Pointer to class data member "::*"

Asked 13 years, 1 month ago Modified 6 months ago Viewed 189k times

291

I came across this strange code snippet which compiles fine:

```
class Car
{
    public:
    int speed;
};

int main()
{
    int Car::*pSpeed = &Car::speed;
    return 0;
}
```

Why does C++ have this pointer to a non-static data member of a class? **What** is the use of this strange pointer in real code?

c++ class pointers c++-faq

Share Edit Follow Flag edited May 23, 2019 at 9:44 L. F. 17.9k ●7 ●41 ●74 asked Mar 22, 2009 at 9:03 Ashwin Nanjappa 72.3k ●76 ●202 ●288

Here's where I found it, confused me too...but makes sense now: stackoverflow.com/a/982941/211160 – HostileFork says dont trust SE Jul 12, 2012 at 6:03

2 Pointer to members are C++'s typesafe alternative to the rather unsafe `offsetof()` construct from C. They both return the information, where inside a `class` or `struct` a certain field is located. – Kai Petzke Feb 18, 2021 at 6:44

18 Answers

Sorted by: Highest score (default)

226

It's a "pointer to member" - the following code illustrates its use:

```
#include <iostream>
using namespace std;

class Car
{
    public:
    int speed;
};

int main()
{
    int Car::*pSpeed = &Car::speed;

    Car c1;
    c1.speed = 1;       // direct access
    cout << "speed is " << c1.speed << endl;
    c1.*pSpeed = 2;     // access via pointer to member
    cout << "speed is " << c1.speed << endl;
    return 0;
}
```

As to why you would want to do that, well it gives you another level of indirection that can solve some tricky problems. But to be honest, I've never had to use them in my own code.

Edit: I can't think off-hand of a convincing use for pointers to member data. Pointer to member functions can be used in pluggable architectures, but once again producing an example in a small space defeats me. The following is my best (untested) try - an Apply function that would do some pre &post processing before applying a user-selected member function to an object:

```
void Apply( SomeClass * c, void (SomeClass::*func)() ) {
    // do hefty pre-call processing
    (c->*func)();  // call user specified function
    // do hefty post-call processing
}
```

The parentheses around `c->*func` are necessary because the `->*` operator has lower precedence than the function call operator.

Share Edit Follow Flag edited Jul 31, 2015 at 20:57 Oktalist 13.7k ●3 ●43 ●60 answered Mar 22, 2009 at 9:13 anon

3 Could you show an example of a tricky situation where this is useful? Thanks. – Ashwin Nanjappa Mar 22, 2009 at 9:31

I have an example of using pointer-to-member in a Traits class in another SO answer. – Mike DeSimone Apr 13, 2011 at 18:08

An example is writing a "callback"-type class for some event-based system. CEGUI's UI event subscription system, for example, takes a templated callback that stores a pointer to a member function of your choosing, so that you can specify a method to handle the event. – Benji XVI Dec 26, 2012 at 21:55

3 There is a pretty cool example of pointer-to-**data**-member usage in a template function in this code – alveko Jul 8, 2013 at 20:43

4 I have recently used pointers to data members in serialization framework. Static marshaller object was initialized with list of wrappers containing pointer to serializable data members. An early prototype of this code. – Alexey Biryukov Apr 8, 2015 at 22:25

105

This is the simplest example I can think of that conveys the rare cases where this feature is pertinent:

```
#include <iostream>

class bowl {
public:
    int apples;
    int oranges;
};

int count_fruit(bowl * begin, bowl * end, int bowl::*fruit)
{
    int count = 0;
    for (bowl * iterator = begin; iterator != end; ++ iterator)
        count += iterator->*fruit;
    return count;
}

int main()
{
    bowl bowls[2] = {
        { 1, 2 },
        { 3, 5 }
    };
    std::cout << "I have " << count_fruit(bowls, bowls + 2, & bowl::apples) << " apples\n";
    std::cout << "I have " << count_fruit(bowls, bowls + 2, & bowl::oranges) << " oranges\n";
    return 0;
}
```

The thing to note here is the pointer passed in to count_fruit. This saves you having to write separate count_apples and count_oranges functions.

