

LESSONS FROM THE TRENCHES

George Martin

Everything Changes

Using the Const Modifier

Sometimes we have the knowledge, but don't utilize all the tools we have available to us. George looks at the forgotten modifier beyond *char*, *int*, *long*, and *float* for writing code in C. Remember the often overlooked *const* qualifier? Well, it can be used to ensure that the data won't be modified during execution, eliminating unexpected changes.



Although it's true that change is inevitable, quite often in embedded software design you want to specify some things as constant. Certainly you don't want any unexpected changes. If you are writing code in C, you are no doubt using *char*, *int*, *long*, and *float* to tell the compiler about your data size and representation. There is another type modifier (or qualifier) that I'll bet you know about and don't use. My New Year's resolution is to use the *const* modifier, and then I'll let the compiler watch out on my behalf.

The *const* modifier declares that the data will not be modified during execution. For example:

```
int a = 5;
// a is an integer type
// set to 5
const int b = 10;
// b is also an integer
// type set to 10
```

```
and won't change
int const b = 10;
// identical in meaning to line
above
```

Both *a* and *b* are variables and have initial values. But, *a* can be modified by the program, whereas *b* cannot. If you go on to write:

```
a = a+7; // increment this
by 7
b++; // increment this
by 1
```

the compiler will flag the second statement, which alters the variable *b*. This example is obvious and probably too simple to be much help unless there are several designers working on different modules for the same project. But, I bet you have a listing that looks something like Listing 1.

I hate questions like, "What do you think this code does?" So I won't ask. *FLINFO* is a structure describing and characterizing the flash memory devices, *DevTable[]* is an array of all the devices you might find soldered on the board, and *FLinfo[MAX_FLASH_DEV]* is a table of all the devices actually found on the board.

As I search and find flash memory devices, I copy the data from *DevTable* into *FLINFO* and update the *BaseAddr*. I don't want *DevTable* to change as I run the program, so it's qualified and *const*. And, any compiled code that would modify any of the *DevTable* values would be flagged at compile time. In fact, if your code has RAM and ROM, the compiler probably will place *DevTable* in ROM.

If you work on larger projects, no doubt you've come across the situation where you've designed a data structure and another programmer is needed to view your data, perhaps to write a report or to log the data. That other

programmer or code module should not be modifying your data. What to do?

DEFINE AND QUALIFY

How about defining a pointer to your data and qualifying that pointer (see Listing 2).

Pointer *p1* can be modified to point to other places, and the contents of *MyData* can be read and written using *p1*. And, pointer *p2* can be modified to point to other places, but because *p2* is pointing to a type *const*, the contents of *MyData* can only be read using *p2*.

If you passed another module *p2* (see Listing 3), that module could read that data until the cows came home but would not be able to modify it. Even if you are the only person working on the project, this would probably be a wise step to take.

I would like to pass another module a pointer to a patient data structure like described in Listing 3. I would like that module to calculate the patient's body mass index because it's a new medical development, and I don't know how to calculate that value. These are two extremely poor reasons but just pretend those arguments work. I want this other module to read the height and weight and write the body mass index (see Listing 4). I don't want them to alter anything except the body mass index.

MyDataBMI is a pointer to a new structure, one where some of the members are *const* and others are not. You could give this pointer and structure definition to the world and not get hurt. So, why not use the *const* modifier? It only feels a bit uncomfortable until you get used to it.

IN OTHER WORDS

While we're here, what other keywords might be of the same sort of use? Volatile comes to mind. The volatile modifier warns the compiler that there may be some special hardware behind the variable causing it to unexpectedly change (see Listing 5).

