Dan Saks

Placing Data into ROM with Standard C

Several months ago I described the problem of using C or C++ to place data into read-only memory (ROM). I explained that declaring an object `const` is necessary but not sufficient to get your compiler to place that object into ROM. (See "Placing Data into ROM," May 1998, p 11.)

In the following months, I elaborated on the `const` qualifier. (See "Placing `const` in Declarations," June 1998, p. 19, and "What `const` Really Means," August 1998, p. 11.) I also introduced the `volatile` qualifier. (See "Volatile Objects," September 1998, p. 101.) With that as background, we can take a closer look at what it takes to place data into ROM.

Hewing to the standard

C and C++ implementations vary across platforms, sometimes significantly. The purpose of a language standard is, among other things, to provide programmers with a common dialect for writing programs that can execute on a wide variety of platforms. Unfortunately, embedded systems often deal with physical resources such as device registers or ROM, which, by their nature, strain the capabilities of a platform-independent dialect.

In practice, it's nearly impossible to write embedded systems entirely in standard C or C++. Most standard-conforming C and C++ implementations provide some non-standard extensions. Although you may have little choice but to use such extensions, you should use them only as needed. You should stick to the standard dialect as much as possible because, all other things being equal, portable code is better than non-portable code.

When it comes to placing data into ROM, both the C and C++ standards say surprisingly little. But looking at what they do say is still worthwhile.

Unfortunately, when it comes to placing data into ROM, both the C and C++ standards say surprisingly little. (Appallingly little, some might say.) Nonetheless, it's still worth looking at what they do say. Since the C++ standard builds on groundwork laid by the C standard, we'll start by looking at what the C standard has to say.

C's abstract machine

Often, the clearest and most direct way to explain the behavior of a program construct is to describe how it executes on some representative computer. This is what the C standard tries to do.

Although real computers have many similarities, they also have differences. To be commercially successful, a programming language standard should avoid unnecessary bias toward or against any particular architecture. Thus, the C standard describes program behaviors in terms of a hypothetical abstract machine rather than any real one.

A real machine that executes a C program need not behave exactly like the abstract machine in every respect. The standard requires only that programs executing on a real machine produce the same results that they would when executing on the abstract machine. This requirement is commonly known as the "as if" rule because a C language implementation is free to do as it pleases as long as compiled programs produce the same observable behavior *as if* they were executing on the abstract machine.

The abstract machine for standard C has separate address spaces for code and data. Although on many real machines, pointers to functions (in the code space) have the same representation as pointers to objects (in the data space), the standard has no such requirement. Thus, a program that converts a function pointer into an object pointer, or vice versa, is not a

strictly conforming program. That is, the program won't execute properly on some standard C platforms.

Providing for read-only storage

All data in the abstract machine resides in one conceptual address space. Neither the stack nor the heap gets a separate data space. Thus, a pointer to an object of type `T` can point to any `T` object, whether that object has automatic storage (on the stack), dynamic storage (in the heap), or static storage.

The single data space encompasses objects with const-qualified and volatile-qualified types, as well as unqualified types. Specifically, the standard says that:

- Qualified and unqualified versions of a type have the same representation and alignment requirements
- Pointers to qualified or unqualified versions of a type have the same representation and alignment requirements

These rules say that for any type `T`, a program can have a "pointer to a qualified `T`" pointing to an "unqualified `T`." Here's an example where `T` is `int`:

```
int i;
...
int const *pci = &i;
```

This declares `pci` as a "pointer to `const int`" pointing to an `int`.

By themselves, the rules also say that a program can have a "pointer to an unqualified `T`" pointing to a "qualified `T`," as in:

```
int const ci;
...
int *pi = &ci;  // not quite
```

However, pointer conversion rules elsewhere in the standard prohibit conversions that strip away qualifiers, unless you use a cast. (I discussed those conversion rules in "What `const` Really Means," August 1998, p. 11.) For example:

```
int *pi = (int *)&ci;  // ok
```

Thus, a pointer to non-const `T` can point to a const `T` object. This seems to suggest that a program can use a pointer to a non-const type to store into a const object, which in turn suggests that const objects really aren't read-only after all. If that's the case, then how can a standard C implementation ever place objects into ROM?

Permission to place objects into ROM comes from the following rule:

- If a program attempts to modify an object defined with a const-qualified type via an expression with a non-const-qualified type, the behavior is undefined

For example, the previous declaration initializes `pi` to point to `ci`. The expression `*pi` has type `int`, which is a non-const-qualified type. However, `*pi` designates `ci`, which is defined as a const-qualified `int`. A program can use `*pi` to inspect the value of `ci`, as in:

```
if (*pi > 0)   // ok
```

However, if it tries to use `*pi` to store into `ci`, the behavior is *undefined*:

```
*pi = 7;   // undefined
```

This assignment is an error, but it's the kind of error that is difficult, if not impossible, to detect at compile time. Acknowledging the difficulty of diagnosing this error, the standard relieves the C implementation of all responsibility. Once a program lapses into undefined behavior, all bets are off. The program has violated its obligations under the standard, so the C implementation is under no obligation to get anything right anymore.

Often, the clearest and most direct way to explain the behavior of a program construct is to describe how it executes on some representative computer. This is what the C standard tries to do.

A footnote in the standard summarizes the freedom that C implementations have to place data into ROM:

- The implementation may place a const object that is not volatile in a read-only region of storage

A C compiler cannot place an object into ROM if it's volatile as well as const. The value of a volatile object might be changed by events outside the program's control. An object can't exhibit volatile behavior if it resides in ROM.

I'm a little intrigued by the exact wording of the footnote. I expected it to say something like:

- The implementation may place a const object *with static storage* that is not volatile in a read-only region of storage

An object can't very well reside in ROM unless it has a fixed address determined at translation time. Maybe omitting the phrase "with static storage" was an oversight, but maybe it wasn't.

I suppose this wording allows for an architecture that can write-protect RAM at run time. Write-protected RAM is not ROM in the sense that I think most embedded programmers speak of ROM, but it's ROM nonetheless. In theory, a program could declare const objects as auto variables and write-protect them at run time. I'm not aware of any architecture that actually does this, but if you know of one, please let me know.

Errata

My previous column ("Volatile Objects," September 1998, p. 101) contained numerous program fragments built around a loop of the form:

```
while (*pc & READY == 0)
  /* do nothing until ready */;
```

Reader Don Starr pointed out that the condition in the loop does not do as I intended. Whereas **&&** has a higher precedence than **==**, **&** has a lower precedence. As written, the loop is equivalent to:

```
while (*pc & (READY == 0))
  /* do nothing until ready */;
```

I should have written:

```
while ((*pc & READY) == 0)
  /* do nothing until ready */;
```

I've never been one to insert redundant parentheses. However, when I showed this one to my brother Joel, he said that his philosophy has been for some time:

"When in doubt about precedence, parenthesize. When certain about precedence, parenthesize. In either case, if you're wrong, your program will still work and no one will be the wiser."

Score one for his side. **esp**

Dan Saks is the president of Saks & Associates, a C/C++ training and consulting company. He is also a contributing editor for the C/C++ Users Journal. *He served for many years as secretary of the C++ standards committee and remains an active member. With Thomas Plum, he wrote* C++ Programming Guidelines. *You can write to him at dsaks@wittenberg.edu.*