Member Function Pointers and the Fastest Possible C++ Delegates

Don Clugston, 6 Apr 2005 CPOL

★★★★★ 4.82 (681 votes)

Rate: vote 1vote 2vote 3vote 4vote 5

A comprehensive tutorial on member function pointers, and an implementation of delegates that generates only two ASM opcodes!

- **Download source files - 18.5 Kb**
- **Download macro files used to develop this library - 18.6 Kb**

Russian Translation

I'm pleased to announce that Denis Bulichenko has translated this article into Russian! It has been published inRSDN

. The complete article will soon be available on-line. I'm deeply indebted to you, Denis.

Introduction

Standard C++ does not have true object-oriented function pointers. This is unfortunate, because object-oriented function pointers, also called 'closures' or 'delegates', have proved their value in similar languages. In Delphi (Object Pascal), they are the basis for Borland's Visual Component Library (VCL). More recently, C# has popularized the delegate concept, contributing to the success of that language. For many applications, delegates simplify the use of elegant design patterns (Observer, Strategy, State[GoF]) composed of very loosely coupled objects. There can be no doubt that such a feature would be useful in standard C++.

Instead of delegates, C++ only provides member function pointers. Most C++ programmers have never used member function pointers, and with good reason. They have their own bizarre syntax (the `->*` and `.*` operators, for example), it's hard to find accurate information about them, and most of the things you can do with them could be done better in some other way. This is a bit scandalous: it's actually easier for a compiler writer to implement proper delegates than it is to implement member function pointers!

In this article, I'll "lift the lid" on member function pointers. After a recap of the syntax and idiosyncrasies of member function pointers, I'll explain how member function pointers are implemented by commonly-used compilers. I'll show how compilers could implement delegates efficiently. Finally, I will show how I used this clandestine knowledge of member function pointers to make an implementation of delegates that is optimally efficient on most C++ compilers. For example, invoking a single-target delegate on Visual C++ (6.0, .NET, and .NET 2003) generates just two lines of assembly code!

Function Pointers

We begin with a review of function pointers. In C, and consequently in C++, a function pointer called`my_func_ptr` that points to a function taking an `int` and a `char *` and returning a `float`, is declared like this:

Hide Copy Code

```
float (*my_func_ptr)(int, char *);
// To make it more understandable, I strongly recommend that you use a typedef.
// Things can get particularly confusing when
// the function pointer is a parameter to a function.
// The declaration would then look like this:
typedef float (*MyFuncPtrType)(int, char *);
MyFuncPtrType my_func_ptr;
```

Note that there is a different type of function pointer for each combination of arguments. On MSVC, there is also a different type for each of the three different calling conventions: `__cdecl`, `__stdcall`, and `__fastcall`. You make your function pointer point to a function `float some_func(int, char *)` like this:

Hide Copy Code

```
 my_func_ptr = some_func;
```

When you want to invoke the function that you stored, you do this:

Hide Copy Code

```
 (*my_func_ptr)(7, "Arbitrary String");
```

You are allowed to cast from one type of function pointer to another. But you are not allowed to cast a function pointer to a `void *` data pointer. The other allowable operations are trivial. A function pointer can be set to 0 to mark it as a null pointer. The full range of comparison operators are available (`==`, `!=`, `<`, `>`, `<=`, `>=`), and you can also test for null pointers using `==0` or via an implicit cast to `bool`. Interestingly, a function pointer can be used as a non-type template parameter. This is fundamentally different from a type parameter, and is also different from an integral non-type parameter. It is instantiated based on *name* rather than type or value. Name-based template parameters are not supported by all compilers, not even by all those with support for partial template specialization.

In C, the most common uses of function pointers are as parameters to library functions like `qsort`, and as callbacks for Windows functions, etc. They have many other applications as well. The implementation of function pointers is simple: they are just "code pointers": they hold the starting address of an assembly-language routine. The different types of function pointers exist only to ensure that the correct calling convention is used.

Member Function Pointers

In C++ programs, most functions are member functions; that is, they are part of a class. You are not allowed to use an ordinary function pointer to point to a member function; instead, you have to use a member function pointer. A member function pointer to a member function of class `SomeClass`, with the same arguments as before, is declared like this:

Hide Copy Code

```
float (SomeClass::*my_memfunc_ptr)(int, char *);
// For const member functions, it's declared like this:
float (SomeClass::*my_const_memfunc_ptr)(int, char *) const;
```

Notice that a special operator (`::*`) is used, and that `SomeClass` is part of the declaration. Member function pointers have a horrible restriction: they can only point to member functions of a single class. There is a different type of member function pointer for each combination of arguments, for both types of const-ness, and for each class. On MSVC, there is also a different type for each of the four different calling conventions: `__cdecl`,`__stdcall`, `__fastcall`, and `__thiscall`. (`__thiscall` is the default. Interestingly, there is no documented`__thiscall` keyword, but it sometimes shows up in error messages. If you use it explicitly, you'll get an error message stating that it is reserved for future use.) If you're using member function pointers, you should always use a `typedef` to avoid confusion.

You make your function pointer point to a function `float SomeClass::some_member_func(int, char *)`like this:

Hide   Copy Code

```
my_memfunc_ptr = &SomeClass::some_member_func;
// This is the syntax for operators:
my_memfunc_ptr = &SomeClass::operator !;
// There is no way to take the address of a constructor or destructor
```

Some compilers (most notably MSVC 6 and 7) will let you omit the `&`, even though it is non-standard and confusing. More standard-compliant compilers (e.g., GNU G++ and MSVC 8 (a.k.a. VS 2005)) require it, so you should definitely put it in. To invoke the member function pointer, you need to provide an instance of `SomeClass`, and you must use the special operator `->*`. This operator has a low precedence, so you need to put it in parentheses:

Hide   Copy Code

```
  SomeClass *x = new SomeClass;
  (x->*my_memfunc_ptr)(6, "Another Arbitrary Parameter");
// You can also use the .* operator if your class is on the stack.
  SomeClass y;
  (y.*my_memfunc_ptr)(15, "Different parameters this time");
```

Don't blame me for the syntax -- it seems that one of the designers of C++ loved punctuation marks!

C++ added three special operators to the C language specifically to support member pointers. `::*` is used in declaring the pointer, and `->*` and `.*` are used in invoking the function that is pointed to. It seems that extraordinary attention was paid to an obscure and seldom-used part of the language. (You're even allowed to overload the `->*` operator, though why you want to do such a thing is beyond my comprehension. I only know of one such usage [Meyers].)

A member function pointer can be set to 0, and provides the operators `==` and `!=`, *but only for member function pointers of the same class*. Any member function pointer can be compared with 0 to see if it is null. [Update, March 2005: This doesn't work on all compilers. On Metrowerks MWCC, a pointer to the first virtual function of a simple class will be equal to zero!] Unlike simple function pointers, the inequality comparisons (`<`, `>`, `<=`, `>=`) are not available. Like function pointers, they can be used as non-type template parameters, but this seems to work on fewer compilers.

