isocpp.org

Pointers to Member Functions, C++ FAQ

Pointers to Member Functions ¶ Δ

Pointers to Member Functions

Is the type of “pointer-to-member-function” different from “pointer-to-function”? ¶ Δ

Yep.

Consider the following function:

```
1.   int f(char a, float b);
```

The type of this function is different depending on whether it is an ordinary function or a non- `static` member function of some class:

- Its type is “ `int (*)(char,float)` ” if an ordinary function
- Its type is “ `int (Fred::*)(char,float)` ” if a non- `static` member function of `class Fred`

Note: if it’s a `static` member function of `class Fred` , its type is the same as if it were an ordinary function: “ `int (*)(char,float)` ”.

How do I pass a pointer-to-member-function to a signal handler, X event callback, system call that starts a thread/task, etc? ¶ Δ

Don’t.

Because a member function is meaningless without an object to invoke it on, you can’t do this directly (if The X Window System was rewritten in C++, it would probably pass references to *objects* around, not just pointers to functions; naturally the objects would embody the required function and probably a whole lot more).

As a patch for existing software, use a top-level (non-member) function as a wrapper which takes an object obtained through some other technique. Depending on the routine you're calling, this "other technique" might be trivial or might require a little work on your part. The system call that starts a thread, for example, might require you to pass a function pointer along with a `void*`, so you can pass the object pointer in the `void*`. Many real-time operating systems do something similar for the function that starts a new task. Worst case you could store the object pointer in a global variable; this might be required for Unix signal handlers (but globals are, in general, undesired). In any case, the top-level function would call the desired member function on the object.

Here's an example of the worst case (using a global). Suppose you want to call `Fred::memberFn()` on interrupt:

```
1.  class Fred {
2.  public:
3.    void memberFn();
4.    static void staticMemberFn();  // A static member function can usually handle it
5.    // ...
6.  };
7.
8.  // Wrapper function uses a global to remember the object:
9.  Fred* object_which_will_handle_signal;
10.
11.  void Fred_memberFn_wrapper()
```

```
12.   {
13.     object_which_will_handle_signal->memberFn();
14.   }
15.
16.   int main()
17.   {
18.     /* signal(SIGINT, Fred::memberFn); */   // Can NOT
      do this
19.     signal(SIGINT, Fred_memberFn_wrapper);  // Okay
20.     signal(SIGINT, Fred::staticMemberFn);   // Okay
      usually; see below
21.     // ...
22.   }
```

Note: `static` member functions do not require an actual object to be invoked, so pointers-to- `static` -member-functions are *usually* type-compatible with regular pointers-to-functions. However, although it probably works on most compilers, it actually would have to be an `extern "C"` non-member function to be correct, since “C linkage” doesn’t only cover things like name mangling, but also calling conventions, which might be different between C and C++.

Why do I keep getting compile errors (type mismatch) when I try to use a member function as an interrupt service routine? ¶ Δ

This is a special case of the previous two questions, therefore read the previous two answers first.

Non- `static` member functions have a hidden parameter that corresponds to the `this` pointer. The `this` pointer points to the instance data for the object. The interrupt hardware/firmware in the system is not capable of providing the `this` pointer argument. You must use “normal” functions (non class members) or `static` member functions as interrupt service routines.

One possible solution is to use a `static` member as the interrupt service routine and have that function look somewhere to find the instance/member pair that should be called on interrupt. Thus the effect is that a member function is invoked on an interrupt, but for technical reasons you need to call an intermediate function first.

Why am I having trouble taking the address of a C++ function? ¶ Δ

Short answer: if you’re trying to store it into (or pass it as) a pointer-to-function, then that’s the problem — this is a corollary to the previous FAQ.

Long answer: In C++, member functions have an implicit parameter which points to the object (the `this` pointer inside the member function). Normal C functions can be thought of as having a different calling convention from member functions, so the types of their pointers (pointer-to-member-function vs pointer-to-function) are different and incompatible. C++ introduces a new type of pointer, called a pointer-to-member, which can be invoked only by providing an object.

