Why use prefixes like m_ on data members in C++ classes?

Asked 15 years, 4 months ago Modified 1 year, 6 months ago

Viewed 157k times

185

A lot of C++ code uses syntactical conventions for marking up data members. Common examples include

- `m_memberName` for public members (where public members are used at all)
- `_memberName` for private members or all members

Others try to enforce using `this->member` whenever a data member is used.

In my experience, most larger code bases fail at applying such rules consistently.

In other languages, these conventions are far less widespread. I see it only occasionally in Java or C# code. I think I have never seen it in Ruby or Python code. Thus, there seems to be a trend with more modern languages to not use special markup for data members.

Is this convention still useful today in C++ or is it just an anachronism, especially as it is used so inconsistently across libraries? Haven't the other languages shown that one can do without member prefixes?

c++ class coding-style naming-conventions

Share

Improve this question

Follow

edited Jun 15, 2023 at 22:45

HolyBlackCat

95.3k ● 13 ● 164 ● 264

asked Aug 4, 2009 at 15:25

VoidPointer

17.8k ● 15 ● 58 ● 58

23 I prefer it; in complex codebases it can be important to know which vars are local and which aren't. I generally use the prefix over forcing this-> which I find to be a lot of extra typing and optional (whereas naming will force you to do it) – Joe Aug 4, 2009 at 15:35

6 You've never seen it in Ruby because of @ for attribute, and the idiom of generating accessors in preference to using attributes directly. – Steve Jessop Aug 4, 2009 at 17:17

7 According to PEP 8 non-public member variables are to be prefixed with an underscore in Python (example: `self._something = 1`). – Nathan Osman Apr 2, 2013 at 1:52

2 Shouldn't editor's syntax highlighting be used to identify these? – Michał Turecki Jul 14, 2015 at 14:49

4 You do have seen the equivalent of `this->member` in Python code. In Python it would typically be `self.member` and it is not only a convention, it is required by the language. – matec May 14, 2016 at 8:23

31 Answers

Sorted by: Highest score (default)

1 2 Next

281

I'm all in favour of **prefixes done well**.

I think (System) Hungarian notation is responsible for most of the "bad rap" that prefixes get.

This notation is largely pointless in strongly typed languages e.g. in C++ "lpsz" to tell you that your string is a long pointer to a nul terminated string, when: segmented architecture is ancient history, C++ strings are by common convention pointers to nul-terminated char arrays, and it's not really all that difficult to know that "customerName" is a string!

However, I do use prefixes to specify the *usage* of a variable (essentially "Apps Hungarian", although I prefer to avoid the term Hungarian due to it having a bad and unfair association with System Hungarian), and this is a very handy **timesaving** and **bug-reducing** approach.

I use:

- m for members
- c for constants/readonlys
- p for pointer (and pp for pointer to pointer)
- v for volatile
- s for static

- i for indexes and iterators
- e for events

Where I wish to make the *type* clear, I use standard suffixes (e.g. List, ComboBox, etc).

This makes the programmer aware of the *usage* of the variable whenever they see/use it. Arguably the most important case is "p" for pointer (because the usage changes from var. to var-> and you have to be much more careful with pointers - NULLs, pointer arithmetic, etc), but all the others are very handy.

For example, you can use the same variable name in multiple ways in a single function: (here a C++ example, but it applies equally to many languages)

```
MyClass::MyClass(int numItems)
{
    mNumItems = numItems;
    for (int iItem = 0; iItem < mNumItems; iItem++)
    {
        Item *pItem = new Item();
        itemList[iItem] = pItem;
    }
}
```

You can see here:

- No confusion between member and parameter
- No confusion between index/iterator and items
- Use of a set of clearly related variables (item list, pointer, and index) that avoid the many pitfalls of

generic (vague) names like "count", "index".

- Prefixes reduce typing (shorter, and work better with auto-completion) than alternatives like "itemIndex" and "itemPtr"

Another great point of "iName" iterators is that I never index an array with the wrong index, and if I copy a loop inside another loop I don't have to refactor one of the loop index variables.

Compare this unrealistically simple example:

```
for (int i = 0; i < 100; i++)
    for (int j = 0; j < 5; j++)
        list[i].score += other[j].score;
```

(which is hard to read and often leads to use of "i" where "j" was intended)

with:

```
for (int iCompany = 0; iCompany < numCompanies; iCompa
    for (int iUser = 0; iUser < numUsers; iUser++)
       companyList[iCompany].score += userList[iUser].
```

(which is much more readable, and removes all confusion over indexing. With auto-complete in modern IDEs, this is also quick and easy to type)

The next benefit is that code snippets *don't require any context* to be understood. I can copy two lines of code into an email or a document, and anyone reading that snippet can tell the difference between all the members, constants, pointers, indexes, etc. I don't have to add "oh, and be careful because 'data' is a pointer to a pointer", because it's called 'ppData'.

And for the same reason, I don't have to move my eyes out of a line of code in order to understand it. I don't have to search through the code to find if 'data' is a local, parameter, member, or constant. I don't have to move my hand to the mouse so I can hover the pointer over 'data' and then wait for a tooltip (that sometimes never appears) to pop up. So programmers can read and understand the code *significantly* faster, because they don't waste time searching up and down or waiting.

> *(If you don't think you waste time searching up and down to work stuff out, find some code you wrote a year ago and haven't looked at since. Open the file and jump about half way down without reading it. See how far you can read from this point before you don't know if something is a member, parameter or local. Now jump to another random location... This is what we all do all day long when we are single stepping through someone else's code or trying to understand how to call their function)*

The 'm' prefix also avoids the (IMHO) ugly and wordy "this->" notation, and the inconsistency that it guarantees (even if you are careful you'll usually end up with a mixture of 'this->data' and 'data' in the same class, because nothing enforces a consistent spelling of the name).

