Dr. Dobb's
THE WORLD OF SOFTWARE DEVELOPMENT

Assertions

Don't read unless you want to see how to use preprocessor, template, overloading, and virtual function tricks at the same time in only a few dozen lines of code. The result is an assertion facility that works when your code doesn't.

April 01, 2003
URL:http://drdobbs.com/assertions/184403861

This installment of "Generic<Programming>" discusses *assertions*, a very powerful tool that belongs in your arsenal. Starting from the good ole **assert**, we'll build a more powerful tool that helps you build better programs. We'll soon see (through a well-placed soapbox, the delight of columnists and the dread of readers everywhere) that assertions are more than simple tools/macros/functions. They are a *modus vivendi*, a deep conviction that divides programmers in two cliques: those who understand, and those who don't understand, the power of... but I'm getting ahead of myself.

Mojo Gets Groovier and Groovier, Baby

Ever since Mojo [1] saw the light of print, or better said, the light of the Internet, the interest in it has been impressive. The C++ community definitely looks ripe for efficient pass-by-value semantics, and this is good.

I've received numerous emails from readers with various proposals for improvements. Most add some convenience to Mojo, such as adding **operator->** to **mojo::temporary**, or shuffle the tradeoffs in various ways.

Dave Abrahams sent me code that addresses a typo and an exception-safety issue in **mojo::uninitialized_move**. (You know Dave's work, so you bet he'd be the first to figure out an exception-safety issue.) The original implementation of **mojo::uninitialized_move** was:

```
template <class Iter1, class Iter2>
Iter2 uninitialized_move(Iter1 begin, Iter1 end, Iter2 dest)
{
    for (; begin != end; ++begin, ++dest)
    {
        new(*dest) T(as_temporary(*begin));
    }
    return dest;
}
```

First off, **new(*dest)** must be replaced with **new(&*dest)**. Second, there's an exception-safety issue in that if **T**'s constructor throws an exception somewhere, the program gets into the awkward state of having built some objects of which it can't keep track.

The correct version sent by Dave is:

```
template <class Iter1, class Iter2>
Iter2 uninitialized_move(Iter1 begin, Iter1 end, Iter2 dest)
{
    typedef typename std::iterator_traits<Iter2>::value_type T;
    Iter2 built = dest;
    try
    {
        for (; begin != end; ++begin, ++built)
        {
            new (&*built) T(as_temporary(*begin));
        }
    }
    catch (...)
    {
        for (; dest != built; ++dest)
        {
            dest->~T();
        }
        throw;
    }
    return built;
}
```

As you see, the modified code keeps track of the objects constructed by copying **dest** to a new iterator **built**. Then it uses **built** throughout the loop that constructs objects. If anything bad happens, **uninitialized_move** nicely sweeps the floor behind it just before exiting so that the function either succeeds at creating all of the objects or fails by creating no object.

A subtle aspect of the new implementation is that it doesn't support output iterators for **Iter2** anymore. By definition, output iterators don't allow you to keep copies (**Iter2 built = dest;**) and use them later — you must do everything in one swoop.

If you think of it, the circle closes so nicely: output iterators are much like ink characters on paper, network packets flying down the wire, or sweet musical tunes being played on a speaker. You can't undo those actions. The bad old version was doing a "fire and forget" iteration. The new version is more careful in that it undoes whatever it did if anything goes wrong. By sheer necessity, the new, more careful version does not support output iterators. If you're like me, you have to love these situations in which theory and practice explain each other so nicely.

Thanks Dave!

Finally, my longtime email friend and up-and-coming guru Rani Sharoni (of porting-Loki-to-Microsoft Visual C++ .NET fame) wrote me that Mojo's first implementation [2] (which was simpler but discovered to be buggy by Peter Dimov) might actually be correct. Here are the relevant links [3, 4]. We'll see how things develop; it all starts feeling like a famous lawsuit in which a smart prosecutor finds new, unexpected evidence. Last time I got word from Rani, he wrote:

> First, I agree with your comment (although I don't think I'm a guru).
>
> I was actually disappointed that someone authorized (such as Steve Adamczyk) didn't reply to my posting, but the comment regarding **auto_ptr** at issue 291 really convinced me that your original implementation is legal according to the current standard just as some other **auto_ptr** tricks are.