NOTE: do *not* attempt to “cast” a pointer-to-member-function into a pointer-to-function; the result is undefined and probably disastrous. E.g., a pointer-to-member-function is *not* required to contain the machine address of the appropriate function. As was said in the last example, if you have a pointer to a regular C function, use either a top-level (non-member) function, or a `static` (class) member function.

How can I avoid syntax errors when creating pointers to members? ¶ Δ

Use a `typedef` .

Yea, right, I know: *you* are different. You are *smart*. You can do this stuff without a `typedef` . Sigh. I have received many emails from people who, like you, refused to take the simple advice of this FAQ. They wasted hours and hours of their time, when 10 seconds worth of `typedef` s would have simplified their lives. Plus, face it, you are not writing code that only you can read; you are hopefully writing your code that others will also be able to read — when they’re tired — when they have their own deadlines and

their own challenges. So why intentionally make life harder on yourself and on others? Be smart: use a `typedef`.

Here's a sample class:

```
1.  class Fred {
2.  public:
3.    int f(char x, float y);
4.    int g(char x, float y);
5.    int h(char x, float y);
6.    int i(char x, float y);
7.    // ...
8.  };
```

The typedef is trivial:

```
1.  typedef  int (Fred::*FredMemFn)(char x, float y);
    // Please do this!
```

That's it! `FredMemFn` is the type name, and a pointer of that type points to any member of `Fred` that takes `(char,float)`, such as `Fred`'s `f`, `g`, `h` and `i`.

It's then *trivial* to declare a member-function pointer:

```
1.  int main()
```

```
2.   {
3.     FredMemFn p = &Fred::f;
4.     // ...
5.   }
```

And it's also *trivial* to declare functions that receive member-function pointers:

```
1.   void userCode(FredMemFn p)
2.   { /*...*/ }
```

And it's also *trivial* to declare functions that return member-function pointers:

```
1.   FredMemFn userCode()
2.   { /*...*/ }
```

So please, use a `typedef`. Either that or *do not send me email* about the problems you have with your member-function pointers!

How can I avoid syntax errors when calling a member function using a pointer-to-member-function? ¶ Δ

If you have access to a compiler and standard library that implements the appropriate parts of the upcoming C++17 standard, use `std::invoke`. Otherwise, use a `#define` macro.

Please.

Pretty please.

I get way too many emails from confused people who refused to take this advice. It's so simple. I know, you don't *need* `std::invoke` or a macro, and the expert you talked to can do it without either of them, but please don't let your ego get in the way of what's important: money. Other programmers will need to read / maintain your code. Yes, I know: you are smarter than everyone else; fine. And you are awesome; fine. But don't add unnecessary complexity to your code.

Using `std::invoke` is trivial. Note: `FredMemFn` is a `typedef` for a pointer-to-member type:

```
1.  void userCode(Fred& fred, FredMemFn p)  // Use a typedef for pointer-to-member types
2.  {
3.    int ans = std::invoke(fred, p, 'x', 3.14);
4.    // Would normally be: int ans = (fred.*p)('x', 3.14);
5.
6.    // ...
7.  }
```

If you can't use `std::invoke`, reduce maintenance cost by, paradoxically, using a `#define` macro in this particular case.

(Normally I dislike `#define` macros, but you should use them with pointers to members because they improve the readability and writability of that sort of code.)

