Catching errors early with compile-time assertions

By Dan Saks
President
Saks & Associates
E-mail: dsaks@wittenberg.edu

Different platforms align and pad data differently, so a particular structure definition that lays out the members properly for one platform may produce an incorrect layout when compiled for a different platform.

All too often, an improperly laid-out structure compiles without complaint, but the resulting program misbehaves at runtime. Rather than struggle to debug the program, you can craft your code so that the compiler can catch the layout errors. The trick is to use assertions that produce overt compile-time errors whenever the structure members have the wrong size or alignment.

C and C++ provide various ways to implement assertions. My preference is for something that provides a compile-time equivalent of the Standard C assert macro. Let's begin with a brief look at that macro.

Runtime assertions

The assert macro is defined in the Standard C header <assert.h> and also in the Standard C++ header <cassert>. A call of the form:

```
assert(condition);
```

expands to code that tests the condition. If the condition is true (it yields a non-zero value), nothing happens–i.e. the program continues executing with the next statement after the macro call. On the other hand, if the condition is false (equal to zero), the program writes a diagnostic message to stderr (the standard error stream) and aborts execution by calling the standard abort function.

The assert macro can help you detect logic errors in your programs. Suppose that calling get_token(f, t, n) scans input from FILE *f and copies the scanned input into the character array starting at *t with length *n*. You can call assert in the body of get_token to detect erroneous argument values that would otherwise cause undefined behavior, as in

```
bool get_token(FILE *f, char
*t, size_t n)
    {
    ...
    assert(f != NULL);
    assert(t != NULL);
    assert(n >= 2);
    ...
    }
```

If your program inadvertently calls get_token with a null pointer as the first argument, the first assertion will write a message to stderr and abort execution. With most compilers, the message looks something like

```
Assertion failed: f != NULL,
file get_token.c, line 18
```

Writing assertions into your code helps you document and enforce the assumptions that you make as you develop your code. Unfortunately, because it writes to stderr, the standard assert macro is useless in embedded environments that lack support for the standard C I/O system. However, it's not all that hard to write your own version of an assert macro that displays the message somewhere else.

Although the assert macro can be a useful debugging aid, it's inappropriate for handling runtime errors in an end-user product that is shipping. A shipping product should produce diagnostic messages that are more meaningful to the average end-user. It should also recover or shut down more gracefully than by calling abort. Consequently, <assert.h> offers an easy way to disable all assertions with little or no change to your source code. You can leave the assertions in your code as documentation, but render them so that they generate no code.

If the macro NDEBUG is defined in the source file before including <assert.h>, the assert macro will be defined as simply

```
#define assert(cond) ((void)0)
```

so that a subsequent call such as:

```
assert(f != NULL);
```

expands as:

```
((void)0);
```

Compilers can optimize this expression into no code at all.

You can write the definition for NDEBUG into the source code just before the include directive for <assert.h>, as in

```
#define NDEBUG
#include <assert.h>}
```

The problem with this approach is that you must modify the source program every time you want to turn the assertions on or off.

Most compilers let you define macros using command line arguments when you invoke the compiler, usually with the -D option. For example, a command line such as

```
cc -DNDEBUG get_token.c
```

compiles get_token.c as if

```
#define NDEBUG
```

appears before the first line in the source, thus turning the assertions off.

Using the preprocessor

You can use assertions to verify that the members in your memory-mapped structures have the proper size and alignment. Suppose that you define the device registers for a timer as

```
typedef struct timer timer;}
struct timer
  {
  uint8_t MODE;
  uint32_t DATA;
  uint32_t COUNT;
  };
```

You can use an assertion and the offsetof macro to verify that the DATA member has an offset of four within the structure, as in:

```
assert(offsetof(timer,
DATA) == 4);
```

The offsetof macro is defined in the Standard C header <stddef.h> and Standard C++ header <cstddef>. An expression of the form offsetof(t, m) returns the offset in bytes of member *m* from the beginning of structure type *t*.

This assertion does indeed catch a potential alignment problem, but it's less than ideal. Using assert to check the offset of a structure member defers until runtime a check that should be done at compile time. Calls to assert can appear only within functions, so you have to wrap the call inside a function and call that function as part of, or very shortly after, program startup.

Just to be very clear here, I'm not suggesting that every assertion can be checked at compile time. For example, an assertion that tests the value of a variable, such as

```
assert(f != NULL);
```

must be done at runtime. However, an assertion that tests the value of a constant expression, such as the size or offset of a structure member, can be done at compile time.