In Listing 5, two variables, *a* and *b*, are declared. *a* is set to 7 and then tested. A good compiler would know that *a* is in fact equal to 7, so no code would be generated for the test and

Listing 1

```
/**
 *
 * The fl_status() routine can return the following
 * information about flash memory.
 *
 */
typedef struct { /* Flash Memory information structure */
    INT16 MfrID; /* manufacturer ID code */
    INT16 DevID; /* device ID code */
    BYTE MaxErase; /* max erase time in seconds */
    BYTE MaxWrite; /* max write time in msec */
    INT16 NumSect1;
    /* number of sectors per chip 1st type */
    INT32 SectorSize1; /* bytes per sector 1st type */
    INT16 NumSect2;
    /* number of sectors per chip 2nd type */
    INT32 SectorSize2; /* bytes per sector 2nd type */
    INT16 NumSect3; /* number of sectors per chip
    3rd type */
    INT32 SectorSize3; /* bytes per sector 3rd type */
    INT16 NumSect4; /* number of sectors per chip
    4th type */
    INT32 SectorSize4; /* bytes per sector 4th type */
    INT16 *BaseAddr; /* Starting Address of device */
    INT32 TotalSize; /* total number of bytes */
    BYTE *pDevName;
    /* pointer to manufacturer & de vice name */
} FLINFO;
//
// Put the known device types here.
//
const FLINFO DevTable[] = {
/*MfgrID DevID Erase Write No.Sect SectSiz Base Addr
TotSize
DevNameString */
{ 0x89, 0x88c2, 0,0, 31, 0x1000L,
8, 0x0200L,
0, 0x0000L,
0, 0x0000L, 0x0L,
0x20000L,
"INTEL 28F160C3"
},
{ 0x89, 0x4471, 0,0, 1, 0x0200L,
2, 0x0100L,
1, 0x0C00L,
3, 0x1000L, 0x0L,
0x4000L,
"INTEL 28F400B5B"
}
};
//
// Create an array of the structure for up to
// MAX_FLASH_DEV number of devices
//
FLINFO FLinfo[MAX_FLASH_DEV];
// Put the devices we find in this table
```

Listing 2—Pointer p1 points to the structure MyData, and pointer p2 points to the structure const MyData

```
/**
 *
 * This data structure is filled in by my code
 *
 ***/
typedef struct {                // Some structure */
    INT16  FirstParameter;      // Stuff
    INT16  SecondParameter;     // More Stuff
    INT16  LastParameter;       // End of my stuff
} MyData;
MyData *p1;
const MyData *p2;
```

Listing 3—This is another example using two pointers.

```
/**
 *
 * Part of this data structure is filled in by my code
 *
 ***/
typedef struct {                // Patient Data Structure
    INT8   *Name[32];           // Name
    INT32  HeightInInches;      // Height
    INT32  WeightInPounds;      // Weight
    INT32  BodyMassIndex;       // A new parameter form the
                                // industry
} PRec;
PRec *p1;
const PRec *p2;
```

Listing 4—Here are pointers to constant data.

```
/**
 *
 * Part of this data structure is filled in by my code
 *
 ***/
typedef struct {                // Patient Data Structure
    const INT8   *Name[32];     // Name
    const INT32  HeightInInches; // Height
    const INT32  WeightInPounds; // Weight
    const INT32  BodyMassIndex; // A new parameter form the industry
} PRec;
PRec *MyDataBMI;
```

Listing 5—Take a look at a volatile type of modifier.

```
/**
 *
 * A Volatile Example
 *
 ***/
INT16 a;
volatile INT16 b;
a=7;
if(a==7) {
    print("a=7");
}
b=9;
if(b==9) {
    print("b=9");
}
```

the `print()` statement would always be executed. The variable `b` is declared as volatile, so even though it was just set to 9, `b` would be loaded and tested before the `print()` statement is reached. If `b` was the serial port status register and `a` was the serial port control register, in a memory-mapped scheme they could both be at the same address—one a read variable and one a write variable. There is hardware behind that common address that could change the values, so always reload it.

Although this example may be obvious, perhaps you can go up another level of optimization on the compiler level. George started his career in the computer hardware industry in 1980. After five years and a few products under his own and cofounded a design and manufacturing firm. Typical systems that George designs include servo-motion control, graphical input and output, data acquisition, and remote control. George is a charter member of the Ciarcia Design Works Team and most recently, he's been working on the people-tracking system for Bill Gates' new house. You can reach him at george.martin@worldnet.att.net.

Circuit Cellar, the Magazine for Computer Applications. Reprinted by permission. For subscription information, call (860) 875-2199, subscribe@circuitcellar.com or www.circuitcellar.com/subscribe.htm.