The macro is trivial:

```
1.  #define CALL_MEMBER_FN(object,ptrToMember)
```

```
((object).*(ptrToMember))
```

Using the macro is also trivial. Note: `FredMemFn` is a `typedef` for a pointer-to-member type:

```
1.  void userCode(Fred& fred, FredMemFn p)  // Use a typedef for pointer-to-member types
2.  {
3.    int ans = CALL_MEMBER_FN(fred,p)('x', 3.14);
4.    // Would normally be: int ans = (fred.*p)('x', 3.14);
5.
6.    // ...
7.  }
```

The reason `std::invoke` or this macro is a good idea is because member function invocations are often a *lot* more complex than the simple example just given. The difference in readability and writability is significant. `comp.lang.c++` has had to endure hundreds and hundreds of postings from confused programmers who couldn't quite get the syntax right. Almost all these errors would have vanished had they used `std::invoke` or the above macro.

Note: `#define` macros are evil in 4 different ways: evil#1, evil#2, evil#3, and evil#4. But they're still useful sometimes. But you should still feel a vague sense of shame after using them.

How do I create and use an array of pointer-to-member-function? ¶ Δ

Use *both* the `typedef` *and* `std::invoke` or the `#define` macro described earlier, and you're 90% done.

Step 1: create a `typedef` :

```
class Fred {
public:
  int f(char x, float y);
  int g(char x, float y);
  int h(char x, float y);
  int i(char x, float y);
  // ...
};

// FredMemFn points to a member of Fred that takes (char,float)
typedef  int (Fred::*FredMemFn)(char x, float y);
```

Step 2: create a `#define` macro if you don't have `std::invoke` :

```
#define CALL_MEMBER_FN(object,ptrToMember) ((object).*(ptrToMember))
```

Now your array of pointers-to-member-functions is straightforward:

```
FredMemFn a[] = { &Fred::f, &Fred::g, &Fred::h,
```

```
    &Fred::i };
```

And your usage of one of the member function pointers is also straightforward:

```
1.  void userCode(Fred& fred, int memFnNum)
2.  {
3.    // Assume memFnNum is between 0 and 3 inclusive:
4.    std::invoke(fred, a[memFnNum], 'x', 3.14);
5.  }
```

or if you don't have `std::invoke` ,

```
1.  void userCode(Fred& fred, int memFnNum)
2.  {
3.    // Assume memFnNum is between 0 and 3 inclusive:
4.    CALL_MEMBER_FN(fred, a[memFnNum]) ('x', 3.14);
5.  }
```

Note: `#define` macros are evil in 4 different ways: evil#1, evil#2, evil#3, and evil#4. But they're still useful sometimes. Feel ashamed, feel guilty, but when an evil construct like a macro improves your software, use it.

How do I declare a pointer-to-member-function that points to a `const` member function? ¶ Δ

Short answer: add a `const` to the right of the `)` when you use a `typedef` to

declare the member-function-pointer type.

For example, suppose you want a pointer-to-member-function that points at `Fred::f`, `Fred::g` or `Fred::h`:

```
1.  class Fred {
2.  public:
3.    int f(int i) const;
4.    int g(int i) const;
5.    int h(int j) const;
6.    // ...
7.  };
```

Then when you use a `typedef` to declare the member-function-pointer type, it should look like this:

```
1.  // FredMemFn points to a const member-function of
    Fred that takes (int)
2.  typedef  int (Fred::*FredMemFn)(int) const;
3.                                        ↑↑↑↑↑ // Points
    only to member functions decorated with const
```

That's it!

Then you can declare/pass/return member-function pointers just like normal:

```
1.   void foo(FredMemFn p)  // Pass a member-function
     pointer
2.   { /*...*/ }
3.
4.   FredMemFn bar()        // Return a member-function
     pointer
5.   { /*...*/ }
6.
7.   void baz()
8.   {
9.     FredMemFn p = &Fred::f;  // Declare a member-
     function pointer
10.    FredMemFn a[10];         // Declare an array of
     member-function pointers
11.    // ...
12.  }
```

What is the difference between the `.*` and `->*` operators? ¶ Δ

You won't need to understand this if you use `std::invoke` or a macro for member-function-pointer calls. Oh yea, *please* use `std::invoke` or a macro in this case. And did I mention that you should use `std::invoke` or a macro in this case??!?

