RETURN

Dan Saks

Volatile Objects

For the past few months, I've been discussing the `const` qualifier, mostly with an eye on using `const` to place objects into ROM. I haven't said all I have to say about `const`, but part of what I have left involves the volatile qualifier, as well. So this month, I'll introduce you to the volatile qualifier.

The volatile qualifier can appear anywhere that the `const` qualifier can. Whereas `const` declares objects that the program can't change, volatile declares objects whose values might be changed by events outside the program's control. A typical example of a volatile object is a memory-mapped input/output (I/O) port.

Memory-mapped I/O

Most modern computers communicate with I/O devices using either memory-mapped I/O or port I/O. Memory-mapped I/O maps device registers into the conventional data space. To a C or C++ programmer, memory-mapped I/O registers look more or less like ordinary objects. That is, storing into a memory-mapped device register sends commands or data to a device; reading from a memory-mapped I/O register retrieves status or data from a device. This is the approach used in the Motorola 68K family of processors.

In contrast, port I/O maps control and data registers into a separate (often small) data space. Port I/O is similar to memory-mapped I/O except that programs must use special instructions, such as the in and out instructions on the Intel x86 processors, to move data to or from the device registers.

My focus here is on explaining the volatile qualifier rather than the details of I/O. To keep things simple, the remaining discussion uses only the memory-mapped model.

A typical hardware device often communicates through a sequence of device registers. Some registers communicate control and status information, while others communicate data. A given device may use separate registers for input and output. Registers may occupy bytes, words, or whatever the architecture demands.

The volatile qualifier declares objects whose value can be changed by events beyond the program's control. Volatile objects are useful for memory-mapped I/O ports.

The simplest representation for a data register is as an object of the appropriate size integer type. For example, you might declare a one-byte register as a `char` or a two-byte register as a `short int`. Then you can move data to a memory-mapped device by storing a value into its output data register, and retrieve data from that device by reading from its input data register.

A control/status register is not really an integer-valued object—it's a collection of bits. One bit may indicate that the device is ready to perform an operation, while another might indicate whether interrupts have been enabled for that device. A device might not use all the available bits in its control/status register.

One common technique for manipulating control/status registers is to use symbolic constants to represent masks for isolating bits, and use bitwise operators (like `&` and `|`) to set, clear, and test bits in registers. For example, you might represent a 16-bit control/status register as:

```
typedef short int control;
#define ENABLE 0x40
            /* enable interrupt */
#define READY 0x80
             /* device is ready */
```

You can then define:

```
control *const pc
  = (control *)0xFF70;
```

which uses a cast expression to initialize `pc` to point to the address of a memory-mapped control/status register at some fixed address. (Although C and C++ both allow cast expressions such as the one above that convert integers to pointers, the exact behavior of such casts varies across platforms.)

Once `pc` points to a memory-mapped control/status register, the program can communicate with the device by testing or setting the value of the register via `pc`. For example:

```
*pc &= ~ENABLE;
          /* clear enable bit */
```

clears the control register's enable bit, so the device will not trigger interrupts. A loop such as:

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

repeatedly polls (tests) the control register's ready bit until that bit is non-zero.

Many devices use control/status and data registers in tandem. The following declarations declare a bi-directional device (supporting both input and output) using a pair of registers for input and another pair for output:

```
typedef short int control;
typedef short int data;
#define ENABLE 0x40
#define READY 0x80

typedef struct port port;
struct port
  {
  control c;
  data d;
  };

typedef struct ioport ioport;
struct ioport
  {
  port in, out;
  };
```

Using the declarations above,

```
ioport *const pio
  = (ioport *)0xFF70;
```

declares `pio` to point to an I/O port, and

```
pio->out.c &= ~ENABLE;
```

disables output interrupts for that I/O port's output port.

An assignment such as:

```
pio->out.d = c;
```

sends the value of character `c` to the output device. It immediately clears the ready bit in the corresponding output control/status register to indicate that the device is busy.

The hardware automatically sets the ready bit when the device completes the current operation. That is, it sets the ready bit to indicate that the device is ready to start another operation. Thus, a loop such as:

```
while (p->out.c & READY == 0)
  ;
```

waits until the output device is ready to receive another character.

As a somewhat more complete example, a function such as:

```
void put(char const *s, ioport *p)
  {
  for (; *s != '\0'; ++s)
    {
    while (p->out.c & READY == 0)
      ;
    p->out.d = *s;
    }
  }
```

sends the characters in null-terminated string `s` to the output device controlled by `*p`.

