by Dan Saks

Placing Data into ROM

Just about every embedded program deals with at least some invariant data; that is, data whose values should never change during program execution. For example, I think most of us would be rather taken aback if calling:

```
printf("Hello");
```

ever printed anything but "Hello." Clearly, string literals such as "Hello" should be invariant. In addition to literals, many programs refer to configuration data, state-transition tables, or numeric coefficients that should also be invariant. We usually speak of invariant data as being read-only, as opposed to variant data which is read-write.

In many desktop applications, the distinction between read-only and read-write data is logical rather than physical. The linker may place all the read-only data together in one data segment to facilitate program loading, but all the data winds up in RAM nonetheless.

In many embedded systems, the distinction isn't just logical. Most embedded programs don't load from disk in the sense that desktop applications do. Rather, an embedded program, including its read-only data, resides permanently in ROM. Clearly, read-write data can't also live in ROM; it must be in RAM. Therefore, a compiler must be able to distinguish the read-only data from the read-write data so it can place the former in ROM and the latter in RAM.

Actually, thinking in terms of just ROM vs. RAM is too limiting for many embedded systems. For example, some systems use a common control program for every model in a product line, and use different configuration data to customize the behavior of each model. These systems place code

I think most of us would be rather taken aback if calling `printf("Hello");` ever printed anything but "Hello."

and read-only data in different storage segments so they can place the data in a ROM separate from the code. Thus, although each model uses a different ROM for configuration data, all models in the product line can use the same ROM(s) for code.

Typical C and C++ compilers for embedded systems respond to these needs by mapping code and data into several logical segments:

- *Code* (also known as *text*): a read-only segment containing the code
- *Literal*: a read-only segment containing initialized data
- *Initialized data* (often called just plain *data*): a read-write segment containing data that's initialized as part of program startup
- *Uninitialized data* (often known as *bss*): a read-write segment containing data that remains uninitialized until the program actually uses it

A compiler and linker can also provide controls, in the form of command line switches or `#pragma` directives, that let you merge some logical segments into a single physical segment. For example, you might merge the literals with the code (in ROM) or with the initialized data (in RAM).

With this segmentation model, placing string literals into ROM is fairly easy. The compiler collects all the string literals into the literal segment. The linker and other back-end tools, with guidance from the appropriate command switches, place the literal segment in ROM.

Placing non-literal data in ROM is more of a problem. The compiler must be able to distinguish initialized read-only data from initialized read-write data. Clearly, uninitialized data can't be read-only. Uninitialized data is useless if you can never store a value into it, so it must be in RAM. It's not so obvious what to do with initialized data.

For example, when a compiler encounters a declaration such as:

```
unsigned char two_to_the[]
  = { 1, 2, 4, 8, 16, 32, 128, 256 };
```

how is it supposed to know whether `two_to_the` is read-only data or read-write data that just happens to have a specified initial value? In other words, how does the compiler know whether to place the data into the literal segment or into the initialized data segment?

By the way, even if `two_to_the` turns out to be read-write data in RAM, a copy of the initial values 1, 2, 4, and so on probably still appear in ROM. The program initializes `two_to_the` by copying the initial values from ROM to RAM during startup.

Most C development tools for embedded systems support the following technique for placing initialized data into ROM. When compiling a source file, the compiler places all the initialized data into one segment. By default, that segment is the initialized data segment. However, you can use a

compiler switch to make the compiler place the initialized data into the literal segment. Thus you can place selected data that should be read-only into ROM by collecting it into a separate source file and by compiling with the switch set to place all the initialized data from that file into the literal segment.

One problem with this technique is that it forces you to organize some parts of your program according to its physical, rather than its logical, requirements. Whereas you might want to place certain read-only and read-write data in the same source file along with the code that uses it, you can't. You must place the read-only data in a separate source file, which can make the program harder to read and maintain.

Many compilers provide pragmas as an alternative to using command switches. The pragmas let you place data from a single source file into different segments. For example, a compiler might let you write something like:

```
#pragma data("literal")
unsigned char two_to_the[]
  = { 1, 2, 4, 8, 16, 32, 128, 256 };
#pragma data()
```

The first pragma tells the compiler to place subsequent definitions for initialized data into the segment named `literal`. The second pragma tells the compiler to go back to placing initialized data into the default segment.

Unfortunately, compilers differ in their use of pragmas, sometimes dramatically. Both the C and C++ standards specify that pragmas exist, but they don't mandate that compilers support any particular pragmas. Therefore, code that relies on pragmas is rarely portable.

Another problem with both of the above techniques (compiler switches and pragmas) is that they don't provide compilers with a way to prevent writing into read-only data. A compiler should be able to issue a diagnostic for an assignment such as:

```
two_to_the[i] = 0;
```

but it can't. You probably won't notice the error until you run the program.

Enter the `const` qualifier. The `const` qualifier provides a way to declare read-only data so that compilers can also detect accidental attempts to modify that data. For example,

```
const unsigned char two_to_the[]
  = { 1, 2, 4, 8, 16, 32, 128, 256 };
```

defines `two_to_the` as an "array of read-only unsigned char." The `const` qualifier is part of the type of `two_to_the` and compilers use this type information to validate subsequent uses of the object. For instance, given the declaration just above, an assignment such as:

```
two_to_the[i] = 0;
```

is a compile-time error.

Simply declaring an object `const` is not enough to get it into ROM. You must still use the appropriate options on the linker and other back-end tools to get the data where you want it. The `const` qualifier is just a safer and more convenient way to get the process started.

Placing data in ROM is just one of several uses for the `const` qualifier. Thus, even with the appropriate support from back-end tools, using the `const` qualifier in a declaration does not assure that the declared object will actually wind up in ROM. The declaration must satisfy other semantic constraints. In the coming months, I'll describe those constraints and explain other uses for the `const` qualifier. 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 *(Plum Hall, 1991). You can write to him at dsaks@wittenberg.edu.*