Lecture 15

Inheritance, pt.1

Konstantin Leladze

OOP in C++

DIHT MIPT 2021

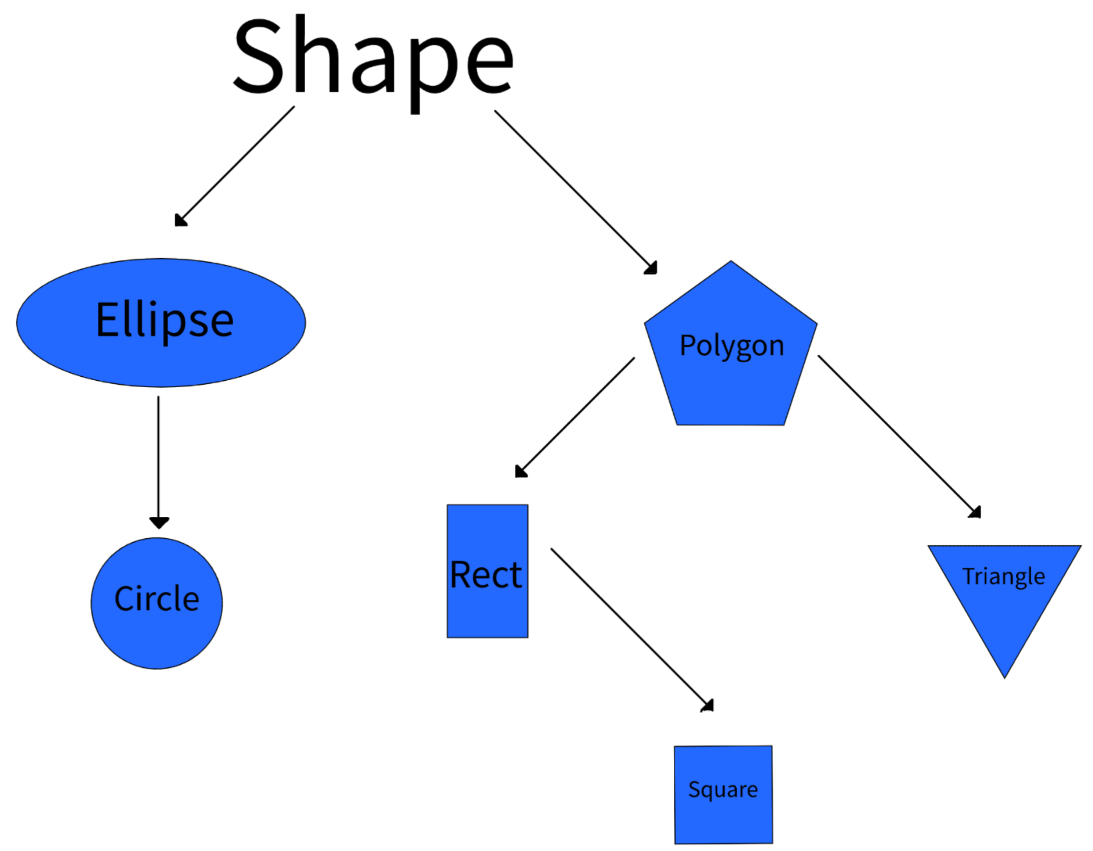
Three principles of OOP

OOP has three principles, they are listed below. These principles are present not only in C++, but in any object-oriented language. Below I have given a list of these principles and **the mechanisms by which they are implemented in C ++**.

- Encapsulation: **private and public access modifiers**
- Polymorphism:
  - static polymorphysm (compile-time): **templates**
  - dynamic polymorphysm (runtime): **virtual functions & inheritance**
- Inheritance**: inheritance**

In green I colored what we have already managed to talk about, in red - what is yet to come.

