f()` throwing `e`:

```
void g() throw (h)
{
    f();        // error: f() might throw e

    try {
        f();    // OK: if f() throws e, g() will throw h
    }
    catch (e) {
        throw h;
    }
}
```

To allow such checking *every* function must be decorated with the exceptions it might throw. This would be too much of a nuisance for most people. For example:

```
#include <stream.h>

main()
{
    cout << "Hello world\n"; // error: might throw an exception!
}
```

Preventing such compile-time errors would be a great bother to the programmer and the checking would consequently be subverted. Such subversion could take may forms. For example, calls through C functions could be used to render the static checking meaningless, programmers could declare functions to throw all exceptions by default, and tools could be written to automatically create lists of thrown exceptions. The effect in all cases would be to eliminate both the bother and the safety checks that were the purpose of the mechanism.

Consequently, we think the only way to make static checking practical would be to say that undecorated functions can potentially throw any exception at all. Unfortunately, that again puts an intolerable burden on the programmer who wishes to write a function that is restricted to a particular set of exceptions. That programmer would have to guard *every* call to *every* unrestricted function − and since functions are

unrestricted by default, that would make restricted functions too painful to write in practice. For instance, *every* function that did not expect to throw I/O exceptions would have to enumerate every possible I/O exception.

8.2 Dynamic checking

A more attractive and practical possibility is to allow the programmer to specify what exceptions a function can potentially throw, but to enforce this restriction at run time only. When a function says something about its exceptions, it is effectively making a guarantee to its caller; if during execution that function does something that tries to abrogate the guarantee, the attempt will be transformed into a call of `unexpected()`. The default meaning of `unexpected()` is `terminate()`, which in turn normally calls `abort()`; see Appendix G for details.

In effect, writing this:

```
void f() throw (e1, e2)
{
    // stuff
}
```

is equivalent to writing this:

```
void f()
{
    try {
        // stuff
    }
    catch (e1) {
        throw;  // re-throw
    }
    catch (e2) {
        throw;  // re-throw
    }
    catch (...) {
        unexpected();
    }
}
```

The advantage of the explicit declaration of exceptions that a function can throw over the equivalent checking in the code is not just that it saves typing. 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 and even if we do have access to the source code of all our libraries we strongly prefer not to have to look at it very often.

Another advantage is that it may still be practical to detect many uncaught exceptions during compilation. For example, if the first `f()` above called a function that could throw something other than `e1` or `e2`, the compiler could warn about it. Of course the compiler has to be careful about such warnings. For example, if every integer addition can potentially throw `Int_overflow`, it is important to suppress warnings about such common operations. Programmers do not appreciate being showered by spurious warnings.

Detecting likely uncaught exceptions may be a prime job for the mythical `lint++` that people have been looking for an excuse to design. Here is a task that requires massive static analysis of a complete program and does not fit well with the traditional C and C++ separate compilation model.

8.3 Type Checking of Exceptions

If a list of exceptions is specified for a function, it behaves as part of the function's type in the sense that the lists must match in the declaration and definition. Like the return type, the exception specification does not take part in function matching for overloaded functions.

A function declared without an exception specification is assumed to throw every exception.

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

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

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

The list of exceptions could be part of the function's signature so that type-safe linkage could be achieved. This, however, would require sophisticated linker support so the exception specification is better left unchecked on systems with traditional linker (just as the return type is left unchecked). Simply forcing a re-write and recompilation whenever an exception specification changed would cause the users to suffer the worst implications of strict static checking.

For example, a function must potentially be changed and recompiled if a function it calls (directly or indirectly) changes the set of exceptions it catches or throws. This could lead to major delays in the production of software produced (partly) by composition of libraries from different sources. Such libraries would *de facto* have to agree on a set of exceptions to be used. For example, if sub-system X handles exceptions from sub-system Y and the supplier of Y introduces a new kind of exception, then X's code will have to be modified to cope. A user of X and Y will not be able to upgrade to a new version of Y until X has been modified. Where many sub-systems are used this can cause cascading delays. Even where the 'multiple supplier problem' does not exist this can lead to cascading modifications of code and to large amounts of re-compilation.

Such problems would cause people to avoid using the exception specification mechanism or else subvert it. To avoid such problems, either the exceptions must not be part of a function's signature or else the linker must be smart enough to allow calls to functions with signatures that differ from the expected in their exception specification only.

An equivalent problem occurs when dynamic checking is used. In that case, however, the problem can be handled using the exception grouping mechanism presented in section 4. A naive use of the exception handling mechanism would leave a new exception added to sub-system Y uncaught or converted into a call to `unexpected()` by some explicitly called interface. However, a well defined sub-system Y would have all its exceptions derived from a class `Yexception`. For example

```
class newYexception : public Yexception { /* ... */ };
```

This implies that a function declared

```
void f() throw (Xexception, Yexception, IOexception);
```

would handle a `newYexception` by passing it to callers of `f()`.