'this' notation is intended to resolve *ambiguity* - but why would anyone deliberately write code that can be ambiguous? Ambiguity *will* lead to a bug sooner or later. And in some languages 'this' can't be used for static members, so you have to introduce 'special cases' in your coding style. I prefer to have a single simple coding rule that applies everywhere - explicit, unambiguous and consistent.

The last major benefit is with Intellisense and auto-completion. Try using Intellisense on a Windows Form to find an event - you have to scroll through hundreds of mysterious base class methods that you will never need to call to find the events. But if every event had an "e" prefix, they would automatically be listed in a group under "e". Thus, prefixing works to group the members, consts, events, etc in the intellisense list, making it much quicker and easier to find the names you want. (Usually, a method might have around 20-50 values (locals, params, members, consts, events) that are accessible in its scope. But after typing the prefix (I want to use an index now, so I type 'i...'), I am presented with only 2-5 auto-complete options. The 'extra typing' people attribute to prefixes and meaningful names drastically reduces the search space and measurably accelerates development speed)

I'm a lazy programmer, and the above convention saves me a lot of work. I can code faster and I make far fewer mistakes because I know how every variable should be used.

Arguments against

So, what are the cons? Typical arguments against prefixes are:

- *"Prefix schemes are bad/evil"*. I agree that "m_lpsz" and its ilk are poorly thought out and wholly useless. That's why I'd advise using a well designed notation designed to support your requirements, rather than copying something that is inappropriate for your context. (Use the right tool for the job).
- *"If I change the usage of something I have to rename it"*. Yes, of course you do, that's what refactoring is all about, and why IDEs have refactoring tools to do this job quickly and painlessly. Even without prefixes, changing the usage of a variable almost certainly means its name *ought* to be changed.
- *"Prefixes just confuse me"*. As does every tool until you learn how to use it. Once your brain has become used to the naming patterns, it will filter the information out automatically and you won't really mind that the prefixes are there any more. But you have to use a scheme like this solidly for a week or two before you'll really become "fluent". And that's when a lot of people look at old code and start to

wonder how they ever managed *without* a good prefix scheme.

- *"I can just look at the code to work this stuff out"*. Yes, but you don't need to waste time looking elsewhere in the code or remembering every little detail of it when the answer is right on the spot your eye is already focussed on.
- *(Some of) that information can be found by just waiting for a tooltip to pop up on my variable*. Yes. Where supported, for some types of prefix, when your code compiles cleanly, after a wait, you can read through a description and find the information the prefix would have conveyed instantly. I feel that the prefix is a simpler, more reliable and more efficient approach.
- *"It's more typing"*. Really? One whole character more? Or is it - with IDE auto-completion tools, it will often reduce typing, because each prefix character narrows the search space significantly. Press "e" and the three events in your class pop up in intellisense. Press "c" and the five constants are listed.
- *"I can use `this->` instead of `m`"*. Well, yes, you can. But that's just a much uglier and more verbose prefix! Only it carries a far greater risk (especially in teams) because to the compiler it is *optional*, and therefore its usage is frequently inconsistent. `m` on the other hand is brief, clear, explicit and not optional, so it's much harder to make mistakes using it.

Share Improve this answer Follow

edited Jan 27, 2013 at 12:36

community wiki
13 revs
Jason Williams

9 I mean to have read that the problem with Hungarien Notation just resulted from Simonyi being misunderstood. He wrote a prefix should be used to indicate the type of a variable where he meant "type" like in "kind of thing" not literal datatype. Later the platform guys at Microsoft picked it up and came up with lpsz... and the rest is history... – VoidPointer Aug 4, 2009 at 23:04

26 "s is for static" sounds pretty much like the "bad" form of Hungarian to me. – Stack Overflow is garbage Aug 24, 2011 at 11:46

7 @Mehrdad: I don't think `z` is very often useful in a language like C++ where that sort of low level implementation detail should be encapsulated in a class, but in C (where zero-termination is an important distinction) I agree with you. IMO any scheme we use should be adapted as required to best suit our own needs - So, if zero-termination affects your usage of the variable, there's nothing wrong with declaring "z" a useful prefix. – Jason Williams Apr 23, 2012 at 19:06

19 `The most important case is "p" for pointer (because the usage changes from var. to var-> and you have to be much more careful with pointers.` I whole heartedly disagree. If I use a pointer wrong, it simply won't compile (`void*` may be an exception for double pointers though). And the whole `->` over `.` is enough to tell me it is a pointer. Also, if you are using autocomplete, your editor probably has declaration tooltips,

eliminating the need for prefixing for variable information. Regardless, good answer. – Thomas Eding Nov 14, 2012 at 7:27

7 Upvoted for the clear, comprehensive and interesting explanations, however there is little here suggesting how this saves time in C++ YET remains largely unused in many other languages. – user234736 Jun 21, 2013 at 17:16

139

I generally don't use a prefix for member variables.

I used to use a `m` prefix, until someone pointed out that "C++ already has a standard prefix for member access: `this->`.

So that's what I use now. That is, *when there is ambiguity*, I add the `this->` prefix, but usually, no ambiguity exists, and I can just refer directly to the variable name.

To me, that's the best of both worlds. I have a prefix I can use when I need it, and I'm free to leave it out whenever possible.

Of course, the obvious counter to this is "yes, but then you can't see at a glance whether a variable is a class member or not".

