From C++ Common Knowledge

Item 15. Pointers to Class Members Are Not Pointers

It's unfortunate that pointers to class members have the term "pointer" in their descriptions, because they don't contain addresses and don't behave like pointers.

The syntax for declaring a pointer to member is really not too horrible (if you're already resigned to the declarator syntax for regular pointers):

```cpp
int *ip;  // pointer to an int
int C::*pmC;  // pointer to an int member of C
```

All you have to do is use `classname::*` rather than a plain `*` to indicate you're referring to a member of `classname`. Otherwise, the syntax is the same as for a regular pointer declarator.

```cpp
void * * *const * weird1;
void *A::*B::*const * weird2;
```

The name `weird1` has the type `pointer to const pointer to pointer to pointer to void`. The name `weird2` has the type `pointer to const pointer to a member of B to a pointer to a member of A`, which is a pointer to `void`. (This is just an example, and you wouldn't normally expect to see a declaration this complex or this silly.)

A regular pointer contains an address. If you dereference a pointer, you get the object at that address:

```cpp
int a = 12;
ip = &a;
*ip = 0;
a = *ip;
```

A pointer to member, unlike a regular pointer, does not refer to a specific memory location. Rather, it refers to a particular member of a class but not to a particular member of a particular object. Mechanically, it's usually clearest to consider a pointer to data member to be an offset. This is not necessarily the case, because the C++ standard says nothing about how a pointer to data member should be implemented; it says only what its syntax and behavior must be. However, most compilers implement pointers to data members as integers that contain the offset of the member referred to, plus one. (The offset is incremented so that the value 0 can represent a null pointer to data member.) The offset tells you how many bytes from the start of an object a particular member is located.

```cpp
class C {
    public:
```
When we set the value of `pimC` to `&C::a_`, we're effectively setting `pimC` with the offset of `a_` within `C`. Let's be clear: Unless `a_` is a static member, using `&` in the expression `&C::a_` does not give us an address; it gives us an offset. Note that this offset applies to any object of type `C`; that is, if the member `a_` can be found 12 bytes from the start of one `C` object, it will be found 12 bytes from the start of any other `C` object.

Given an offset of a member within a class, we need the address of an object of that class in order to get to the data member at that offset. That's where the unusual-looking `.*` and `->*` operators enter. When we write `pC->*pimC`, we are requesting that the address in `pC` be augmented by the offset in `pimC` in order to access the appropriate data member in the `C` object referred to by `pC`. When we write `aC.*pimC`, we are requesting that the address of `aC` be augmented by the offset in `pimC` in order to access the appropriate data member in the `C` object referred to by `pC`.

Pointers to data members are not as commonly used as pointers to member functions, but they are handy for illustrating the concept of contravariance. There is a predefined conversion from a pointer to a derived class to a pointer to any of its public base classes. We often say that there is an is-a relationship from the derived class to its public base classes, and this relationship often arises naturally from an analysis of the problem domain (see Polymorphism [2, 3]). Therefore, we can state (for example) that a `Circle` is-a `Shape` through public inheritance, and C++ backs us up by providing an implicit conversion from `Circle *` to `Shape *`. No implicit conversion exists from a `Shape *` to a `Circle *` because such a conversion would not make sense; many different types of `Shape` may exist, and not all of them are `Circle`s. (It also just sounds silly to say, "A `Shape` is a `Circle`.")

In the case of pointers to class members, the opposite situation holds: There is an implicit conversion from a pointer to a member of a base class to a pointer to a member of a publicly derived class, but there is no conversion from a pointer to a member of a derived class to a pointer to a member of any of its bases. This concept of contravariance seems counterintuitive until we remember that a pointer to data member is not a pointer to an object; it's an offset into an object.
class Shape {
    //...
    Point center_;  //...
};
class Circle : public Shape {
    //...
    double radius_;  //...
};

A Circle is-a Shape, so a Circle object contains a Shape subobject. Therefore, any offset within Shape is also a valid offset within Circle.