Exception handling mechanisms descending from the Clu mechanism tend to provide a distinguished *failure* exception that is raised (that is, thrown) when an unexpected exception occurs. In that case, the scheme presented here calls `unexpected()`. This is a more serious response to an unexpected exception since a call of `unexpected()` cannot be caught the way an exception can and is intended to trigger a program's response to a catastrophic failure. This choice partly reflects the observation that in a mixed language environment it is not possible to intercept all exceptional circumstances so it would be dangerous to encourage programmers to believe they can. For example, a C function may call the standard C library function `abort()` and thus bypass attempts to intercept all returns from a call to a subsystem. Having a distinguished failure exception also allows easy subversion of interfaces by providing a single catch-all for (almost) every kind of unexpected event. Stating that a function raises only a specific set of exceptions becomes less meaningful because one must always remember ''and failure'' − and failure can be caught elsewhere leaving a program running ''correctly'' after the explicit assumption of an interface specification has been violated. In other words, the motivation for not having a failure exception is that in our model, raising it would always indicate a design error and such design errors should cause immediate termination whenever possible.

The grouping mechanism provided for C++ allows programmers to use separate exception classes for major sub-systems, such as basic I/O systems, window managers, networks, data base systems, etc. By deriving the individual exceptions thrown by such a sub-system from the sub-system's ''main exception,'' that main exception becomes the equivalent to a failure exception for that sub-system. This technique promotes more precise and explicit specification of exceptions compared with the unique distinguished failure exception technique. Having a universal base class for exceptions, which one might call `failure`, would simulate the Clu technique and is consequently not recommended.

Occasionally, the policy of terminating a program upon encountering an unexpected exception is too Draconian. For example, consider calling a function `g()` written for a non-networked environment in a

distributed system. Naturally, `g()` will not know about network exceptions and will call `unexpected()` when it encounters one. To use `g()` in a distributed environment we must provide code that handles network exceptions − or rewrite `g()`. Assuming a re-write is infeasible or undesirable we can handle the problem by redefining the meaning of `unexpected()`. The function `set_unexpected()` can be used to achieve that. For example:

```
void rethrow() { throw; }

void networked_g()
{
    PFV old = set_unexpected(&rethrow);
    // now unexpected() calls rethrow()

    try {
        void g();
    }
    catch (network_exception) {
        set_unexpected(old);
        // recover
    }
    catch (...) {
        set_unexpected(old);
        unexpected();
    }
}
```

See Appendix G for details and for a more elegant way of setting and restoring `unexpected()`.

9 Asynchronous Events

Can exceptions be used to handle things like signals? Almost certainly not in most C environments. The trouble is that C uses functions like `malloc` that are not re-entrant. If an interrupt occurs in the middle of `malloc` and causes an exception, there is no way to prevent the exception handler from executing `malloc` again.

A C++ implementation where calling sequences and the entire run-time library are designed around the requirement for reentrancy would make it possible for signals to throw exceptions. Until such implementations are commonplace, if ever, we must recommend that exceptions and signals be kept strictly separate from a language point of view. In many cases, it will be reasonable to have signals and exceptions interact by having signals store away information that is regularly examined (polled) by some function that in turn may throw appropriate exceptions in response to the information stored by the signals.

10 Concurrency

One common use of C++ is to emulate concurrency, typically for simulation or similar applications. The AT&T task library is one example of such concurrency simulation[16]. The question naturally arises: how do exceptions interact with such concurrency?

The most sensible answer seems to be that unwinding the stack must stop at the point where the stack forks. That is, an exception must be caught in the process in which it was thrown. If it is not, an exception needs to be thrown in the parent process. An exception handled in the function that is used to create new processes can arrange that by providing a suitable exception handler.

The exception handling scheme presented here is easily implemented to pass information from the throw-point to handlers in a way that works correctly in a concurrent system; see Appendix A for details. The ‘resource acquisition is initialization’ technique mentioned in section 6 can be used to manage locks.

11 C Compatibility

Since ANSI C does not provide an exception handling mechanism, there is no issue of C compatibility in the traditional sense. However, a convenient interface to the C++ exception handling mechanism from C can be provided. This would allow C programmers to share at least some of the benefits of a C++ exception handling implementation and would improve mixed C/C++ systems. In 'plain' ANSI C this would require tricky programming using the functions implementing the C++ exception handling mechanism directly. This would, however, still be simpler and better than most current C error handling strategies. Alternatively, C could be extended with a `try` statement for convenience, but if one were going that far there would be little reason not to switch completely to C++.

12 Standard Exceptions

A set of exceptions will eventually be standard for all implementations. In particular, one would expect to have standard exceptions for arithmetic operations, out-of-range memory access, memory exhaustion, etc. Libraries will also provide exceptions. At this point we do not have a suggested list of standard exceptions such as one would expect to find in a reference manual.

This paper describes a set of language mechanisms for implementing error handling strategies − not a complete strategy.

13 So How Can We Use Exceptions?

The purpose of the exception handling mechanism 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 typically written independently and that the part of the program that handles the exception often can do something sensible about it.

