Is shadowing a member variable from a base class a bad thing? Maybe, but maybe not.

devblogs.microsoft.com/oldnewthing/20240304-00

March 4, 2024

Raymond Chen

What is shadowing? In C++, *shadowing* is the name given to the phenomenon when a name in one scope hides an identical name in another scope.

Is shadowing bad? That's a harder question.

Whether shadowing is good or bad depends on the order in which the conflicting names were introduced.

Suppose you have a class library, and one of the classes is this:

```
struct Tool {
    int a;
};
```

And suppose some customer uses your class like this:

```
class Wrench : public Tool {
private:
    int a;
};
```

In this case, shadowing is probably unintended. The customer has accidentally shadowed `Tool::a`, and any references to `a` in the `Wrench` class will refer to `Wrench::a`, even if the author meant to access `Tool::a`.

Meanwhile, you have another customer who writes this:

```
class Pliers : public Tool {
private:
    int b;
};
```

There is no shadowing going on here. Everything is just fine. Your library's `Tool` class happily accesses the `a` member, and the customers `Pliers` class can also access the `Tool::a` member.

In the next version of your library, you add a new member to `Tool`.

```
struct Tool {
    int a;
    int b; // new member variable
};
```

Does this create a problem with customers of your library? In particular, does this break the `Pliers` class?

No, it doesn't create any problems.

All of your library code that uses `Tool` will continue to use `Tool` pointers and references, and it can write `tool->b` or `tool.b` to access that new `b` member. Even if the `Pliers` object is passed to a function that expects a `Tool`, the fact that `Pliers::b` shadows `Tool::b` has no effect on code that uses the `Tool`. Inside the `Tool` class, `this` is a `Tool*`, so writing `b` refers to `this->b`, which is the `Tool`'s `b`. Outside the `Tool` class, `tool->b` and `tool.b` still access the `Tool::b`.

Meanwhile, the `Pliers` is also unaffected. When code in the `Pliers` class writes `b`, it gets the `Pliers`'s `b` member, just like it did in version 1 of your tool library. And if you have a `Pliers*` or a `Pliers&`, you can write `pliers->b` or `pliers.b`, and you will get `Pliers::b`. Since there was no `Tool::b` at the time the original `Pliers` code was written, there was no possibility of writing `tool->b` in the `Pliers` project because there was no `Tool::b` in version 1 of the library.

Shadowing saves the day! You can add a member to the base class without breaking existing code.

Now, of course, if after upgrading to version 2 of your tools library, the `Pliers` wants to start taking advantage of the new `b` member, then the developer of the `Pliers` object will have to do some extra work to resolve the conflict by writing `pliers->b` to access `Pliers::b` and `pliers->Tool::b` to access `Tool::b`.

So sometimes shadowing is a good thing that keeps code working. And sometimes shadowing is a bad thing that prevents new code from working. Whether it's more good than bad depends on the order in which the conflicting names were introduced. And the order in which the code was written is not something a compiler has insight into. It's something that only you, the human being who assembled the project from a library and some personally-written code, will understand.