

Code that tests itself

Using conditions and invariants to debug your code

Charles Weir

© Charles Weir, November 2004

Introduction

It is virtually impossible to write any significant amount of code without some errors in it. To remove errors we can do 'system testing', providing external input and checking the code behaves as expected. We can 'module test' components by creating frameworks to exercise them through their external programming interfaces (APIs). Also we can give the code itself to other people to review, to identify bugs by inspection and analysis just as a detective identifies criminals by inspecting for clues.

However there is a further possibility: we can use the code we write to check itself. Automatically. By adding a few lines of code here and there within the application code we can make it tell us when it's going wrong, long before any error appears to the user. Of course the extra code has a cost: it occupies space and takes extra execution time. However with the help of the pre-processor we can ensure that we pay these costs only while we are debugging and not in the final version of the system.

In this article I'll look at how we implement this checking in C++, and examine a systematic way to design these checks in applications. The C++ technique is called an *assertion*, and the design approach is called *pre- and post- conditions*, and *class invariants*. We'll examine what these are, look some traditional and not-so-traditional techniques to implement them in the C++ language. In particular we'll consider the problem of what happens after an exception: how we can be sure that our objects are left in a valid state.

What is a C++ assertion?

Consider a common problem for a programmer: you have a possible situation that you're pretty sure will never happen. However you can't be absolutely certain. Maybe it depends on third-party code that you've no way of verifying, or maybe the situation is too complicated for you to be sure that you've worked through every possible situation. You have two options:

1. You can ignore the possibility, or
2. You can put in unnecessary code to deal with the situation and behave in some sensible way if it did happen.

Option 1 will cause a difficult-to-debug crash if it happens; option 2 leads to code bloat and lots of code you could never test.

Fortunately there is a third option: an assertion. This is simply a line of C++ that states the assumption you're making, and which makes the problem easy to debug if it turns out not to be true. It's written as a macro, which takes the true expression as a parameter:

```
ASSERT( x == 0 ); // x is always non-zero at this point.
```

We can think of an assertion as an *executable comment*. It has no effect on the final system, yet it shows the intention of the programmer and proves that the intention has been carried out.

How are assertions implemented?

The implementation of assertions must reflect their use as a debugging tool. A good implementation of assertions will make sure of three things:

- Compiling assertions must be optional. We need to ensure we can get good performance and code size by telling the compiler not to generate the assertion checking code.
- If an assertion is false at any point, we need to get as much information as possible about the problem. In a UNIX environment this usually implies a core dump; in the MS Windows environment this means invoking the debugger. If there's no debugger available, we should at least be told which expression caused the assertion failure.
- If someone accidentally produces code that relies on an assertion being the function call it looks like, then code will continue to work similarly even if the assertion is compiled out:

```
if ( x )
    ASSERT( x == 1 ); // Should behave whether or not assertions are compiled.
```

Typical implementations of assertions use the standard pre-processor macros `__FILE__` and `__LINE__` to give the location, and use pre-processor macros to remove the code. Unfortunately in many environments the standard implementation - a macro called `assert()` - makes it very difficult to invoke the debugger. Traditionally it calls `exit()` after displaying a dialogue box; however calling `exit()` makes debugging virtually impossible. There are some good modern implementations: `ASSERT()` in the MFC development environment is one example. For many environments, though, it's worth implementing your own. One possibility is shown below:

```
#ifndef NDEBUG
// #define DEBUGGER() abort() // UNIX, or
# define DEBUGGER() asm { int 3; } // MS Windows
# define ASSERT( x ) \
    { if (!(x)) { cerr << "Assertion failed: " #x " , file: " \
        __FILE__ " , line: " << __LINE__ ; DEBUGGER(); }}
#else
# define ASSERT( x ) {}
#endif
```

In a GUI environment we might display a dialog rather than output to `cerr`.

How do we use assertions?