What kind of code could one reasonably expect to find in an exception handler? Here are some examples:

```
int f(int arg)
{
    try {
        g(arg);
    }
    catch (x1) {
        // fix something and retry:
        g(arg);
    }

    catch (x2) {
        // calculate and return a result:
        return 2;
    }

    catch (x3) {
        // pass the bug
        throw;
    }

    catch (x4) {
        // turn x4 into some other exception
        throw xxii;
    }

    catch (x5) {
        // fix up and carry on with next statement
    }
```

```
        catch (...) {
            // give up:
            terminate();
        }
        // ...
    }
```

Does this actually make error handling easier than 'traditional techniques?' It is hard to *know* what works without first trying this exact scheme, so we can only conjecture. The exception handling scheme presented here is synthesized from schemes found in other languages and from experiences with C++, but learning from other people's mistakes (and successes) is not easy and what works in one language and for a given set of applications may not work in another language for different range of applications.

Consider what to do when an error is detected deep in a library. Examples could be an array index range error, an attempt to open a non-existent file for reading, falling off the stack of a process class, or trying to allocate a block of memory when there is no more memory to allocate. In a language like C or C++ without exceptions there are only few basic approaches (with apparently infinite variations). The library can

[1] Terminate the program.

[2] Return a value representing 'error.'

[3] Return a legal value and leave the program in an illegal state.

[4] Call a function supplied to be called in case of 'error.'

What can one do with exceptions that cannot be achieved with these techniques? Or rather − since anything can be fudged given sufficient time and effort − what desirable error handling techniques become easier to write and less error-prone when exceptions are used?

One can consider throwing an exception as logically equivalent to case [1]: 'terminate the program' with the proviso that if some caller of the program thinks it knows better it can intercept the 'terminate order' and try to recover. The *default* result of throwing an exception is exactly that of termination (or entering the debugger on systems where that is the default response to encountering a run-time error).

Exception handling is not really meant to deal with 'errors' that can be handled by [4] 'call an error function.' Here, a relation between the caller and the library function is already established to the point where resumption of the program is possible at the point where the error was detected. It follows that exception handling is not even as powerful as the function call approach. However, should the 'error function' find itself unable to do anything to allow resumption then we are back to cases [1], [2], or [3] where exceptions may be of use.

Case [2], 'returning an error value,' such as `0` instead of a valid pointer, `NaN` (not a number, as in IEEE floating point) from a mathematical function, or an object representing an error state, implies that the caller will test for that value and take appropriate action when it is returned. Experience shows that

[1] There are often several levels of function calls between the point of error and a caller that knows enough to handle the error, and

[2] it is typically necessary to test for the error value at most intermediate levels to avoid consequential errors and to avoid the 'error value' simply being ignored and not passed further up the call chain. Even if error values such as `NaN` can be propagated smoothly, it can be hard to find out what went wrong when a complicated computation produces `NaN` as its ultimate result.

Where this is the case one of two things happens:

[1] Sufficient checking is done and the code becomes an unreadable maze of tests for error values, or

[2] insufficient checking is done and the program is left in an inconsistent state for some other function to detect.

Clearly, there are many cases where the complexity of checking error values is manageable. In those cases, returning error values is the ideal technique; in the rest, exception handling can be used to ensure that [2], 'insufficient checking,' will not occur and that the complexity induced by the need for error handling is minimized through the use of a standard syntax for identifying the error handling code and through a control structure that directly supports the notion of error handling. Using exception handling instead of 'random logic' is an improvement similar to using explicit loop-statements rather than gotos; in both cases benefits only occur when the 'higher level' construct is used in a sensible and systematic manner. However, in both cases that is easier to achieve than a sensible and systematic use of the 'lower level' construct.

Case [3], 'return a legal value leaving the program in an illegal state,' such setting the global variable

`errno` in a C program to signal that one of the standard math library functions could not compute a valid result relies on two assumptions

[1] that the legal value returned will allow the program to proceed without subsequent errors or extra coding, and

[2] someone will eventually test for the illegal state and take appropriate action.

This approach avoids adding complex error handling tests and code to every function, but suffers from at least four problems:

[1] Programmers often forget to test for the error state.

[2] Subsequent errors sometimes happen before the execution gets back to the test for the error condition.

[3] Independent errors modify the error state (e.g. overwrite `errno`) so that the nature of the problem encountered becomes confused before getting back to the error test. This is particularly nasty where several threads of control are used.

[4] Separate libraries assign the same values (error states) to designate different errors, thus totally confusing testing and handling of errors.

Another way of describing the exception handling scheme is as a formalization of this way of handling errors. There is a standard way of signaling an error that ensures that two different errors cannot be given the same 'error value,' ensures that if someone forgets to handle an error the program terminates (in whichever way is deemed the appropriate default way), and (since the ordinary execution path is abandoned after an exception is thrown) all subsequent errors and confusion will occur in exception handling code − most of which will be written specifically to avoid such messes.