At any rate, the good news is that Mojo works and works well within its design constraints. Moreover, barring the Sharoni factor mentioned above, Mojo seems to be the most compact framework that provides 100% elimination of unnecessary copying. Many people started to apply Mojo, and it is very likely that soon we'll start seeing reports and performance numbers brought by Mojo's large-scale utilization. By the way, if you have any, send them in. Any numbers — as long as they're good.

assert(cool);

All right, so what's in an assertion? Why should you care about assertions, when do you want to use them, and, just as important, when you do *not* want to use them?

Assertions (represented, for example, by the standard **assert** macro) are in my opinion the single-most simple powerful tool for ensuring program correctness. It is hard, at least for the projects I've been involved in, to overestimate the power of assertions. I'd go as far as saying that the success of a project can be conditioned by the effectiveness with which developers use assertions in their code.

One important thing about assertions is that they generate code in debug mode only (when the **NDEBUG** macro is *not* defined), so they are in a sense "free." This allows you to test for truths that seem so obvious that it would be a waste of cycles to verify them. But cycles are not wasted in release mode, so you have the psychological comfort to insert assertions liberally.

Philosophers say that reality defies imagination by being more complex than what our mind can concoct. This is certainly true in the case of software development. There are tons of stories about assertions that were just "impossible to fire," but they actually fired. Time and again, it goes the same way. "This value is certainly positive! That pointer is obviously not null! The array? It's undoubtedly sorted; I'd bet money on that! Why recheck tautologies?" If you're a beginner, you feel like you are wasting your time when you write some **assert**, but much more often than you initially thought, it's going to fire to save your life. It's all because software development is complex, and anything could happen in a program that is changing. Assertions verify that what you believe is "obviously true" actually stays true.

A subtle aspect of assertions is that the "sillier" they are, the more information they convey to you when they do fire — and the more valuable they are. This is because, according to the theory of information, the quantity of information in an event decreases with the likelihood of that event happening. The less likely some **assert** is to fire, the more information it will bring to you when it does fire. For example, when debugging some code without assertions, you would check the more obvious failure possibilities first, and you'd think of the conditions that "can't happen" only later (later, like in "later during the night, after exhausting all explainable potential causes").

So when to use **assert**, and when to use a genuine run-time check followed by signaling an error? After all, **assert** is for signaling errors, but there are other methods of signaling errors as well. How do you decide which one to choose in each context?

There is an effective litmus test to differentiate the cases in which you need to use **assert** and when you need to use genuine error checking: you use error checking for things that *could* happen, even very improbably. You use **assert** only for things that *you truly believe cannot possibly happen under any circumstances*. An assertion that fails always signals a design or a programmer error — not a user error.

Almost always, assertions verify conditions that in theory could be verified at compile time. You just *know* they hold. Proving some things, however, is impractical in terms of prohibitively long compilation time, lack of availability of source code, etc.

On the other hand, you do not want to use assertions to validate return values of functions that might fail. You don't use **assert** to make sure that **malloc** worked [5], that a window creation succeeded, or that a thread was started. You can, use, however, **assert** to make sure that APIs work as documented. For example, if some API function is documented to always return a positive value, but somehow you suspect it might have a bug, you might want to plant an **assert**.

