# Interface vs Implementation in C++. What does this mean? [closed]

I am learning the concepts of inheritance, especially about access-specifiers, here I am confused about the `protected` access specifier. The members under protected can be accessible by the base class member functions and derived class member functions. There is a chance of messing up `implementation` of base class if we declare protected as access specifier. Its always better to declare data members under `private` rather than protected as only the `interface` is exposed and not the `implementation` section. We are declaring only the variables in the private section of a class and how it becomes `implementation` ? Implementation will be done in the `member functions` right? The terms are confusing, can anyone clarify and explain me the terms?

c++ inheritance interface protected

asked Aug 5 '16 at 11:33

Kyle Reese
**3** 3

**closed** as unclear what you're asking by Martin, Galik, EdChum, πάντα ῥεῖ, Rakete1111 Aug 6 '16 at 2:07

Please clarify your specific problem or add additional details to highlight exactly what you need. As it's currently written, it’s hard to tell exactly what you're asking. See the How to Ask page for help clarifying this question.

If this question can be reworded to fit the rules in the help center, please edit the question.

Provide a comment atleast rather than down voting :-/ – Kyle Reese Aug 5 '16 at 11:35

Protected member function can be good for utility-type functions that might be needed by the sub-classes, but still not be publicly available. They can also be used for virtual functions that are called by the parent class and then overridden in the child-classes to provide some specific, but still not public, operation. – Some programmer dude Aug 5 '16 at 11:36

3 These ideas will become clearer if you use them. As it stands your question isn't very clear. – doctorlove Aug 5 '16 at 11:36

@JoachimPileborg What are utility-type functions? – Kyle Reese Aug 5 '16 at 11:37

From the Merriam-Webster dictionary: "something useful or designed for use" and "a program or routine designed to perform or facilitate especially routine operations (as copying files or editing text) on a computer". I.e. common tool-like functions. – Some programmer dude Aug 5 '16 at 11:40

## 1 Answer

Interface and implementation are not ideas specific to C++ and it sounds like you're confused about what interfaces vs. implementations are *in general*, so hopefully by explaining what they are it will be easier to understand it in C++.

This SO question (though not exactly what you're asking) has a good definition for what an interface is:

> An interface is a **contract**: the guy writing the interface says, "hey, I accept things looking that way", and the guy using the interface says "Ok, the class I write looks that way".

> **An interface is an empty shell**, there are only the signatures of the methods, which implies that the methods do not have a body. The interface can't do anything. It's just a pattern.

And in his example the **interface** is (translated to C++):

```
class MotorVehicle
{
public:
    virtual void run() const = 0;
    virtual int getFuel() const = 0;
}
```

And then the **implementation** is:

```
class Car : public MotorVehicle
{
    int fuel;

public:
    void run() const override
    {
        printf("Wrroooooooom\n");
    }


    int getFuel() const override
    {
        return this->fuel;
    }
}
```

The **implementation** is the actual substance behind the *idea*, the actual definition of how the *interface* will do what we expect it to. Another example: in terms of algorithms we talk about a Depth First Search (DFS) and it has a clearly defined behavior, but how we *code*, or *implement*, that algorithm can vary. We could use recursion or a stack data structure, for instance.

Now as regards **access specifiers**: it is not bad to use `protected` access. We talk about inheritance as an **"is-a"** relationship. When we say `Cat` inherits from `Animal` , we also say `Cat` **is an** `Animal` . So for the `Cat` to use some of the instance variables of the `Animal` is perfectly normal because it *should* belong to the `Cat` anyway.

You're worried that something the subclass does will mess up what the superclass does by changing the instance variables. You can certainly do that by throwing in meaningless data from the subclass, but usually you don't do that. You use the instance variables as the superclass intended them to be used (otherwise you would indeed screw up the functionality), which should be documented. If you are still thinking that someone should really not use your instance variables then that's what the `private` specifier is for.

One last thing: overriding superclass methods should also prevent misuse of superclass variables. By accessing and writing to `protected` variables you might change the behavior of superclass methods to something undesired, but then those methods should be overridden to do the new thing that your subclass is intending to do.

edited Aug 8 '16 at 8:06

answered Aug 5 '16 at 12:57

Matthew Woo
**526** 2 16

It sounded like he was confused about what an interface and implementation are in general, so I thought I'd answer that part. Should I change the quoted example to C++? – Matthew Woo Aug 5 '16 at 15:18

*"Should I change the quoted example to C++?"* Yes that would be definitiely better, everything else will be confusing future visitors. – πάντα ῥεῖ Aug 5 '16 at 15:20