So, does the exception handling mechanism solve our error handling problems? No, it is only a mechanism. Does the exception handling mechanism provide a radically new way of dealing with errors? No, it simply provides a formal and explicit way of applying the standard techniques. The exception handling mechanism

[1] Makes it easier to adhere to the best practices.

[2] Gives error handling a more regular style.

[3] Makes error handling code more readable.

[4] Makes error handling code more amenable to tools.

The net effect is to make error handling less error-prone in software written by combining relatively independent parts.

One aspect of the exception handling scheme that will appear novel to C 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 on and hope for the best. Thus exception handling makes programs more 'brittle' in the sense that more care and effort will have to be taken to get a program to run acceptably. This seems far preferable, though, to getting wrong results later in the development process (or after the development process was considered complete and the program handed over to innocent users).

The exception handling mechanism can be seen as a run-time analog to the C++ type checking and ambiguity control mechanisms. It makes the design process more important and the work involved in getting a program to compile harder than for C while providing a much better chance that the resulting program will run as expected, will be able to run as an acceptable part of a larger program, will be comprehensible to other programmers, and 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 could only be practiced informally and incompletely in languages such as C.

It should be recognized that error handling will remain a difficult task and that the exception handling mechanism while far more formalized than the techniques it replaces, still is relatively unstructured compared with language features involving only local control flow.

For example, exceptions can be used as a way of exiting from loops:

```
void f()
{
    class loop_exit { };

    // ...

    try {
        while (g()) {
            // ...
            if (I_want_to_get_out) throw loop_exit();
            // ...
        }
    }
    catch (loop_exit) {
        // come here on 'exceptional' exit from loop
    }

    // ...
}
```

We don't recommend this technique because it violates the principle that exceptions should be exceptional: there is nothing exceptional about exiting a loop. The example above simply shows an obscure way of spelling `goto`.

14 Conclusions

The exception handling scheme described here is flexible enough to cope with most synchronous exceptional circumstances. Its semantics are independent of machine details and can be implemented in several ways optimized for different aspects. In particular, portable and run-time efficient implementations are both possible. The exception handling scheme presented here should make error handling easier and less error-prone.

15 Acknowledgements

Michael Jones from CMU helped crystallize early thoughts about exception handling in C++ and demonstrate the power of some of the basic exception handling implementation techniques originating in the Clu and Modula2+ projects. Jim Mitchell contributed observations about the problems with the Cedar/Mesa and Modula exception handling mechanisms. Bill Joy contributed observations about interactions between exception handling mechanisms and debuggers. Dan Weinreb provided the 'network file system exception' example.

Jerry Schwarz and Jonathan Shopiro contributed greatly to the discussion and development of these ideas. Doug McIlroy pointed out many deficiencies in earlier versions of this scheme and contributed a healthy amount of experience with and skepticism about exception handling schemes in general. Dave Jordan, Jonathan Shopiro, and Griff Smith contributed useful comments while reviewing earlier drafts of this paper.

Discussions with several writers of optimizing C compilers were essential.

A discussion in the `c.plus.plus/new.syntax` conference of the BIX bulletin board helped us improve the presentation of the exception handling scheme.

The transformation of the exception handling scheme presented at the *C++ at Work* conference into the scheme presented here owes much to comments by and discussions with Toby Bloom, Dag Brück, Peter Deutsch, Keith Gorlen, Mike Powell, and Mike Tiemann. The influence from ML is obvious.

We are also grateful to United Airlines for the period of enforced airborne idleness between Newark and Albuquerque that gave us the time to hatch some of these notions.

16 References

[1] L. Cardelli, J. Donahue, L. Glassman, M. Jordan, B. Kalsow, G. Nelson: *Modula-3 Report*. DEC Systems Research Center. August 1988.

[2] Flaviu Cristian: *Exception Handling*. in *Dependability of Resilient Computers*, T. Andersen Editor, BSP Professional Books, Blackwell Scientific Publications, 1989.

[3] Margaret A. Ellis and Bjarne Stroustrup: *The Annotated C++ Reference Manual*. Addison Wesley 1990.

[4] J. Goodenough: *Exception Handling: Issues and a Proposed Notation*. CACM December 1975.

[5] Steve C. Glassman and Michael J. Jordan: *Safe Use of Exceptions*. Personal communication.

[6] Steve C. Glassman and Michael J. Jordan: *Preventing Uncaught Exceptions*. Olivetti Software Technology Laboratory. August 1989.

[7] Griswold, Poage, Polonsky: *The SNOBOL4 Programming Language*. Prentice-Hall 1971

[8] Andrew Koenig and Bjarne Stroustrup: *Exception Handling for C++*. Proc. C++ at Work Conference, SIGS Publications, November 1989.

[9] Andrew Koenig and Bjarne Stroustrup: *Exception Handling for C++ (revised).* Proc. USENIX C++ Conference, San Francisco, April 1990.

[10] Bertrand Meyer: *Object-oriented Software Construction*. Prentice Hall. 1988.