Inheritance, idea

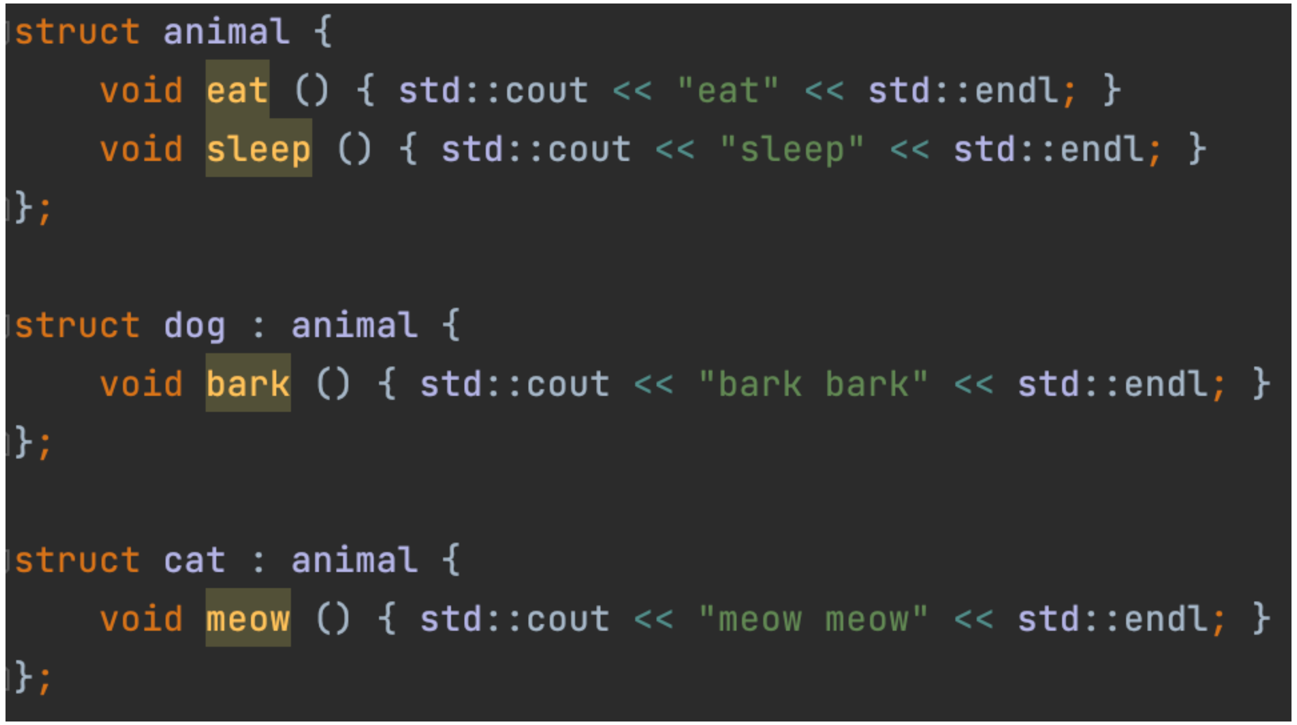


Inheritance is one of the key features of Object-oriented programming in C++. It allows us to create a new class (derived class) from an existing class (base class).

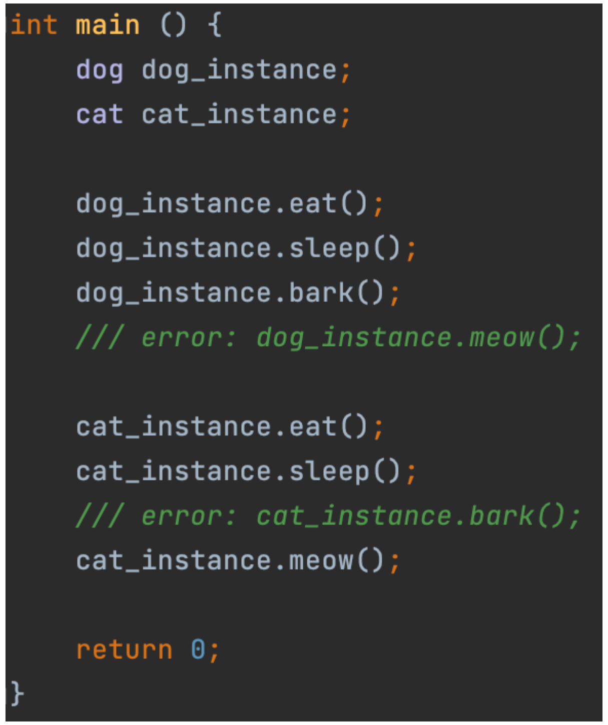
The derived class inherits the features (members and methods) from the base class and can have additional features (members and methods) of its own.