We can use an assertion any time we're not absolutely sure of our program logic. In particular we might use one:

- When we're writing a function that is called from code written by other developers. We will want to know if they pass invalid parameters or if they call the function when the object's not in an appropriate state. This check is also called a *pre-condition*.

```
void foo( int * x ) {
    ASSERT( x != 0 );
```

- When we call a third-party function and we want to verify that it behaves as we expect.

```
int * p = bar();
ASSERT( p != 0 );
```

- When we've done a complicated operation, such as sorting a data structure, and want to check that the results are what we intended. If this check happens at the end of the function, it might also be called a *post-condition*.

```
sort( theArray );
for ( int i = 1; i < theArray.length(); i++ )
    ASSERT( theArray[i-1] <= theArray[i] );
```

A word of warning

However it is vital not to misuse assertions. Assertions are useless for handling errors that may occur in the final system: out-of-memory conditions, network problems and the like. The assertion macro compiles to nothing in the final system so the checks simply will not work.

```
MyClass * p = new(nothrow()) MyClass;
ASSERT( p != 0 ); // Wrong!
```

Self-checking and object-orientation

So far the techniques I've discussed work just as well with procedural programming as with object-oriented work. However object-orientation gives us much more powerful tools for code self-testing. This arises from the two vital features of object-orientation:

Encapsulation In a well-written class, the only way to modify data is through the code associated with the class¹. So we can write checks to test the state of the data, and be certain they will execute whenever the data changes.

Instances Every instance of a class follows the same rules for behaviour. Thus we can talk about sets of rules on the data members in a class, and use self-checking to verify these rules.

Bertrand Meyer first introduced the principles of self-checking and object-orientation in his book *Object oriented software construction* (reference 2). In it he introduces the concepts of pre-conditions, post-conditions and invariants and describes the language of Eiffel, which implements these very effectively.

A pre-condition for a member function is true at the start of a member function. It checks, for example, that the parameters are reasonable and that it is sensible to call that function given the current state of the object. As we saw earlier in the article, we can implement pre-conditions in C++ using assertions.

A post-condition applies only to mutator member function, and describes the result of the function. Post-conditions are often expressed in terms of the initial state of the object. In C++ it's not particularly easy to retain the initial state of the object, so in practice we usually write only simple post-conditions.

However the most powerful checking tool is the *invariant*. This is a statement about the internal state of the object which is always true as perceived by a client. In C++ we can implement an invariant as a member function that uses assertions to check the instance's internal state.

A note about words

In this discussion, I'll use the term *client* to mean external code that uses our objects through the public interfaces.

In order to talk about invariants we need to distinguish two kinds of member function: *updater* functions change the data within an object; *observer* functions do not. This corresponds to the idea of *logical const-ness* (see reference 3). In C++ observer functions will generally be `const`; updater functions generally not.

Implementing invariants in C++

There are several points we must bare in mind when implementing the invariant function:

¹ Of course we could break encapsulation by having public data members. However C++ inline functions make this unnecessary.

- Inheritance The derived classes must make sure that the invariant is still true for the base class. Otherwise base class functions that assume the invariant holds would fail. In C++ terms this means that the invariant function for a derived class must call the base class function.
- Virtual When an updatator function in a base class changes the state of an object, the state change must be valid for the actual object. So we must check the invariant of the derived class. In C++, this means that the invariant function must be virtual.
- Compile out Like assertions, we want our invariants to have no performance or code size effects on the final system. So the invariant declaration in the class, and calls to the invariant must be through macros. Similarly the implementation of the invariant functions must be surrounded by `#ifndef DEBUG... #endif` statements.
- Constant The invariant function doesn't change the member data; it must be `const`.

So a simple implementation of the support files for invariants might be as follows:

```
#ifndef NDEBUG
# define DECLARE_INVARIANT      virtual void CheckInvariant() const
# define CHECK_INVARIANT()      CheckInvariant()
#else
# define DECLARE_INVARIANT
# define CHECK_INVARIANT()
#endif
```

When should the invariant be checked?

The naive approach to invariants would be to say the invariant is always true except during an updatator member function. So the rule would be to check the invariant at the end of the constructor, at the start and end of every updatator function, and at the start of every accessor function. However this is often not a practical rule for software implementation.

The first problem is that frequently we don't want to set up all the state of an object in constructor. Sometimes it may be easier to use updatator functions to set the state; sometimes the C++ rules on what we can do in a constructor are too restrictive. So there can be a period following the construction of an object before the invariant is true. In practice the rule becomes: an invariant need not be true until a client depends on it. In other words the invariant must be true only when an accessor function is called.

So in theory we might choose to check the invariant at the start of any accessor function and (usually) at the start of the destructor. This approach is logical; unfortunately it is unhelpful for debugging. By the time we're calling an accessor function we've lost any knowledge of how the object came to be in its current state. It's far too late.