[11] Mike Miller: *Exception Handling Without Language Extensions*. Proceedings of the 1988 USENIX C++ Conference.

[12] B. Milner, M. Tofte, R. Harper: *The Definition of Standard ML*. MIT Press 1990.

[13] James G. Mitchell, William Maybury, and Richard Sweet: *Mesa Language Manual*. Version 5.0. April 1979. XEROX PARC CSL-79-3.

[14] Barbara Liskov and Alan Snyder: *Exception Handling in CLU*. IEEE ToSE. November 1979.

[15] Paul Rovner: *Extending Modula-2 to Build Large, Integrated Systems.* IEEE Software Vol.3 No.6 November 1986. pp 46-57.

[16] Jonathan Shopiro and Bjarne Stroustrup: *A Set of C++ Classes for Co-Routine Style Programming*. Proc USENIX C++ Workshop. Santa Fe, NM. November 1987.

[17] Sun Common Lisp Advanced User's Guide. Section 3: *The Error Handling Facility*.

[18] David Waitzman: *Exception Handling in Avalon/C++*. CMU Dept. of Computer Science. August 6. 1987.

[19] USA Department of Defense: *Reference Manual for the ADA Programming Language*. ANSI/MID-STD-1815A-1983.

[20] Shaula Yemini and Daniel M. Berry: *A Modular Verifiable Exception-Handling Mechanism*. ACM ToPLaS. Vol.7, No.2, April 1985, pp 214-243. San Francisco, April 1990.

16.1 Appendix A: Implementations

We have described the design of an exception handling mechanism. Several different implementations are possible. In particular, it is possible to write an implementation that is fully portable in the sense that it produces ANSI C (only) and also to write an implementation that is close to ideal in its use of run-time.

16.2 Run-time representation of Types

All implementations must provide a way of representing the type of the exception thrown for matching with the types specified in the handlers. At the point where an exception is thrown, the compiler might give to the library functions that search for a handler a pointer to a string that names the exception class and all its base classes, along with a pointer to the copy constructor for each. A variant of the encoding of types used to achieve type-safe linkage on systems with traditional linkers could be used. For example:

```
class X { /* ... */ };

class Y : public X { /* ... */ };

class A { /* ... */ };

class Z : public Y : public A { /* ... */ };
```

could be represented as:

center; cfB|cfB lfCW|lfCW. = **type run-time representation** _ X "_1X" Y "_1Y:_1X" Z "_1Z:(_1Y:_1X,_1A)" Y& "R_1Y:_1X" Z* "P_1Z:(_1Y:_1X,_1A)" const char* "PCc" _

The reason that strings are a reasonable representation and that the complete inheritance hierarchy is needed is that separate compilation of the *throw-expression* and the *handler* will be the norm and that the resolution must be done at run time.

Schemes based on other kinds of objects representing the classes are of course also feasible and will be more appropriate than strings in some environments. Whatever representation is chosen should be suitably encapsulated in a class. It will also be necessary to define a set of C functions for creating and using such run-time representations. We present the string encoding scheme simply as an existence proof.

16.3 Exception Temporaries

Another issue that every implementation must face is the need to introduce a ‘‘temporary object’’ to hold the current exception during stack unwinding and during the execution of a handler (in case a re-throw operation occurs in the handler or in a function called by the handler). The use of such a temporary object is completely under control of the exception handling mechanism. One strategy would be to allocate such objects using the general purpose free store mechanisms. However, that could make the exception handling mechanism vulnerable to free store corruption and exhaustion. A better strategy would be to allocate such objects out of a pre-allocated pool of storage used only by the exception handling mechanism and maybe also optimize away the ‘‘temporary object’’ for exceptions passed by value and not re-thrown.

16.4 Outline of a Portable Implementation

Here we describe an implementation that is fully portable in the sense that it makes it possible to generate ANSI C (only) from a C++ program. The importance of such an implementation is that it allows programmers to experiment with C++ exception handling without waiting for implementations to be specifically tuned for their particular system and guarantees portability of programs using exception handling across a greater range of machines than would be economically feasible had each machine required a ‘hand crafted’ implementation. The price of such an implementation is run-time cost. We have no really good estimates of that cost but it is primarily a function of the cost of `setjmp()` and `longjmp()` on a given system.

To simplify the discussion, we need to define a few terms. A *destructible object* is an object that has a non-empty destructor or contains one or more destructible objects. The *destructor count* of a class is the number of destructors that must be executed before freeing an object of that class, including the destructor for the class itself. The destructor count of an array is the number of elements in the array times the destructor count of an element.

The general strategy is to lay down a destructor for every destructible object, even if the user did not specify one (for example, consider an object without an explicit destructor that contains two objects with destructors). Every destructible object on the stack acquires a *header* containing a pointer to the destructor for that object, a pointer to the next most recently created automatic destructible object, and a *skip count* that indicates how much of that object has been constructed:

```
struct _header {
    void* destructor;
    void* backchain;
    unsigned skipcount;
};
```

Thus, for example, a C++ class that looks like this:

```
class T {
public:
    T();
    ~T();
};
```

will be augmented internally to look like this:

```
class T {
    _header _h;
public:
    T();
    ~T();
}
```

Every `_header` object is part of some other object. All destructible objects on the stack therefore form a chain, with the most recent first. The head of that chain is a (single) global variable created for the purpose, initially null.

