C/C++ Users Journal™
Advanced Solutions for Professional Developers

Compile-Time Assertions

C/C++ Users Journal November, 2004

For when memory footprint, performance, and portability are important

By Ralf Holly

Ralf Holly is principal of PERA Software Solutions and can be contacted at rholly@ pera-software.com.

Traditional assertions check assumptions made by developers at runtime. Checking assumptions at runtime is surely a good thing; however, in certain cases, the compiler is able to check assertions at compile time. This is a big advantage, as assertions checked at compile time do not consume any code and do not affect performance. In this article, I describe a static (that is, compile time) assert facility that supplements the classic assert and give examples on when and how to use it.

Traditional Assertions

Much has been written about the many advantages of assertions; see, for instance, Steve Maguire's *Writing Solid Code* (Microsoft Press, 1993). In a nutshell, assertions are a means of automatically verifying assumptions made by developers; see Listing 1.

If an assumption is wrong (the expression passed to assert evaluates to 0), the program is terminated abnormally and a diagnostic message is displayed. Of course, assertions are no substitute for proper error handling. Still, assertions are extremely powerful; they can be viewed as "dynamic documentation," since they are checked at runtime. Contrast this to the traditional approach of documenting assumptions via plain /* comments */. Plain comments—even if embellished with words like "note" or "important"—tend to be overlooked or misinterpreted by maintenance programmers. Worse yet, over time, comments tend to get out of sync with the code they are supposed to clarify.

Typical Assumptions

I was once working on an embedded project where memory was extremely tight. In many places, we needed to check a status flag like this:

```
#define UART_READY_MASK 0x80
#define IS_UART_READY() (gIOStatus
  & UART_READY_MASK)
 ...
if (IS_UART_READY()) {
 ...
```

```
}
 ...
```

A colleague found out that on our hardware platform, it was more efficient to replace the bit test with a sign check. Since the bit test was already encapsulated in a macro, the change was fairly easy:

```
#define IS_UART_READY ((S08)gIOStatus < 0)
 ...
if (IS_UART_READY()) {
 ...
}
 ...
```

This five-minute change worked fine and saved quite a bit of code and everyone was happy—until we changed to a different hardware platform a year later. What was the problem? Well, even though the optimization worked nicely, it is based on a fair amount of assumptions:

- **S08** is a signed type. **S08** suggests that **S** stands for "signed;" however, this is not enforced anywhere. If **S08** has been sloppily defined as:

```
typedef char S08; // Better: type
                  // def signed char S08
```

- it might end up being signed or unsigned, depending on the compiler you are using, since the C Standard doesn't define whether **char** is signed or unsigned. (This is only true for ANSI C89, on which most contemporary compilers are still based. ISO C99 and ISO C++98 require **char**s to be signed.) If **char** happens to be unsigned, this test always evaluates to **false**.
- **S08** comprises exactly 8 bits. Portable data types such as **S08**, **U08**, **U16**, and so on, are often implemented to have at least the specified sized, not the exact size. This strategy avoids speed penalties on certain platforms that work more efficiently with their native (larger) types. It is not unlikely that **S08** is defined as:

```
// Where sizeof(short) == 2
typedef signed short S08;
```

- which would turn the optimization just described into a bug, because **(short)0x80** is still **0x80 (+128)** and, hence, greater than zero.
- **UART_READY_FLAG** is at bit offset 7. What if you use a different **UART** someday whose "ready" flag is at offset 3 instead of 7? Obviously, the sign comparison trick only works when the ready flag and sign bit (bit 7) coincide. Thus, the "optimized" code would still check the state of bit 7, which would have a totally different meaning for the new UART.

For our first platform, all of these assumptions were true and the optimization worked fine. These assumptions appeared so self evident that the programmer(s) took them for granted.

But to a team of maintenance programmers who were given the task of porting the code to a new hardware platform, these assumptions were not as self evident as to the original developers. They managed to improve the overall performance on the new platform by changing the portable types. An **S08** now looked like this:

```
// Where sizeof(short) == 2
typedef signed short S08;
```

Of course, this violation of the second assumption of the original programmer caused the maintenance programmers lots of debugging woes.

Asserts to the Rescue

One way to document and enforce these assumptions is to use asserts. While asserts are definitely many times better than hidden assumptions, traditional (runtime checked) asserts have the following shortcomings:

- Traditional asserts are only checked in debug builds. While it is a good idea to leave assertions enabled as often and as long as possible, there are times when you have to switch them off; for instance, when you have run out of memory before all of your features are implemented. (Some experts even suggest leaving assertions enabled in the release version. While there are some advantages to this approach, I wouldn't generally recommend it. Today's systems are just too diverse and one size hardly ever fits all.) It is clear that when asserts are disabled, they cannot warn you about violations of assumptions.
- Traditional asserts rely on thorough testing. Since classic asserts are checked at runtime, violations of assumptions will only be reported if you have test cases that exercise the assert.
- Traditional asserts slow down performance. Obviously, additional checks cost time. This is a big problem in interrupt service routines and/or multithreaded environments where asserts affect the timing. This quite often leads to hard-to-debug concurrency problems.
- Traditional asserts increase the memory footprint. While there are more efficient assert implementations possible than the one that typically ships with your compiler, on some systems even today every byte counts; if you sell large quantities, cents quickly accumulate.

