

august 1999 www.embedded.com volume 12, number 8

Embedded Systems

P R O G R A M M I N G

Nuts to OOP!

One Programmer's
Perspective

GUI Development

Handling **Xmit Enable Lines**

Using Functions to Represent **Transients**

Saks on **Name Mangling**

Special Report: **16-Bit MCUs**



A MILLER FREEMAN PUBLICATION

THOMAS NIEMANN

Nuts to OOP!

Seasoned programmer Thomas Niemann thinks the increasingly widespread use of object-oriented programming in embedded development is largely unwarranted. *ESP* technical editor Michael Barr disagrees.



For the last 25 years I have worked as a software engineer, specializing in compilers and embedded systems. As I enter my geezerhood, I'm noticing a shift toward C++, and object-oriented programming (OOP) in general. But after maintaining several systems designed this way, I have one comment to make: nuts! Perhaps elaborating a bit would make my argument more persuasive.

For some perspective, I'll briefly discuss my background. Then I'll present a C++ class from actual code used in an industrial controller. This code happened to have a bug that surfaced only when we changed compilers. I'll detail the discovery of the bug and follow this with an implementation of the same code without objects. While you can't make generalizations based on one test case, many of the features of OOP that I find objectionable are well illustrated by this simple example.

Background

In 25 years of programming I've seen a lot of code. This doesn't necessarily mean that I'm a good programmer—it just means I've seen a lot of code. This distinction is important, and I'll mention it again in the conclusion.

One of the largest projects in which I ever participated involved writing the code generator for a COBOL compiler. I was the technical leader for this project, and a total of eight man-years of effort went into the solution. At the end of the project we received positive feedback from three places:

- *A customer.* Our first customer arrived from France with a reel of COBOL programs under his arm. After several days of testing, we found one bug. That bug was isolated and fixed, within two hours, by someone who was not familiar with the particular section of code
- *A developer.* One of the developers came onto the project toward the end to maintain the code. He stopped me in the hall one day to tell me that he found the code exceptionally easy to follow
- *The project manager.* Our manager gave each developer a substantial bonus

As I enter my geezerhood, I'm noticing a shift toward C++, and object-oriented programming in general. But after maintaining several systems designed this way, I have one comment to make: nuts!

Oh yes, I almost forgot to point out that we developed this software with procedural code. Not one object was used. I mention this to give you some perspective, and to point out that it's possible to develop high-quality large systems without objects. I've seen it done.

An object-oriented example

The following OOP example is derived from a real piece of software that downloads 16-bit values, via dual-port memory, to a device. Here's the code that downloads the value `dval`:

```
DEVICE *device;
...
device->download = dval;
```

To find out what's happening underneath, we examine the `DEVICE` class in file `device.h`:

```
class DEVICE: {
public:
...
    Register<volatile unsigned
        short> upLoad;
    Register<volatile unsigned
        short> downLoad;
...
}
```

You'll notice that `download` is a `Register`, defined in file `reg.h`:

```
template <class T> class Register {
public:
    Register(unsigned int address) {
        data = (T*)(address); }
    T& operator=(T value) { *data =
        value; return *data; }
    T& operator|=(T value) { return
        *data |= value; }
    T& operator&=(T value) { return
        *data &= value; }
    T& operator^=(T value) { return
```

```
    *data ^= value; }
    T& operator<<=(int bits) {
        return *data <<= bits; }
    T& operator>>=(int bits) {
        return *data >>= bits; }

private:
    Register();
    T *data;
}
```

The `Register` class defines its default constructor as private, forcing us to use the constructor with an address parameter. This is good practice, as registers don't make much sense unless they're tied to an address. The address is assigned to a private data member on construction, thus protecting this value. The remaining member operators allow operations on the register. Note the assignment operator returns the value assigned. This allows for chained assignments.

To find out where the register is mapped, we need to examine the initializer for the constructor of `download`. This is found by examining the `DEVICE` class constructor, in file `device.cpp`:

```
DEVICE::DEVICE(unsigned int base) {
....
    upLoad(base + 0x00000020);
    downLoad(base + 0x00000022);
....
}
```

Here we see that that `download` is constructed with a value of `base + 0x00000022`. The value of `base` is found in the instantiation of `download`, in file `main.cpp`:

```
DEVICE *device =
    new DEVICE(0x80000000);
```

Putting it all together, we're able to

determine that the `download` register is at location `0x80000022` in physical memory. Notice how we had to look all over the place to discover this simple fact. This is one of the disadvantages of information hiding, a basic tenet of OOP. While I'm not proposing we expose everyone's underwear, the fact remains that in a developing system, we often don't know the source of a bug. It can just as easily be in our source or the source of a function we utilize. Thus, rather than being helpful, information hiding is a hindrance.