Implementing Assertions

The standard-provided **assert** has a very simple implementation that looks like this:

```
// From the standard include file cassert or assert.h
#ifndef NDEBUG
    void __assert(const char *, const char *, int);
    #define assert(e) ((e) ? (void)0 : __assert(#e, __FILE__, __LINE__))
```

```
#else
    #define assert(unused) ((void)0)
#endif
```

The **__assert** helper function prints an error message to the standard error stream and aborts the program by calling **abort()**. There might be variations from one implementation to the other; for example, Microsoft Visual C++ shows you a dialog window that gives you the opportunity to break into the debugger and see the source code. You still cannot ignore the assertion and go on with execution — **abort()** will still be called as soon as you resume execution inside the debugger. (This is a bit unfortunate — you have the same privilege as Orpheus seeing Eurydice just before she's taken back to Hades. There are ways around it though — for Microsoft Visual C++ users, not for Orpheus [6].)

Terminating the program via **abort()** is too harsh. Many times, you might want to ignore a particular assertion because you notice it's benign. Then, some operating systems and debuggers allow you to nicely break into your source code to trace what's going on. In that case, again you don't want to **abort()**; instead you'd like to have the option of tracing forward through the program.

That's why it is often recommended that you write your own assertion mechanism. To do so, you should start by looking through your compiler's documentation to see how you can break into the debugger. For example, in Microsoft Visual C++ running under x86 processors, the magic incantation is **__asm { int 3 }**. We'll abstract the breakpoint code under a little macro:

```
#define BREAK_HERE __asm { int 3 }
```

So now we can implement an assertion mechanism like this:

```
inline void Assert(const char * e, const char * file, int line)
{
    switch (AskUser(e, file, line))
    {
    case giveUp:
        abort();
    case debug:
        BREAK_HERE;
        break;
    case ignore:
        // nothing to do
        break;
    }
}
#define ASSERT(e) ((e) ? (void)0 : Assert(#e, __FILE__, __LINE__))
```

You still have to define the **AskUser** function that uses some I/O to ask the user what action the program should take. A more refined assertion mechanism would also offer the option of logging the assertion before aborting, debugging, or ignoring.

A problem with **BREAK_HERE** is that it often really breaks into the debugger *at the exact place of its invocation*. However, you want the code to break in the spot *where you used* the **ASSERT**, not inside **AskUser**'s definition. It follows that you'd need to insert **BREAK_HERE** inside the **ASSERT** macro, not inside a function that the **ASSERT** macro calls [7].

Expression or Statement?

This brings us to a problematic point: so far, **ASSERT** as defined above is an *expression*. However, if you want **ASSERT** to invoke **BREAK_HERE**, you might need to transform **ASSERT** into a *statement*. This is because **BREAK_HERE** might be a statement on your system.

Expression macros are always more flexible than statement macros because you can use expressions in more contexts, and you can transform them into statements by appending a semicolon. However, statement macros allow you more flexibility (and require more care) in defining the macro.

An **ASSERT** transformed into a statement would look like this:

```
#define ASSERT(e) do\
    if (!(e)) switch (AskUser(#e, __FILE__, __LINE__))\
    {\
    case giveUp:\
        abort();\
    case debug:\
        BREAK_HERE;\
        break;\
    }\
} while (false)
```

The apparently futile "**do**/**while**" construct is a syntactic trick. Its purpose is to require the programmer to insert a semicolon right after using the **ASSERT** so that there's no confusion created.

In release mode, you can define **ASSERT** like this:

```
#define ASSERT(unused) do {} while (false)
```

which does nothing, but in an "intelligent" fashion.

Assertions and Exceptions

Often, it would make a lot of sense to throw an exception in case an assertion fails. For one thing, you can nicely test your code's safety in the presence of exceptions that are hard to cause "naturally." Then, having assertions makes a lot of sense in "beta" releases. In such cases, you do want to check for design bugs, but your users don't have the option to break into the debugger. In such cases, it is most natural to throw an exception.

From a syntactic viewpoint, we'd like to use **ASSERT** in the following ways:

```
ASSERT(a != b); // as before
ASSERT<std::runtime_error>(!Santa.bag.empty(),
    "Looks like Santa's bag is empty – and it surely shouldn't!");
```

Ideally, the two uses of **ASSERT** would share the same name, so as not to pollute too much the crowded macro identifier space. The second version would throw an exception containing the file and line as above, plus some nice programmer-defined error message.