What is needed is a way to put the burden of checking assumptions on the compiler, not the program. If the compiler is able to perform the check, no valuable code space and CPU cycles are wasted. Assumptions would be checked when the system is built, not when the system is tested. Enter static asserts...

Static Asserts

In its most basic form, a static assert could look like this:

```
#define assert_static(e)     1/(e)
```

This implementation takes advantage of the fact that if an expression **e** is **false** (that is, 0), the compiler encounters a divide-by-zero error at compile time, which causes the compilation process to abort. If **e** is nonzero (**true**), the resulting expression is **1;** or **0;** depending on the exact value of the divisor. (If **e** is **1**, the whole expression evaluates to **1**, whereas if **e** is greater than **1**, the whole expression evaluates to **0**.) Regardless of whether the value of the resulting expression is **1;** or **0;**, the compiler will happily carry on compiling the rest of the module.

In the previous example, all hidden assumptions made by developers can be verified at compile time like this:

```
// Ensure S08 is a signed type
assert_static((S08)-1 == -1);

// Ensure S08 comprises exactly 8 bits
assert_static(sizeof(S08) == 1);

// Ensure 'ready' flag and sign flag
// coincide
```

```
assert_static(UART_READY_MASK == 0x80);
```

If you are a paranoid programmer, you might want to add additional checks because who says that a byte must have 8 bits and the most significant bit of a byte is used as a sign bit?

```
// Ensure that a byte comprises
// exactly 8 bits
#include <limits.h>
assert_static(CHAR_BIT == 8);

// Ensure that MSB is used as a sign bit
assert_static((S08)0x80 == -128);
```

Alternative Implementations

Even though the **1/(e)** approach just described looks fine, it isn't perfect. First of all, a compiler message such as "Divide-by-zero error" is not very descriptive. Of course, since static asserts are not part of the language standard, we will not be able to find an implementation that outputs "Assertion failed: UART_READY_MASK == 0x80." Nevertheless, it should be possible to find something better.

Another shortcoming is that most compilers I know spit out warnings such as "Useless code," even if the expression evaluates to **true** (good case). This is annoying, especially if a company follows the "zero compiler warnings" paradigm. By far the most dangerous weakness of **1/(e)**, however, is that it silently fails if somebody mistakenly uses a static assertion where a runtime-checked assertion should have been used:

```
extern U16 gWriteMode;
 ...
assert_static(gWriteMode !=
  WRITE_MODE_READ); // Wrong
```

This expression cannot be evaluated at compile time and, hence, must be evaluated at runtime. Therefore, the developer should have used a dynamic assert instead:

```
assert(gWriteMode != WRITE_MODE_READ); // Correct
```

With **static_assert** implemented as **1/(e)**, the best a compiler can do in such a situation is warn about "Useless code." If the compiler doesn't produce such a warning, or the warning is overlooked by the developer, the assert is simply ignored and the developer is left with a false sense of security. If you are working on a C++ project, you should first consider using **BOOST_STATIC_ASSERT** (http://www.boost.org/), which is implemented more or less like this:

```
template<bool> struct CompileTimeAssert;
template<> struct CompileTimeAssert
  <true> { };
#define BOOST_STATIC_ASSERT(e)
  (CompileTimeAssert <(e) != 0>())
```

This approach is based on template specialization; if the expression evaluates to **1**, a dummy object of type **CompileTimeAssert<true>** is instantiated. If the expression is false, the compiler will generate an error message, since there is no definition for **CompileTimeAssert<false>**.

For a detailed discussion of C++ implementations of static asserts as well as more sophisticated variations, have a look at Chapter 2.1 of Andrei Alexandrescu's *Modern C++ Design* (Addison-Wesley, 2001).

If you are working on a C or embedded C++ project where templates and/or template specialization are not available, you might consider these implementations:

The first approach takes advantage of the fact that a switch case value may only be defined once:

```
#define assert_static(e)
     switch(0){case 0:case (e):;}
```

whereas the second implementation works because it is illegal to define an array of negative size:

```
#define assert_static(e) \
   { char assert_static__[(e) ? 1 : -1] }
```

Both alternatives work and avoid the static/dynamic assert confusion problem. They still suffer from nondescriptive compiler messages in cases where the assertion fails. (Most compilers report something like "case value '0' already used" and "negative subscript or subscript is too large," respectively.) Moreover, both produce annoying good-case warnings with most C/C++ compilers, just like **1/(e)**.