To which I say "so what? If you need to know that, your class probably has too much state. Or the function is too big and complicated".

In practice, I've found that this works extremely well. As an added bonus it allows me to promote a local variable to a class member (or the other way around) easily, without having to rename it.

And best of all, it is consistent! I don't have to do anything special or remember any conventions to maintain consistency.

By the way, you *shouldn't* use leading underscores for your class members. You get uncomfortably close to names that are reserved by the implementation.

The standard reserves all names starting with double underscore or underscore followed by capital letter. It also reserves all names starting with a single underscore *in the global namespace*.

So a class member with a leading underscore followed by a lower-case letter is legal, but sooner or late you're going to do the same to an identifier starting with upper-case, or otherwise break one of the above rules.

So it's easier to just avoid leading underscores. Use a postfix underscore, or a `m_` or just `m` prefix if you want to encode scope in the variable name.

Share Improve this answer Follow

answered Aug 4, 2009 at 17:01

Stack Overflow is garbage

247k ● 52 ● 353 ● 555

"So a class member with a leading underscore followed by a lower-case letter is legal, but sooner or late you're going to do the same to an identifier starting with upper-case, or otherwise break one of the above rules." -- class member variables aren't in the global namespace, so a leading underscore is safe, regardless of whether it's followed by a lower or uppercase letter. – user185104 Sep 8, 2011 at 6:22

4 @mbarnett: No, underscore followed by upper-case is reserved *in general*, not just in the global namespace. – Stack Overflow is garbage Sep 8, 2011 at 7:45

10 surprised that this answer's vote is less than the prefix one. – Marson Mao Sep 2, 2014 at 8:47

I agree with this answer, just use `this->` if you need to specify that its a member variable, or don't, that's good too. – David Jan 10, 2017 at 22:14

Moreover, you don't have to document your convention to give your code to other people. Everyone understand what `this->` means. – Caduchon Sep 21, 2017 at 8:41

56

You have to be careful with using a leading underscore. A leading underscore before a capital letter in a word is reserved. For example:

_Foo

_L

are all reserved words while

_foo

_l

are not. There are other situations where leading underscores before lowercase letters are not allowed. In my specific case, I found the _L happened to be reserved by Visual C++ 2005 and the clash created some unexpected results.

I am on the fence about how useful it is to mark up local variables.

Here is a link about which identifiers are reserved: [What are the rules about using an underscore in a C++ identifier?]()

edited May 23, 2017 at 12:18
Community Bot
1 ● 1

answered Aug 4, 2009 at 15:33

Juan
3,756 ● 3 ● 32 ● 33

6 Actually, both _foo and _l are reserved at namespace scope.
– anon Aug 4, 2009 at 15:36

13 But they are ok as member variable names. I don't prefix underscores, because the rules are too confusing, and I had gotten burned in the past. – Juan Aug 4, 2009 at 15:40

13 These are not reserved words. They are reserved names. If they were reserved words, you couldn't use them at all. Because they are reserved names, you can use them, but at your own risk. – TonyK Nov 30, 2010 at 21:27

34

I prefer postfix underscores, like such:

```
class Foo
{
   private:
      int bar_;

   public:
      int bar() { return bar_; }
};
```

Share Improve this answer

Follow

answered Aug 4, 2009 at 16:46

jkeys
3,945 ● 11 ● 40 ● 63

1 Me too. I also give accessors/mutators the same name.
– Rob Aug 4, 2009 at 18:46

4 Interesting. Looks a bit ugly at first, but I can see how it can be beneficial. – ya23 Nov 16, 2009 at 13:57

7 I'd say it's a lot lot less ugly than:" mBar" or "m_bar". – sydan Apr 21, 2015 at 8:18

10 but then you have `vector<int> v_;` and writing `v_.push_back(5)` is pretty ugly too – avim Apr 29, 2015 at 21:24

4 That's Google C++ style. – Justme0 Nov 26, 2015 at 5:01

24

Lately I have been tending to prefer m_ prefix instead of having no prefix at all, the reasons isn't so much that its important to flag member variables, but that it avoids ambiguity, say you have code like:

```
void set_foo(int foo) { foo = foo; }
```

That of cause doesn't work, only one `foo` allowed. So your options are:

- `this->foo = foo;`

  I don't like it, as it causes parameter shadowing, you no longer can use `g++ -Wshadow` warnings, its also longer to type then `m_`. You also still run into naming conflicts between variables and functions when you have a `int foo;` and a `int foo();`.

- `foo = foo_;` or `foo = arg_foo;`

  Been using that for a while, but it makes the argument lists ugly, documentation shouldn't have do deal with name disambiguity in the implementation. Naming conflicts between variables and functions also exist here.

- `m_foo = foo;`

  API Documentation stays clean, you don't get ambiguity between member functions and variables and its shorter to type then `this->`. Only disadvantage is that it makes POD structures ugly, but as POD structures don't suffer from the name ambiguity in the first place, one doesn't need to use it with them. Having a unique prefix also makes a few search&replace operations easier.

- `foo_ = foo;`

  Most of the advantages of `m_` apply, but I reject it for aesthetic reasons, a trailing or leading underscore just makes the variable look incomplete and unbalanced. `m_` just looks better. Using `m_` is also more extendable, as you can use `g_` for globals and `s_` for statics.

PS: The reason why you don't see `m_` in Python or Ruby is because both languages enforce the their own prefix, Ruby uses `@` for member variables and Python requires `self.`.