However there's an important feature of object-orientation that helps solve this problem: encapsulation. Usually the only way to change the internal state of an object is through its updatator (non-const) member functions. So it will be sufficient to check the invariant as we exit from any updatator function. We'll also check the invariant on exit from the constructor unless, as above, it the invariant isn't true until the user has called a further updatator function (see Figure 1 below).

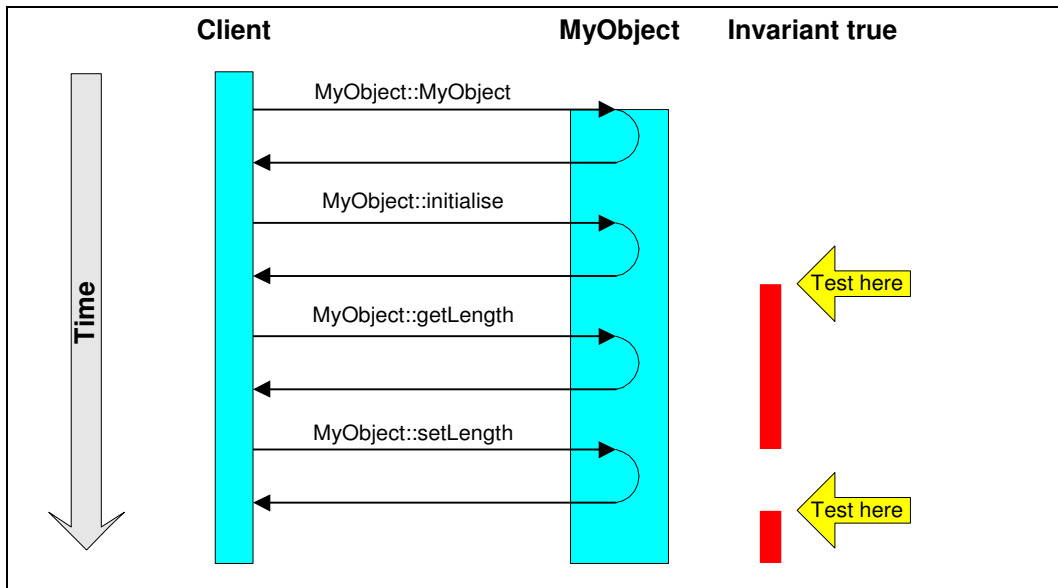


Figure 1: When the invariant is true

This rule works well for objects whose contents can be changed only through the public interface. However this is not always true. For example consider an object that uses a widget in a graphical user interface (GUI) environment. Maybe the environment might allow user input to change the internal data of the widget without reference to the object. And if we define an invariant for the object that depends on the state of the widget, then the invariant may yet be invalid at the start of a function even if it was valid at the end of the previous one. In this (unusual) case, it would be worth checking the invariant also at the *start* of any function that depends on the state of the widget.

Invariants and Exceptions

So we wish to check invariants as we exit from any updater function. In practice this is more difficult than it sounds. A function may have multiple exit points; we must be careful to check before every **return** statement.

There's a much bigger problem: a function may also exit using an *exception* (see reference 4). We should check the invariant even then. Although the most common problem with exceptions is the memory leaks they can cause, there are other significant problems caused when they leave objects in an inconsistent state. Checking the invariant will detect this problem.

Implementing this in C++ is not trivial. The straightforward approach is too clumsy to be useful:

```
#ifndef NDEBUG
try {
#endif
int foo = FunctionWhichMayThrowAnException();
#ifdef NDEBUG
catch (...) { CHECK_INVARIANT(); throw; }
#endif
```

It would make the code almost unreadable to surround every exception-capable function call with this kind of boiler-plate.

A better approach

As an alternative we can use the most important feature of C++ exceptions: as an exception unwinds the stack, it invokes the destructors on all stack-based objects. If we can create a stack object whose destructor invokes the invariant test, we've got exactly the check we want. Better

still, such a stack object is also destroyed before any return statement so we'd have no need of checks before each return statement either.

Creating such a *checker* object is not trivial, and the implementation given here has only become possible relatively recently as compilers have become able to handle templates effectively. This implementation uses the template technique sometimes called a '*virtual constructor*'. It uses a template function to deduce the C++ type of a parameter (here, the **this** parameter) and creates an instance of a template class parameterised on this type. This means the checker object can know the type of the object being checked, and call the right invariant function.