The best implementation of **assert_static** for C that I've found so far is:

```
#define assert_static(e) \
   do { \
      enum { assert_static__ = 1/(e) }; \
      } while (0)
```

I haven't seen a compiler that produces good-case warnings with this implementation. In case of a failed assertion, most compilers report "expected constant expression," which is pretty close to ideal. This version is based on the original **1/(e)** approach; however, it avoids the aforementioned shortcomings by assigning the result to an **enum** member. Since **enum** members can only be initialized with compile-time constants, the compiler will report a compile-time error if developers erroneously use **assert_static** where they should have used a traditional assert. The **do/while(0)** loop serves two purposes. First, it introduces a local namespace that avoids multiple redefinitions of the **assert_static** enumerator; second, it forces you to add a trailing semicolon after **assert_static** because the C Standard requires **do/while** loops to be followed by a semicolon.

I recommend that you try out different implementations until you find a solution that works best with your compiler(s). If there is no single implementation that works satisfactorily for all compilers (for instance, it produces good-case warnings on a particular compiler), you can always use conditional compilation:

```
/* Static assert for Meta Foo compiler */
#if COMPILER == META_FOO
   #define assert_static(e) \
      { char assert_static__[(e) ? 1 : -1] }
/* Default implementation */
#else
   #define assert_static(e) \
      do { \
         enum { assert_static__ = 1/(e) }; \
```

```
      } while (0)
#endif
```

Using Static Assertions

It takes a while to understand how and when to use static assertions. By and large, static assertions are best used to tackle portability issues. To get you up to speed, I've listed some more examples and use cases.

One trivial thing to do is check whether Boolean constants have been properly defined:

```
assert_static(TRUE == 1 && FALSE == 0);
```

Some projects employ ISO C99's portable integer types. The constraints defined in ISO C99 can easily be checked through the use of static assertions; for example:

```
// inttypes.h
// ISO C99 integer type definitions
 ...
// Now check if constraints are obeyed
assert_static(sizeof(int16_t) == 2);
assert_static(sizeof(int_least16_t) >= 2);
assert_static(sizeof(int_fast16_t) >= 2);
 ...
```

A couple of years ago, I needed to store **struct**s in nonvolatile memory (NVM). NVM was a scarce resource, so I used a special compiler switch that disabled **struct** member alignment (padding). I knew that almost every compiler supports such a feature; however, I wanted to automatically alert developers porting the code to another platform. Here is what I did:

```
typedef struct ListNode {
   U08 flags;
   U16 value;
   U16* pNext;
} LIST_NODE;
// For efficiency, we need to store list nodes without
// padding bytes. Ensure that compiler settings are
// set to 'no struct member alignment'
assert_static(sizeof(LIST_NODE) ==
   sizeof(U08) + sizeof(U16) + sizeof(U16*));
```

I presume that this has saved the porting team quite a bit of debugging time. Sometimes you need to store pointers in variables of integral type. By using this check you can ensure that such conversions are safe:

```
// Ensure that a pointer can safely be converted to an int
assert_static(sizeof(pInBuffer) <= sizeof(int));

int myint = (int)pInBuffer; // Safe
```

Or think of cases where a couple of data structures need to have the same size or number of elements, as in this example:

```
const char* TRACE_TEXTS[] = { "success", "warning", "fail" };
const U16 TRACE_IDS[] = { 17, 42, 99 };
```

Of course, there are tricks (such as x-macros) that help in such situations, but how can you ensure that both arrays have the same number of elements? It turns out to be quite easy:

```
#define ARRAY_SIZE(x) (sizeof((x)) / sizeof((x)[0]))
// Ensure TRACE_TEXTS and TRACE_IDS have the same
// number of elements
assert_static(ARRAY_SIZE(TRACE_TEXTS) == ARRAY_SIZE(TRACE_IDS));
```

There are many more uses of static assertions, but I assume that these examples are enough to get you started.

Since the purpose of a particular **static_assert** is not always obvious (especially to novice developers), it is important that all uses are accompanied by comments that clearly indicate the intent. Just look at the previous examples that I've given: Without the comments, would you have known what they are for?

Conclusion

In systems programming, memory footprint, performance, and portability are of utmost importance—and that's exactly where **static_assert** shines. While compile-time assertions are no cure-all, they nicely supplement their runtime-checked cousins. Since it is easy to add **static_assert** support, there is no reason for not adding them to your toolchest.

Acknowledgment

Thanks to Jens Steen Krogh for introducing me to static asserts many years ago and for his support in reviewing this article.