

# Bullet-proofing Your Code

## How to Expect the Unexpected

Charles Weir

© Object Designers Limited, December 1994

What should our code do when it finds a problem? How should it handle something unexpected? Bad parameters to a function, missing configuration files, garbage typed in by a user: a robust system will detect all of these, and take some action for each. If we as programmers write a system that doesn't do this, the very least that we can expect is rude phone calls from our users as the software misbehaves in unpredictable ways.

In a large project it is also important that all of the code handles errors consistently: if Jim's functions all return an error status code, and Joan's throw C++ exceptions, then Jim and Joan are inflicting significant confusion on developers who must interface to both. They're not doing much, either, for the poor programmer who must maintain the system when they've gone.

So consistency in error handling is 'a good thing'. It matters less which approach we use, than that the approach is consistent across the entire project. Thus decisions on error handling are part of the project strategy, and we must make them early in the project. This article will describe an approach to C++ error handling and the technical ramifications of some possible choices.

## Types of Error

What is an error? Consider for example a function that returns a list of the file names corresponding to a given wild card specification string (using a list template class):

```
getFileList( const char* specification, List<char*>& result );
```

The errors it might detect might include: a call with a null pointer as parameter; an operating system error (no such directory, or out of memory); or simply that there are no files matching the given specification. Observe, though, that these three errors are different in kind: the invalid parameter will never happen if our code is correct; the lack of memory may terminate either the program or just the action; the invalid string is quite likely, and needs to be handled in a helpful way. Table 1 shows how our `getFileList` function might handle different situation in two different applications, a file listing utility, and a file server: the key difference between the two is that the file server may have many clients and a failure in the server should not bring all the clients down, whereas failure of the directory utility can be allowed to cause failure in its clients.

Situation	Simple Directory Utility	File Server
Some files match	Answers a list of files	Answers a list of files
No matching files	Answers an empty list	Answers an empty list
File system error (e.g. invalid directory).	Terminates program abnormally with an error message	Terminates request with an error status reply
Invalid pointer parameter	Terminates program abnormally with an error message	Terminate program, logs error message, and restarts.

*Table 1: How `getFileList` might handle situations in two different applications.*

In practice, we can categorise the types of error:

- *Internal Errors* are errors that cannot happen, at least in theory. Examples include data corruption, invalid functions to parameters, or calling functions in a sequence they do not support.
- *External Errors* are (rare) conditions where the processing of an action cannot continue due to a fault within the computer system. Examples might include a network fault, an unexpected file system error, or running out of memory.
- *Anticipated Errors* are situations where the external input to the program is not what was expected. Examples might be garbage input typed by the user, or an out-of-range reading from a temperature sensor.

The key point about this categorisation is that we are not pre-judging what action to take as a result; it is, after all, likely to be different in different applications. The following sections examine each type of error in more detail, and suggest an approach to handling each one.

## Handling Internal Errors

The important feature of internal errors is that they do not happen. At least in a fully tested, production quality, released, and quality assured application, they shouldn't happen. In practice, of course, during the development and early test phases of a project some such errors almost certainly will occur. There will be bugs. We should like to check for them to find them early so as to generate meaningful error messages rather than obscure system crashes, and this requires us to insert tests into the code: parameter checks and the like. Yet there is an overhead in processing time and code size to these tests; we want to avoid keeping them in the release version.

The solution to this is a C++ macro, `assert()`, or a variant of it. This library macro checks that its single parameter is true; if false it uses clever pre-processor directives to display a helpful message with the file and line number where the problem occurred (ANSI versions usually also display the checking code as text), and then terminates the program. To compile the final version of the program we define the pre-processor macro `NDEBUG`, and all of the `assert` statements compile out to nothing. Thus we can make the checks as complicated as we like, with no final performance penalty.

## Using Assertions

There are several points to keep in mind when using assertions. First, it is crucial that we don't use `assert` to handle errors that may still occur in the final system; since assertions don't exist in the final version, neither will this error handling. In addition, the location (file and line number) displayed by the error message is the place where the error was detected, while often the actual error is elsewhere. For example, in the `getFileList` example the assertion will fail if the caller passes an invalid parameter, but the fact that a bad parameter was passed tells us little; to debug the error we need to examine the stack. Some implementations of `assert` display an error message then call `exit()`, which loses all information about the program state. In the MFC and Borland OWL MS Windows environment the error display is a dialog box; so long as we are running under a debugger we can halt the application at this point and examine the stack. In UNIX environments, it is usually best to rewrite the `assert` macro (in a different header file) so that it calls `abort()` rather than `exit()`; then we can use a UNIX debugger to examine the resulting core file. Figure 2 shows a typical UNIX implementation of `assert`; note the clever use of pre-processor features. In either environment an examination of the function call stack is often enough to identify the problem.

```

#ifndef NDEBUG
#  define assert(p) ((p) ? (void)0 : (void) (cerr << \
        "Assertion failed: " #p ", file " __FILE__ ", line " \
        << __LINE__ << endl, abort() ) )
#else
#  define assert( p )
#endif

```

Figure 2: Possible UNIX implementation of `assert()`.

