

Case Study: Converting C Programs to C++

Jerry Fitzpatrick

C++ is most effective when applied to new software projects that use an object-oriented design from the start. However, many C programs can benefit from object-oriented extensions even when a complete rewrite is impractical. By converting a C application to compile using a C++ compiler, object-oriented elements can be introduced incrementally.

Over the last several months I converted a medium-sized (34 KLOC) graphics adapter test program written in C to C++ and then implemented several new features using C++ classes. As you know, C++ was designed to be backward compatible with C, but my experience with this graphics test program illustrates that the conversion process can be tricky.

Most C code was not written in anticipation of C++, and C compilers are more lax in flagging potential problems in areas such as type checking and function prototyping. Anyone converting legacy C code is likely to experience many of the same problems, so the solutions I describe here should be applicable to most programs. This article briefly outlines the application I converted, analyzes the key problems I encountered, and explains the techniques I use to overcome these problems.

Background

The program I converted from C to C++ tests graphics adapters for IBM-PC compatible computers. These tests exercise various video resolutions and color depths, the color palette, hardware cursor, and many other graphics functions.

When the test program was originally written seven to ten years ago, most computers used character-based CGA or EGA graphics. Pixel-based graphics became more common after VGA adapters became widely used. Creating test programs for these adapters was relatively easy since most manufacturers conformed to the CGA, EGA, and VGA standards. Graphics operations could also be performed using system BIOS calls, minimizing the need for low-level adapter programming.

The authors of the original program had decided that one program could support all three types of adapters. Although the application is compiled as an MS-DOS program, it is linked with special code that allows it to communicate with a proprietary front-end shell. Unfortunately that also prevents it from being run as a standalone program.

In addition, the test program is not allowed to use DOS device drivers. Because DOS programs can only access the first megabyte of system memory, every installed driver uses memory that could otherwise be used for the test program itself.

Although this strategy is restrictive, it worked fairly well until a new generation of adapters came along. Visually oriented programs fueled a demand for higher-performance graphics boards. In the push to market, manufacturers provided higher speeds and additional capabilities without the benefit of industry standards. As a result these adapters are often controlled very differently from one another.

The biggest differences relate to the high-resolution graphics modes, hardware cursor, line and polygon drawing, and bit-blitting (used to copy areas of the screen at high speed). These functions are quite sophisticated and require a significant amount of support code. As a result the test program started to grow considerably.

My client had originally hired me to add support for a new adapter. The engineer who had been working on it had been promoted to management, leaving no one to continue development. Yet the demand for features and new adapter test support continued to grow rapidly.

After examining the code it was obvious that it had become bloated over the years and was sorely in need of rework. Having read about the benefits of object-oriented programming, several staff members favored experimenting with object-oriented design techniques and the C++ programming language.

Table 1 – Initial Application Statistics

	C Language	Assembly Language
Number of source files	64	15
Number of header files	72 (a)	6
Source lines of code	27,676	6675 (b)
LOC per file (mean)	432	445
LOC per file (standard deviation)	330	243

(a) not including standard library headers
(b) not including header file LOC

When we first considered using C++ the program had about 34,000 lines of code, and an executable size of about 190K bytes (see Table 1). The front-end program has an executable size of about 250K and allocates additional heap memory for its services. For this reason, programs larger than about 200K would not load, or would crash instead of run. Clearly the test program was headed for disaster!

Because of my previous C++ and object-oriented design experience my supervisor encouraged me to convert the program as the initial step toward enhancement and eventual redesign.

Starting Off

Adding the required functionality would have been difficult without converting the entire application. With a few exceptions, the code was not organized into libraries or reusable functions. This lack of modularity is primarily a design problem that makes restructuring very difficult. The transition to C++ makes this even more evident because the language promotes modularity, encapsulation, and reuse unmatched by C.

It's useful to note that C functions can be called from C++ but not the other way around. Since the front-end interface and entry points to test functions were in C, I couldn't cleanly and transparently add low-level C++ code.

The original C program produced about fifty warnings using the Borland C/C++ 3.1 compiler. Each test subroutine had identical parameters to allow an array of pointers to the functions. The warning were due to unused parameters in these routines and couldn't be eliminated without rewriting all the parsing and dispatching code, or by using a preprocessor `#pragma` directive.

I knew of several other things the C++ compiler would complain about, but I decided to compile the program without changes just to see how big the job would be. The result was 102 warnings and 125 errors, as summarized in Table 2.