Weird Things about Member Function Pointers

There are some weird things about member function pointers. Firstly, you can't use a member function to point to a static member function. You have to use a normal function pointer for that. (So the name "member function pointer" is a bit misleading: they're actually "non-static member function pointers".) Secondly, when dealing with derived classes, there are some surprises. For example, the code below will compile on MSVC if you leave the comments intact:

Hide   Copy Code

```
class SomeClass {
 public:
    virtual void some_member_func(int x, char *p) {
       printf("In SomeClass"); };
};
class DerivedClass : public SomeClass {
 public:
 // If you uncomment the next line, the code at line (*) will fail!
//    virtual void some_member_func(int x, char *p) { printf("In DerivedClass"); };
};
int main() {
    // Declare a member function pointer for SomeClass
    typedef void (SomeClass::*SomeClassMFP)(int, char *);
    SomeClassMFP my_memfunc_ptr;
    my_memfunc_ptr = &DerivedClass::some_member_func; // ---- line (*)
}
```

Curiously enough, &DerivedClass::some_member_func is a member function pointer of class SomeClass. It is not a member of DerivedClass! (Some compilers behave slightly differently: e.g., for Digital Mars C++,&DerivedClass::some_member_func is undefined in this case.) But, if DerivedClass overridessome_member_func, the code won't compile, because &DerivedClass::some_member_func has now become a member function pointer of class DerivedClass!

Casting between member function pointers is an extremely murky area. During the standardization of C++, there was a lot of discussion about whether you should be able to cast a member function pointer from one class to a member function pointer of a base or derived class, and whether you could cast between unrelated classes. By the time the standards committee made up their mind, different compiler vendors had already made implementation decisions which had locked them into different answers to these questions. According to the Standard (section 5.2.10/9), you can use reinterpret_cast to store a member function for one class inside a member function pointer for an unrelated class. The result of invoking the casted member function is undefined. The only thing you can do with it is cast it back to the class that it came from. I'll discuss this at length later in the article, because it's an area where the Standard bears little resemblance to real compilers.

On some compilers, weird stuff happen even when converting between member function pointers of base and derived classes. When multiple inheritance is involved, using `reinterpret_cast` to cast from a derived class to a base class may or may not compile, depending on what order the classes are listed in the declaration of the derived class! Here's an example:

Hide Copy Code

```
class Derived: public Base1, public Base2 // case (a)
class Derived2: public Base2, public Base1 // case (b)
typedef void (Derived::* Derived_mfp)();
typedef void (Derived2::* Derived2_mfp)();
typedef void (Base1::* Base1mfp) ();
typedef void (Base2::* Base2mfp) ();
Derived_mfp x;
```

For case (a), `static_cast<Base1mfp>(x)` will work, but `static_cast<Base2mfp>(x)` will fail. Yet for case (b), the opposite is true. You can safely cast a member function pointer from a derived class to its first base class only! If you try, MSVC will issue warning C4407, while Digital Mars C++ will issue an error. Both will protest if you use`reinterpret_cast` instead of `static_cast`, but for different reasons. But, some compilers will be perfectly happy no matter what you do. Beware!

There's another interesting rule in the standard: you can declare a member function pointer before the class has been defined. You can even invoke a member function of this incomplete type! This will be discussed later in the article. Note that a few compilers can't cope with this (early MSVC, early CodePlay, LVMM).

It's worth noting that, as well as member function pointers, the C++ standard also provides member data pointers. They share the same operators, and some of the same implementation issues. They are used in some implementations of `stl::stable_sort`, but I'm not aware of many other sensible uses.

Uses of Member Function Pointers

By now, I've probably convinced you that member function pointers are somewhat bizarre. But what are they useful for? I did an extensive search of published code on the web to find out. I found two common ways in which member function pointers are used:

a. contrived examples, for demonstrating the syntax to C++ novices, and
b. implementations of delegates!

They also have trivial uses in one-line function adaptors in the STL and Boost libraries, allowing you to use member functions with the standard algorithms. In such cases, they are used at compile-time; usually, no function pointers actually appear in the compiled code. The most interesting application of member function pointers is in defining complex interfaces. Some impressive things can be done in this way, but I didn't find many examples. Most of the time, these jobs can be performed more elegantly with virtual functions or with a refactoring of the problem. But by far, the most famous uses of member function pointers are in application frameworks of various kinds. They form the core of MFC's messaging system.

When you use MFC's message map macros (e.g., `ON_COMMAND`), you're actually populating an array containing message IDs and member function pointers (specifically, `CCmdTarget::*` member function pointers). This is the reason that MFC classes must derive from `CCmdTarget` if they want to handle messages. But the various message handling functions have different parameter lists (for example, `OnDraw` handlers have `CDC *` as the first parameter), so the array must contain member function pointers of various types. How does MFC deal with this? They use a horrible hack, putting all the possible member function pointers into an enormous union, to subvert the normal C++ type checking. (Look up the `MessageMapFunctions` union in *afximpl.h* and *cmdtarg.cpp* for the gory details.) Because MFC is such an important piece of code, in practice, all C++ compilers support this hack.

In my searches, I was unable to find many examples of good usage of member function pointers, other than at compile time. For all their complexity, they don't add much to the language. It's hard to escape the conclusion that C++ member function pointers have a flawed design.

In writing this article, I have one key point to make: *It is absurd that the C++ Standard allows you to cast between member function pointers, but doesn't allow you to invoke them once you've done it*. It's absurd for three reasons. Firstly, the cast won't always work on many popular compilers (so, casting is standard, but not portable). Secondly, on **all** compilers, if the cast is successful, invoking the cast member function pointer behaves exactly as you would hope: there is no need for it to be classed as "undefined behavior". (Invocation is portable, but not standard!) Thirdly, allowing the cast without allowing invocation is completely useless; but if both cast and invocation are possible, it's easy to implement efficient delegates, with a huge benefit to the language.

To try to convince you of this controversial assertion, consider a file consisting solely of the following code. This is legal C++.

Hide Copy Code

```
class SomeClass;
typedef void (SomeClass::* SomeClassFunction)(void);
void Invoke(SomeClass *pClass, SomeClassFunction funcptr) {
  (pClass->*funcptr)(); };
```

Note that the compiler has to produce assembly code to invoke the member function pointer, knowing **nothing**about the class `SomeClass`. Clearly, unless the linker does some _extremely_ sophisticated optimization, the code**must** work correctly regardless of the actual definition of the class. An immediate consequence is that you can safely invoke a member function pointer that's been cast from a completely different class.

To explain the other half of my assertion, that casting doesn't work the way the standard says it should, I need to discuss in detail exactly how compilers implement member function pointers. This will also help to explain why the rules about the use of member function pointers are so restrictive. Accurate documentation about member function pointers is hard to obtain, and misinformation is common, so I've examined the assembly code produced by a large range of compilers. It's time to get our hands dirty.

Member Function Pointers - why are they so complex?

A member function of a class is a bit different from a standard C function. As well as the declared parameters, it has a hidden parameter called `this`, which points to the class instance. Depending on the compiler, `this` might be treated internally as a normal parameter, or it might receive special treatment. (For example, in VC++, `this` is usually passed using the `ECX` register). `this` is fundamentally different to normal parameters. For virtual functions, it controls *at runtime* which function is executed. Even though a member function is a real function underneath, there is no way in standard C++ to make an ordinary function behave like a member function: there's no`thiscall` keyword to make it use the correct calling convention. Member functions are from Mars, ordinary functions are from Venus.

You'd probably guess that a "member function pointer", like a normal function pointer, just holds a code pointer. You would be wrong. On almost all compilers, a member function pointer is bigger than a function pointer. Most bizarrely, in Visual C++, a member function pointer might be 4, 8, 12, or 16 bytes long, depending on the nature of the class it's associated with, and depending on what compiler settings are used! Member function pointers are more complex than you might expect. But it was not always so.

Let's go back in time to the early 1980's. When the original C++ compiler (CFront) was originally developed, it only had single inheritance. When member function pointers were introduced, they were simple: they were just function pointers that had an extra `this` parameter as the first argument. When virtual functions were involved, the function pointer pointed to a short bit of 'thunk' code. (Update, Oct 04: A discussion on*comp.lang.c++.moderated* has established that CFront didn't actually use thunks, it was much less elegant. But it*could* have used this method, and pretending that it did, makes the following discussion a bit easier to understand.)