Point Circle::*loc = &Shape::center_; // OK, base to derived

However, a Shape is not (necessarily) a Circle, so the offset of a member of Circle is not (necessarily) a valid offset within a Shape.

double Shape::*extent =
    &Circle::radius_; // error! derived to base

It makes sense to say that a Circle contains all the data members of its Shape base class (that is, it inherits those members from Shape), and C++ backs us up with an implicit conversion from a pointer to member of a Shape to a pointer to member of a Circle. It doesn't make sense to say that a Shape contains all the data members of a Circle (Shape doesn't inherit anything from Circle), and C++ reminds us of that by disallowing the conversion from pointer to member of Circle to pointer to member of Shape.
Item 16. Pointers to Member Functions Are Not Pointers

When you take the address of a non-static member function, you don't get an address; you get a pointer to member function.

class Shape {
    public:
        //...
        void moveTo( Point newLocation );
        bool validate() const;
        virtual bool draw() const = 0;
        //...
};
class Circle : public Shape {
    //...
    bool draw() const;
    //...
};
//...
void (Shape::*mf1)( Point ) = &Shape::moveTo; // not a pointer

The declaration syntax of a pointer to member function is really no more difficult than that of a pointer to a regular function (which, admittedly, is bad enough as it is; see Dealing with Function and Array Declarators [17, 61]). As with pointers to data members, all that's necessary is to use classname::* rather than * to indicate that the function referred to is a member of classname. Unlike a regular pointer to function, though, a pointer to member function can refer to a const member function:

bool (Shape::*mf2)() const = &Shape::validate;

As with a pointer to data member, we need an object or pointer to an object in order to dereference a pointer to member function. In the case of a pointer to data member, we need to add the object's address to the member's offset (contained in the pointer to data member) in order to access the member. In the case of a pointer to member function, we need the object's address to use as (or to calculate; see Meaning of Pointer Comparison [28, 97]) the value of the this pointer for the function call and possibly for other reasons as well.

Circle circ;
Shape *pShape = &circ;
(pShape->*mf2)(); // call Shape::validate
(circ.*mf2)(); // call Shape::validate

The ->* and .* operators must be parenthesized because they have lower precedence than the () operator, and we have to first find out what function to call before we call it! This is entirely analogous to the use of parentheses in an expression such as (a+b)*c, where we want to ensure that the lower-precedence addition is carried out before the higher-precedence multiplication.
Note that there is no such thing as a "virtual" pointer to member function. Virtualness is a property of the member function itself, not the pointer that refers to it.

mf2 = &Shape::draw; // draw is virtual
(pShape->*mf2)(); // call Circle::draw

This is one reason why a pointer to member function cannot be implemented, in general, as a simple pointer to function. The implementation of the pointer to member function must store within itself information as to whether the member function to which it refers is virtual or nonvirtual, information about where to find the appropriate virtual function table pointer (see The Compiler Puts Stuff in Classes [11, 37]), an offset to be added to or subtracted from the function's this pointer (see Meaning of Pointer Comparison [28, 97]), and possibly other information. A pointer to member function is commonly implemented as a small structure that contains this information, although many other implementations are also in use. Dereferencing and calling a pointer to member function usually involves examining the stored information and conditionally executing the appropriate virtual or nonvirtual function calling sequence.

As with pointers to data members, pointers to member functions exhibit contravariance: there is a predefined conversion from a pointer to a member function of a base class to a pointer to a member function of a derived class, not the reverse. This makes sense if you consider that a base class member function will attempt to access only base class members through its this pointer whereas a derived class function may attempt to access members that are not present in the base class.

class B {
    public:
    void bset( int val ) { bval_ = val; }
    private
    int bval_
};
class D : public B {
    public:
    void dset( int val ) { dval_ = val; }
    private:
    int dval_
};
B b;
D d;
void (B::*f1)(int) = &D::dset; // error! derived func in base ptr
(b.*f1)(12); // oops! access nonexistent dval member!
void (D::*f2)(int) = &B::bset; // OK, base func in derived ptr
(d.*f2)(11); // OK, set inherited bval data member