Share Improve this answer Follow

edited Jul 22, 2010 at 13:34

answered Aug 29, 2009 at 10:29

Grumbel
6,983 ● 7 ● 45 ● 50

2 to be fair, you missed at least 2 other options, e.g. (a) use full names like `foo` only for members and instead use single-letter or short names for parameters or other locals/throwaways, such as `int f`; or (b) prefix the *parameters* or other locals with something. good point re `m_` and pods, though; i've independently arrived at a preference to follow both of those guidelines, for the most part. – underscore_d Jan 28, 2016 at 1:08

3 @underscore_d Parameter names are part of the public interface of a class. That should be the last place you're adding weird naming conventions. Also, single letter variable names are awful and should be avoided at all costs with *very* few exceptions (i in a loop). – Dan Bechard Feb 13, 2021 at 9:47

@DanBechard Good point about param names in APIs, but the name conflicts we're discussing don't happen there, they happen in the *implementation*. And there you're free to have some cosmetic deviations from the interface. (Of course, that would require some extra effort for something that's not even good practice, just the ifc/docs argument doesn't hold is all I'm saying.) One-letter vars have their legit roles, they can even help reducing noise and clutter, esp. in small/straightforward sections. *Orthodoxy* is considered more harmful in my book than those. – Sz. Feb 12, 2023 at 23:51

"longer to type" is a non-argument; remember that "code is read much more than it is written" - so readability trumps everything else. Also, re. "documentation" is non-argument too (at least for 2 reasons; first, IF it is such a huge concern - nobody has said that declaration == definition; second, if "documentation" is meant as doxygen - the only use for it I've seen in my 30+ years - including 20+ at architect positions, is to make managers happy, so its quality doesn't matter at all). – No-Bugs Hare May 13, 2023 at 18:10

15

When reading through a member function, knowing who "owns" each variable is absolutely essential to understanding the meaning of the variable. In a function like this:

```
void Foo::bar( int apples )
{
    int bananas = apples + grapes;
    melons = grapes * bananas;
    spuds += melons;
}
```

...it's easy enough to see where apples and bananas are coming from, but what about grapes, melons, and spuds? Should we look in the global namespace? In the class declaration? Is the variable a member of this object or a member of this object's class? Without knowing the answer to these questions, you can't understand the code. And in a longer function, even the declarations of local variables like apples and bananas can get lost in the shuffle.

Prepending a consistent label for globals, member variables, and static member variables (perhaps g_, m_, and s_ respectively) instantly clarifies the situation.

```
void Foo::bar( int apples )
{
    int bananas = apples + g_grapes;
    m_melons = g_grapes * bananas;
    s_spuds += m_melons;
}
```

These may take some getting used to at first—but then, what in programming doesn't? There was a day when even { and } looked weird to you. And once you get used to them, they help you understand the code much more quickly.

(Using "this->" in place of m_ makes sense, but is even more long-winded and visually disruptive. I don't see it as a good alternative for marking up all uses of member variables.)

A possible objection to the above argument would be to extend the argument to types. It might also be true that knowing the type of a variable "is absolutely essential to understanding the meaning of the variable." If that is so, why not add a prefix to each variable name that identifies its type? With that logic, you end up with Hungarian notation. But many people find Hungarian notation laborious, ugly, and unhelpful.

```
void Foo::bar( int iApples )
{
    int iBananas = iApples + g_fGrapes;
    m_fMelons = g_fGrapes * iBananas;
    s_dSpuds += m_fMelons;
}
```

Hungarian *does* tell us something new about the code. We now understand that there are several implicit casts in the Foo::bar() function. The problem with the code now is that the value of the information added by Hungarian prefixes is small relative to the visual cost. The C++ type system includes many features to help types either work well together or to raise a compiler warning or error. The compiler helps us deal with types—we don't need notation to do so. We can infer easily enough that the variables in Foo::bar() are probably numeric, and if that's all we know, that's good enough for gaining a general understanding of the function. Therefore the value of knowing the precise type of each variable is relatively low. Yet the ugliness of a variable like "s_dSpuds" (or even just "dSpuds") is great. So, a cost-benefit analysis rejects Hungarian notation, whereas the benefit of g_, s_, and m_ overwhelms the cost in the eyes of many programmers.

Share Improve this answer Follow

answered Oct 21, 2011 at 18:00

Jeff Wofford
11.5k ● 14 ● 52 ● 77

Thanks for the s_ idea. Seems very useful, and somehow had never occurred to me. – Chris Olsen May 3, 2020 at 22:03

10

I can't say how widespred it is, but speaking personally, I always (and have always) prefixed my member variables with 'm'. E.g.:

```
class Person {
   ....
   private:
       std::string mName;
};
```

It's the only form of prefixing I do use (I'm very anti Hungarian notation) but it has stood me in good stead over the years. As an aside, I generally detest the use of underscores in names (or anywhere else for that matter), but do make an exception for preprocessor macro names, as they are usually all uppercase.

Share Improve this answer Follow

answered Aug 4, 2009 at 15:30
anon

5 The problem with using m, rather than m_ (or _) is with the current fashion for camel case it makes it difficult to read some variable names. – Martin Beckett Aug 4, 2009 at 15:44

1 @Neil I am with you. @mgb: I hate names starting with '_' It is just an invitation for things to go wrong in the future. – Loki Astari Aug 4, 2009 at 16:20

1 @Neil: Which convention do you use then, if you don't use underscores, and don't use camelcase? – Stack Overflow is garbage Aug 4, 2009 at 17:07

2 My understanding was that it is camelCase which makes using just m for variables like 'apData' confusing - it becomes 'mapData' rather than 'm_apData'. I use _camelCase for protected/private member variables because it stands out – Martin Beckett Sep 4, 2009 at 21:16

11 @MartinBeckett: You should capitalize the `a` in that scenario-- you're not doing it right otherwise. `mApData` (`m` prefix, then the variable name is `apData`). – Platinum Azure Oct 21, 2011 at 18:27

10

The main reason for a member prefix is to distinguish between a member function and a member variable with the same name. This is useful if you use getters with the name of the thing.