Uncovering a bug

Wouldn't you know it—even in this simple example, there's a bug. This bug turned out to be rather serious, and was worked on by three programmers before I tracked it down. It took me three days to isolate the problem, making several dead-end attempts before arriving at the solution. I'll eliminate the dead ends, so it should take you less than three minutes to understand the explanation.

When the application begins execution, it initializes the device. During initialization, a dialog box indicated that one of the device commands had failed. Diagnostics, stored in an error file, also showed the device had interrupted us with bad news. On the first day, I determined that nothing was wrong with the command, and that the interrupt had occurred for a previous action. The second day, I peppered the source with print statements, and narrowed the problem down to an assignment statement that downloaded state information to the device. The third day was spent debugging that statement:

```
1 device->download = dval;
2 movzwl %di,%eax
```

```

3  pushl %eax    ; dval
4  leal 276(%esi),%eax
5  pushl %eax   ; this pointer
6  call __as__t8Register1ZUVsUs
7  addl $8,%esp
8  movl %eax,%edx ; EDX = pointer to data
9  movw (%edx),%ax ; read *data
10 movb 1(%ebp),%dl
11 movb %dl,%al
12
13 T& operator=(T value) { *data
  = value; return *data; }
14 __as__t8Register1ZUVsUs
15 pushl %ebp ; entry sequence
16 movl %esp,%ebp
17 pushl %esi
18 pushl %ebx
19 movl 8(%ebp),%ecx ; this
  pointer
20 movl 12(%ebp),%edx ; EDX has
  dval
21 movl %edx,%ebx ; EBX has dval
22 movl (%ecx),%eax ; EAX =
  pointer to data
23 movw %bx,(%eax) ; move dval
  to *data
24 movl (%ecx),%esi ; ESI =
  pointer to data
25 movl %esi,%eax ; EAX =
  pointer to data
26 leal 8(%ebp),%esp ; return
  sequence
27 popl %ebx
28 popl %esi
29 leave
30 return

```

When the C++ statement in Line 1 was executed, the device generated an interrupt indicating it wasn't happy with the assignment. Using a debugger, I checked the pointer and `dval` to ensure all was okay. It was. I then tried a simple C solution, which you'll see in the next section. This tactic worked adequately. I concluded that the new C++ compiler must have changed the behavior of Line 1. Let's see just how.

Lines 2 to 6 push `dval` and the `this` pointer on the stack and call the assignment operator. Lines 15 to 18 are the entry sequence for the assignment operator. After several instructions, Line 23 does the actual move of

`dval` to the memory-mapped location. We then return to the caller, at Line 7. Notice that on return, we're left with the data pointer in `EAX`. This is placed in `EDX` at Line 8, and then dereferenced at Line 9. Dereferenced! Whoa, Betsy! There's our bug.

Recall that we're trying to write to the device. The actual piece of hardware has a write-only register, and attempting to read from this location is strictly prohibited. Let's take another look at the assignment operator code:

```
T& operator=(T value) { *data =
value; return *data; }
```

After we assign `value`, we return `*data`. This was done so we could chain assignment operators. The new C++ compiler interpreted this as a requirement to actually read the value stored at that location. This behavior is correct, since `data` was declared `volatile`. The solution is to rewrite the operator and return a `Register`, or `*this`. This bug surfaced when we changed to a more recent release of the GNU C++ compiler. The older version of the compiler failed to dereference the return value, while the newer version correctly loaded the value, uncovering a mistake in our coding.

Let's make a rough estimate for the cost of this bug. Since four programmers worked on it for three days, we have about 100 hours of effort. Assuming a rate of \$100/hour, we're talking about a \$10,000 bug!

Without objects

In an attempt to find the bug, I recoded the assignment statement in C. First I defined a `Register` in file `device.h`:

```
typedef volatile unsigned short
int Register;
extern Register* const
deviceDownload;
```

The register is initialized in file `device.c`:

```
Register* const deviceDownload =
```

```
(Register *)0x80000022;
```

A reference to the register is done as follows:

```
*deviceDownload = dval;
movl deviceDownload,%eax
movl dval,%edx
movw %edx,(%eax)
```

To trace a reference, scan the include files for `deviceDownload`. Once you've found it is declared in `device.h`, you'll know the definition is in `device.c`. Of course, grouping all the device variables and functions in `device.c` and prefixing them with "device" also helps.

At this point, several things should be apparent. The bug in the C++ implementation is not present in this version. The other thing you should notice is that this version is a lot simpler and \$10,000 cheaper.