We now have two challenging problems in redefining **ASSERT**. One is that **ASSERT** must support template syntax as well as normal, non-template syntax. The second is that **ASSERT** must support either one or two parameters. If you've ever written your own macro, you know for sure that both these features are beyond what macros can do [8].

But why be obstinate about having **ASSERT** be a macro? We need this because in **NDEBUG** mode there must be no evaluation going on at all. As mentioned before, it is important that programmers see **ASSERT** as a zero-cost debugging device. With current compiler technology, even in the presence of aggressive inlining, only macros can ensure that an expression, be it side-effect free and totally futile, is guaranteed to not be evaluated.

An idea is to have **ASSERT** be a macro that evaluates to a function name, in which case we can use template arguments and overloading with it:

```
void Assert(bool expression);
template <class E> void Assert(bool expression, const char* message = "");
#define ASSERT Assert
```

Now both **ASSERT(a != b)** and **ASSERT<std::runtime_error>(a != b)** will work swell. As it always happens, however, it isn't as easy as it seems. Now we lost three important tidbits of information: the textual representation of the expression being tested, the current file, and the current line. The latter two can be used to infer the third. Some version information would be useful as well, in which case three other standard predefined macros would help, namely **__DATE__**, **__TIME__**, and **__TIMESTAMP__**. The nonstandard but increasingly popular extension macros **__FUNCTION__** and **__PRETTY_FUNCTION__** would be very helpful as well.

But what can we do now that **ASSERT** defers to a function? We need to, well, insert some code before calling that function, code that will be executed after the function will be called, if you see what I mean! An interesting solution is to define a class right on the spot in the **ASSERT** macro:

```
#define ASSERT\
    struct Local {\
        Local(const Asserter& info)\
 {\
            if (!info.Handle(__FILE__, __LINE__))\
 BREAK_HERE;\
        }\
    } localAsserter = Asserter::Make
```

Whoa, whoa! What is going on here? Well, not a lot in a way, but I have to tell it's that kind of code in which the little details count. So, the code depends on a class **Asserter** defined elsewhere and creates a local **struct** called, well, **Local** (very original). Now if you say:

```
ASSERT(a != b);
```

the code will expand to:

```
struct Local {
    Local(const Asserter& info)
    {
  if (!info.Handle(__FILE__, __LINE__))
            BREAK_HERE;
    }
} localAsserter = Asserter::Make(a != b);
```

If you're like me, you're mildly enthused by this point: the syntactic and semantic tricks we've done work in tandem to keep inside the macro what belongs to the macro (**__FILE__**, **__LINE__**, and especially **BREAK_HERE**), and to keep everything else out of the macro. If you indeed are mildly enthused, great, it's possible you'll be more enthused in a few paragraphs; if you're not, who knows, maybe you'll still become mildly enthused later.

Now all you need to do is define a class **Asserter** like this:

```
class Asserter
{
protected:
    const bool holds_;
public:
    Asserter(bool holds) : holds_(holds)
    {
    }
    virtual bool Handle(const char* file, int line) const
    {
```

```
        if (holds_) return true;
        switch (AskUser(file, line))
        {
        case giveUp:
            abort();
        case ignore:
            return true;
        }
        return false;
    }
    static Asserter Make(bool flag)
    {
        return Asserter(flag);
    }
};
```

So the whole plan works like this: when **ASSERT(a != b)** is invoked, an object of type **Local** is created and initialized with a call to **Asserter::Make(a != b)**. In turn, this function passes down the Boolean condition to **Asserter**'s constructor and passes that **Asserter** object to **Local**'s constructor.

Local's constructor evaluates **Asserter::Handle(__FILE__, __LINE__)**. If that yields **true**, **Local** considers the problem has been handled and does nothing. Otherwise, **Local** breaks into your debugger. That's pretty much it!

We're Hosed! We're Not!