This idyllic world was shattered when CFront 2.0 was released. It introduced templates and multiple inheritance. Part of the collateral damage of multiple inheritance was the castration of member function pointers. The problem is, with multiple inheritance, you don't know what `this` pointer to use until you make the call. For example, suppose you have the four classes defined below:

Hide   Copy Code

```
class A {
 public:
       virtual int Afunc() { return 2; };
};
class B {
 public:
      int Bfunc() { return 3; };
};
// C is a single inheritance class, derives only from A
class C: public A {
 public:
     int Cfunc() { return 4; };
};
// D uses multiple inheritance
class D: public A, public B {
 public:
    int Dfunc() { return 5; };
};
```

Suppose we create a member function pointer for class `C`. In this example, `Afunc` and `Cfunc` are both member functions of `C`, so our member function pointer is allowed to point to `Afunc` or `Cfunc`. But `Afunc` needs a `this`pointer that points to `C::A` (which I'll call `Athis`), while `Cfunc` needs a `this` pointer that points to `C` (which I'll call`Cthis`). Compiler writers deal with this situation by a trick: they ensure that `A` is physically stored at the start of `C`. This means that `Athis == Cthis`. We only have one `this` to worry about, and all's well with the world.

Now, suppose we create a member function pointer for class `D`. In this case, our member function pointer is allowed to point to `Afunc`, `Bfunc`, or `Dfunc`. But `Afunc` needs a `this` pointer that points to `D::A`, while `Bfunc`needs a `this` pointer that points to `D::B`. This time, the trick doesn't work. We can't put both `A` *and* `B` at the start of `D`. So, a member function pointer to `D` needs to specify not only what function to call, but also what `this`pointer to use. The compiler does know how big `A` is, so it can convert an `Athis` pointer into a `Bthis` just by adding an offset (`delta = sizeof(A)`) to it.

If you're using virtual inheritance (i.e., virtual base classes), it's much worse, and you can easily lose your mind trying to understand it. Typically, the compiler uses a virtual function table ('vtable') which stores for each virtual function, the function address, and the virtual_delta: the amount in bytes that needs to be added to the supplied`this` pointer to convert it into the `this` pointer that the function requires.

None of this complexity would exist if C++ had defined member function pointers a bit differently. In the code above, the complexity only comes because you are allowed to refer to `A::Afunc` as `D::Afunc`. This is probably bad style; normally, you should be using base classes as interfaces. If you were forced to do this, then member function pointers could have been ordinary function pointers with a special calling convention. IMHO, allowing them to point to overridden functions was a tragic mistake. As payment for this rarely-used extra functionality, member function pointers became grotesque. They also caused headaches for the compiler writers who had to implement them.

Implementations of Member Function Pointers

So, how do compilers typically implement member function pointers? Here are some results obtained by applying the `sizeof` operator to various structures (an `int`, a `void *` data pointer, a code pointer (i.e., a pointer to a static function), and a member function pointer to a class with single-, multiple-, virtual- inheritance, or unknown (i.e., forward declared)) for a variety of 32, 64 and 16-bit compilers.

Compiler	Options	int	DataPtr	CodePtr	Single	Multi	Virtual	Unknown
MSVC		4	4	4	4	8	12	16
MSVC	/vmg	4	4	4	16#	16#	16#	16
MSVC	/vmg /vmm	4	4	4	8#	8#	--	8
Intel_IA32		4	4	4	4	8	12	16
Intel_IA32	/vmg /vmm	4	4	4	4	8	--	8
Intel_Itanium		4	8	8	8	12	16	20
G++		4	4	4	8	8	8	8
Comeau		4	4	4	8	8	8	8
DMC		4	4	4	4	4	4	4
BCC32		4	4	4	12	12	12	12
BCC32	/Vmd	4	4	4	4	8	12	12
WCL386		4	4	4	12	12	12	12
CodeWarrior		4	4	4	12	12	12	12
XLC		4	8	8	20	20	20	20
DMC	small	2	2	2	2	2	2	2
	medium	2	2	4	4	4	4	4
WCL	small	2	2	2	6	6	6	6

compact	2	4	2	6	6	6	6
medium	2	2	4	8	8	8	8
large	2	4	4	8	8	8	8

# Or 4,8, or 12 if the `__single`/ `__multi`/ `__virtual_inheritance` keyword is used.

The compilers are Microsoft Visual C++ 4.0 to 7.1 (.NET 2003), GNU G++ 3.2 (MingW binaries, www.mingw.org), Borland BCB 5.1 (www.borland.com), Open Watcom (WCL) 1.2 (www.openwatcom.org), Digital Mars (DMC) 8.38n (www.digitalmars.com), Intel C++ 8.0 for Windows IA-32, Intel C++ 8.0 for Itanium (www.intel.com), IBM XLC for AIX (Power, PowerPC), Metrowerks Code Warrior 9.1 for Windows (www.metrowerks.com), and Comeau C++ 4.3 (www.comeaucomputing.com). The data for Comeau applies to all their supported 32-bit platforms (x86, Alpha, SPARC, etc.). The 16-bit compilers were also tested in four DOS configurations (tiny, compact, medium, and large) to show the impact of various code and data pointer sizes. MSVC was also tested with an option (/vmg) that gives "full generality for pointers to members". (If you have a compiler from a vendor that is not listed here, please let me know. Results for non-x86 processors are particularly valuable.)

Amazing, isn't it? Looking at this table, you can readily see how easy it is to write code that will work in some circumstances but fail to compile in others. The internal implementations are obviously very different between compilers; in fact, I don't think that any other language feature has such a diversity of implementation. Looking in detail at the implementations reveals some surprising nastiness.

The Well-Behaved Compilers

For almost all compilers, two fields I've called `delta` and `vindex` are used to adjust the provided `this` pointer into an `adjustedthis` which is passed to the function. For example, here is the technique used by Watcom C++ and Borland:

Hide Copy Code

```
struct BorlandMFP { // also used by Watcom
   CODEPTR m_func_address;
   int delta;
   int vindex; // or 0 if no virtual inheritance
};
if (vindex==0) adjustedthis = this + delta;
else adjustedthis = *(this + vindex -1) + delta
CALL funcadr
```

If virtual functions are used, the function pointer points to a two-instruction 'thunk' to determine the actual function to be called. Borland applies an optimization: if it knows that the class only used single inheritance, it knows that `delta` and `vindex` will always be zero, so it can skip the calculation in this most common case. Importantly, it only changes the invocation calculation, it does not change the structure itself.

Many other compilers use the same calculation, often with a minor reordering of the structure.

Hide Copy Code

```
// Metrowerks CodeWarrior uses a slight variation of this theme.
// It uses this structure even in Embedded C++ mode, in which
// multiple inheritance is disabled!
struct MetrowerksMFP {
   int delta;
   int vindex; // or -1 if no virtual inheritance
   CODEPTR func_address;
};
// An early version of SunCC apparently used yet another ordering:
struct {
   int vindex; // or 0 if a non-virtual function
   CODEPTR func_address; // or 0 if a virtual function
   int delta;
};
```

Metrowerks doesn't seem to inline the calculation. Instead, it's performed in a short 'member function invoker' routine. This reduces code size a little but makes their member function pointers a bit slower.

Digital Mars C++ (formerly Zortech C++, then Symantec C++) uses a different optimization. For single-inheritance classes, a member function pointer is just the address of the function. When more complex inheritance is involved, the member function pointer points to a 'thunk' function, which performs the necessary adjustments to the `this` pointer, and then calls the real member function. One of these little thunk functions is created for every member function that's involved in multiple inheritance. This is easily my favorite implementation.