Share Edit Follow Flag answered Apr 29, 2011 at 16:05 John McFarlane 4,722 ●4 ●31 ●35

3 Shouldn't it be `&bowls.apples` and `&bowls.oranges`? `&bowl::apples` and `&bowl::oranges` doesn't point to anything. – Dan Nissenbaum Mar 30, 2014 at 9:20

24 `&bowl::apples` and `&bowl::oranges` do not point to members of an *object*; they point to members of a *class*. They need to be combined with a pointer to an actual object before they point to something. That combination is achieved with the `->*` operator. – John McFarlane Mar 30, 2014 at 20:56

Thank you very much for this very illustrative example! Nevertheless, I think I still don't fully understand the expression: `int bowl::*fruit`. What is the type and what is the parameter name of this expression? – fabian Apr 12, 2021 at 17:37

1 @fabian YW! The parameter name is `fruit`. It's type says, "I point to an `int` that's a member of the `bowl` class." Under the hood, it's typically implemented as an offset from the start of the class, i.e. 0 bytes for `apples` or 4 bytes for `oranges`. Here's a simpler example with a function that just increments a member. `fruit` specifies where in `b` that member variable is as a byte offset. And here's the calling code which either passes `0` or `4` in as that offset. – John McFarlane Apr 12, 2021 at 20:53

67

Another application are intrusive lists. The element type can tell the list what its next/prev pointers are. So the list does not use hard-coded names but can still use existing pointers:

```
// say this is some existing structure. And we want to use
// a list. We can tell it that the next pointer
// is apple::next.
struct apple {
    int data;
    apple * next;
};

// simple example of a minimal intrusive list. Could specify the
// member pointer as template argument too, if we wanted:
// template<typename E, E *E::*next_ptr>
template<typename E>
struct List {
    List(E *E::*next_ptr):head(0), next_ptr(next_ptr) { }

    void add(E &e) {
        // access its next pointer by the member pointer
        e.*next_ptr = head;
        head = &e;
    }

    E * head;
    E *E::*next_ptr;
};

int main() {
    List<apple> lst(&apple::next);

    apple a;
    lst.add(a);
}
```

Share Edit Follow Flag edited Mar 23, 2009 at 0:25 answered Mar 23, 2009 at 0:19 Johannes Schaub - litb 481k ●123 ●868 ●1188

If this is truly a linked list wouldn't you want something like this: void add(E* e) { e->*next_ptr = head; head = e; } ?? – eeeeaaii Aug 25, 2011 at 16:56

5 @eee I recommend you to read about reference parameters. What I did is basically equivalent to what you did. – Johannes Schaub - litb Aug 25, 2011 at 18:55

+1 for your code example, but I didn't see any necessity for the use of pointer-to-member, any other example? – Alcott Aug 14, 2012 at 8:32

8 @Alcott: You can apply it to other linked-list-like structures where the next pointer is not named `next`. – icktoofay May 18, 2013 at 23:13

48

Here's a real-world example I am working on right now, from signal processing / control systems:

Suppose you have some structure that represents the data you are collecting:

```
struct Sample {
    time_t time;
    double value1;
    double value2;
    double value3;
};
```

Now suppose that you stuff them into a vector:

```
std::vector<Sample> samples;
... fill the vector ...
```

Now suppose that you want to calculate some function (say the mean) of one of the variables over a range of samples, and you want to factor this mean calculation into a function. The pointer-to-member makes it easy:

```
double Mean(std::vector<Sample>::const_iterator begin,
    std::vector<Sample>::const_iterator end,
    double Sample::* var)
{
    float mean = 0;
    int samples = 0;
    for(; begin != end; begin++) {
        const Sample& s = *begin;
        mean += s.*var;
        samples++;
    }
    mean /= samples;
    return mean;
}

...
double mean = Mean(samples.begin(), samples.end(), &Sample::value2);
```

Note Edited 2016/08/05 for a more concise template-function approach

And, of course, you can template it to compute a mean for any forward-iterator and any value type that supports addition with itself and division by size_t:

```
template<typename Titer, typename S>
S mean(Titer begin, const Titer& end, S std::iterator_traits<Titer>::value_type::* var) {
    using T = typename std::iterator_traits<Titer>::value_type;
    S sum = 0;
    size_t samples = 0;
    for( ; begin != end ; ++begin ) {
        const T& s = *begin;
        sum += s.*var;
        samples++;
    }
    return sum / samples;
}

struct Sample {
    double x;
}

std::vector<Sample> samples { {1.0}, {2.0}, {3.0} };
double m = mean(samples.begin(), samples.end(), &Sample::x);
```

EDIT - The above code has performance implications

You should note, as I soon discovered, that the code above has some serious performance implications. The summary is that if you're calculating a summary statistic on a time series, or calculating an FFT etc, then you should store the values for each variable contiguously in memory. Otherwise, iterating over the series will cause a cache miss for every value retrieved.