But if you really want to avoid `std::invoke` or the macro, sigh, groan, okay, here it

is: use `.*` when the left-hand argument is a reference to an object, and `->*` when it is a pointer to an object.

For example:

```
class Fred { /*...*/ };

typedef  int (Fred::*FredMemFn)(int i, double d);
// use a typedef!!! please!!!

void sample(Fred x, Fred& y, Fred* z, FredMemFn
func)
{
  x.*func(42, 3.14);
  y.*func(42, 3.14);
  z->*func(42, 3.14);
}
```

BUT please consider using a `std::invoke` or macro instead:

```
void sample(Fred x, Fred& y, Fred* z, FredMemFn
func)
{
```

```
3.      std::invoke(x, func, 42, 3.14);
4.      std::invoke(y, func, 42, 3.14);
5.      std::invoke(*z, func, 42, 3.14);
6.    }
```

or

```
1.    void sample(Fred x, Fred& y, Fred* z, FredMemFn
      func)
2.    {
3.      CALL_MEMBER_FN(x,func)(42, 3.14);
4.      CALL_MEMBER_FN(y,func)(42, 3.14);
5.      CALL_MEMBER_FN(*z,func)(42, 3.14);
6.    }
```

As discussed earlier, real-world invocations are often much more complicated than the simple ones here, so using a `std::invoke` or macro will typically improve your code's writability and readability.

Can I convert a pointer-to-member-function to a `void*` ? ¶ Δ

No!

```
1.    class Fred {
2.    public:
```

```
 3.     int f(char x, float y);
 4.     int g(char x, float y);
 5.     int h(char x, float y);
 6.     int i(char x, float y);
 7.     // ...
 8.   };
 9.
10.   // FredMemFn points to a member of Fred that takes
     (char,float)
11.   typedef  int (Fred::*FredMemFn)(char x, float y);
12.
13.   #define CALL_MEMBER_FN(object,ptrToMember)
     ((object).*(ptrToMember))
14.
15.   int callit(Fred& o, FredMemFn p, char x, float y)
16.   {
17.     return CALL_MEMBER_FN(o,p)(x, y);
18.   }
19.
```

```
20.   int main()

21.   {

22.     FredMemFn p = &Fred::f;

23.     void* p2 = (void*)p;                  // ←
      illegal!!

24.     Fred o;

25.     callit(o, p, 'x', 3.14f);             // okay

26.     callit(o, FredMemFn(p2), 'x', 3.14f); // might
      fail!!

27.     // ...

28.   }
```

Technical details: pointers to member functions and pointers to data are not necessarily represented in the same way. A pointer to a member function might be a data structure rather than a single pointer. Think about it: if it's pointing at a virtual function, it might not actually be pointing at a statically resolvable pile of code, so it might not even be a normal address — it might be a different data structure of some sort.

Please do not email me if the above *seems to work* on your particular version of your particular compiler on your particular operating system. I don't care. It's illegal, period.

Can I convert a pointer-to-function to a `void*` ? ¶ Δ

No!

```
1.   int f(char x, float y);
```

```
 2.   int g(char x, float y);
 3.
 4.   typedef int(*FunctPtr)(char,float);
 5.
 6.   int callit(FunctPtr p, char x, float y)
 7.   {
 8.     return p(x, y);
 9.   }
10.
11.   int main()
12.   {
13.     FunctPtr p = f;
14.     void* p2 = (void*)p;              // ← illegal!!
15.     callit(p, 'x', 3.14f);            // okay
16.     callit(FunctPtr(p2), 'x', 3.14f); // might fail!!
17.     // ...
18.   }
```

Technical details: `void*` pointers are pointers to data, and function pointers point to

functions. The language does not require functions and data to be in the same address space, so, by way of *example* and *not limitation*, on architectures that have them in different address spaces, the two different pointer types will not be comparable.