Therefore, to construct an automatic object, the implementation must make the object header point to the destructor for that object, set the skip count to indicate that construction has not yet started, and link the header into the global chain. If the variable that heads the chain is called `_DO_head` (`DO` stands for 'destructible objects'), the following code would be inserted into `T::T()` to maintain the chain:

```
_h.backchain = _DO_head;
_h.skipcount = 1;   // destructor count of T
_h.destructor = (void*) &T::~T();
_DO_head = &h;
```

We will use this chain *only* to destroy objects while unwinding the stack; we do not need it when a destructor is called during normal exit from a block. However, an exception could occur in a destructor during normal block exit, so it is necessary to keep the global chain header up to date while destroying objects. The safest way to do this is probably for the compiler to avoid following the actual chain. Instead, the compiler should point the chain header directly at each object it is about to destroy.

In other words, rather than generating code inside `T::~T()` to update `_DO_head`, the compiler should reset `_DO_head` explicitly and then call the relevant destructor where possible:

```
_DO_head = previous value;
t->T::~T();
```

The skip count indicates how many destructors have to be skipped when destroying an object. When we are about to begin constructing an object, we set its skip count to its destructor count. Each constructor decrements the skip count, so that a completely constructed object has a skip count of zero.

The elements of an object are destroyed in reverse order. Each destructor *either* decrements the skip count *or* destroys its associated object if the skip count is already zero.

To do all this, we give every constructor and destructor the address of the skip count as an 'extra' argument. If the object is not on the stack, we pass the address of a dummy word.

For example, code generated for the destructor `T::~T()` might look like this:

```
T_dtor(/* other arguments */, unsigned* skipcount) /* generated code */
{
    if (*skipcount)
        --*skipcount;
    else {
        /* destroy '*this' */
    }
}
```

If destructors leave a zero skip count untouched, we can safely give a destructor for an off-stack object the address of a static zero word somewhere. Having a destructor do this costs little, as destructors must test for zero anyway.

The test is presumably unnecessary for constructors, and it would be a shame to have to include it anyway just to avoid having a constructor for an off-stack object scribble a global value. However, we can use a different global value for constructors. We don't care what the value is, so all constructors can safely scribble it. By making all skip counts `unsigned`, we can avoid the possibility of overflow or underflow.

The actual stack unwinding must be done by `longjmp` and takes place in two stages. First we track back through the stack object chain, calling destructors along the way. After that, we execute the C `longjmp` function to pop the stack.

What is the overhead of this scheme? Every constructor gets an extra argument (which takes one

instruction to push it) and has to decrement the skip count (one or two instructions, two memory references). Every destructor gets an extra argument (one instruction to push it) and has to decrement and test the skip count (three or four instructions). In addition, each destructible object gets three extra words (which takes three instructions to fill them) and the list head must be updated when the object is created (one instruction). There is no extra overhead for classes, such as `Complex`, that do not have destructors.

Thus the overhead per destructible object is somewhere near 10-12 instructions. This overhead decreases if constructors or destructors are inline: the skip counts can be set directly instead of indirectly and the inline destructors don't even have to test them. It is still necessary to lay down an out-of-line destructor with full generality.

16.5 Outline of an Efficient Implementation

Here we outline an implementation technique (derived from Clu and Modula2+ techniques) that has close to ideal run-time performance. This means that (except possibly for some table building at startup time), the implementation carries absolutely no run time overhead until an exception is thrown; every cycle spent is spent when exceptions are thrown. This could make throwing an exception relatively expensive, but that we consider a potential advantage because it discourages the use of exception handling where alternative control structures will do.

Since this implementation requires detailed knowledge of the function calling mechanism and of the stack layout on a particular system, it is portable only though a moderate amount of effort.

The compiler must construct, for each block, a map from program counter values to a 'construction state.' This state is the information necessary to determine which objects need to be destroyed if execution is interrupted at the given point.

Such a table can be organized as a list of address triples, the first two elements representing a region in which a particular destruction strategy is appropriate, and the third element representing a location to which to jump if execution is interrupted in that region. The number of entries in such tables could be greatly reduced (typically to one) at a modest execution cost by adding a fourth element to entries and using the `__counter` trick presented in section 5.

This table is searched only if an exception is thrown. In that case, the exception dispatcher unwinds the stack as necessary, using the appropriate table for each stack frame to decide how to destroy objects for that stack frame. The dispatcher can regain control after each level of unwinding by 'hijacking' the return address from each frame before jumping to the destruction point.

It is possible to merge all pre-frame exception tables into a single table either at run time or compile time. Where dynamic linking is used such table merging must be done at run time.