When legacy code like this has so much potential for improvement, it is tempting to dive in and make major changes. Doing so involves considerable risk, however, even when the changes seem simple and obvious. For this reason, I chose to use a more conservative approach base on the maxim: *first make it work, then make it better*.

Let's look at some of the errors and warnings in more details and examine my approach to fixing them.

Table 2 – Initial Errors and Warnings

	Quantity	Percentage
Errors		
Nonexistent function prototype	50 (65)	65
Improper typecasting	21 (54)	27
Parameter type mismatch	4	5
Miscellaneous	2	3
Total	77 (125)	100 %
Warnings		
Unused function parameters	43	51
Enumeration initialized with an integer	12	14
Non-identical symbol redefinition	11 (28)	13
Obsolete style of function declaration	11	13
Missing function return value	5	6
Variable assigned but not used	3	4
Total	85 (102)	101 %

Note: Numbers in parentheses include redundant error or warning messages.

Nonexistent Function Prototype

Function prototypes were originally grouped into two header files. They were obviously out of date and my initial approach was to simply add the ones that were missing. This was a reasonable idea in principle, but it was difficult to implement.

Monolithic header files are a troublesome source of coupling between C and C++ source modules. For example, let's say one header contains the prototypes for 60 source files. One of the prototypes is for the function `slime()` which is used only in source module 15. Any time you change the signature for `slime()` you'll end up compiling not only module 15, but all 60 source modules. Even if your make procedure doesn't do this automatically, you'd have to be very confident about all the places where `slime()` was used to simply compile module 15.

After a few recompiles, I revised my strategy a bit. First I tried to determine which functions could be declared `static`, and then moved those prototypes to the source modules where they were implemented. Then I divided the public prototypes into six header files based on general functionality and manually added missing prototypes before attempting another recompile. Unfortunately this reorganization revealed even more problems!

Many of the header files included other header files. Analysis showed that several of the same files were being included three or four times during compiling. Other headers were being included, but they were not really needed. Worse yet, changing the order of inclusion generated no additional compiler errors or warnings, but it did cause the program to crash when run. This elusive problem was probably caused by having the same symbols redefined differently in more than one header file (see the section "Other Warnings" later in this article).

Perhaps the best way to prevent multiple inclusion is to wrap the headers using preprocessor directives. For example:

```
// beginning of header file

#ifndef _VID_PALETTE_
#define _VID_PALETTE_

... all other definitions, macros, etc.

#endif // _VID_PALETTE_
```

Here we check if the symbol assigned to this header file has been defined. If not, we define it and perform all the other header file operation. Otherwise we do nothing.

Many C++ preprocessors can actually detect this wrapper idiom and optimize the preprocessing. This requires the preprocessor to keep track of which headers they've already included by checking for the `#ifndef ... #endif` wrappers and skipping over those files if they're included more than once.

For this approach to work correctly each header file symbol (in this case `_VID_PALETTE_`) must be unique. I usually use the class or module name in addition to the library or subsystem name to ensure uniqueness. Therefore, this header would contain my color palette class and related definitions which are part of my video library.

Even though you can prevent multiple inclusion using this technique, it's still a good idea to remove unnecessary `#include`'s and properly reorder the remaining ones. This enhances the long-term maintainability of the application by making the structure clearer. Moreover, it will enhance performance for preprocessors that don't optimize for multiple inclusion of headers.

It took two full days to untangle all the headers for this project.

Improper Typecasting

The typecasting errors were pretty easy to fix. Most complaints were about assignment operations such as:

```
char *video = MK_FP(0xA000,0);
```

The compiler didn't like the fact that a `char*` pointer was being assigned to the `void*` pointer create by the `MK_FP` (make-far-pointer) macro. This was fixed by simply adding the appropriate cast:

```
char *video = (char*)MK_FP(0xA000,0);
```

The only tricky ones were conversions from void pointers to function pointers, whose syntax I can never remember. Here's one for your viewing pleasure:

```
void (*reset)(void) = (void (*)(void))MK_FP(0xC000,3);
```

Although I chose the direct approach to resolve these errors, I could have added a bit more abstraction by defining pointer types and a different make-far-pointer macro such as:

```
typedef char *pc_t;           // pointer to character type
typedef void (*pf_t)(void);  // pointer to a function type

#define MAKE_FP(type, segment, offset) (type)MK_FP(segment, offset)
```

Thus the previous declarations would become:

```
pc_t video = MAKE_FP(pc_t, 0xA000, 0);
pf_t reset = MAKE_FP(pf_t, 0xC000, 0);
```