So `assert` will validate the parameters of a function. Yet if the parameter is a pointer or reference, how can `assert` check it? We can check if it is zero, certainly, but that is hardly a very strong test. Instead there is a convention here which helps: a validation function. For each class, we define a function to verify that the instance data is in a consistent state. This function has the same name for every class - let's call it `isValid()` - and it is called from within an `assert` statement (so that the calls are compiled out in the final version). Yet rather than return `FALSE` if the check fails, it uses assertions itself to give a better idea where the error occurred. For convenience it always returns `TRUE`. It will often be a virtual function, so that a call through a base class pointer or reference will call the correct version<sup>1</sup>. Figure 3 shows an example implementation of the macro `assert`, and an example of its use in implementing part of a simple string class.

```

class String {
public:
    int isValid() const;
    size_t length() const;
    // ...Constructor, destructor and other member functions go here.
private:
    char* ptr;
};
int String::isValid() const
{ assert( this != 0 );
  assert( ptr != 0 );
  return 1; }
size_t String::length() const
{ assert( isValid() ); return strlen( ptr ); }

```

Figure 3: Using `assert()` with a validation function.

## Handling External Errors

How do we handle errors that are unusual - errors which mean that the computer system is not behaving as expected?

In most cases the desired behaviour is to stop the unit or processing that caused the error; log, display, or send an error message; and return to wait for another request. What do we mean by 'a unit of processing'? We could find other names for it: an 'action', a 'scenario', or the 'processing of a single event'; but in C++ terms what we usually want to terminate is the sequence of function calls resulting from a single initial entry point. Alternatively, if we do not have a recovery plan for a particular kind of error we shall want to terminate the entire program with an appropriate error statement (message, log, or dialog box).

Consider, for example, a class representing a file, where the constructor opens the file itself (figure 4). What do we do if there is an error on opening the file?

---

<sup>1</sup>In the Microsoft Foundation Class library, this virtual function is defined on `CObject`, and called `AssertValid()`.

```

class File
{
public:
    File( const char* name ) { f = fopen( name, "r+" );
                            if ( f == NULL ) { /* What happens here? */ }
    }
private:
    FILE* f;
};

```

Figure 4: Part of a file implementation, showing a problem situation..

The answer, of course, lies in the new C++ feature, *exceptions*. A C++ exception answers our needs. The syntax of exceptions adds three keywords to the language: to raise an exception we use the `throw` keyword, while for code that implements the recovery from an exception, there is the `try...catch` construction. The book “The Design and Evolution of C++” describes C++ exceptions in more detail; figure 5 summarises the key points.

- Exceptions cannot resume: a `throw` statement will never return to execute the following statement. Either execution will continue in a `catch` block, or the program will terminate.
- Each `throw` makes a copy of its single parameter, which is usually a temporary instance of a special exception class. This copy is passed, by value or by reference, as the single parameter to the `catch` block. This `catch` parameter must match (be of the same class, or a base class of) the parameter thrown.
- Exceptions unwind the stack, calling destructors for all the previously initialised stack instances until they reach the corresponding `catch` block.
- There are two functions used by the exception handler to deal with problems: the *terminate* and *unexpected* functions. Each may be redefined by the user. The *terminate* function must stop the program (it may not return); by default the *unexpected* function will do the same.
- Any further exception, thrown in a destructor while unwinding the stack, must be caught before the destructor returns or else the program will terminate (with the *terminate* function).
- Functions may supply `catch` signatures specifying the exceptions they are allowed to pass back to their caller. The signature generates only run-time checks. If a function attempts to pass an exception that doesn't match its signature, this generates a run-time call to the *unexpected* function.

Figure 5: Key points of C++ exceptions

## Using C++ Exceptions

As with most C++ features, exceptions allow a good deal of flexibility in the way they are used. The key decisions still rest with the programmer. In particular, we still have to define the classes that represent the different kinds of exception, the classes whose instances are thrown. What should these contain, what is the exception class hierarchy, and how should the exception classes and the `catch` statements be structured to access their information?

### Contents of an Exception Class

Each exception needs to contain enough information to allow it to be processed effectively. Thus each instance of an exception class needs to contain error codes, error severity codes, and any other information pertinent to its particular error type. Similarly each error will usually require a string explaining the cause of the error.