Consider:

```
class person
{
public:
    person(const std::string& full_name)
        : full_name_(full_name)
    {}

    const std::string& full_name() const { return full
private:
    std::string full_name_;
};
```

The member variable could not be named `full_name` in this case. You need to rename the member function to `get_full_name()` or decorate the member variable somehow.

Share Improve this answer Follow

edited Aug 1, 2022 at 7:39
Wolf
10.2k ●8 ●66 ●112

answered Aug 5, 2009 at 4:08
matthewinrandwick

2 This is the reason I prefix. I think `foo.name()` is much more readable than `foo.get_name()` in my opinion.

– Terrabits Jul 14, 2016 at 16:49

1 @Wolf what do you mean by a member function *local*? I think the OP's original intention is just distinguishing a member function's name, isn't it? – starriet 차주녕 Jul 28, 2022 at 23:42

1 @Wolf This answer (and also the code) is talking about the member function's name, not the member function's local variables' name. The local variables of the member function `full_name()` don't even exist in this code example. (BTW, I meant 'answerer' when I said 'OP' in my previous comment) – starriet 차주녕 Aug 1, 2022 at 1:16

1 @starriet: I now rolled back my first edit. To explain my misunderstanding: I never focus on getters and setters in C++. So thanks for your intransigence on this point. – Wolf Aug 1, 2022 at 7:42

6

I don't think one syntax has real value over another. It all boils down, like you mentionned, to uniformity across the source files.

The only point where I find such rules interesting is when I need 2 things named identicaly, for example :

```
void myFunc(int index){
  this->index = index;
}

void myFunc(int index){
  m_index = index;
}
```

I use it to differentiate the two. Also when I wrap calls, like from windows Dll, *RecvPacket(...)* from the Dll might be wrapped in *RecvPacket(...)* in my code. In these particular occasions using a prefix like "_" might make the two look alike, easy to identify which is which, but different for the compiler

Share Improve this answer Follow

answered Aug 4, 2009 at 15:33

Eric
19.9k ● 19 ● 88 ● 149

6

Some responses focus on refactoring, rather than naming conventions, as the way to improve readability. I don't feel that one can replace the other.

I've known programmers who are uncomfortable with using local declarations; they prefer to place all the declarations at the top of a block (as in C), so they know where to find them. I've found that, where scoping allows for it, declaring variables where they're first used decreases the time that I spend glancing backwards to find the declarations. (This is true for me even for small functions.) That makes it easier for me to understand the code I'm looking at.

I hope it's clear enough how this relates to member naming conventions: When members are uniformly prefixed, I never have to look back at all; I know the declaration won't even be found in the source file.

I'm sure that I didn't start out preferring these styles. Yet over time, working in environments where they were used consistently, I optimized my thinking to take advantage of them. I think it's possible that many folks who currently feel uncomfortable with them would also come to prefer them, given consistent usage.

Share Improve this answer Follow

answered Aug 4, 2009 at 18:42

Dan Breslau
11.5k ● 2 ● 36 ● 44

5

Those conventions are just that. Most shops use code conventions to ease code readability so anyone can easily look at a piece of code and quickly decipher between things such as public and private members.

Share Improve this answer Follow

answered Aug 4, 2009 at 15:31

Mr. Will
2,308 ● 3 ● 21 ● 28

"between things such as public and private members" - how common is this really? i don't recall seeing it, but then again, i don't go around reviewing codebases or anything.
– underscore_d Jan 28, 2016 at 1:10

I don't do it in my own coding, but I've worked at places where we had to do it based on their code convention guides. I prefer to not do it as almost all IDEs will show private variables a different color. – Mr. Will Jan 28, 2016 at 2:42

Hmm, I guess it only happens in different situations than mine. Normally I use either `class`es all of whose members are `private` / `protected`, or POD `struct`s all of whose

variables are `public` (and often also `const`). So, I never need to wonder about the access level of any given member.
– underscore_d Jan 28, 2016 at 10:36

5

> Others try to enforce using this->member whenever a member variable is used

That is usually *because there is no prefix*. The compiler needs enough information to resolve the variable in question, be it a unique name because of the prefix, or via the `this` keyword.

So, yes, I think prefixes are still useful. I, for one, would prefer to type '_' to access a member rather than 'this->'.

Share Improve this answer Follow

edited Sep 27, 2011 at 15:26
user142162

answered Aug 4, 2009 at 15:30
Kent Boogaart
179k ● 37 ● 398 ● 395

3 the compiler can resolve it anyways... local variables will hide ones in higher scope in most langauges. It's for the (dubious) benefit of the humans reading the code. Any decent IDE will highlight locals/members/globals in different ways so there's no need for this sort of stuff – rmeador Aug 4, 2009 at 15:37

2 Exactly. Locals will hide class members. Consider a constructor that sets these members. Usually it makes sense

to name the parameters the same as the members.
– Kent Boogaart Aug 4, 2009 at 15:44

6 Why is that a code smell? I'd say it's perfectly common and reasonable, especially when it comes to constructors.
– Kent Boogaart Aug 4, 2009 at 17:19