A special case of typecasting involved interrupt handlers. With the Borland compiler, the signature of an interrupt handler is different in C++ than in C. For example:

```
void interrupt MyHandler();           // for C
void interrupt MyHandler(...);      // for C++
```

For dual-purpose programs Borland recommends the following approach:

```
#ifdef _cplusplus
#define _CPPARGS ...
#else
#define _CPPARGS
#endif

// create a typedef for the sake of clarity
typedef void interrupt (*ISR)(_CPPARGS);

// declare prototypes
void interrupt IntService9(_CPPARGS);
void interrupt IntService13(_CPPARGS);

// declare save areaa for the old handlers
ISR OldService9;
ISR OldService13;
```

Keep in mind that this implementation is somewhat compiler-dependent. Although Borland C/C++ requires a specific interrupt handler signature, the Microsoft Visual C++ compiler allows either signature, and prefers the function pointer syntax to be:

```
typedef void (interrupt *ISR)(_CPPARGS);
```

Interrupt semantics are not a standard part of the C++ language. It's possible that the Symantec, Watcom, and other compilers use yet a different signature. Other than using conditional compiling - which is tedious and error-prone - I'm not sure there is a good way to make these declarations portable.

Other Errors

Most of the parameter type mismatches were easily eliminated. In two cases, the mismatches had the potential for runtime problems under certain conditions. For example, `ints` and `longs` may be different sizes on different machines.

Other errors were merely annoying. A C structure had been defined with a member named "new", which conflicted with the C++ operator `new()`. A subroutine had declared a local variable named "class", again conflicting with a C++ keyword. Naturally, each of these errors produced numerous misleading messages.

Function-Related Warnings

All of the missing return value problems were innocuous. The functions had been declared with an integer return value, but nothing was actually returned. I imagine someone had intended to return a value, but later changed their mind without changing the code. I opted to re-declare the functions as `void`.

Quite a few functions were declared using the original C convention, such as:

```
void DrawText(row, column, text)
int row;
int column;
char *text;
{
    ...
}
```

This seems prehistoric in the C++ world; hence the warning. It is easily remedied by moving the parameters into the parentheses as:

```
void DrawText(int row, int column, char *text)
{
    ...
}
```

Unused parameters were also a significant problem. To create a uniform interface, the original designers had decided to pass two pointers as parameters to the high-level testing functions. One of these was a pointer to a structure containing incoming values (“passed parameters”). The other was an array of integers containing status values (“returned parameters”). This resulted in dozens of functions declared like this:

```
int ColorBarTest(PASS_PARAMS *pparm, int *rparm)
{
    ...
}
```

Unfortunately many of these functions had no use for the passed parameters, while others had no status information to return via the array.

Neither the C nor C++ compiler is happy when function parameters go unused. In C++ however, the warning can be eliminated by simply eliding the name of the parameter. For example, if the color bar test doesn’t use any passed parameters, its declaration could be changed to:

```
int ColorBarTest(PASS_PARAMS *pparm, int *rparm)
{
    ...
}
```

Although I’m not generally in favor of declaring functions with unused parameters, there are occasions when it’s unavoidable, so I’m pleased that the C++ compiler has this capability.

Enumeration Initialized with Integer

Integer initialization of enumerations occurs in situations like this:

```
enum Color { RED, GREEN, BLUE };

void DrawColorBar(int row, int width, Color c);

void DrawColorPattern(void)
{
    Color color = 0;                // compiler warning here

    for(color=0; color < 3; color++) // compiler warning here
        DrawColorBar(color*10, 9, color);
}
```

The warnings may be eliminated by rewriting the function as:

```
void DrawColorPattern(void)
{
    Color color ;

    for(color=RED; color <= BLUE; color++)
        DrawColorBar(color*10, 9, color);
}
```

Frankly, I think it’s clearer to separate the loop index from the color choice and apply a more generic approach to loop termination. Specifically:

```

#define NUM_ELEMS(x)      (sizeof(x) / sizeof(*x))

void DrawColorPattern(void)
{
    const Color barColor[numBars] = { RED, GREEN, BLUE };
    unsigned i;

    for(i=0; i < NUM_ELEMS(barColor); i++)
        DrawColorBar(i*10, 9, barColor[i]);
}

```

Again the choice may be governed by the compiler, which generally has an option to select whether enumerations are treated as integers or not.

