

# One at a Time, Please

*by Stephen C. Dewhurst*

The way programmers grouse about maintenance, you'd think they'd try to produce the minimum number of programs required to solve a problem. Yet, in case after case, it's common to find teams of otherwise intelligent programmers writing and maintaining many more programs than necessary. This column discusses how that happens and what one can do about it. Note that these problems and solutions definitely fall under the rubric of "Common Knowledge," but that knowledge is so commonly ignored that it bears repeating.

## One Program or Two?

How do we insert debugging code into our programs? Simple, everyone knows we use the preprocessor.

```
void buggy() {
#ifdef NDEBUG
    // some debugging code...
#endif
    // some actual code...
#ifdef NDEBUG
    // more debug code...
#endif
}
```

Everyone's wrong. Most veteran programmers have long and tedious stories about how the debug version of a program worked perfectly, but "simply" defining `NDEBUG` caused the production version to fail mysteriously. Well, there's nothing mysterious about it. We're actually discussing two unrelated programs that happen to be generated from the same source files. You'd have to compile the same source code twice to see if it's even syntactically correct. The correct way to write the code is to dispense with the idea of a debugging version and write just a single program.

```
void buggy() {
    if( debug ) {
        // some debugging code...
    }
    // some actual code...
    if( debug ) {
```

```
        // more debugging code...
    }
}
```

What about the problem of all the debug code remaining in the executable of the production version? Isn't that going to waste space? Aren't the unnecessary conditional branches going to cost time? Not if the debug code isn't present in the executable. Compilers are very, very good at identifying and removing unusable code. They're a whole lot better at this than we are with our pathetic `#ifdefs`. All we have to do is make things easy for the compiler.

```
const bool debug = false;
```

The expression `debug` is what the standard calls a “constant-expression,” and what used to be called an “integral-constant-expression.” (Don't ask me where the “integral” went; I have no idea.) Every C++ compiler must be able to evaluate constant-expressions like this at compile time in order to translate array bound expressions, case labels, and bitfield lengths. Every minimally competent compiler can perform elimination of unreachable code of the form

```
if( false ) {
    // unreachable code...
}
```

Yes, even the compiler you've been complaining about to your management for the last five years can handle this. Even though the compiler removes the unreachable code, it must still perform a full parse and static semantic check. Given the definition of constant-expression in the standard, your compiler can even eliminate unreachable code that is guarded by more complex expressions, provided that the expression is a constant-expression that can be evaluated at compile time.

```
if( debug && debuglvl > 5 && debugopts&debugmask ) {
    // potentially unreachable code...
}
```

Your compiler may even perform the code elimination in more complex cases. For example, we might attempt to involve my favorite inline function in the conditional expression.

```
typedef unsigned short Bits;
inline Bits
repeated( Bits b, Bits m )
    { return b & m & (b & m)-1; }

//...

if( debug && repeated( debugopts, debugmask ) ) {
    // potentially unreachable code...
    error( "One at a Time, Please" );
}
```

However, with the use of a function call (whether inline or not), the expression is no longer a constant-

expression, we have no guarantee that the compiler will be able to evaluate it at compile time, and therefore the code elimination may not take place. If you require the code elimination, this approach is not portable. Some readers who have been programming in C for too long may suggest the following fix:

```
#define repeated(b, m) ((b) & (m) & ((b) & (m)) - 1)
```

I'm not even going to comment on this, except to note that it's a good thing you've got debugging code available. You'll need it.

Note that it may be advisable to have some conditionally compiled code in an application, in order to be able to set the values of constants from the compile line.

```
#ifndef NDEBUG
    const bool debug = false;
#else
    const bool debug = true;
#endif
```

Even the presence of this minimal conditionally compiled code is not necessary, however. A generally better approach is to select between debug and production versions in a makefile or similar facility.

## Using #if for Portability

“However,” you state with a knowing look, “my code is platform-independent. I have to use #if to handle the different platform requirements.” To prove your point, such as it is, you display the following code.

```
void operation() {
    // some portable code...
#ifdef WIN32
    // do something...
    a(); b(); c();
#endif
#ifdef UNIX
    // do same thing...
    d(); e();
#endif
}
```

This code is not platform-independent. It's multi-platform dependent. Any change to any of the platforms requires not only a recompilation of the source, but actual change to the source for all platforms. We've achieved maximal coupling among platforms; a remarkable achievement, if somewhat impractical.