Since we catch any exception class using a `catch` clause for its base class, it is very important that these base classes represent useful categories of errors. Thus we have to be very careful to achieve a useful inheritance hierarchy for our exception classes. A key question is: are the types of error determined by the module of code where they occur, or by the nature of the error they represent. To take an example, if we write a library module which can have problems with a shortage of memory, do we throw an instance of a 'Memory Error' class (`xalloc`, for example), or do we throw an error specific to the library module (a `MyLibraryError` instance, perhaps)?

To answer this, consider the data we may need to include with the exception instance in order to process it. For a memory error, typically we might want to know the number of bytes requested, which will be implemented as a `bytesRequested()` member function on the exception class. But to have such information using the `MyLibraryError` class would mean having a `bytesRequested` data member. Either this would be defined in class `MyLibraryError`, and be redundant for other errors thrown by the library, or it would be defined in a specialist `LibraryMemoryError` class, deriving from `MyLibraryError`. The first option is distasteful and makes it awkward to add new types of error; the second leads to a proliferation of exception classes. So instead we derive global exception classes, such as the proposed ANSI standard `xalloc` class, or maybe one such as `FileError`, differentiating them according to the type of the exception thrown. After all, in processing a memory error, we seldom care precisely where it occurred.

Is there one base class for all the exception classes? We have some guidance from the ANSI committee's deliberations, in the form of the classes already defined for ANSI library errors: all exception classes derive from the class `xmsg`. Figure 6 shows a possible implementation of `xmsg` that conforms to the draft standard.

```
class xmsg
{
public:
    // Uses compiler generated destructor, copy ctor,
    // and assignment operator
    // (granted a sane implementation of the class string).
    xmsg(const string& s) : str(s) {}
    string why() const { return str; }
private:
    string str;
};
```

Figure 6: Possible implementation of `xmsg`

In creating an exception class derived from `xmsg`, we shall often want to create a composite string for the error description, by adding parameters to the basic text. We do this either by initialising the string as we want it in the constructor initialisation list, or by using the assignment operator for `xmsg` to change it later.<sup>2</sup> Figure 7 gives an example. It is an exception class to handle operating system errors; the POSIX `strerror()` function encapsulated in `getErrorString` is related to the more familiar `perror()` function.

---

<sup>2</sup> Beware: One or two current implementations of `xmsg` do not support this assignment operator correctly.

```

class SystemError : public xmsg {
public:
    SystemError( const string& s )
        : Errno( ::errno ),
          xmsg( s + ": " + strerror( ::errno ) )
    {}
    int  errno() const { return Errno; }
    // Uses compiler generated destructor, copy ctor,
    // and assignment operator.

private:
    int Errno;    // Copy of the OS error number for this error.
};

```

Figure 7: Possible implementation of a SystemError Exception reporting class

We might use this class as follows shown in figure 8.

```

try
{
    ifstream s( "x" );
    if (!s) throw SystemError( "Cannot open file x" );
}
catch (xmsg& x)
{ cout << x.why() << endl; }
// Which displays: "Cannot open file x: No such file or directory"

```

Figure 8: Example of using class SystemError

Note how we have encapsulated as much work as possible within this class; it takes on the work of retrieving the error information and of producing a composite error message, so that a user accessing the error message using the `xmsg` class gets a meaningful and helpful message. If the project required such errors to be logged to file, or multiple language support, we would add this functionality to the class as well. This is a good example of object oriented reuse: by encapsulating functionality in the class we avoid the need to duplicate it in more than one location.

### Catching C++ Exceptions

C++ has a policy of having no policies. In exception handling, it supports two different approaches to the catch statement: pass by reference, and pass by value; in other words, the parameter to the `catch` clause may be either an instance, or a reference. Which should we use?

```

class Exception
{ virtual char* reason(); };

class ParticularException : public Exception
{ virtual char* reason(); };

void foo()          { throw ParticularException(); }

// --- Value approach to catching exceptions:
try { foo(); }
catch (Exception x)          { cerr << x.reason() << endl; }

// Reference approach to catching exceptions:
try { foo(); }
catch ( const Exception& x ) { cerr << x.reason() << endl; }

```

Figure 9: Value and Reference approaches to catching exceptions

Figure 9 shows the two approaches in practice. The difference is clear: the value approach will call `Exception::reason()`; the reference approach will honour the type of the original exception and call `ParticularException::reason()`. In other words, the value approach does not respect virtual functions. So we have a clear choice: either we use virtual functions to distinguish exceptions; or we use the value approach with no virtual functions. In the latter case we must ensure that the derived exception classes set up the data in the base classes correctly so that virtual functions are not required.