Consider the performance of this code:

```
struct Sample {
  float w, x, y, z;
};

std::vector<Sample> series = ...;

float sum = 0;
int samples = 0;
for(auto it = series.begin(); it != series.end(); it++) {
  sum += *it.x;
  samples++;
}
float mean = sum / samples;
```

On many architectures, one instance of `Sample` will fill a cache line. So on each iteration of the loop, one sample will be pulled from memory into the cache. 4 bytes from the cache line will be used and the rest thrown away, and the next iteration will result in another cache miss, memory access and so on.

Much better to do this:

```
struct Samples {
  std::vector<float> w, x, y, z;
};

Samples series = ...;

float sum = 0;
float samples = 0;
for(auto it = series.x.begin(); it != series.x.end(); it++) {
  sum += *it;
  samples++;
}
float mean = sum / samples;
```

Now when the first x value is loaded from memory, the next three will also be loaded into the cache (supposing suitable alignment), meaning you don't need any values loaded for the next three iterations.

The above algorithm can be improved somewhat further through the use of SIMD instructions on eg SSE2 architectures. However, these work **much** better if the values are all contiguous in memory and you can use a single instruction to load four samples together (more in later SSE versions).

YMMV - design your data structures to suit your algorithm.

Share Edit Follow Flag edited Aug 5, 2016 at 13:49 answered Nov 2, 2010 at 16:11 Tom 6,507 ●1 ●34 ●57

This is excellent. I'm about to implement something very similar, and now I don't have to figure out the strange syntax! Thanks! – Nicu Stiurca Mar 26, 2013 at 2:47

1 This is the best answer. The `double Sample::*` part is key! – Eyal Nov 6, 2019 at 16:23

This is called generally AoS vs SoA: en.wikipedia.org/wiki/AoS_and_SoA – hftd Sep 22, 2021 at 16:48

42

You can later access this member, on **any** instance:

```
int main()
{
  int Car::*pSpeed = &Car::speed;
  Car myCar;
  Car yourCar;

  int mySpeed = myCar.*pSpeed;
  int yourSpeed = yourCar.*pSpeed;

  assert(mySpeed > yourSpeed); // ;-)

  return 0;
}
```

Note that you do need an instance to call it on, so it does not work like a delegate.
It is used rarely, I've needed it maybe once or twice in all my years.

Normally using an interface (i.e. a pure base class in C++) is the better design choice.

Share Edit Follow Flag edited May 22, 2013 at 16:58 answered Mar 22, 2009 at 9:10 peterchen 39.7k ●19 ●101 ●181

But surely this is just bad practice? should do something like youcar.setspeed(mycar.getpspeed) – thecoshman Oct 6, 2010 at 21:08

13 @thecoshman: entirely depends - hiding data members behind set/get methods is not encapsulation and merely a milkmaids attempt at interface abstraction. In many scenarios, "denormalization" to public members is a reasonable choice. But that discussion probably exceeds the confines of the comment functionality. – peterchen Oct 12, 2010 at 15:21

5 +1 for pointing out, if I understand correctly, that this is a pointer to a member of any instance, and not a pointer to a specific value of one instance, which is the part I was completely missing. – johnbakers May 21, 2013 at 9:49

@Fellowshee You do understand correctly :) (emphasized that in the answer). – peterchen May 22, 2013 at 16:58

27

IBM has some more documentation on how to use this. Briefly, you're using the pointer as an offset into the class. You can't use these pointers apart from the class they refer to, so:

```
  int Car::*pSpeed = &Car::speed;
  Car mycar;
  mycar.*pSpeed = 65;
```

It seems a little obscure, but one possible application is if you're trying to write code for deserializing generic data into many different object types, and your code needs to handle object types that it knows absolutely nothing about (for example, your code is in a library, and the objects into which you deserialize were created by a user of your library). The member pointers give you a generic, semi-legible way of referring to the individual data member offsets, without having to resort to typeless void * tricks the way you might for C structs.