But that's a minor annoyance compared to the real problem that lurks inside this implementation of `operation`. Functions are abstractions. The `operation` function is an abstraction of an operation that has different implementations on different platforms. When we use high-level languages, we can often use the same source code to implement the same abstraction for different platforms. For example,

the expression `a = b + c` where `a`, `b`, and `c` are `ints` has to be rendered in very different ways for different processors, but the meaning of the expression is sufficiently close across processors that we can (generally) use the same source code for all platforms. This isn't always the case, particularly when our operation must be defined in terms of operating system or library specific operations.

The implementation of `operation` indicates that the “same” thing is supposed to happen under both supported platforms, and this may even be the case initially. Under maintenance, however, bug reports tend to be reported and repaired on a platform-specific basis. Over a breathtakingly short period of time, the meaning of `operation` on different platforms will diverge, and you really will be maintaining totally different applications for each platform. Note that these different behaviors are different *required* behaviors, because users will come to depend on the platform specific meanings of `operation`. A correct initial implementation of `operation` would have accessed platform dependent code through a platform independent interface.

```
void operation() {
    // some portable code...
    doSomething(); // portable interface...
}
```

In making the abstraction explicit, it is far more likely that, under maintenance, different platforms will remain in conformance with the meaning of the operation. The declaration of `doSomething` belongs in the platform independent portion of the source. The various implementations of `doSomething` are defined in the various platform dependent portions of the source (if `doSomething` is inline, then it will be defined in a platform specific header file). Selection of platform is handled in the makefile. No `#ifs`. Note also that adding or removing a particular platform requires no source code changes.<sup>1</sup>

## What About Classes?

Well, what about them? Like a function, a class is an abstraction. An abstraction has an implementation that can vary either at compile time or runtime depending on its implementation. As with a function, use of `#if` for varying a class's implementation is fraught with peril.

```
class Doer {
#   if ONSERVER
    ServerData x;
#   else
    ClientData x;
#   endif
    void doit();
    // ...
};

void Doer::doit() {
#   if ONSERVER
    // do server things...
#   else
```

```
    // do client things...
#   endif
}
```

Strictly speaking, this code is not illegal unless the `Doer` class is defined with the `ONSERVER` symbol both defined and undefined in different translation units. But sometimes it would be nice if it were illegal.<sup>2</sup> It's common for different versions of `Doer` to be defined in different translation units and then linked together without error. The runtime errors that appear are unusually arcane and difficult to track down.

Fortunately, this technique for introducing bugs is not now as common as it once was. The most obvious way to express variation of this kind is to use polymorphism.

```
class Doer { // platform independent
public:
    virtual ~Doer();
    virtual void doit() = 0;
};
class ServerDoer : public Doer { // platform specific
    void doit();
    ServerData x;
};
class ClientDoer : public Doer { // platform specific
    void doit();
    ClientData x;
};
```

If more flexibility is required, then the implementation of `Doer` can use a `Bridge`.

```
class Doer {
public:
    Doer();
    ~Doer();
    void doit();
private:
    DoerImpl *impl_;
};
```

In those very rare cases in which even a simple virtual function call is too expensive, templates may be used to implement what might be called “compile time polymorphism.”

```
template <class T>
class Doer;
```

```
template <>
class Doer<ServerData> {
public:
    void doit();
private:
    ServerData x;
};
```

```
template <>
class Doer<ClientData> {
public:
    void doit();
private:
    ClientData x;
};
```

In any case, the structure of each “Doer” class itself is invariant from platform to platform, and variations in behavior can be relegated to platform specific source files.

## Reality Check

We’ve looked at some fairly simple manifestations of attempts to make a single source represent different programs. From these simple examples, it looks like a straightforward task to re-engineer the source code to be more maintainable through application of the idioms and patterns illustrated above.

Unfortunately, the reality is often far worse and far more complex. Typically, the source is not parameterized by a single symbol (like `NDEBUG`), but is subject to several different symbols, each of which may take on a number of values, and which may be used in combination. As we illustrated above, each combination of symbols and symbol values gives rise to an essentially different application with different required, abstract behaviors. From a practical standpoint, even if it is possible to tease apart the separate applications defined by these symbols, re-engineering will unavoidably result in a change in behavior of the application on at least one platform.

However, such re-engineering eventually becomes necessary when the abstract meaning of the program can no longer be easily determined, and when many hundreds of compilations with different symbol settings are required simply to determine whether the source code is syntactically correct. It’s far better to avoid the use of `#if` for versioning of source.

*end of article*

---

<sup>1</sup> Many of the most commonly used design patterns speak to similar concerns, especially Abstract Factory, Bridge, and various augmentations of Factory Method (Gamma et al., *Design Patterns*, Addison Wesley 1995).

<sup>2</sup> Yes, this is a real example (simplified) from one of my clients. And yes, we fixed it before it became dangerous.