Please do not email me if the above *seems to work* on your particular version of your particular compiler on your particular operating system. I don't care. It's illegal, period.

I need something like function-pointers, but with more flexibility and/or thread-safety; is there another way? ¶ Δ

Use a functionoid.

What the heck is a functionoid, and why would I use one? ¶ Δ

Functionoids are functions on steroids. Functionoids are strictly more powerful than functions, and that extra power solves some (not all) of the challenges typically faced when you use function-pointers.

Let's work an example showing a traditional use of function-pointers, then we'll translate that example into functionoids. The traditional function-pointer idea is to have a bunch of compatible functions:

```
1.   int funct1( /*...params...*/ ) { /*...code...*/ }

2.   int funct2( /*...params...*/ ) { /*...code...*/ }

3.   int funct3( /*...params...*/ ) { /*...code...*/ }
```

Then you access those by function-pointers:

```
1.   typedef int(*FunctPtr)( /*...params...*/ );

2.

3.   void myCode(FunctPtr f)
```

```
4.   {
5.     // ...
6.     f( /*...args-go-here...*/ );
7.     // ...
8.   }
```

Sometimes people create an array of these function-pointers:

```
1.   FunctPtr array[10];
2.   array[0] = funct1;
3.   array[1] = funct1;
4.   array[2] = funct3;
5.   array[3] = funct2;
6.   // ...
```

In which case they call the function by accessing the array:

```
1.   array[i]( /*...args-go-here...*/ );
```

With functionoids, you first create a base class with a pure-virtual method:

```
1.   class Funct {
```

```
2.   public:
3.     virtual int doit(int x) = 0;
4.     virtual ~Funct() = 0;
5.   };
6.
7.   inline Funct::~Funct() { }  // defined even though
    it's pure virtual; it's faster this way; trust me
```

Then instead of three functions, you create three derived classes:

```
1.   class Funct1 : public Funct {
2.   public:
3.     virtual int doit(int x) { /*...code from
    funct1...*/ }
4.   };
5.
6.   class Funct2 : public Funct {
7.   public:
8.     virtual int doit(int x) { /*...code from
    funct2...*/ }
9.   };
```

```
10.
11.   class Funct3 : public Funct {
12.   public:
13.     virtual int doit(int x) { /*...code from
      funct3...*/ }
14.   };
```

Then instead of passing a function-pointer, you pass a `Funct*`. I'll create a `typedef` called `FunctPtr` merely to make the rest of the code similar to the old-fashioned approach:

```
1.   typedef Funct* FunctPtr;
2.
3.   void myCode(FunctPtr f)
4.   {
5.     // ...
6.     f->doit( /*...args-go-here...*/ );
7.     // ...
8.   }
```

You can create an array of them in almost the same way:

```
1.   FunctPtr array[10];

2.   array[0] = new Funct1( /*...ctor-args...*/ );

3.   array[1] = new Funct1( /*...ctor-args...*/ );

4.   array[2] = new Funct3( /*...ctor-args...*/ );

5.   array[3] = new Funct2( /*...ctor-args...*/ );

6.   // ...
```

This gives us the first hint about where functionoids are strictly more powerful than function-pointers: the fact that the functionoid approach has arguments you can pass to the ctors (shown above as …*ctor-args*…) whereas the function-pointers version does not. Think of a functionoid object as a freeze-dried function-call (emphasis on the word *call*). Unlike a pointer to a function, a functionoid is (conceptually) a pointer to a *partially called* function. Imagine for the moment a technology that lets you pass some-but-not-all arguments to a function, then lets you freeze-dry that (partially completed) call. Pretend that technology gives you back some sort of magic pointer to that freeze-dried partially-completed function-call. Then later you pass the *remaining* args using that pointer, and the system magically takes your original args (that were freeze-dried), combines them with any local variables that the function calculated prior to being freeze-dried, combines all that with the newly passed args, and continues the function's execution where it left off when it was freeze-dried. That might sound like science fiction, but it's conceptually what functionoids let you do. *Plus* they let you repeatedly "complete" that freeze-dried function-call with various different "remaining parameters," as often as you like. *Plus* they allow (not require) you to change the freeze-dried state when it gets called, meaning functionoids can remember information from one call to the next.