Which approach should we use in practice? Personally, given a free choice, I would favour the reference approach; it seems more, well, object oriented. However consider what happens when we write a library function for reuse. If we throw value-type exceptions, then they behave correctly whether the user chooses to catch them as values or as references. Whereas if we throw reference-type exceptions (containing virtual functions) and the user catches them by value, then the caller obtains the wrong versions of our member functions. So the only completely safe strategy to use in a library is to support no public virtual functions in exception classes. Indeed, this is the approach that the ANSI exception classes `xmsg` and `xalloc` have adopted.

However it is safe to catch by reference whichever approach the exception classes take. So the rule we can adopt is always to catch by reference, but in exception classes for reuse in many systems, to design exception classes not to use virtual functions.

## Handling Anticipated Errors

Last we might consider the situation of the *anticipated* error: something that is quite likely during normal processing. Examples might include asking for a list of file names and finding none, or putting up a warning dialog box “This is dangerous. Do you wish to continue?”, and having the user select “no”; or indeed any other form of unexpected input typed by the user or received from the outside world.

I strongly recommend against using exceptions to handle such errors. In terms of the C++ language clearly they will work for this; the problem lies in software maintenance and debugging. The writers of a function must anticipate that an exception not caught may terminate the program. However for *anticipated* errors it is important both that it is clear from the interface that such exceptions can and will happen, and that they are handled gracefully. Exceptions also make it difficult to follow the flow of control; they are, after all, thinly disguised `goto` statements.

Most important of all, exceptions hamper portability, since not all compilers support them as yet. So instead, where possible, we prefer to handle *anticipated* errors using other C++ constructions: for example an empty collection (in the example where no files were found), a null pointer value, or perhaps a returned status code.

## Summary and Conclusion

This article has identified three kinds of error condition which may occur in almost any C++ program, and suggested ways of handling each:

- *Internal errors*, representing faults in the code.
- *System errors*, which are unexpected problems with the computer system or configuration.
- *Anticipated errors*, which are problems that may well occur as part of normal processing.

Figure 10 shows in a simple flowchart how we might decide between the error types in a given project, and how we would handle them.

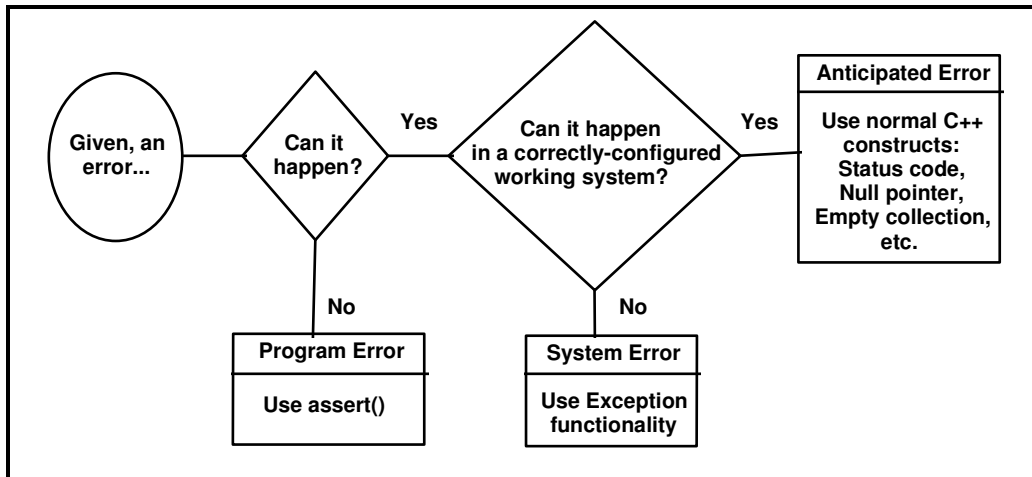


Figure 10: Deciding between the error types

Of course, in practice there will always be some choices and strategic decisions involved in choosing between the error types: for example, if the user provides the configuration file, is an error in this a system error or an anticipated error? The answer depends on your project. Nevertheless these questions will usually allow us to categorise and handle the majority of our error situations.

Whether we choose this approach or a different one, it is vital to establish a consistent strategy for error handling early in the project. With this in place, it becomes straightforward to anticipate and code for error conditions. Which leads to fewer bugs, faster delivery, and a more robust product. And that, after all, is what good programming is all about!

---

## References

- [ARM] *The Annotated C++ Reference Manual*, Margaret Ellis & Bjarne Stroustrup, Addison-Wesley 1990, ISBN 0-201-51459-1.
- [Evolution] *The Design and Evolution of C++*, Bjarne Stroustrup, Addison-Wesley 1994, 0-201-54330-3