3 A constructor should (generally) set locals in its initialization list. And there, parameters don't shadow field names, but both are accessible - so you can write `struct Foo { int x; Foo(int x) : x(x) { ... } };` – Pavel Minaev Aug 4, 2009 at 17:33

3 I assume the problem with that comes when you do `Foo(int x, bool blee) : x(x) { if (blee) x += bleecount; } // oops, forgot this->` I prefer to call my member variables something useful and then give constructor parameters that match them abbreviated names: `Foo(int f) : foo(f) {...}` – Steve Jessop Aug 4, 2009 at 17:40

4

Other languages will use coding conventions, they just tend to be different. C# for example has probably two different styles that people tend to use, either one of the C++ methods (_variable, mVariable or other prefix such as Hungarian notation), or what I refer to as the StyleCop method.

```
private int privateMember;
public int PublicMember;

public int Function(int parameter)
{
  // StyleCop enforces using this. for class members.
```

```
    this.privateMember = parameter;
 }
```

In the end, it becomes what people know, and what looks best. I personally think code is more readable without Hungarian notation, but it can become easier to find a variable with intellisense for example if the Hungarian notation is attached.

In my example above, you don't need an m prefix for member variables because prefixing your usage with this. indicates the same thing in a compiler-enforced method.

This doesn't necessarily mean the other methods are bad, people stick to what works.

Share Improve this answer Follow

answered Aug 4, 2009 at 15:33
Will Eddins
13.9k ● 7 ● 52 ● 85

3

When you have a big method or code blocks, it's convenient to know immediately if you use a local variable or a member. it's to avoid errors and for better clearness !

Share Improve this answer Follow

answered Aug 4, 2009 at 15:33
Matthieu
2,751 ● 19 ● 21

3 If you have a big method, for better clearness break it down.
– sbi Aug 4, 2009 at 15:52

4 There are lots of reasons not to break down some big methods. For example, if your method needs to keep a lot of local state, you either have to pass lots of parameters into your subordinate methods, create new classes that exist solely for the purpose of passing data between these methods, or declaring the state data as member data of the parent class. All of these have problems that would affect the clarity or maintainability of the method, compared to a single long-ish method (especially one whose logic is straightforward). – Steve Broberg Aug 4, 2009 at 17:12

3 @sbi: Guidelines are just that; guidelines, not rules. Sometimes you need large methods that don't logically lend themselves to being split apart, and sometimes parameter names clash with members. – Ed Swangren Aug 4, 2009 at 17:23

Please don't be making your member variables public. Just use accessors. The parentheses should tell the reader that it is a member variable. – jkeys Aug 5, 2009 at 0:34

Note that there is a warning in gcc (>= 4.6) to detect clash of names : `-Wshadow` – Caduchon Sep 21, 2017 at 8:27

3

IMO, this is personal. I'm not putting any prefixes at all. Anyway, if code is meaned to be public, I think it should better has some prefixes, so it can be more readable.

Often large companies are using it's own so called 'developer rules'.
Btw, the funniest yet smartest i saw was DRY KISS (Dont Repeat Yourself. Keep It Simple, Stupid). :-)

Share Improve this answer answered Aug 4, 2009 at 15:36

Follow

Andrejs Cainikovs
28.4k ● 2 ● 78 ● 96

3

As others have already said, the importance is to be colloquial (adapt naming styles and conventions to the code base in which you're writing) and to be consistent.

For years I have worked on a large code base that uses both the "this->" convention as well as using a *postfix* underscore notation for member variables. Throughout the years I've also worked on smaller projects, some of which did not have any sort of convention for naming member variables, and other which had differing conventions for naming member variables. Of those smaller projects, I've consistently found those which lacked any convention to be the most difficult to jump into quickly and understand.

I'm very anal-retentive about naming. I will agonize over the name to be ascribed to a class or variable to the point that, if I cannot come up with something that I feel is "good", I will choose to name it something nonsensical and provide a comment describing what it really is. That way, at least the name means exactly what I intend it to mean--nothing more and nothing less. And often, after using it for a little while, I discover what the name should *really* be and can go back and modify or refactor appropriately.

One last point on the topic of an IDE doing the work--that's all nice and good, but IDEs are often not available in environments where I have perform the most urgent work. Sometimes the only thing available at that point is a copy of 'vi'. Also, I've seen many cases where IDE code completion has propagated stupidity such as incorrect spelling in names. Thus, I prefer to not have to rely on an IDE crutch.

Share Improve this answer Follow

answered Aug 4, 2009 at 16:08
Chris Cleeland
4,890 ● 3 ● 27 ● 28

3

The original idea for prefixes on C++ member variables was to store additional type information that the compiler didn't know about. So for example, you could have a string that's a fixed length of chars, and another that's variable and terminated by a '\0'. To the compiler they're both `char *`, but if you try to copy from one to the other you get in huge trouble. So, off the top of my head,

```
char *aszFred = "Hi I'm a null-terminated string";

char *arrWilma = {'O', 'o', 'p', 's'};
```

where "asz" means this variable is "ascii string (zero-terminated) and "arr" means this variable is a character array.

Then the magic happens. The compiler will be perfectly happy with this statement:

```
strcpy(arrWilma, aszFred);
```

But you, as a human, can look at it and say "hey, those variables aren't really the same type, I can't do that".

Unfortunately a lot places use standards such as "m_" for member variables, "i" for integers no matter how used, "cp" for char pointers. In other words they're duplicating what the compiler knows, and making the code hard to read at the same time. I believe this pernicious practice should be outlawed by statute and subject to harsh penalties.