For assertions involving only constant expressions, some C and C++ compilers will let you use a preprocessor conditional statement to test the assertion, as in

```
#if (offsetof(timer, DATA)
!= 4)}
    #error DATA must be at
offset 4 in timer
#endif
```

Using this approach, the compiler evaluates the condition at compile time during preprocessing. If the assertion fails (the #if condition is true), the preprocessor executes the #error directive, which displays a

message containing the text in the directive and terminates the compilation. The exact form of the message varies from compiler to compiler, but you should expect to see something that looks like

```
timer.h, line 14: #error: DATA must be at offset 4 in timer
```

Using #error directives offers you the ability to write clear diagnostic messages.

Since this approach evaluates assertions at compile time, the assertions never incur a runtime penalty, so you never have to turn them off. While you can ship a program that might violate a runtime assertion, you can't ship a program that violates a compile-time assertion. A program that fails a compile-time assertion simply fails to compile.

Unlike an assert call, which must appear in a function body, preprocessor directives can appear anywhere–globally, locally, or even within a class or structure definition.

Despite these advantages, using #if directives to implement assertions has at least a couple of problems. The first problem is minor–you must invert (negate) the assertion condition in an #if from what you would normally write using the assert macro. For example, to test that the offset of the timer's DATA member is four, you write the runtime assertion as:

```
assert(offsetof(timer, DATA) == 4);
```

To test the same condition at compile time, you replace the == operator with !=, as in:

```
#if (offsetof(timer, DATA) != 4)}
  #error ...
#endif
```

or logically negate the entire condition, as in

```
#if(!(offsetof(timer, DATA) == 4))
  #error ...
#endif
```

or leave the condition alone and put the #error directive in the #else part, as in

```
#if (offsetof(timer, DATA) == 4)}
#else
  #error ...
#endif
```

The second problem with using #if directives to implement assertions is more serious: Standard C and C++ don't recognize sizeof and offsetof in #if conditions. They don't recognize enumeration constants in #if conditions, either. A few compilers allow sizeof, offsetof and enumeration constants in #if conditions as an extension, but most don't. Fortunately, you can write compile-time assertions in another way that doesn't have this limitation.

Invalid declarations

In both C and C++, a constant expression that specifies the number of elements in an array declaration must have a positive value. For example,

```
int w[10];
int x[1];
```

are valid array declarations, while

```
int y[0];
```

is not. A constant array dimension may have multiple operands and operators, including sizeof and offsetof subexpressions, as in

```
int z[2 * sizeof(w) / sizeof(w[0])];
```

This declares array *z* with twice as many elements as array *w*.

You can exploit the requirement that constant array dimensions must be positive to implement compile-time assertions as a macro:

```
#define compile_time_assert(cond)\
  char assertion[(cond) ? 1 : 0]
```

If *x* is an expression that evaluates to true, then calling

```
compile_time_assert(x);
```

expands to a valid array declaration (with dimension one). Otherwise, it expands to an invalid array declaration (with dimension zero), which produces a compile-time diagnostic message (an error or warning).

Unfortunately, the text of the error message that you see when an assertion fails varies with the compiler. I've seen messages such as "array must have at least one element," or "negative subscript or subscript is too large."

If you're lucky, your compiler produces a message that includes the array name, such as "size of array 'assertion' is zero." In that case, it helps to make the array name an additional macro parameter, as in

```
#define compile_time_assert
(cond, msg) \
char msg[(cond) ? 1 : 0]
```

Then you can use the array name to describe the reason for the assertion failure. For example, if calling

```
compile_time_assert(offsetof(timer,
DATA) == 4,}
DATA_must_be_at_offset_4);
```

causes an assertion failure, then you might see an error message that looks like

```
size of array 'DATA_must_be_
at_offset_4' is zero
```

As written, this macro has a minor problem, which is easy to fix. The problem is that, in some cases, the array declaration may be a definition that allocates storage. You can avoid the problem by turning the array declaration into a typedef, as in

```
#define compile_time_assert
(cond, msg) \
typedef char msg[(cond) ? 1 : 0]
```

You can't have two typedefs with the same name in the same scope, so you must use the msg parameter to give each typedef a distinct name. If you'd rather not bother with the msg parameter, you can declare the array as extern, as in

```
#define compile_time_assert(cond)\
extern char assertion[(cond) ? 1 : 0]
```

Unfortunately, if you use this approach, you won't be able to use the macro within a C++ class because you can't declare a C++ class member as extern.

You may find that your compiler doesn't complain about zero-sized arrays. In that case, you might try changing the 0 to a -1, as in

```
#define compile_time_assert
(cond, msg) \
typedef char msg[(cond) ? 1 : -1]
```

The Boost library offers C++ programmers another way to do compile-time assertions in the form of a macro called BOOST_STATIC_ASSERT. The macro has a clever implementation using C++ templates. If you're a C++ programmer and you understand explicit template specialization, you might want to check it out. □

[Embedded Systems Programming]