In practice, very often you have to create object hierarchies in order to select their common properties, which allows you to reduce the amount of code.

Inheritance, motivation



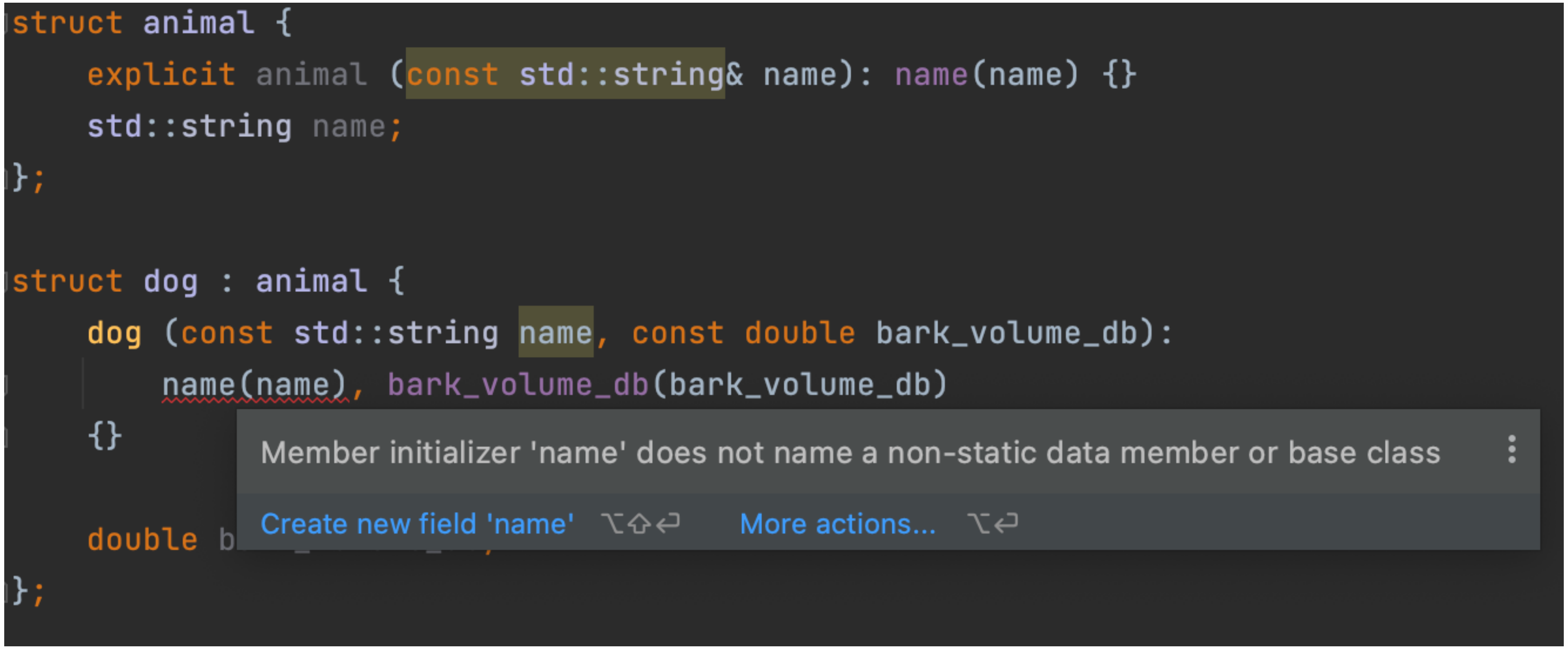
```cpp
struct animal {
    void eat () { std::cout << "eat" << std::endl; }
    void sleep () { std::cout << "sleep" << std::endl; }
};

struct dog : animal {
    void bark () { std::cout << "bark bark" << std::endl; }
};

struct cat : animal {
    void meow () { std::cout << "meow meow" << std::endl; }
};
```



```cpp
int main () {
    dog dog_instance;
    cat cat_instance;

    dog_instance.eat();
    dog_instance.sleep();
    dog_instance.bark();
    /// error: dog_instance.meow();

    cat_instance.eat();
    cat_instance.sleep();
    /// error: cat_instance.bark();
    cat_instance.meow();

    return 0;
}
```

Both the cat and the dog can eat and sleep, but the dog has a unique option - to bark. Cat – to meow. The motivation for using inheritance in this case is obvious, we write less code due to the reuse of the eat and sleep methods.