Overly aggressive optimization

Using the previous declaration for `ioport`, here's a sequence of code for a rudimentary device driver that writes a `'\r'` (carriage return) and `'\n'` (newline or line feed) to the output device of an I/O port (inspired by P.J. Plauger's "Touching Memory: Standard C Makes The Act More Precise," *C Users Journal*, May 1988):

```
while (pio->out.c & READY == 0)
  ;
pio->out.d = '\r';
while (pio->out.c & READY == 0)
  ;
pio->out.d = '\n';
```

A compiler might not realize that `pio->out.c` is actually a device register whose value could change due to some external event such as the completion of an I/O operation. The compiler's optimizer might therefore conclude that either the ready bit in `pio->out.c` is always set, or the ready bit in `pio->out.c` is never set. In generating code, the compiler considers both possibilities.

If the ready bit is always set, the program never enters this loop:

```
while (pio->out.c & READY == 0)
  ;
```

If the ready bit is never set, the program never leaves the loop. In either case, there's no need to test the condition more than once. The optimizer transforms the loop into:

```
if (pio->out.c & READY == 0)
  for (;;)
    ;
```

which tests the condition only once, and then either loops forever or never loops at all.

After this optimization, the driver code looks like:

```
if (pio->out.c & READY == 0)
  for (;;)
    ;
pio->out.d = '\r';
if (pio->out.c & READY == 0)
  for (;;)
    ;
pio->out.d = '\n';
```

Again, as far as the compiler can tell, the ready bit is either always set or never set. If it's always set, then the program skips both loops. If it's never set, the program falls into the first loop and never escapes. In either event, the program never executes the second loop. That loop is dead code, and the optimizer can eliminate it.

After this optimization, the driver code looks like:

```
if (pio->out.c & READY == 0)
```

```
  for (;;)
    ;
pio->out.d = '\r';
pio->out.d = '\n';
```

As far as the compiler can see, the second assignment overwrites the result of the first assignment. Therefore, only the last assignment is worth keeping. The final "optimized" code looks like:

```
if (pio->out.c & READY == 0)
  for (;;)
    ;
pio->out.d = '\n';
```

It does the wrong thing, but much more efficiently!

The problem here is that a memory-mapped I/O port isn't an ordinary object in RAM. With an object in RAM, a compiler can assume that if the program places a value in the object, the value will remain in that object until the program places a different value there. Thus, a compiler can "optimize away" references to the object if the compiler can determine that the value hasn't changed since the last reference.

With a memory-mapped port, a compiler can't make the same assumption. The value in a port may change spontaneously. If the program has an expression that looks at the port's value, the compiler must generate code that actually fetches the value. It can't optimize away those fetches.

The volatile qualifier

In days gone by, programmers often solved this problem by placing device driver code in a separate source file. You had to compile that one file with optimizations turned off, but then you could compile the rest of the program with optimizations turned on.

Some compilers offer pragmas that will turn off compiler optimizations for a portion of a source file. You could then wrap the driver code inside a pair of pragmas, as in:

```
#pragma optimization = off
/* driver code goes here*/
#pragma optimization = on
```

You had to hope you turned optimizations off in just the right places.

The volatile qualifier eliminates the guesswork. It identifies objects, such as memory-mapped ports, that may be changed by events that compilers cannot detect. For example,

```
ioport volatile *const pio
  = (ioport *)0xFFA0;
```

declares that `pio` has type "`const` pointer to a volatile `ioport`." Thus, `*pio` is an object whose value may change spontaneously. This tells the compiler that it shouldn't "optimize away" references to `*pio`, even if it appears that the value of `*pio` hasn't changed since the last reference.

The volatile qualifier prevents optimizations only for accesses to volatile-qualified objects. It does not inhibit any other optimizations.

An object can be both `const` and volatile. For example, given:

```
iport volatile const *pi = ...;
```

then `*pi` is a `const` volatile `iport` object. The program can't write to `*pi`, but it must act as if `*pi`'s value might change spontaneously. This is typical behavior for an input port.

Of course, there's more to the volatile qualifier than what I've covered in this brief discussion. Although I will resume my discussion of `const` in upcoming articles, I will fill in details on volatile as well. **esp**

Dan Saks is the president of Saks & Associates, and a contributing editor for the C/C++ Users Journal. *Write to him at dsaks@wittenberg.edu.*