Finally, there's two points I should mention:

- Judicious use of C++ features allows the compiler to know the information you had to encode in raw C-style variables. You can make classes that will only allow valid operations. This should be done as much as practical.
- If your code blocks are so long that you forget what type a variable is before you use it, they are *way* too long. Don't use names, re-organize.

Share Improve this answer Follow

answered Aug 4, 2009 at 16:40

A. L. Flanagan
1,168 ● 8 ● 22

Prefixes that indicate type or kind of variable are also something worth a discussion, but I was referring mainly to prefixes indicating whether something is a (private) member/field. The reverse Hungarian notation you are mentioning can be quite handy when applied intelligently (like in your example). My favorite example where it makes sense is relative and absolute coordinates. when you see absX = relX you can plainly see that something might be wrong.you can also name functions accordingly: absX = absFromRel(relX, offset); – VoidPointer Aug 4, 2009 at 23:10

Note: the initialization of aszFred is questionable (offering non-const access to a literal string), and the initialization of arrWilma won't even compile. (You probably intended to declare arrWilma as an array, instead of a pointer!) No problem though, as you wrote that it's just off the top of your head... :-) – Niels Dekker Aug 7, 2009 at 13:48

Oops, you're absolutely right. Kids, don't try that at home. Do this: 'const char *aszFred = "Hi I'm a null-terminated string"; char arrWilma[] = {'O', 'o', 'p', 's'};' – A. L. Flanagan Dec 31, 2009 at 21:11

3

Our project has always used "its" as a prefix for member data, and "the" as a prefix for parameters, with no prefix for locals. It's a little cutesy, but it was adopted by the early developers of our system because they saw it used as a convention by some commercial source libraries we were using at the time (either XVT or RogueWave - maybe both). So you'd get something like this:

```
void
MyClass::SetName(const RWCString &theName)
{
   itsName = theName;
}
```

The big reason I see for scoping prefixes (and no others - I hate Hungarian notation) is that it prevents you from getting into trouble by writing code where you think you're referring to one variable, but you're really referring to another variable with the same name defined in the local scope. It also avoids the problem of coming up with a variable names to represent that same concept, but with different scopes, like the example above. In that case, you would have to come up with some prefix or different name for the parameter "theName" anyway - why not make a consistent rule that applies everywhere.

Just using this-> isn't really good enough - we're not as interested in reducing ambiguity as we are in reducing coding errors, and masking names with locally scoped identifiers can be a pain. Granted, some compilers may have the option to raise warnings for cases where you've masked the name in a larger scope, but those warnings may become a nuisance if you're working with a large set of third party libraries that happen to have chosen names for unused variables that occasionally collide with your own.

As for the its/the itself - I honestly find it easier to type than underscores (as a touch typist, I avoid underscores whenever possible - too much stretching off the home rows), and I find it more readable than a mysterious underscore.

Share Improve this answer Follow

answered Aug 4, 2009 at 17:29

Steve Broberg

4,384 ● 3 ● 31 ● 44

This is the most intuitive solution with the fastest learning curve I've ever heard. I wish spoken languages were more flexible to handle all those so that we didn't have to think about coming up with new techniques to solve ambiguities in code. – Guney Ozsan Jun 5, 2016 at 13:07

2

I use it because VC++'s Intellisense can't tell when to show private members when accessing out of the class. The only indication is a little "lock" symbol on the field icon in the Intellisense list. It just makes it easier to identify private members(fields) easier. Also a habit from C# to be honest.

```
class Person {
   std::string m_Name;
public:
   std::string Name() { return m_Name; }
   void SetName(std::string name) { m_Name = name; }
};

int main() {
  Person *p = new Person();
  p->Name(); // valid
  p->m_Name; // invalid, compiler throws error. but in
this..
  return 1;
}
```

Share Improve this answer Follow

answered Aug 4, 2009 at 17:52

Zack
2,485 ● 4 ● 39 ● 51

2

I think that, if you need prefixes to distinguish class members from member function parameters and local variables, either the function is too big or the variables are badly named. If it doesn't fit on the screen so you can easily see what is what, refactor.

Given that they often are declared far from where they are used, I find that naming conventions for global constants (and global variables, although IMO there's rarely ever a need to use those) make sense. But otherwise, I don't see much need.

That said, I used to put an underscore at the end of all private class members. Since all my data is private, this implies members have a trailing underscore. I usually don't do this anymore in new code bases, but since, as a programmer, you mostly work with old code, I still do this a lot. I'm not sure whether my tolerance for this habit comes from the fact that I used to do this always and am still doing it regularly or whether it really makes more sense than the marking of member variables.

Share Improve this answer Follow

edited Aug 6, 2009 at 14:32

answered Aug 4, 2009 at 15:46

sbi
224k ● 46 ● 264 ● 447

2 This very much reflects my on feeling about this issue. Code should be readable without resorting to prefixes. Maybe we

don't see so much prefix uses in more modern languages because their user communities have embraced readability a bit more than what you sometimes see in C++. Of course, C++ can and should be readable. It's just that a lot of unreadable C++ has been written over the years.
– VoidPointer Aug 4, 2009 at 15:55

2

In python leading double underscores are used to emulate private members. For more details see this answer

Share Improve this answer Follow

edited May 23, 2017 at 12:02
Community Bot
1 ● 1

answered Nov 16, 2009 at 14:50
Konstantin Tenzin
12.8k ● 3 ● 24 ● 20

2

I like variable names to give only a meaning to the values they contain, and leave how they are declared/implemented out of the name. I want to know what the value means, period. Maybe I've done more than an average amount of refactoring, but I find that embedding how something is implemented in the name makes refactoring more tedious than it needs to be. Prefixes indicating where or how object members are declared are implementation specific.

```
color = Red;
```

Most of the time, I don't care if Red is an enum, a struct, or whatever, and if the function is so large that I can't remember if color was declared locally or is a member, it's probably time to break the function into smaller logical units.