As before, the decision about whether to enter an exception handler can be made by code generated for the purpose, the address of which can be placed in the table in place of the destruction point for those blocks with exception handlers.

The only overhead of this scheme if exceptions are not used is that these tables will take up some space during execution. Moreover, the stack frame layout must be uniform enough that the exception dispatcher can locate return addresses and so on.

In addition, we need some way of identifying these tables to the exception dispatcher. A likely way to do that while still living with existing linkers is to give the tables names that can be recognized by a suitable extension to the `munch` or `patch` programs.

16.6 Appendix B: Grammar

In the C++ grammar, the exception handling mechanism looks like this:

try-block:
`try` *compound-statement handler-list*

handler-list:
handler handler-list$_{opt}$

handler:
`catch (` *exception-declaration* `)` *compound-statement*

exception-declaration:
type-specifier-list declarator
type-specifier-list abstract-declarator
type-specifier-list
`...`

A *try-block* is a *statement*.

throw-expression:
`throw` *expression*$_{\texttt{opt}}$

A *throw-expression* is a *unary-expression* of type `void`.

A *exception-specification* can be used as a suffix in a function declaration:

exception-specification:
`throw (` *type-list*$_{opt}$ `)`

type-list:
type-name
type-list `,` *type-name*

16.7 Appendix C: Local Scope Issues

Allowing the exception handling code in a *handler* to access variables in the `try` block to which it is attached would violate the notion that a block establishes a scope. For example:

```
void f(int arg)
{
    int loc1;

    try { int loc2; /* ... */ g(); } catch (x1) { /* ... */ }
    try { int loc3; /* ... */ g(); } catch (x2) { /* ... */ }
}
```

Here, both exception handlers can access the local variable `loc1` and the argument `arg`, but neither of them can get at `loc2` or `loc3`.

Disallowing access to variables in the block to which the handler is attached is also important to allow good code to be generated. For example:

```
void f(int arg)
{
    int loc1;

    // ...

    try {
        int loc2 = 1;
        g1();
        loc2++;
        g2();
    }
    catch (x1) {
        if (loc2 == 1) // ...
    }
}
```

Were this allowed, the compiler and run-time systems would have to ensure that the value of `loc2` is well defined upon entry to the handler. On many architectures, this implies that its value must be allocated in memory before the calls of `g1()` and `g2()`. However, this involves an increase of the size of the generated code and poor use of registers. One could easily lose a factor of two in run-time performance while at the same time enlarging the code significantly.

Access to local variables in the surrounding scope (including the function arguments) is allowed and is important in the cases where recovery is possible. For example:

```
void f(int arg)
{
    // ...

    try {
        // ...
    }
    catch (x1) {
        // fix things
        return f(arg);  // try again
    }
}
```

When re-trying this way, one should beware of the infinite recursion that can occur in case of repeated failure.

Naturally, the value of local variables accessible from a handler, such as `arg`, must be correct upon entry of the handler. This implies compilers must take a certain amount of care about the use of registers for such variables.

16.8 Appendix D: Syntax Details

It might be possible to simplify the

```
try { ... } catch (abc) { ... }
```

syntax by removing the apparently redundant `try` keyword, removing the redundant parentheses, and by allowing a handler to be attached to any statement and not just to a block. For example, one might allow:

```
void f()
{
    g(); catch (x1) { /* ... */ }
}
```

as an alternative to

```
void f()
{
    try { g(); } catch (x1) { /* ... */ }
}
```

The added notational convenience seems insignificant and may not even be convenient. People seem to prefer syntactic constructs that start with a prefix that alerts them to what is going on, and it may be easier to generate good code when the `try` keyword is required. For example,

```
void f(int arg)
{
    int i;
    // ...
    try { /* ... */ } catch (x) { /* ... */ }
    // ...
}
```

Here, the `try` prefix enables a single-pass compiler to ensure, at the beginning of the block with a handler, that local variables in the surrounding block are not stored in registers. Without the prefix, register allocation would have to be postponed until the end of the function. Naturally, optimal register allocation based on flow analysis does not require `try` as a hint, but it is often important for a language design to enable simple code generation strategies to be applied without incurring devastating run-time penalties.

Allowing exception handlers to be attached to blocks only and not to simple statements simplifies syntax analysis (both for humans and computers) where several exceptions are caught and where nested exception handlers are considered (see Appendix E). For example, assuming that we allowed *handler*s to be attached to any statement we could write:

```
try try f(); catch (x) { ... } catch (y) { ... } catch (z) { ... }
```

The could be interpreted be in at least three ways:

```
try { try f(); catch (x) { ... } } catch (y) { ... } catch (z) { ... }

try { try f(); catch (x) { ... } catch (y) { ... } } catch (z) { ... }

try { try f(); catch (x) { ... } catch (y) { ... } catch (z) { ... } }
```

There seems to be no reason to allow these ambiguities even if there is a trivial and systematic way for a parser to chose one interpretation over another. Consequently, a `{` is required after a `try` and a matching `}` before the first of the associated sequence of `catch` clauses.