Hide Copy Code

```
struct DigitalMarsMFP { // Why doesn't everyone else do it this way?
   CODEPTR func_address;
};
```

Current versions of the GNU compiler use a strange and tricky optimization. It observes that, for virtual inheritance, you have to look up the vtable in order to get the `voffset` required to calculate the `this` pointer. While you're doing that, you might as well store the function pointer in the vtable. By doing this, they combine the`m_func_address` and `m_vtable_index` fields into one, and they distinguish between them by ensuring that function pointers always point to even addresses but vtable indices are always odd:

Hide Copy Code

```
// GNU g++ uses a tricky space optimisation, also adopted by IBM's VisualAge and XLC.
struct GnuMFP {
   union {
     CODEPTR funcadr; // always even
     int vtable_index_2; //  = vindex*2+1, always odd
   };
   int delta;
};
adjustedthis = this + delta
if (funcadr & 1) CALL (* ( *delta + (vindex+1)/2) + 4)
else CALL funcadr
```

The G++ method is well documented, so it has been adopted by many other vendors, including IBM's VisualAge and XLC compilers, recent versions of Open64, Pathscale EKO, and Metrowerks' 64-bit compilers. A simpler scheme used by earlier versions of GCC is also very common. SGI's now discontinued MIPSPro and Pro64 compilers, and Apple's ancient MrCpp compiler used this method. (Note that the Pro64 compiler has become the open source Open64 compiler).

Hide Copy Code

```
struct Pro64MFP {
     short delta;
     short vindex;
     union {
       CODEPTR funcadr; // if vindex==-1
       short __delta2;
     } __funcadr_or_delta2;
   };
// If vindex==0, then it is a null pointer-to-member.
```

Compilers based on the Edison Design Group front-end (Comeau, Portland Group, Greenhills) use a method that is almost identical. Their calculation is (PGI 32-bit compilers):

Hide Copy Code

```
// Compilers using the EDG front-end (Comeau, Portland Group, Greenhills, etc)
struct EdisonMFP{
    short delta;
    short vindex;
    union {
     CODEPTR funcadr; // if vindex=0
     long vtordisp;   // if vindex!=0
    };
};
if (vindex==0) {
   adjustedthis=this + delta;
   CALL funcadr;
} else {
   adjustedthis = this+delta + *(*(this+delta+vtordisp) + vindex*8);
   CALL *(*(this+delta+funcadr)+vindex*8 + 4);
};
```

Most compilers for embedded systems don't allow multiple inheritance. Thus, these compilers avoid all the quirkiness: a member function pointer is just a normal function pointer with a hidden '`this`' parameter.

The Sordid Tale of Microsoft's "Smallest For Class" Method

Microsoft compilers use an optimization that is similar to Borland's. They treat single inheritance with optimum efficiency. But unlike Borland, the default is to discard the entries which will always be zero. This means that single-inheritance pointers are the size of simple function pointers, multiple inheritance pointers are larger, and virtual inheritance pointers are larger still. This saves space. But it's not standard-compliant, and it has some bizarre side-effects.

Firstly, casting a member function pointer between a derived class and a base class can change its size! Consequently, information can be lost. Secondly, when a member function is declared before its class is defined, the compiler has to work out how much space to allocate to it. But it can't do this reliably, because it doesn't know the inheritance nature of the class until it's defined. It has to guess. If it guesses wrongly in one source file, but correctly in another, your program will inexplicably crash at runtime. So, Microsoft added some reserved words to their compiler: `__single_inheritance`, `__multiple_inheritance`, and `__virtual_inheritance`. They also added a compiler switch: */vmg*, which makes all MFPs the same size, by retaining the missing zero fields. At this point, the tale gets sordid.

The documentation implies that specifying */vmg* is equivalent to declaring every class with the`__virtual_inheritance` keyword. It isn't. Instead, it uses an even bigger structure which I've called`unknown_inheritance`. It also uses this whenever it has to make a member function pointer for a class when all it's seen is a forward declaration. They can't use their `__virtual_inheritance` pointers because they use a really silly optimization. Here's the algorithm they use:

Hide   Copy Code

```
// Microsoft and Intel use this for the 'Unknown' case.
// Microsoft also use it when the /vmg option is used
// In VC1.5 - VC6, this structure is broken! See below.
struct MicrosoftUnknownMFP{
   FunctionPointer m_func_address; // 64 bits for Itanium.
   int m_delta;
   int m_vtordisp;
   int m_vtable_index; // or 0 if no virtual inheritance
};
 if (vindex=0) adjustedthis = this + delta
 else adjustedthis = this + delta + vtordisp + *(*(this + vtordisp) + vindex)
 CALL funcadr
```

In the virtual inheritance case, the `vtordisp` value is not stored in the `__virtual_inheritance` pointer! Instead, the compiler hard-codes it into the assembly output when the function is invoked. But to deal with incomplete types, you need to know it. So they ended up with *two* types of virtual inheritance pointers. But until VC7, the unknown_inheritance case was hopelessly buggy.
The `vtordisp` and `vindex` fields were always zero! Horrifying consequence: on VC4-VC6, specifying the */vmg* option (without */vmm* or */vms*) can result in the wrong function being called! This would be extremely difficult to track down. In VC4, the IDE had a box for selecting the*/vmg* option, but it was disabled. I suspect that *someone* at MS knew about the bug, but it was never listed in their knowledge base. They finally fixed it in VC7.

Intel uses the same calculation as MSVC, but their */vmg* option behaves quite differently (it has almost no effect - it only affects the unknown_inheritance case). The release notes for their compiler state that conversions between pointer to member types are not fully supported in the virtual inheritance case, and warns that compiler crashes or incorrect code generation may result if you try. This is a very nasty corner of the language.

And then there's CodePlay. Early versions of Codeplay's VectorC had options for link compatibility with Microsoft VC6, GNU, and Metrowerks. But, they always used the Microsoft method. They reverse-engineered it, just as I have, but they didn't detect the `unknown_inheritance` case, or the `vtordisp` value. Their calculations implicitly (and incorrectly) assumed `vtordisp=0`, so the wrong function could be called in some (obscure) cases. But Codeplay's upcoming release of VectorC 2.2.1 has fixed these problems. The member function pointers are now binary compatible with either Microsoft or GNU. With some high powered optimisations and a massive improvement in standards conformance (partial template specialisation, etc), this is becoming a very impressive compiler.

What have we learned from all of this?

In theory, all of these vendors could radically change their technique for representing MFPs. In practice, this is extremely unlikely, because it would break a lot of existing code. At MSDN, there is a very old article that was published by Microsoft which explains the run-time implementation details of Visual C++ [JanGray]. It's written by Jan Gray, who actually wrote the MS C++ object model in 1990. Although the article dates from 1994, it is still relevant - excluding the bug fix, Microsoft hasn't changed it for 15 years. Similarly, the earliest compiler I have (Borland C++ 3.0, (1990)) generates identical code to Borland's most recent compiler, except of course that 16 bit registers are replaced with 32 bit ones.

By now, you know far too much about member function pointers. What's the point? I've dragged you through this to establish a rule. Although these implementations are very different from one another, they have something useful in common: *the assembly code required to invoke a member function pointer is identical, regardless of what class and parameters are involved.* Some compilers apply optimizations depending on the inheritance nature of the class, but when the class being invoked is of incomplete type, all such optimizations are impossible. This fact can be exploited to create efficient delegates.