The astute reader might have noticed a fatal flaw in **ASSERT** as defined above. Consider the following sequence:

```
ASSERT(a != b);
ASSERT(c != d);
```

After the macro expansion and all, you'll get some compile-time errors about duplicate symbols: indeed, **Local** and **localAsserter** are defined twice in the same scope! Ouch! What to do?

I tried two solutions to this problem. One is messy, complicated, and doesn't work on all compilers. The other is simple, nice, and works on all compilers without problems.

Making a choice is not complicated, but it's worth telling both for informational purposes.

The bad solution relies on **__LINE__** and some complicated preprocessor trickery, which I personally never understood 100%, to generate a unique identifier for each line of the program. I won't bother you with the details. On Microsoft Visual C++ the trick doesn't work, and it seems like it never will. Because of that, they defined a separate macro **__COUNTER__** with which you can play those dreaded preprocessor tricks.

Even after all the effort, the technique of generating unique IDs falls flat on its face if you invoke **ASSERT** twice on the same line:

```
ASSERT(a != b); ASSERT(c != d); // Error! Duplicate identifiers!
```

Been there, done that, can't remember most of it.

The good solution is to... here it is, I won't spoil the surprise for you. Put it in your macro tricks bag.

```
#define ASSERT\
    if (false) ; else struct Local\
    {\
        Local(const Asserter& info)\
        {\
            if (!info.Handle(__FILE__, __LINE__))\
 BREAK_HERE;\
        }\
    } localAsserter = Asserter::Make
```

Templates Stick Their Nose

(You should be warned that seatbelts are mandatory from this point on.)

Ok, now how about the promised templated construct:

```
ASSERT<std::runtime_error>(!Santa.bag.empty(),
    "Looks like Santa's bag is empty – and it surely shouldn't!");
```

No problem. All we have to do is derive a parameterized class from **Asserter**, something like this:

```
template <class E>
class AsserterEx : public Asserter
{
    const char* const msg_;
public:
    AsserterEx(bool holds, const char* msg)
        : Asserter(holds), msg_(msg)
    {
    }
```

```
    virtual bool Handle(const char* file, int line) const
    {
        const Action action = AskUser(file, line, msg_);
        if (action == throwUp) throw E(msg_);
        return Asserter::Handle(file, line);
    }
};
```

So **AsserterEx<E>::Handle** asks the user on the action requested and either throws an exception or passes the decision down to **Asserter::Handle**.

Accordingly, we add a template function **Make** inside **Asserter**:

```
class Asserter
{
    ... as before ...
    template <class E> static AsserterEx<E> Make(bool flag, const char* msg)
    {
        return AsserterEx<E>(flag, msg);
    }
};
```

That's it — now there are two versions of **Asserter::Make**. The one we just introduced responds to **ASSERT<std::runtime_error>(a != b, "Something weird is going on, Watson")**.

Ok, all's good and... hey, *wait*. How come **Handle** is **virtual**, yet there's no dynamic allocation in sight? Does **Asserter** have polymorphic semantics or sheer value semantics?

What a succulent little detail. Let's recap the process through which **Local** is created. **Local**'s constructor takes a *reference to a **const** **Asserter***, and the function template **Asserter::Make** returns by value an object derived from **Asserter**. That object will be directly bound to the reference taken by **Local**'s constructor, without any slicing. So inside the mentioned constructor, the reference will act fully polymorphic, which is exactly what's needed to complete a little design cute as a bug's ear... ah, not that it has any bugs.

Those cool people around here who have read the article on **ScopeGuard** written by Petru Marginean and yours dedicatedly [9] might notice that here and there similar techniques are used to obtain "stealth polymorphism" without dynamic allocation.

Finally, how would you define **ASSERT** in release mode? There would be many solutions. Here is how:

```
#define ASSERT\
    if (true) ; else struct Local\
    {\
        Local(const Asserter& info)\
        {\
            if (!info.Handle(__FILE__,__LINE__))\
 BREAK_HERE;\
        }\
    } localAsserter = Asserter::Make
```

The whole machinery is there, but always will be skipped by the **if (true)** test. Today's compilers are easily able to eliminate the dead code that hangs onto the unused **else** clause.