16.9 Appendix E: Nested Exceptions

What happens if an exception is thrown while an exception handler is executing? This question is more subtle than it appears at first glance. Consider first the obvious case:

```
try { f(); } catch (e1 e) { throw e; }
```

This is the simplest case of throwing an exception inside an exception handler and appears to be one of the more common ones: a handler that does something and then passes the same exception on to the surrounding context. This example argues strongly that re-throwing `e1` should not merely cause an infinite loop, but rather that the exception should be passed up to the next level.

What about throwing a different exception?

```
try { f(); } catch (e2) { throw e3; }
```

Here, if exception `e2` occurs in `f()`, it seems to make sense to pass `e3` on to the surrounding context, just as if `f()` had thrown `e3` directly. Now, let's wrap a block around this statement:

```
try {
    try {
        f();
    }
    catch (e2) {
        throw e3;
    }
    catch (e3) {
        // inner
    }
}
catch (e3) {
    // outer
}
```

If `f()` throws `e2`, this will result in the handler for `e3` marked ‘outer’ being entered.

From the language’s perspective an exception is considered caught at the point where control is passed to a user-provided handler. From that point on all exceptions are caught by (lexically or dynamically) enclosing handlers.

The handling of destructors naturally gives raise to a form of nested handlers − and these *must* be dealt with. Nested handlers may naturally occur in machine generated code and could conceivably be used to improve locality of error handling.

16.10 Appendix F: Arguments to Throw Operations

We expect that it will be commonplace to pass information from the point where an exception is thrown to the point where it is caught. For example, in the case of a vector range exception one might like to pass back the address of the vector and the offending index. Some exception handling mechanisms provide arguments for the throw operation to allow that[13,14,15]. However, since (in the scheme presented here) exceptions are simply objects of user-defined types, a user can simply define a class capable of holding the desired information. For example:

```
class Vector {

    int* p;
    int sz;

public:
    class Range {
        Vector* id;
        int index;
    public:
        Range(Vector* p, int i): id(p), index(i) { }
    };

    int& operator[](int i)
    {
        if (0<=i && i<sz) return p[i];
         throw Range(this, i);
    }
};
```

```
void f(Vector& v)
{
    try {
        do_something(v);
    }
    catch (Vector::Range r) {
        // r.id points to the vector
        // r.index is the index
        // ...
    }
}
```

This solution works nicely even under a system with concurrency. Throwing an exception involves creating a new object on the stack; catching it involves unwinding the stack to the catch point after copying the exception object into the handler. Neither of these operations is a problem for a system that can handle concurrency at all.

16.11 Appendix G: `terminate()` and `unexpected()`

Occasionally, exception handling must be abandoned for less subtle error handling techniques. Examples of this are when the exception handling mechanism cannot find a handler for a thrown exception, when the exception handling mechanism finds the stack corrupted, and when a destructor called during stack unwinding cause by an exception tries to exit using an exception. In such cases

```
void terminate();
```

is called.

One other kind of error is treated specially: when a function tries to raise an exception that it promised in the *throw-clause* in its declaration not to raise. In that case

```
void unexpected();
```

is called. This case does not necessarily call `terminate()` because although the failure is severe in some sense, its semantics are well defined and it may therefore be possible to recover in a reliable way.

The `terminate()` function executes the last function given as an argument to the function `set_terminate()`:

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

The previous function given to `set_terminate()` will be the return value.

Similarly, the `unexpected()` function executes the last function given as an argument to the function `set_unexpected()`.

This enables users to implement a stack strategy for using `terminate()` or `unexpected()`:

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

void my_function()
{
    STC xx(&my_terminate_handler);

    // my_terminate_handler will be called in case of disasters here

    // the destructor for xx will reset the terminate handler
}
```

This is an example of using a local object to represent the acquisition and relinquishing of a resource as mentioned in section 6. Without it, it guaranteeing the reset of the function to be called by `terminate()`

to its old value would have been tricky. With it, `terminate()` will be reset even when `my_function()` is terminated by an exception.

We avoided making `STC` a local class because with a suitable name it would be a good candidate for a standard library.

By default `unexpected()` will call `terminate()` and `terminate()` will call `abort()`. These default are expected to be the correct choice for most users.

A call of `terminate()` is assumed not to return to its caller.

Note that the functions `abort()` and `terminate()` indicate 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.

A call of `abort()` invokes no destructors. A call of `exit()` invokes destructors for constructed static objects only. One could imagine a third way of exiting a program where first constructed automatic objects were invoked in (reverse) order and then then destructors for constructed static objects. Such an operation would naturally be an exception. For example:

```
throw Exit(99);
```

where `Exit` must be defined something like this:

```
class Exit {
public:
    int val;
    Exit(int i) { val=i; }
};
```

One might further define the invocation of a C++ program like this:

```
void system()   // pseudo code
{
    try {
        return main(argv,argc);
    }
    catch (Exit e) {
        return (e.val);
    }
    catch (...) {
        return UNCAUGHT_EXCEPTION;
    }
}
```