Other Warnings

Symbol redefinitions were primarily caused by too many authors and too many lengthy header files. This resulted in typedefs like `BYTE` being declared `char` in one place and `unsigned char` in another. Similar redefinitions could have been disastrous (e.g. if the symbol `ok` were defined as 0 in one file but 1 in another file). Unfortunately, the only way I know to resolve these conflicts is to carefully examine the program to determine what the correct type should be.

Unused variables, although a minor problem, could easily have been removed by the original programmer. It seems to me that released code should not contain these artifacts, so I removed them.

During the course of fixing all the errors and warnings I found two program bugs unrelated to the conversion. One was a loop that wrote data outside the bounds of a static array. The other was a signed integer parameter that should have been unsigned to prevent overflow. I caught these simply as a result of the code inspection needed to perform the conversion.

Linking Errors

After eliminating all compiler errors and warnings, I still had many linker errors. I had forgotten to declare the assembly routines and front-end interface C functions as “extern C”.

These additions eliminated all but three errors which were quite mysterious. The linker was unable to resolve some C++ functions that were clearly present. After a bit of detective work I found that some of the assembly routines made calls to C functions, or rather, what used to be C functions.

The idea of an assembly routine calling a C routine seems backwards to me, but I didn’t want to rewrite the assembly code or experiment with C++ function calls from assembler. Some of the code would have to remain in C. That code, in turn, would not be able to call C++ functions. This really worried me because, if the existing code was too tightly knit, I’d end up unraveling much of my work!

Eventually, by modifying some routines and juggling others around, I ended up with two small C files. By declaring the functions in these files “extern C” my linker problems went away.

To my surprise the executable was slightly smaller than the original (by a few hundred bytes). On the other hand, the symbol table grew a whopping 50% from about 240K bytes to 360K bytes. It was now too large to load for source-level debugging!

After a bit of muttering I solved the problem by setting a compiler option to compress the debug information (apparently Borland had encountered this before). I also had to remove the network login from my autoexec file to increase the available DOS memory.

Other Issues

It took a week to complete the conversion. While I was in the thick of it, my supervisor mentioned that memory allocation might not be possible. He explained that the front-end shell uses a memory allocation scheme that is incompatible with `malloc` and `free`, and that it also allocated much of free memory for its own use.

Unfortunately C++ loses many of its benefits when you take away memory allocation. I could see classes such as strings and containers fading before my eyes. More importantly, polymorphism would be impractical because I wouldn't be able to instantiate the appropriate classes at runtime.

Fortunately, the restriction wasn't as severe as first described and I was able to use `new` and `delete` as long as I didn't allocate huge blocks of memory.

Memory model turned out to be another concern. The program had been built using the medium memory model and the code was littered with `near` and `far` qualifiers. In addition to the added complexity, the program simply would not run if built using the large memory model.

Briefly, here is how objects and memory models work together.

Pointers to objects work basically like any other pointer. Consider the following code as compiled using the medium memory model:

```
class Display;

Display *slime;          // a near pointer to a Display object
slime = new Display;    // Display object allocated off the near heap

Display far *mold;      // a far pointer to a Display object
mold = new Display;    // Display object allocated off the far heap
```

As you can see, pointers to objects are `near` by default (for this memory model), but they can be explicitly made `far` in the declaration. There is, however, another way to set the default size of object pointers. The class definition can include memory model information as follows:

```
class far Display       // class declared as far
{
    ...
};

Display *slime;        // a far pointer, even using medium memory model
```

All this can get pretty complicated, especially if class member functions are also declared `near` or `far`. I recommend using 32-bit linear addressing whenever possible. This is not an option for standard DOS, but it is available for Windows programming by using the Win32 interface.

If you must use 16-bit addressing and can afford the overhead, I recommend using the large or huge memory model to avoid the need for `near` and `far` qualifiers. Keep it simple, right?

Conclusion

Even with my C++ experience I was a bit surprised by all the "gotchas" that surfaced during conversion. Although I overcame the problems, a C++ novice might have been discouraged enough to give up.

Naturally, conversion to the C++ compiler does not produce an object-oriented design (if only it were that easy). However, the conversion process clearly revealed several problems and questionable programming practices. It also extended the useful life of this program by several months and provided valuable data about the applicability of C++ to this type of project.

This success prompted management to approve a complete rewrite of the test program starting with an object-oriented design. The new design has much greater modularity and about 70% fewer lines of code, as I'll describe in a follow-up article.

References

1. Reeves, J. "Migrating from C to C++" *C++ Report*, July /Aug. 1995