Share Edit Follow Flag edited Feb 18, 2013 at 4:54 answered Mar 22, 2009 at 9:13 AHelps 1,762 ●11 ●16

Could you share a code snippet example where this construct is useful? Thanks. – Ashwin Nanjappa Mar 22, 2009 at 9:32

2 I'm currently doing alot of this due to doing some DCOM work and using managed resource classes which involves doing a bit of work before each call, and using data members for internal representation to send off to com, plus templating, makes a lot of boiler plate code *much* smaller – Dan Aug 10, 2009 at 21:30

21

It makes it possible to bind member variables and functions in the uniform manner. The following is example with your Car class. More common usage would be binding `std::pair::first` and `::second` when using in STL algorithms and Boost on a map.

```
#include <list>
#include <algorithm>
#include <iostream>
#include <iterator>
#include <boost/lambda/lambda.hpp>
#include <boost/lambda/bind.hpp>


class Car {
public:
    Car(int s): speed(s) {}
    void drive() {
        std::cout << "Driving at " << speed << " km/h" << std::endl;
    }
    int speed;
};

int main() {

    using namespace std;
    using namespace boost::lambda;

    list<Car> l;
    l.push_back(Car(10));
    l.push_back(Car(140));
    l.push_back(Car(130));
    l.push_back(Car(60));

    // Speeding cars
    list<Car> s;

    // Binding a value to a member variable.
    // Find all cars with speed over 60 km/h.
    remove_copy_if(l.begin(), l.end(),
                   back_inserter(s),
                   bind(&Car::speed, _1) <= 60);

    // Binding a value to a member function.
    // Call a function on each car.
    for_each(s.begin(), s.end(), bind(&Car::drive, _1));

    return 0;
}
```

Share Edit Follow Flag edited Mar 22, 2009 at 13:08 answered Mar 22, 2009 at 13:02 Alex B 79k ●40 ●196 ●275

11

You can use an array of pointer to (homogeneous) member data to enable a dual, named-member (i.e. x.data) and array-subscript (i.e. x[idx]) interface.

```
#include <cassert>
#include <cstddef>

struct vector3 {
    float x;
    float y;
    float z;

    float& operator[](std::size_t idx) {
        static float vector3::*component[3] = {
            &vector3::x, &vector3::y, &vector3::z
        };
        return this->*component[idx];
    }
};

int main()
{
    vector3 v = { 0.0f, 1.0f, 2.0f };

    assert(&v[0] == &v.x);
    assert(&v[1] == &v.y);
    assert(&v[2] == &v.z);

    for (std::size_t i = 0; i < 3; ++i) {
        v[i] += 1.0f;
    }

    assert(v.x == 1.0f);
    assert(v.y == 2.0f);
    assert(v.z == 3.0f);

    return 0;
}
```

Share Edit Follow Flag edited Mar 23, 2009 at 17:33 answered Mar 23, 2009 at 4:24 Functastic 606 ●3 ●12

I've more often seen this implemented using an anonymous union including an array field v[3] since that avoids an indirection, but clever nonetheless, and potentially useful for non-contiguous fields. – Dwayne Robinson Apr 21, 2015 at 4:28

3 @DwayneRobinson but using a `union` to type-pun in that fashion is not allowed by the standard as it invokes numerous forms of undefined behaviour... whereas this answer is ok. – underscore_d Sep 7, 2016 at 22:25

That's a neat example but operator[] can be rewritten without pointer-to-component: `float *component[] = { &x, &y, &z }; return *component[idx];` I.e., the pointer-to-component seems to serve no purpose except obfuscation. – tobi_s May 21, 2020 at 3:20

2

One way I've used it is if I have two implementations of how to do something in a class and I want to choose one at run-time without having to continually go through an if statement i.e.

```
class Algorithm
{
public:
    Algorithm() : m_impFn( &Algorithm::implementationA ) {}
    void frequentlyCalled()
    {
        // Avoid if ( using A ) else if ( using B ) type of thing
        (this->*m_impFn)();
    }
private:
    void implementationA() { /*...*/ }
    void implementationB() { /*...*/ }

    typedef void ( Algorithm::*IMP_FN ) ();
    IMP_FN m_impFn;
};
```

Obviously this is only practically useful if you feel the code is being hammered enough that the if statement is slowing things done eg. deep in the guts of some intensive algorithm somewhere. I still think it's more elegant than the if statement even in situations where it has no practical use but that's just my opnion.