Delegates

Unlike member function pointers, it's not hard to find uses for delegates. They can be used anywhere you'd use a function pointer in a C program. Perhaps most importantly, it's very easy to implement an improved version of the Subject/Observer design pattern [GoF, p. 293] using delegates. The Observer pattern is most obviously applicable in GUI code, but I've found that it gives even greater benefits in the heart of an application. Delegates also allow elegant implementations of the Strategy [GoF, p. 315] and State [GoF, p. 305] patterns.

Now, here's the scandal. Delegates aren't just far more useful than member function pointers. They are much simpler as well! Since delegates are provided by the .NET languages, you might imagine that they are a high-level concept that is not easily implemented in assembly code. This is emphatically not the case: invoking a delegate is intrinsically a very low-level concept, and can be as low-level (and fast) as an ordinary function call. A C++ delegate just needs to contain a `this` pointer and a simple function pointer. When you set a delegate, you provide it with the `this` pointer at the same time that you specify the function to be called. The compiler can work out how the `this` pointer needs to be adjusted at the time that the delegate is set. There's no work to be done when invoking the delegate. Even better, the compiler can frequently do all the work at compile time, so that even setting the delegate is a trivial operation. The assembly code produced by invoking a delegate on an x86 system*should* be as simple as:

Hide Copy Code

```
mov ecx, [this]
call [pfunc]
```

However, there's no way to generate such efficient code in standard C++. Borland solves this problem by adding a new keyword (`__closure`) to their C++ compiler, allowing them to use convenient syntax and generate optimal code. The GNU compiler also adds a language extension, but it is incompatible with Borland's. If you use either of these language extensions, you restrict yourself to a single compiler vendor. If instead, you constrain yourself to standard C++, it's still possible to implement delegates, they are just not as efficient.

Interestingly, in C# and other .NET languages, a delegate is apparently dozens of times slower than a function call (MSDN). I suspect that this is because of the garbage collection and the .NET security requirements. Recently, Microsoft added a "unified event model" to Visual C++, with the keywords `__event`, `__raise`, `__hook`,`__unhook`, `event_source`, and `event_receiver`. Frankly, I think this feature is appalling. It's completely non-standard, the syntax is ugly and doesn't even look like C++, and it generates very inefficient code.

Motivation: The need for extremely fast delegates

There is a plethora of implementations of delegates using standard C++. All of them use the same idea. The basic observation is that member function pointers act as delegates -- but they only work for a single class. To avoid this limitation, you add another level of indirection: you can use templates to create a 'member function invoker' for each class. The delegate holds the `this` pointer, and a pointer to the invoker. The member function invoker needs to be allocated on the heap.

There are many implementations of this scheme, including several here at CodeProject. They vary in their complexity, their syntax (especially, how similar their syntax is to C#), and in their generality. The definitive implementation is boost::function. Recently, it was adopted into the next version of the C++ standard [Sutter1]. Expect its use to become widespread.

As clever as the traditional implementations are, I find them unsatisfying. Although they provide the desired functionality, they tend to obscure the underlying issue: a low-level construct is missing from the language. It's frustrating that on all platforms, the 'member function invoker' code will be identical for almost all classes. More importantly, the heap is used. For some applications, this is unacceptable.

One of my projects is a discrete event simulator. The core of such a program is an event dispatcher, which calls member functions of the various objects being simulated. Most of these member functions are very simple: they just update the internal state of the object, and sometimes add future events to the event queue. This is a perfect situation to use delegates. However, each delegate is only ever invoked once. Initially, I used `boost::function`, but I found that the memory allocation for the delegates was consuming over a third of the entire program running time! I wanted real delegates. For crying out loud, it should be just two ASM instructions!

I don't always (often?) get what I want, but this time I was lucky. The C++ code I present here generates optimal assembly code in almost all circumstances. Most importantly, *invoking a single-target delegate is as fast as a normal function call*. There is no overhead whatsoever. The only downside is that, to achieve this, I had to step outside the rules of standard C++. I used the clandestine knowledge of member function pointers to get this to work. Efficient delegates are possible on any C++ compiler, if you are *very* careful, and if you don't mind some compiler-specific code in a few cases.

The trick: casting any member function pointer into a standard form

The core of my code is a class which converts an arbitrary class pointer and arbitrary member function pointer into a generic class pointer and a generic member function. C++ doesn't have a 'generic member function' type, so I cast to member functions of an undefined `CGenericClass`.

Most compilers treat all member function pointers identically, regardless of the class. For most of them, a straightforward `reinterpret_cast<>` from the given member function pointer to the generic member function pointer would work. In fact, if this doesn't work, the compiler is non-standard. For the remaining compilers (Microsoft Visual C++ and Intel C++), we have to transform multiple- or virtual-inheritance member function pointers into single-inheritance pointers. This requires some template magic and a horrible hack. Note that the hack is only necessary because these compilers are not standards-compliant, but there is a nice reward: the hack gives optimal code.

Since we know how the compiler stores member function pointers internally, and because we know how the `this`pointer needs to be adjusted for the function in question, we can adjust the `this` pointer ourselves when we are setting the delegate. For single inheritance pointers, no adjustment is required; for multiple inheritance, there's a simple addition; and for virtual inheritance ... it's a mess. But it works, and in most cases, all of the work is done at compile time!

How can we distinguish between the different inheritance types? There's no official way to find out whether a class used multiple inheritance or not. But there's a sneaky way, which you can see if you look at the table I presented earlier -- on MSVC, each inheritance style produces a member function pointer with a different size. So, we use template specialization based on the size of the member function pointer! For multiple inheritance, it's a trivial calculation. A similar, but much nastier calculation is used in the unknown_inheritance (16 byte) case.

For Microsoft's (and Intel's) ugly, non-standard 12 byte `virtual_inheritance` pointers, yet another trick is used, based on an idea invented by John Dlugosz. The crucial feature of Microsoft/Intel MFPs which we exploit is that the `CODEPTR` member is *always* called, regardless of the values of the other members. (This is *not* true for other compilers, e.g., GCC, which obtain the function address from the vtable if a virtual function is being called.) Dlugosz's trick is to make a fake member function pointer, in which the `codeptr` points to a probe function which returns the '`this`' pointer that was used. When you call this function, the compiler does all the calculation work for you, making use of the secretive `vtordisp` value.

Once you can cast any class pointer and member function pointer into a standard form, it's easy (albeit tedious) to implement single-target delegates. You just need to make template classes for all of the different numbers of parameters.

A very significant additional benefit of implementing delegates by this non-standard cast is that they can be compared for equality. Most existing delegate implementations can't do this, and this makes it difficult to use them for certain tasks, such as implementing multi-cast delegates [Sutter3].

Static functions as delegate targets

Ideally, a simple non-member function, or a static member function, could be used as a delegate target. This can be achieved by converting the static function into a member function. I can think of two methods of doing this, in both of which the delegate points to an 'invoker' member function which calls the static function.

The Evil method uses a hack. You can store the function pointer instead of the `this` pointer, so that when the invoker function is called, it just needs to cast `this` into a static function pointer and call that. The nice thing about this is that it has absolutely no effect on the code for normal member functions. The problem with it is that it is a hack as it requires casting between code and data pointers. It won't work on systems where code pointers are larger than data pointers (DOS compilers using the medium memory model). It will work on all 32 and 64 bit processors that I know of. But because it's Evil, we need an alternative.

The Safe method is to store the function pointer as an extra member of the delegate. The delegate points to its own member function. Whenever the delegate is copied, these self-references must be transformed, and this complicates the `=` and `==` operators. This increases the size of the delegate by four bytes, and increases the complexity of the code, but it has no effect on the invocation speed.