Bells and Whistles

There are two useful amenities that experience has proved useful.

Sometimes you discover that an assertion is benign, and you'd like to disable it for the rest of the execution of the program. You can achieve that purpose very elegantly if you hold a **static** Boolean variable inside the constructor:

```
#define ASSERT\
    if (false) ; else struct Local\
    {\
        Local(const Asserter& info)\
        {\
            static bool ignore;
            if (!ignore && !info.Handle(__FILE__, __LINE__, ignore))\
 BREAK_HERE;\
        }\
    } localAsserter = Asserter::Make
```

If the user selects an option such as "ignore assertions on this line for the rest of this execution," **Asserter::Handle** will set **ignore** to **true** and won't bother you again — very useful.

Another cool feature is to be able to ignore all assertions for a run of the program. This can be easily achieved by holding a **static** Boolean member inside the **Asserter** class and manipulating it accordingly.

We could go further with supporting more amenities, such as passing more macros (**__DATE__**, **__TIMESTAMP__**, **__FUNCTION__**, et al.), but hopefully the main point has been made.

You may want to consult the attached code that contains a functioning version of **ASSERT** on Microsoft Visual C++ Everett Beta.

Conclusion

Assertions are important. They are also simple. This is almost unfortunate, because some seem to think that they're too simple to be important. Do use assertions liberally throughout your code; they are watchful, reliable guards that protect you (your program) from insanity.

This article introduces an assertion facility that supports immediate breaks into the debugger (subject to your compiler and OS supporting such a feature), overloading, exception throwing, and per-line and global ignoring of assertions during a session.

ASSERT puts to work together a bouillabaisse of macro tricks, template code, and polymorphic functions to obtain a useful little assertion facility.

Acknowledgment

Many of the ideas in the **ASSERT** implementation above are the direct result of a long discussion with Jason Shirk (of "The Visual C++ 7.1 compiler is so cool!" fame) during the ACCU conference in April.

Bibliography and Notes

[1] Andrei Alexandrescu. "Generic<Programming>: Move Constructors," *C/C++ Users Journal C++ Experts Forum*, February 2003, <www.cuj.com/experts/2102/alexandr.htm>.

[2] <www.moderncppdesign.com/mojo/old/>

[3] Usenet posting by Rani Sharoni, <http://groups.google.com/groups?dq=&hl=en&lr=&ie=UTF-8&safe=off&selm=df893da6.0301021305.d8f76af%40posting.google.com>.

[4] Standard C++ Defect Report, <http://anubis.dkuug.dk/jtc1/sc22/wg21/docs/cwg_active.html#291>.

[5] By the way, I did have a memory allocation request fail recently on my 256 MB desktop machine, during normal use. There were many applications started, but nothing really out of the ordinary — a good reinforcement of the fact that out-of-memory situations can occur in the real world.

[6] If you right-click in the editor during debugging and select "Set Next Statement," you can "go to" past the **assert**, thus continuing execution.

[7] The fact that the **Assert** function is inline doesn't help: most debuggers would step through inline functions.

[8] It should be noted that GNU C++ (gcc) has a non-standard extension that allows macros with variable number of arguments.

[9] Andrei Alexandrescu and Petru Marginean. "Generic<Programming>: Simplify Your Exception-Safe Code," *C/C++ Users Journal C++ Experts Forum*, December 1999, <www.cuj.com/experts/1812/alexandr.htm>.

Download the Code

<alexandr.zip>

About the Author

Andrei Alexandrescu is a Ph.D. student at University of Washington in Seattle, and author of the acclaimed book *Modern C++ Design*. He may be contacted at andrei@metalanguage.com. Andrei is also one of the featured instructors of The C++ Seminar (<http://thecppseminar.com>).