Share Edit Follow Flag edited Mar 23, 2009 at 18:40 answered Mar 22, 2009 at 11:57 Troubadour 13.1k ●2 ●36 ●56

Basically, you can achieve the same with the abstract `Algorithm` and two derived classes, e.g., `AlgorithmA` and `AlgorithmB`. In such a case both algorithms are well separated and are ensured to be tested independently. – shycha Apr 10, 2019 at 13:31

2

Pointers to classes are not *real* pointers; a class is a logical construct and has no physical existence in memory, however, when you construct a pointer to a member of a class it gives an offset into an object of the member's class where the member can be found; This gives an important conclusion: *Since static members are not associated with any object so a pointer to a member CANNOT point to a static member(data or functions) whatsoever* Consider the following:

```
class x {
public:
    int val;
    x(int i) { val = i;}

    int get_val() { return val; }
    int d_val(int i) {return i+i; }
};

int main() {
    int (x::* data) = &x::val;               //pointer to data member
    int (x::* func)(int) = &x::d_val;        //pointer to function member

    x ob1(1), ob2(2);

    cout <<ob1.*data;
    cout <<ob2.*data;

    cout <<(ob1.*func)(ob1.*data);
    cout <<(ob2.*func)(ob2.*data);


    return 0;
}
```

Source: The Complete Reference C++ - Herbert Schildt 4th Edition

Share Edit Follow Flag edited Apr 10, 2019 at 15:11 shycha 432 ●4 ●13 answered Jun 27, 2018 at 18:06 Arijit Dey 21 ●3

1

with pointer to member, we can write generic code like this

```
template<typename T, typename U>
struct alpha{
   T U::*p_some_member;
};

struct beta{
   int foo;
};

int main()
{

   beta b{};

   alpha<int, beta> a{&beta::foo};

   b.*(a.p_some_member) = 4;

   return 0;
}
```

Share Edit Follow Flag answered Jul 22, 2021 at 10:25 Spyros Tsimboulis 31 ●3

1

I love the `*` and `&` operators:

```
struct X
{
    int a {0};
    int *ptr {NULL};

    int &fa() { return a; }
    int *&fptr() { return ptr; }
};

int main(void)
{
    X x;
    int X::*p1 = &X::a;     // pointer-to-member 'int X::a'. Type of p1 = 'int X::*'
    x.*p1 = 10;

    int *X::*p2 = &X::ptr;  // pointer-to-member-pointer 'int *X::ptr'. Type of p2 =
'int *X::*'
    x.*p2 = nullptr;
    X *xx;
    xx->*p2 = nullptr;

    int& (X::*p3)() = X::fa; // pointer-to-member-function 'X::fa'. Type of p3 = 'int &
(X::*)()'
    (x.*p3)() = 20;
    (xx->*p3)() = 30;

    int *&(X::*p4)() = X::fptr;  // pointer-to-member-function 'X::fptr'. Type of p4 =
'int *&(X::*)()'
    (x.*p4)() = nullptr;
    (xx->*p4)() = nullptr;
}
```

Indeed all is true as long as the members are public, or static

Share Edit Follow Flag answered Nov 1, 2021 at 14:38 mada 362 ●9

0

Pointer to members are C++'s type safe equivalent for C's `offsetof()`, which is defined in `stddef.h`: Both return the information, where a certain field is located within a `class` or `struct`. While `offsetof()` may be used with certain simple enough classes also in C++, it fails miserably for the general case, especially with virtual base classes. So pointer to members were added to the standard. They also provide easier syntax to reference an actual field:

```
struct C { int a; int b; } c;
int C::* intptr = &C::a;       // or &C::b, depending on the field wanted
c.*intptr += 1;
```

is much easier than:

```
struct C { int a; int b; } c;
int intoffset = offsetof(struct C, a);
* (int *) (((char *) (void *) &c) + intoffset) += 1;
```

As to why one wants to use `offsetof()` (or pointer to members), there are good answers elsewhere on stackoverflow. One example is here: How does the C offsetof macro work?