I've implemented both methods, because both have merit: the Safe method is guaranteed to work, and the Evil method generates the same ASM code that a compiler would probably generate if it had native support for delegates. The Evil method can be enabled with a `#define` (`FASTDELEGATE_USESTATICFUNCTIONHACK`).

Footnote: Why does the evil method work at all? If you look closely at the algorithm that is used by each compiler when invoking a member function pointer, you can see that for all those compilers, when a non-virtual function is used with single inheritance (ie `delta=vtordisp=vindex=0`), the instance pointer doesn't enter into the calculation of what function to call. So, even with a garbage instance pointer, the correct function will be called. Inside that function, the `this` pointer that is received will be `garbage + delta = garbage`. (To put it another way -- garbage in, *unmodified* garbage out!) And at that point we can convert our garbage back into a function pointer. This method would not work if the "static function invoker" was a virtual function.

Using the code

The source code consists of the FastDelegate implementation (*FastDelegate.h*), and a demo *.cpp* file to illustrate the syntax. To use with MSVC, create a blank console app, and add these two files to it. For GNU, just type "g++ demo.cpp" on the command line.

The fast delegates will work with any combination of parameters, but to make it work on as many compilers as possible, you have to specify the number of parameters when declaring the delegate. There is a maximum of eight parameters, but it's trivial to increase this limit. The namespace `fastdelegate` is used. All of the messiness is in an inner namespace called `detail`.

`Fastdelegate`s can be bound to a member function or a static (free) function, using the constructor or `bind()`. They default to 0 (`null`). They can also be set to `null` with `clear()`. They can be tested for `null` using operator`!` or `empty()`.

Unlike most other implementations of delegates, equality operators (`==`, `!=`) are provided. They work even when inline functions are involved.

Here's an excerpt from *FastDelegateDemo.cpp* which shows most of the allowed operations. `CBaseClass` is a virtual base class of `CDerivedClass`. A flashy example could easily be devised; this is just to illustrate the syntax.

Hide Shrink Copy Code

```
using namespace fastdelegate;
int main(void)
{
    // Delegates with up to 8 parameters are supported.
    // Here's the case for a void function.
    // We declare a delegate and attach it to SimpleVoidFunction()
    printf("-- FastDelegate demo --\nA no-parameter
          delegate is declared using FastDelegate0\n\n");

    FastDelegate0 noparameterdelegate(&SimpleVoidFunction);
    noparameterdelegate();
    // invoke the delegate - this calls SimpleVoidFunction()
    printf("\n-- Examples using two-parameter delegates (int, char *) --\n\n");
    typedef FastDelegate2<int, char *> MyDelegate;
    MyDelegate funclist[10]; // delegates are initialized to empty
    CBaseClass a("Base A");
    CBaseClass b("Base B");
    CDerivedClass d;
    CDerivedClass c;

  // Binding a simple member function
    funclist[0].bind(&a, &CBaseClass::SimpleMemberFunction);
  // You can also bind static (free) functions
    funclist[1].bind(&SimpleStaticFunction);
  // and static member functions
    funclist[2].bind(&CBaseClass::StaticMemberFunction);
  // and const member functions
    funclist[3].bind(&a, &CBaseClass::ConstMemberFunction);
  // and virtual member functions.
    funclist[4].bind(&b, &CBaseClass::SimpleVirtualFunction);
  // You can also use the = operator. For static functions,
  // a fastdelegate looks identical to a simple function pointer.
    funclist[5] = &CBaseClass::StaticMemberFunction;
  // The weird rule about the class of derived
  // member function pointers is avoided.
  // Note that as well as .bind(), you can also use the
  // MakeDelegate() global function.
    funclist[6] = MakeDelegate(&d, &CBaseClass::SimpleVirtualFunction);
  // The worst case is an abstract virtual function of a
  // virtually-derived class with at least one non-virtual base class.
  // This is a VERY obscure situation, which you're unlikely to encounter
  // in the real world, but it's included as an extreme test.
    funclist[7].bind(&c, &CDerivedClass::TrickyVirtualFunction);
  // ...BUT in such cases you should be using the base class as an
  // interface, anyway. The next line calls exactly the same function.
    funclist[8].bind(&c, &COtherClass::TrickyVirtualFunction);
  // You can also bind directly using the constructor
    MyDelegate dg(&b, &CBaseClass::SimpleVirtualFunction);
    char *msg = "Looking for equal delegate";
    for (int i=0; i<10; i++) {
        printf("%d :", i);
        // The ==, !=, <=,<,>, and >= operators are provided
        // Note that they work even for inline functions.
        if (funclist[i]==dg) { msg = "Found equal delegate"; };
        // There are several ways to test for an empty delegate
        // You can use if (funclist[i])
        // or          if (!funclist.empty())
        // or          if (funclist[i]!=0)
        // or          if (!!funclist[i])
        if (funclist[i]) {
            // Invocation generates optimal assembly code.
            funclist[i](i, msg);
        } else {
            printf("Delegate is empty\n");
        };
    }
};
```

Non-void return values

Version 1.3 of the code adds the ability to have non-void return values. Like `std::unary_function`, the return type is the *last* parameter. It defaults to `void`, which preserves backwards compatibility, and also means that the most common case remains simple. I wanted to do this without losing performance on *any* platform. It turned out to be easy to do this for any compiler except MSVC6. There are two significant limitations of VC6:

1. you can't have `void` as a default template argument.
2. you can't return a `void`.

I've got around it with two tricks:

1. I create a dummy class called `DefaultVoid`. I convert this to a `void` when required.
2. Whenever I need to return `void`, I return a `const void *` instead. Such pointers are returned in the `EAX`register. From the compiler's point of view, there is absolutely no difference between a `void` function and a`void *` function where the return value is never used. The final insight was the realization that it is not possible to convert from a `void` to a `void *` within the invocation function without generating inefficient code. But if you convert the function pointer the instant that you receive it, all the work is done at compile time (i.e., you need to convert the definition of the function, not the return value itself).

There is one breaking change: all instances of `FastDelegate0` must be changed to `FastDelegate0<>`. This change can be performed with a global search-and-replace

through all your files, so I believe it should not be too onerous. I believe this change leads to more intuitive syntax: the declaration of any kind of `void FastDelegate`is now the same as a function declaration, except that `()` is replaced with `<>`. If this change really annoys anyone, you can modify the header file: wrap the `FastDelegate0<>` definition inside a separate `namespace newstyle`
`{` and `}` `typedef newstyle::FastDelegate0<> FastDelegate0;`. You would need to change the corresponding `MakeDelegate` function too.

Passing a FastDelegate as a function parameter

The `MakeDelegate` template allows you to use a `FastDelegate` as a drop-in replacement for a function pointer. A typical application is to have a `FastDelegate` as a private member of a class and use a modifier function to set it (like a Microsoft `__event`). For example:

Hide Copy Code

```
// Accepts any function with a signature like: int func(double, double);
class A {
public:
    typedef FastDelegate2<double, double, int> FunctionA;
    void setFunction(FunctionA somefunc){ m_HiddenDelegate = somefunc; }
private:
    FunctionA m_HiddenDelegate;
};
// To set the delegate, the syntax is:
A a;
a.setFunction( MakeDelegate(&someClass, &someMember) ); // for member functions, or
a.setFunction( &somefreefunction ); // for a non-class or static function
```

Natural syntax and Boost compatibility (new to 1.4)

Jody Hagins has enhanced the FastDelegateN classes to allow the same attractive syntax provided by recent versions of `Boost.Function` and `Boost.Signal`. On compilers with partial template specialization, you have the option of writing `FastDelegate< int (char *, double)>` instead of `FastDelegate2<char *, double,int>`. Fantastic work, Jody! If your code needs to compile on VC6, VC7.0, or Borland, you need to use the old, portable syntax. I've made changes to ensure that the old and new syntax are 100% equivalent and can be used interchangeably.