If your cyclomatic complexity is so great that you can't keep track of what is going on in the code without implementation-specific clues embedded in the names of things, most likely you need to reduce the complexity of your function/method.

Mostly, I only use 'this' in constructors and initializers.

Share Improve this answer Follow

edited Sep 25, 2015 at 19:16
hata
12.4k ● 6 ● 48 ● 73

answered Sep 25, 2015 at 18:46
ChrisG65
79 ● 2

2

I use m_ for member variables just to take advantage of Intellisense and related IDE-functionality. When I'm coding the implementation of a class I can type m_ and see the combobox with all m_ members grouped together.

But I could live without m_ 's without problem, of course. It's just my style of work.

Share Improve this answer Follow

answered Apr 30, 2017 at 3:39

Hernán
4,587 ● 4 ● 34 ● 48

2 You could also type `this->` – Toast Apr 18, 2019 at 19:54

1 But that's more typing ;-) – Traummaennlein May 3, 2022 at 7:41

1

It is useful to differentiate between member variables and local variables due to memory management. Broadly speaking, heap-allocated member variables should be destroyed in the destructor, while heap-allocated local variables should be destroyed within that scope. Applying a naming convention to member variables facilitates correct memory management.

Share Improve this answer Follow

answered Aug 4, 2009 at 16:32

frankster
1,528 ● 2 ● 16 ● 20

how so? The destructor doesn't have access to local variables declared in other functions, so there's no room for confusion there. Besides, heap-allocated local variables *shouldn't exist*. And heap-allocated member variables should only exist inside RAII classes, pretty much.
– Stack Overflow is garbage Aug 4, 2009 at 17:02

"heap-allocated local variables shouldn't exist" is a bit strong. But if/when you use them, its super-important to make sure that they get deallocated corectly, so a disciplined naming convention for member versus local variables assists immeasurably with ensuring this. – frankster Aug 4, 2009 at 19:30

1

Code Complete recommends m_varname for member variables.

While I've never thought the m_ notation useful, I would give McConnell's opinion weight in building a standard.

Share Improve this answer Follow

answered Aug 4, 2009 at 18:04

Paul Nathan
40.2k ● 30 ● 120 ● 215

2 Not unless he explains why the underscore. I'm a big fan of his "Rapid Development" book, which I've recommended here numerous times, but much less of "Code Complete" (which I will admit I haven't read since it first came out). – anon Aug 4, 2009 at 18:09

1

I almost never use prefixes in front of my variable names. If you're using a decent enough IDE you should be able to refactor and find references easily. I use very clear names and am not afraid of having long variable names. I've never had trouble with scope either with this philosophy.

The only time I use a prefix would be on the signature line. I'll prefix parameters to a method with _ so I can program defensively around them.

Share Improve this answer Follow

answered Aug 4, 2009 at 21:13

James
6,499 ● 11 ● 60 ● 86

1

You should never need such a prefix. If such a prefix offers you any advantage, your coding style in general needs fixing, and it's not the prefix that's keeping your code from being clear. Typical bad variable names include "other" or "2". You do not fix that with requiring it to be mOther, you fix it by getting the developer to think about what that variable is doing there in the context of that function. Perhaps he meant remoteSide, or newValue, or secondTestListener or something in that scope.

It's an effective anachronism that's still propagated too far. Stop prefixing your variables and give them proper names whose clarity reflects how long they're used. Up to 5 lines you could call it "i" without confusion; beyond 50 lines you need a pretty long name.

Share Improve this answer Follow

answered Dec 5, 2013 at 12:09

dascandy
7,292 ● 1 ● 30 ● 50

1

Wrap all private data with a struct

```
class Demo {
public:
    Demo() {
        self.name = "Harry Porter";
        self.age = 18;
    }
protected:
    struct self_type {
        std::string name;
        int age;
    } self = {};
};
```

Share Improve this answer Follow

edited Feb 20, 2023 at 7:15

answered Sep 22, 2022 at 1:03

Sunding Wei
2,208 ● 20 ● 13

Please don't. Dollar is not portable. stackoverflow.com/questions/26301737/... – vines Nov 18, 2022 at 14:57

0

According to JOINT STRIKE FIGHTER AIR VEHICLE C++ CODING STANDARDS (december 2005):

> AV Rule 67

> Public and protected data should only be used in structs—not classes. Rationale: A class is able to maintain its invariant by controlling access to its data. However, a class cannot control access to its members if those members non-private. Hence all data in a class should be private.

Thus, the "m" prefix becomes unuseful as all data should be private.

But it is a good habit to use the p prefix before a pointer as it is a dangerous variable.

0

Many of those conventions are from a time without sophisticated editors. I would recommend using a proper IDE that allows you to color every kind of variable. Color is by far easier to spot than any prefix.

If you need to get even more detail on a variable any modern IDE should be able to show it to you by moving the caret or cursor over it. And if you use a variable in a wrong way (for instance a pointer with the . operator) you will get an error, anyway.

Share Improve this answer Follow

edited Aug 24, 2019 at 8:30

answered Aug 24, 2019 at 8:24

Elias Mueller

31 ● 4

1 2 Next

Highly active question. Earn 10 reputation (not counting the association bonus) in order to answer this question. The reputation requirement helps protect this question from spam and non-answer activity.