Share Edit Follow Flag answered Feb 20, 2021 at 15:45 Kai Petzke 1,427 ●15 ●24

0

Here is an example where pointer to data members could be useful:

```
#include <iostream>
#include <list>
#include <string>

template <typename Container, typename T, typename DataPtr>
typename Container::value_type searchByDataMember (const Container& container, const T&
t, DataPtr ptr) {
    for (const typename Container::value_type& x : container) {
        if (x->*ptr == t)
            return x;
    }
    return typename Container::value_type{};
}

struct Object {
    int ID, value;
    std::string name;
    Object (int i, int v, const std::string& n) : ID(i), value(v), name(n) {}
};

std::list<Object*> objects { new Object(5,6,"Sam"), new Object(11,7,"Mark"), new
Object(9,12,"Rob"),
    new Object(2,11,"Tom"), new Object(15,16,"John") };

int main() {
    const Object* object = searchByDataMember (objects, 11, &Object::value);
    std::cout << object->name << '\n';  // Tom
}
```

Share Edit Follow Flag answered Nov 18, 2015 at 18:48 prestokeys 4,719 ●3 ●16 ●39

0

A realworld example of a pointer-to-member could be a more narrow aliasing constructor for std::shared_ptr:

```
template <typename T>
template <typename U>
shared_ptr<T>::shared_ptr(const shared_ptr<U>, T U::*member);
```

What that constructor would be good for

assume you have a struct foo:

```
struct foo {
    int ival;
    float fval;
};
```

If you have given a shared_ptr to a foo, you could then retrieve shared_ptr's to its members ival or fval using that constructor:

```
auto foo_shared = std::make_shared<foo>();
auto ival_shared = std::shared_ptr<int>(foo_shared, &foo::ival);
```

This would be useful if want to pass the pointer foo_shared->ival to some function which expects a shared_ptr

https://en.cppreference.com/w/cpp/memory/shared_ptr/shared_ptr

Share Edit Follow Flag edited Sep 20, 2020 at 17:09 answered Sep 20, 2020 at 15:56 benb 156 ●2 ●4

Yes, but `auto ival_shared = std::shared_ptr<int>(foo_shared, &foo_shared->ival);` would do the same job, using the existing standard library, and without ever using pointers-to-members. So this answer leaves OP still asking "but *why* would I want to do that?" – Quuxplusone Jan 13, 2021 at 20:39

0

Suppose you have a structure. Inside of that structure are * some sort of name * two variables of the same type but with different meaning

```
struct foo {
    std::string a;
    std::string b;
};
```

Okay, now let's say you have a bunch of `foo`s in a container:

```
// key: some sort of name, value: a foo instance
std::map<std::string, foo> container;
```

Okay, now suppose you load the data from separate sources, but the data is presented in the same fashion (eg, you need the same parsing method).

You could do something like this:

```
void readDataFromText(std::istream & input, std::map<std::string, foo> & container,
std::string foo::*storage) {
    std::string line, name, value;

    // while lines are successfully retrieved
    while (std::getline(input, line)) {
        std::stringstream linestr(line);
        if ( line.empty() ) {
            continue;
        }

        // retrieve name and value
        linestr >> name >> value;

        // store value into correct storage, whichever one is correct
        container[name].*storage = value;
    }
}

std::map<std::string, foo> readValues() {
    std::map<std::string, foo> foos;

    std::ifstream a("input-a");
    readDataFromText(a, foos, &foo::a);
    std::ifstream b("input-b");
    readDataFromText(b, foos, &foo::b);
    return foos;
}
```

At this point, calling `readValues()` will return a container with a unison of "input-a" and "input-b"; all keys will be present, and foos with have either a or b or both.

Share Edit Follow Flag answered Apr 12, 2016 at 15:08 inetknght 4,002 ●1 ●24 ●52

-1

Just to add some use cases for @anon's & @Oktalist's answer, here's a great reading material about pointer-to-member-function and pointer-to-member-data.

https://www.dre.vanderbilt.edu/~schmidt/PDF/C++-ptmf4.pdf

Share Edit Follow Flag edited May 12, 2020 at 19:15 Kuba hasn't forgotten Monica 92.4k ●13 ●137 ●288 answered Dec 25, 2016 at 7:22 Dragonly 35 ●7

the link has been dead. That's why link-only answers are not expected here. At least summarize the content of the link, otherwise your answer becomes invalid when the link rots – phuclv Sep 3, 2019 at 5:35

-1

I think you'd only want to do this if the member data was pretty large (e.g., an object of another pretty hefty class), and you have some external routine which only works on references to objects of that class. You don't want to copy the member object, so this lets you pass it around.

Share Edit Follow Flag answered Mar 22, 2009 at 10:47 Andrew Jaffe 25.4k ●4 ●47 ●58