Jody also contributed a helper function, `bind`, to allow code written for `Boost.Function` and `Boost.Bind` to be rapidly converted to `FastDelegate`s. This allows you to quickly determine how much the performance of your existing code would improve if you switched to FastDelegates. It can be found in "*FastDelegateBind.h*". If we have the code:

Hide Copy Code

```
    using boost::bind;
    bind(&Foo:func, &foo, _1, _2);
```

we should be able to replace the "using" with using fastdelegate::bind and everything should work fine. Warning: The arguments to bind are ignored! No actual binding is performed. The behavior is equivalent toboost::bind only in the trivial (but most common) case where only the basic placeholder arguments _1, _2, _3, etc. are used. A future release may support boost::bind properly.

Ordered comparison operators (new to 1.4)

FastDelegates of the same type can now be compared with <, >, <=, >=. Member function pointers don't support these operators, but they can be emulated with a simple binary comparison using memcmp(). The resulting strict weak ordering is physically meaningless, and is compiler-dependent, but it allows them to be stored in ordered containers like std:set.

The DelegateMemento class (new to 1.4)

A new class, DelegateMemento, is provided to allow disparate collections of delegates. Two extra members have been added to each FastDelegate class:

Hide Copy Code

```
const DelegateMemento GetMemento() const;
void SetMemento(const DelegateMemento mem);
```

DelegegateMementos can be copied and compared to one another (==, !=, >, <, >=, <=), allowing them to be stored in any ordered or unordered container. They can be used as a replacement for a union of function pointers in C. As with a union of disparate types, it is your responsibility to ensure that you are using the same type consistently. For example, if you get a DelegateMemento from a FastDelegate2 and save it into aFastDelegate3, your program will probably crash at runtime when you invoke it. In the future, I may add a debug mode which uses the typeid operator to enforce type safety. DelegateMementos are intended primarily for use in other libraries, rather than in general user code. An important usage is Windows messaging, where a dynamic std::map<MESSAGE, DelegateMemento> can replace the static message maps found in MFC and WTL. But that's another article.

Implicit conversion to bool (new to 1.5)

You can now use the `if (dg) {...}` syntax (where `dg` is a fast delegate) as an alternative to `if(!dg.empty())`, `if (dg!=0)` or even the ugly but efficient `if (!!dg)`. If you're just using the code, all you need to know is that it works correctly on all compilers, and it does not inadvertently allow other operations.

The implementation was more difficult than expected. Merely supplying an `operator bool` is dangerous because it lets you write things like `int a = dg;` when you probably meant `int a = dg();`. The solution is to use the Safe Bool idiom [Karlsson]: a conversion to a private member data pointer is used instead of `bool`. Unfortunately, the usual Safe Bool implementation mucks up the `if (dg==0)` syntax, and some compilers have bugs in their implementations of member data pointers (hmm, another article?), so I had to develop a couple of tricks. One method which others have used in the past is to allow comparisons with integers, and `ASSERT` if the integer is non-zero. Instead, I use a more verbose method. The only side-effect is that comparisons with function pointers are more optimal! Ordered comparisons with the constant 0 are not supported (but it is valid to compare with a function pointer which happens to be null).

License

The source code attached to this article is released into the public domain. You may use it for any purpose. Frankly, writing the article was about ten times as much work as writing the code. Of course, if you create great software using the code, I would be interested to hear about it. And submissions are always welcome.

Portability

Because it relies on behavior that is not defined by the standard, I've been careful to test the code on many compilers. Ironically, it's more portable than a lot of 'standard' code, because most compilers don't fully conform to the standard. It has also become safer through being widely known. The major compiler vendors and several members of the C++ Standards commitee know about the techniques presented here (in many cases, the lead compiler developers have contacted me about the article). There is negligible risk that vendors will make a change that would irreparably break the code. For example, no changes were required to support Microsoft's first 64 bit compilers. Codeplay has even used FastDelegates for internal testing of their VectorC compiler (it's not quite an endorsement, but very close).

The `FastDelegate` implementation has been been tested on Windows, DOS, Solaris, BSD, and several flavors of Linux, using x86, AMD64, Itanium, SPARC, MIPS, .NET virtual machines, and some embedded processors. The following compilers have been tested successfully:

- Microsoft Visual C++ 6.0, 7.0 (.NET), 7.1 (.NET 2003) and 8.0 (2005) Beta (including /clr 'managed C++').
- {Compiled and linked, and ASM listing inspected, but not run} Microsoft 8.0 Beta 2 for Itanium and for AMD64.
- GNU G++ 2.95, 3.0, 3.1, 3.2 and 3.3 (Linux, Solaris, and Windows (MingW, DevCpp, Bloodshed)).
- Borland C++ Builder 5.5.1 and 6.1.
- Digital Mars C++ 8.38 (x86, both 32-bit and 16-bit, Windows and all DOS memory models).
- Intel C++ for Windows (x86) 8.0 and 8.1.
- Metrowerks CodeWarrior for Windows 9.1 (in both C++ and EC++ modes).
- CodePlay VectorC 2.2.1 (Windows, Playstation 2). Earlier versions are *not* supported.
- Portland Group PGI Workstation 5.2 for Linux, 32-bit.
- {Compiled, but not linked or run} Comeau C++ 4.3 (x86 NetBSD).
- {Compiled and linked, and ASM listing inspected, but not run} Intel C++ 8.0 and 8.1 for Itanium, Intel C++ 8.1 for EM64T/AMD64.

Here is the status of all other C++ compilers that I know of which are still in use:

- Open Watcom WCL: Will work once member function templates are added to the compiler. The core code (casting between member function pointers) works on WCL 1.2.
- LVMM: The core code works, but compiler has too many bugs at present.
- IBM Visual Age and XLC: Should work, because IBM claims 100% binary compatibility with GCC.
- Pathscale EKO: Should work, it is also binary compatible with GCC.
- All compilers using the EDG front-end (GreenHills, Apogee, WindRiver, etc.) should also work.
- Paradigm C++: UNKNOWN, but seems to be just a repackaging of on an early Borland compiler.
- Sun C++: UNKNOWN.
- Compaq CXX: UNKNOWN.
- HP aCC: UNKNOWN.

And yet some people are still complaining that the code is not portable! (Sigh).

Conclusion

What started as an explanation of a few lines of code I'd written has turned into a monstrous tutorial on the perils of an obscure part of the language. I also discovered previously unreported bugs and incompatibilities in six popular compilers. It's an awful lot of work for two lines of assembly code!

I hope that I've cleared up some of the misconceptions about the murky world of member function pointers and delegates. We've seen that much of the weirdness of member function pointers arise because they are implemented very differently by various compilers. We've also seen that, contrary to popular opinion, delegates are not a complicated, high-level construct, but are in fact very simple. I hope that I've convinced you that they should be part of the language. There is a reasonable chance that some form of direct compiler support for delegates will be added to C++ when the C++0x standard is released. (Start lobbying the Standards committee!)

To my knowledge, no previous implementation of delegates in C++ is as efficient or easy to use as the FastDelegates I've presented here. It may amuse you to learn that most of it was programmed one-handed while I tried to get my baby daughter to sleep... I hope you find it useful.

References

- [GoF] "Design Patterns: Elements of Reusable Object-Oriented Software", E. Gamma, R. Helm, R. Johnson, and J. Vlissides.