Here is an implementation that uses the technique. For conciseness I've omitted the **NDEBUG** versions.

```
class DoInvariantCheck { public:
    typedef void (*PFV)( void* );
    DoInvariantCheck( PFV cF, void* obj ) :
        checkFunction( cF ), object( obj ) {}
    ~DoInvariantCheck() { checkFunction( object ); }
private:
    PFV checkFunction;
    void* object;
};

template <class T> class InvariantChecker { public:
    static void DoCheck( void* object )
    { ((T*)object)->CheckInvariant(); }
};

template <class T> DoInvariantCheck::PFV CreateInvariantChecker( T* )
{ return InvariantChecker<T>::DoCheck; }

#define DECLARE_INVARIANT( CLASS ) \
    friend class InvariantChecker<CLASS >; \
    virtual void CheckInvariant() const

#define CHECK_INVARIANT_ON_EXIT \
    DoInvariantCheck _iCheck( CreateInvariantChecker( this ), this )
```

Example

The following code shows an example of a simple class which uses standard library classes to sort an input stream and output it. The requirements for this class are that it should be able to fail safely: even if the system runs out of memory loading the file, processing will still continue and the output will still be sorted. This is typical of the requirement for partial failure that application writers face with small machines such as Windows CE machines and Psion organisers.

```

class SortedFile { public:

    SortedFile() { CHECK_INVARIANT_ON_EXIT; }

    void ReadFile( istream& file );

    void WriteFile( ostream& os ) const {
        ostream_iterator<string> it( os, "\n" );
        copy( lines.begin(), lines.end(), it );
    }

private:
    vector<string> lines;
    DECLARE_INVARIANT( SortedFile );
};

void SortedFile::ReadFile( istream& file )
{
    CHECK_INVARIANT_ON_EXIT;
    lines.erase( lines.begin(), lines.end() );
    while ( file && !file.eof() ) // (1) loop reading lines
    {
        string line;
        file >> line;           // (2) possible memory failure.
        lines.push_back( line ); // (3) also possible failure
    }
    sort( lines.begin(), lines.end() );
}

#ifdef NDEBUG
void SortedFile::CheckInvariant() const
{
    // The collection is correctly sorted...
    for (int i=1; i < lines.size(); i++)
        ASSERT( lines[i-1] <= lines[i] );
}
#endif

```

On the face of it, this implementation looks correct. However if we simulate memory failures² there are two places where the function **ReadFile** may throw a memory failure exception: either of the points marked (2) and (3).

When this happens, the **CHECK_INVARIANT_ON_EXIT** macro causes the invariant to be checked during the exception, and this fails, pointing the finger at a bug that would be difficult to track down any other way. Once the bug is found, of course, there are many possible fixes, including simply wrapping loop (1) in a **try** . . **catch** block

Conclusion

In this article, I've shown how we can use assertion and invariant macros to help debug and maintain systems. As 'executable comments', they have two significant benefits:

1. They find errors quickly.
2. They provide comments to help future maintainers of the system. These 'executable comments' are virtually certain to be correct, given always that the system is still compiled in debug mode.

We've also examined two specific types of these macros:

Assertions verify that something is true at a specific location in code. These are best represented as a macro call.

² Some development environments make it easy to simulate memory failure. For others it may be necessary to redefine the global C++ **operator new()** to produce effective tests.

Invariants verify something that is true of any instance of a class, except while it is actually executing a member function. Invariants are best represented using an invariant function, with a set of supporting code and macros to ensure that the invariants are checked on exit from any updater function.

Finally, we've looked at guidelines and examples for implementers providing the support infrastructure and header files to implement these macros. In particular:

- Assertions should be designed to invoke the debugger on failure.
- The invariant function should be virtual, const, and invoke any base class invariant functions.
- Invariant checks require an automatic object whose destructor invokes the invariant check.

References

1. *Design Patterns*, Gamma, Helm, Johnson and Vlissides, Addison-Wesley 1994, ISBN 0-201-63361-2
2. *Object-oriented software construction*, Meyer, Prentice Hall 1988, ISBN 0-13-629049-3
3. *Effective C++*, Meyers, Addison-Wesley 1992, ISBN 0-201-56-364-9.
4. *The Design and Evolution of C++*, Stroustrup, Addison-Wesley 1994, 0-201-54330-3

The Author

Charles Weir is a consultant software architect specialising in O-O design. He is based in England and can be reached as Charles.Weir@iee.org.uk. The code examples are available from the web site <http://www.cix.co.uk/~cweir/>