Myths and facts

Let's examine some of the myths associated with OOP:

- *Objects are needed to protect data.* In fact, this is the sole purpose of the class in my example. By implementing the class with a private member for the pointer, the pointer was protected. Otherwise, it's just a global variable, available for anyone to clobber. I balance these comments with my experience. Frankly, I haven't had problems with global variables. I do keep the number of global variables to a minimum, but when they make sense, I use them. Employing member functions to access variables is simply being overprotective, adds unnecessary code, and requires the programmer to follow extra steps of indirection to determine exactly what the code does. I find this frustrating because every time I do it, I'm reminded of how much easier this would have been in C
- *Objects are needed to group data and procedures.* Just put related functions in a file (`device.c`). If you

want, use a common prefix (device) to the functions. Then place their prototypes in an include file (device.h). To find a function, simply scan the include files for the declaration to determine the corresponding source file that contains the function

- *Objects are needed when implementing large programs.* No objects were used in the code generator project that I told you about at the beginning of the article. That was a large project
- *Objects are easier to maintain.* Because information is hidden, as is the wont of OO programmers, finding what you need to know takes more time. This fact was illustrated in the previous example, where we had to examine four files before we understood the assignment statement
- *The switch statement is bad.* Ever read one of those introductory C++ books that describes a graphics interface? You typically have a Shape class that's inherited by Circle, Square, and Triangle classes. The upshot of all this is that you can avoid those horrible switch statements by using inheritance and polymorphism. You can also avoid pimples by chopping off your head. There's nothing wrong with switch statements. I've never had problems with too many switch statements. So what's your beef?

Well, if those are the myths, I bet you're wondering what the facts are. Try these:

- *Object-oriented programming requires more time, up front, doing design work.* I don't think I'll get much argument here. Even OO folks agree with this.
- *Object-oriented programming requires more time to code a solution.* This follows from the previous statement. Let me quote from one of the Microsoft Help files that accompany Visual Studio 6 (Providing Polymorphism): "More seriously, an overemphasis on inheritance-driven polymorphism typically

results in a massive shift of resources from development tasks to upfront design tasks, doing nothing to address development backlogs or to shorten the time before the end user can discover—through hands-on experience—whether the system actually fulfills the intended purpose."

- *Object-oriented programming results in a more complex solution.* If you don't believe me, ask the engineer maintaining your code after you've started working on something else
- *Object-oriented programming results in code that is more error-prone.* This follows from the previous point. More complexity yields more errors
- *Object-oriented programming results in increased maintenance costs.* Again, this results from increased complexity and information-hiding

When to use objects

I'm not advocating that we never use objects. I do advocate their use when they provide new functionality, or the objects have been proven correct. For example, ActiveX components have the ability to supply an interface for more than one language platform, and may provide much-needed functionality for inter-language communication. Objects derived from the Standard Template Library have been used by thousands of programmers, and tested millions of times. This fact effectively removes them from consideration when bug-tracing.

In a developing system, bugs may exist in the code being examined or any of the user-coded objects that support this code. In other words, I think it's okay to use a vendor-supplied string class, but I suggest you don't try to write one. I definitely suggest you refrain from basing your entire software architecture on the class idiom.

A conspiracy of theorists

I know words like *inheritance*, *polymorphism*, and *information-hiding* sound good to the ears of programmers and their managers. However, I believe these constructs need closer scrutiny. I

see a new generation of programmers coming on the scene, armed with C++ and objects, ready to take on the world. Unfortunately, they're lacking one thing: a knowledge that procedural programming is simpler and works.

The myth that using objects results in code that is easier to maintain is perpetuated by instructors who frequently have little or no experience in procedural programming. It is further perpetuated by a mountain of books that are published every year proclaiming the advantages of OOP. Because object-oriented solutions are more complex, publishers are guaranteed a captive audience, and big sales, for their books. A sense of the magnitude of the problem may be found by counting the number of posts, in a single day, to newsgroup comp.lang.c++. Under these circumstances, one can hardly fault beginning programmers for admiring the emperor's new clothes and their failure to deliver software in a timely fashion.

In the early '70s I was a student at the University of Michigan. I was putting the finishing touches on a programming assignment when another student entered the room with a big grin, and said "I got it working with no GOTOs!" I smiled. Why would anyone want to do the assignment with one hand tied behind their back? I mean, the assignment was hard enough already. Next semester, I took a compiler class. We were using William M. McKeeman's book, *A Compiler Generator* (McKeeman, J.J. Horning, and D.B. Wortman, Prentice-Hall, Englewood Cliffs, NJ, 1970), that had a listing several thousand lines long in the appendix. It was written in XPL, a subset of PL/1, and there was one GOTO. As I started working on the assignment, I noticed how easy it was to understand and modify the code. I became an instant convert. This took two days. Later I learned about structured programming and control constructs, and understood why it worked. I have maintained C++, written by others, off and on for five years. I'm still waiting for that light to turn on.