I've looked at dozens of websites while researching this article. Here are a few of the most interesting ones:

- [Boost]. Delegates can be implemented with a combination of `boost::function` and `boost::bind`.`Boost::signals` is one of the most sophisticated event/messaging system available. Most of the boost libraries require a highly standards-conforming compiler.
- [Loki]. Loki provides 'functors' which are delegates with bindable parameters. They are very similar to`boost::function`. It's likely that Loki will eventually merge with boost.
- [Qt]. The Qt library includes a Signal/Slot mechanism (i.e., delegates). For this to work, you have to run a special preprocessor on your code before compiling. Performance is poor, but it works on compilers with very poor template support.
- [Libsigc++]. An event system based on Qt's. It avoids the Qt's special preprocessor, but requires that every target be derived from a base object class (using virtual inheritance - yuck!).
- [JanGray] An MSDN "Under the Hood" article describing the object model for Microsoft C/C++ 7. It is still applicable to all subsequent compiler versions.
- [Hickey]. An old (1994) delegate implementation that avoids memory allocations. Assumes that all pointer-to-member functions are the same size, so it doesn't work on MSVC. There's a helpful discussion of the code here.
- [Haendal]. A website dedicated to function pointers?! Not much detail about member function pointers though.

- [Karlsson]. The Safe Bool Idiom.
- [Meyers]. Scott Meyer's article on overloading `operator ->*`. Note that the classic smart pointer implementations (Loki and boost) don't bother.
- [Sutter1]. Generalized function pointers: a discussion of how `boost::function` has been accepted into the new C++ standard.
- [Sutter2]. Generalizing the Observer pattern (essentially, multicast delegates) using `std::tr1::function`. Discusses the limitations of the failure of `boost::function` to provide operator `==`.
- [Sutter3]. Herb Sutter's Guru of the Week article on generic callbacks.
- [Dlugosz]. A recent implementation of delegates/closures which, like mine, is optimally efficient, but which only works on MSVC7 and 7.1.

History

- 1.0 - 24 May 2004. Initial release. (Thanks to Arnold the Aardvaark for suggesting I call them 'delegates' instead of 'bound member function pointers'.)
- 1.1 - 28 May 2004. Minor changes (mostly cosmetic) to the code:
  - CODE: Prevents unsafe use of evil static function hack.
  - CODE: Improved syntax for horrible_cast (thanks Paul Bludov).
  - CODE: Now works on Metrowerks MWCC and Intel ICL (IA32), and compiles on Intel Itanium ICL.
- 1.2 - 27 June 2004. Major improvements in portability and robustness:
  - ARTICLE: Added 'Sordid Tale of Microsoft's Method' + discussion of VC6 bug.
  - ARTICLE: Corrected errors regarding GCC and Intel's implementation.
  - ARTICLE: About twenty minor corrections and additions.
  - CODE: Now works on Borland C++ Builder 5.5.
  - CODE: Now works on /clr "managed C++" code on VC7, VC7.1.
  - CODE: Comeau C++ now compiles without warnings.
  - CODE: Improved warning and error messages. Non-standard hacks now have compile-time checks to make them safer.
  - CODE: Prevented the virtual inheritance case from generating incorrect code on VC6 and earlier.
  - CODE: If calling a const member function, a const class pointer can be used.
  - CODE: `MakeDelegate()` global helper function added to simplify pass-by-value.
  - CODE: Added `fastdelegate.clear()`.
- 1.2.1 - 16 July 2004. Minor bugfix.
  - CODE: Workaround for GCC bug (const member function pointers in templates).
- 1.3 - 25 Oct 2004. Non-void return values and seamless support for Microsoft compilers. Incorporated suggestions from Neville Franks, Ratheous, and others.
  - ARTICLE: Corrected an error about the MFP representation in CFront 1 (thanks Scott Meyers).
  - ARTICLE: Added details about EDG member pointer representation.
  - ARTICLE: Added sections on return values, license, and the `MakeDelegate` function.
  - ARTICLE: Numerous minor additions and clarifications.

    - CODE: Support for (non-void) return values.
    - CODE: Now uses a clever hack invented by John M. Dlugosz to eliminate the need for the`FASTDELEGATEDECLARE` macro and the problems with VC6.
    - CODE: A simple program ("Hopter") was made to auto-generate the *FastDelegate.h* file, reducing the use of macros. Error messages should be more comprehensible.
    - CODE: Added include guards.
    - CODE: Added `FastDelegate::empty()` to test if invocation is safe.
    - CODE: Now works on VS 2005 Express Beta, Portland Group C++.
- 1.4 - 6 Jan 2005. Ordered comparisons, the `DelegateMemento` class, function declarator syntax, and limited Boost compatibility. Incorporated code by Jody Hagins and suggestions by Rob Marsden.
    - ARTICLE: Major rewrite of the compiler implementation section. Added details about CodePlay.
    - ARTICLE: Added sections describing the new features.
    - ARTICLE: Improvements were made to the wording in almost every section to reduce ambiguity, and corrected minor errors.
    - CODE: Now supports the simple `boost::function`-style syntax. For example, `FastDelegate<int (char *, double)>`.
    - CODE: Added `DelegateMemento` class to allow collections of disparate delegates.
    - CODE: Added `>`, `<`, `>=`, `<=` comparison operators to allow delegates to be stored in ordered collections.
    - CODE: Added `bind()` function in *FastDelegateBind.h* for limited Boost compatibility (Thanks Jody Hagins).
    - CODE: Now compiles without warnings on *all* of the supported compilers.
- 1.4.1 - 14 Feb 2005. Minor bugfix.
    - ARTICLE: Added details about implementation used by Open64, Pathscale EKO, and related compilers.
    - CODE: Confirmed compatibility with MSVC-Itanium and MSVC-AMD64, ICL-EM64T, and Open64-Itanium. (Thanks Stuart Dootson).
    - CODE: Bugfix: Now treats null function pointers as null delegates, so that `dg==0` is now equivalent to`dg.empty()`, and `dg=0` is now equivalent to `dg.clear()`. (Thanks elfric).
- 1.5 - 30 Mar 2005. Safe_bool idiom and full CodePlay support. Special thanks to Jens-Uwe Dolinsky, (lead developer of the CodePlay VectorC front end).
    - ARTICLE: Added "Conversion to Bool" section.
    - ARTICLE: Added link to Russian translation (Thanks Denis Bulichenko).
    - CODE: Confirmed that CodePlay VectorC 2.2.1 can now compile the code.
    - CODE: Bugfix: On Metrowerks, `empty()` could sometimes return `true` for a non-null delegate. (This is because that compiler does not have a unique value for a null member function pointer, in violation of the Standard).
    - CODE: Now supports the safe_bool idiom, so that `if (dg)` is now equivalent to `if(!dg.empty())`. (This feature was requested by Neville Franks, several hundred years ago 😊).

- CODE: Assignment and comparison of static function pointers has been optimised.

License

This article, along with any associated source code and files, is licensed under The Code Project Open License (CPOL)

About the Author

Don Clugston

Engineer
Germany 🇩🇪

I'm an Australian physicist/software engineer living in Leipzig, Germany. I've published papers about windows (the glass kind), semiconductor device physics, environmental philosophy and theology, vacuum systems, and now (at CodeProject) the innards of C++. (Yes, my C.V. is a terrible mess.) I'm a major contributor to the D programming language. I can read New Testament Greek, and can speak German with a dreadful accent.

I have one wife, one son (pictured), and one daughter, who are all much cuter than I am.

"The light shines in the darkness, but the darkness has not overcome it."