Let's get our feet back on the ground and we'll work a couple of examples to explain what all that mumbo jumbo really means.

Suppose the original functions (in the old-fashioned function-pointer style) took slightly different parameters.

```
1.   int funct1(int x, float y)
2.   { /*...code...*/ }
3.
4.   int funct2(int x, const std::string& y, int z)
5.   { /*...code...*/ }
6.
7.   int funct3(int x, const std::vector<double>& y)
8.   { /*...code...*/ }
```

When the parameters are different, the old-fashioned function-pointers approach is difficult to use, since the caller doesn't know which parameters to pass (the caller merely has a pointer to the function, not the function's *name* or, when the parameters are different, the number and types of its parameters) (do *not* write me an email about this; yes you can do it, but you have to stand on your head and do messy things; but do *not* write me about it — use functionoids instead).

With functionoids, the situation is, at least sometimes, much better. Since a functionoid can be thought of as a freeze-dried function *call*, just take the un-common args, such as the ones I've called `y` and/or `z`, and make them args to the corresponding ctors. You may also pass the common args (in this case the `int` called `x`) to the ctor, but you don't have to — you have the option of passing it/them to the pure virtual `doit()` method instead. I'll assume you want to pass `x` into `doit()` and `y` and/or `z` into the ctors:

```
1.   class Funct {
2.   public:
```

```
3.     virtual int doit(int x) = 0;
4.   };
```

Then instead of three functions, you create three derived classes:

```
1.   class Funct1 : public Funct {
2.   public:
3.     Funct1(float y) : y_(y) { }
4.     virtual int doit(int x) { /*...code from
     funct1...*/ }
5.   private:
6.     float y_;
7.   };
8.
9.   class Funct2 : public Funct {
10.  public:
11.    Funct2(const std::string& y, int z) : y_(y), z_(z)
     { }
12.    virtual int doit(int x) { /*...code from
     funct2...*/ }
13.  private:
```

```
14.     std::string y_;
15.     int z_;
16.   };
17.
18.   class Funct3 : public Funct {
19.   public:
20.     Funct3(const std::vector<double>& y) : y_(y) { }
21.     virtual int doit(int x) { /*...code from
      funct3...*/ }
22.   private:
23.     std::vector<double> y_;
24.   };
```

Now you see that the ctor's parameters get freeze-dried into the functionoid when you create the array of functionoids:

```
1.   FunctPtr array[10];
2.
3.   array[0] = new Funct1(3.14f);
4.
```

```
 5.    array[1] = new Funct1(2.18f);
 6.
 7.    std::vector<double> bottlesOfBeerOnTheWall;
 8.    bottlesOfBeerOnTheWall.push_back(100);
 9.    bottlesOfBeerOnTheWall.push_back(99);
10.    // ...
11.    bottlesOfBeerOnTheWall.push_back(1);
12.    array[2] = new Funct3(bottlesOfBeerOnTheWall);
13.
14.    array[3] = new Funct2("my string", 42);
15.
16.    // ...
```

So when the user invokes the `doit()` on one of these functionoids, he supplies the "remaining" args, and the call conceptually combines the original args passed to the ctor with those passed into the `doit()` method:

```
 1.    array[i]->doit(12);
```

As I've already hinted, one of the benefits of functionoids is that you can have several instances of, say, `Funct1` in your array, and those instances can have different parameters freeze-dried into them. For example, `array[0]` and `array[1]` are both of type `Funct1`, but the behavior of `array[0]->doit(12)` will be

different from the behavior of `array[1]->doit(12)` since the behavior will depend on both the 12 that was passed to `doit()` *and* the args passed to the ctors.

Another benefit of functionoids is apparent if we change the example from an array of functionoids to a local functionoid. To set the stage, let's go back to the old-fashioned function-pointer approach, and imagine that you're trying to pass a comparison-function to a `sort()` or `binarySearch()` routine. The `sort()` or `binarySearch()` routine is called `childRoutine()` and the comparison function-pointer type is called `FunctPtr` :

```
1.   void childRoutine(FunctPtr f)
2.   {
3.     // ...
4.     f( /*...args...*/ );
5.     // ...
6.   }
```

Then different callers would pass different function-pointers depending on what they thought was best:

```
1.   void myCaller()
2.   {
3.     // ...
4.     childRoutine(funct1);
5.     // ...
```

```
6.    }
7.
8.    void yourCaller()
9.    {
10.     // ...
11.     childRoutine(funct3);
12.     // ...
13.   }
```

We can easily translate this example into one using functionoids:

```
1.    void childRoutine(Funct& f)
2.    {
3.      // ...
4.      f.doit( /*...args...*/ );
5.      // ...
6.    }
7.
8.    void myCaller()
```

```
 9.   {
10.     // ...
11.     Funct1 funct( /*...ctor-args...*/ );
12.     childRoutine(funct);
13.     // ...
14.   }
15.
16.   void yourCaller()
17.   {
18.     // ...
19.     Funct3 funct( /*...ctor-args...*/ );
20.     childRoutine(funct);
21.     // ...
22.   }
```

Given this example as a backdrop, we can see two benefits of functionoids over function-pointers. The “ctor args” benefit described above, plus the fact that functionoids can maintain state between calls *in a thread-safe manner*. With plain function-pointers, people normally maintain state between calls via static data. However static data is not intrinsically thread-safe — static data is shared between all threads. The functionoid approach provides you with something that is *intrinsically* thread-safe since the code ends up with thread-local data. The implementation is trivial: change the old-fashioned static

datum to an instance data member inside the functionoid's `this` object, and poof, the data is not only thread-local, but it is even safe with recursive calls: each call to `yourCaller()` will have its own distinct `Funct3` object with its own distinct instance data.

Note that we've gained something without losing anything. If you *want* thread-global data, functionoids can give you that too: just change it from an instance data member inside the functionoid's `this` object to a static data member within the functionoid's class, or even to a local-scope static data. You'd be no better off than with function-pointers, but you wouldn't be worse off either.

The functionoid approach gives you a third option which is not available with the old-fashioned approach: the functionoid lets callers decide whether *they* want thread-local or thread-global data. They'd be responsible to use locks in cases where they wanted thread-global data, but at least they'd have the choice. It's easy:

```
1.   void callerWithThreadLocalData()
2.   {
3.     // ...
4.     Funct1 funct( /*...declare ctor args here...*/ );
5.     childRoutine(funct);
6.     // ...
7.   }
8.
9.   void callerWithThreadGlobalData()
10.  {
11.    // ...
```

```
12.     static Funct1 funct( /*...declare ctor args
    here...*/ );  // The static is the only difference

13.     childRoutine(funct);

14.     // ...

15.   }
```

Functionoids don't solve every problem encountered when making flexible software, but they are strictly more powerful than function-pointers and they are worth at least evaluating. In fact you can easily prove that functionoids don't lose any power over function-pointers, since you can imagine that the old-fashioned approach of function-pointers is equivalent to having a global(!) functionoid object. Since you can always make a global functionoid object, you haven't lost any ground. QED.

Can you make functionoids faster than normal function calls? ¶ Δ

Yes.

If you have a small functionoid, and in the real world that's rather common, the cost of the function-call can be high compared to the cost of the work done by the functionoid. In the previous FAQ, functionoids were implemented using virtual functions and will typically cost you a function-call. An alternate approach uses templates.

The following example is similar in spirit to the one in the previous FAQ. I have renamed `doit()` to `operator()()` to improve the caller code's readability and to allow someone to pass a regular function-pointer:

```
1.   class Funct1 {

2.   public:

3.     Funct1(float y) : y_(y) { }

4.     int operator()(int x) { /*...code from funct1...*/
```

```
    }
 5.  private:
 6.    float y_;
 7.  };
 8.
 9.  class Funct2 {
10.  public:
11.    Funct2(const std::string& y, int z) : y_(y), z_(z)
    { }
12.    int operator()(int x) { /*...code from funct2...*/
    }
13.  private:
14.    std::string y_;
15.    int z_;
16.  };
17.
18.  class Funct3 {
19.  public:
20.    Funct3(const std::vector<double>& y) : y_(y) { }
```

```
21.     int operator()(int x) { /*...code from funct3...*/
    }

22.   private:

23.     std::vector<double> y_;

24.   };
```

The difference between this approach and the one in the previous FAQ is that the fuctionoid gets “bound” to the caller at compile-time rather than at run-time. Think of it as passing in a parameter: if you know at compile-time the kind of functionoid you ultimately want to pass in, then you can use the above technique, and you can, at least in typical cases, get a speed benefit from having the compiler inline-expand the functionoid code within the caller. Here is an example:

```
1.   template <typename FunctObj>

2.   void myCode(FunctObj f)

3.   {

4.     // ...

5.     f( /*...args-go-here...*/ );

6.     // ...

7.   }
```

When the compiler compiles the above, it might inline-expand the call which might improve performance.

Here is one way to call the above:

```
1.   void blah()
2.   {
3.     // ...
4.     Funct2 x("functionoids are powerful", 42);
5.     myCode(x);
6.     // ...
7.   }
```

Aside: as was hinted at in the first paragraph above, you may also pass in the names of normal functions (though you might incur the cost of the function call when the caller uses these):

```
1.   void myNormalFunction(int x);
2.
3.   void blah()
4.   {
5.     // ...
6.     myCode(myNormalFunction);
7.     // ...
8.   }
```

What's the difference between a functionoid and a functor? ¶ Δ

A functionoid is an object that has one major method. It's basically the OO extension of a C-like function such as printf(). One would use a functionoid whenever the function has more than one entry point (i.e., more than one "method") and/or needs to maintain state between calls in a thread-safe manner (the C-style approach to maintaining state between calls is to add a local "static" variable to the function, but that is horribly unsafe in a multi-threaded environment).

A functor is a special case of a functionoid: it is a functionoid whose method is the "function-call operator," operator()(). Since it overloads the function-call operator, code can call its major method using the same syntax they would for a function call. E.g., if "foo" is a functor, to call the "operator()()" method on the "foo" object one would say "foo()". The benefit of this is in templates, since then the template can have a template parameter that will be used as a function, and this parameter can be either the name of a function or a functor-object. There is a performance advantage of it being a functor object since the "operator()()" method can be inlined (whereas if you pass the address of a function it must, necessarily, be non-inlined).

This is very useful for things like the "comparison" function on sorted containers. In C, the comparison function is always passed by pointer (e.g., see the signature to "qsort()"), but in C++ the parameter can come in either as a pointer to function OR as the name of a functor-object, and the result is that sorted containers in C++ can be, in some cases, a lot faster (and never slower) than the equivalent in C.

Since Java has nothing similar to templates, it must use dynamic binding for all this stuff, and dynamic binding of necessity means a function call. Normally not a big deal, but in C++ we want to enable extremely high performance code. That is, C++ has a "pay for it only if you use it" philosophy, which means the language must never arbitrarily impose any overhead over what the physical machine is capable of performing (of course a programmer may, optionally, use techniques such as dynamic binding that will, in general, impose some overhead in exchange for flexibility or some other "ility", but it's up to the designer and programmer to decide whether they want the benefits (and costs) of such constructs).