How to initialize base class



```
struct animal {
    explicit animal (const std::string& name): name(name) {}
    std::string name;
};

struct dog : animal {
    dog (const std::string name, const double bark_volume_db):
        name(name), bark_volume_db(bark_volume_db)
    {}

    double b
};
```

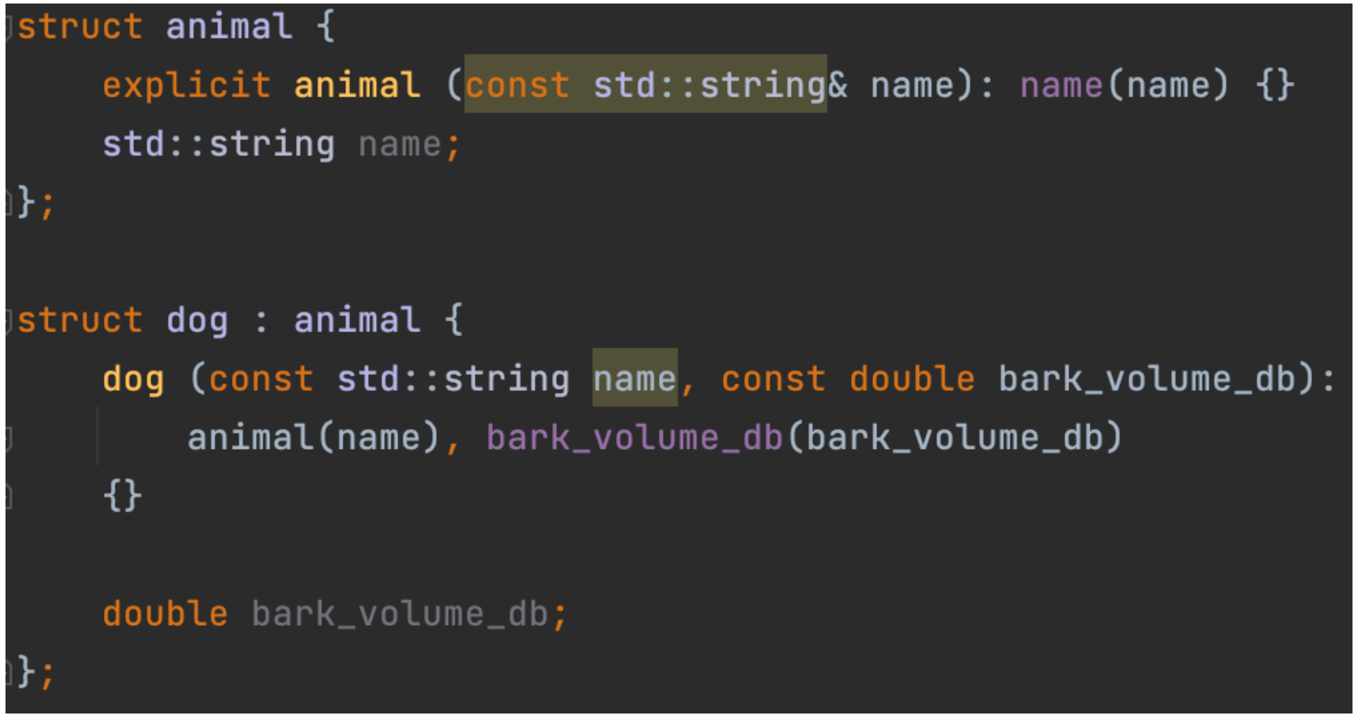
Member initializer 'name' does not name a non-static data member or base class

Create new field 'name' ⌥⇧↩ More actions... ⌥↩

If we inherit from a class that accepts a value in the constructor, then we will need to somehow initialize the base class (class which we inherit the derived class from).

How to initialize base class: solution

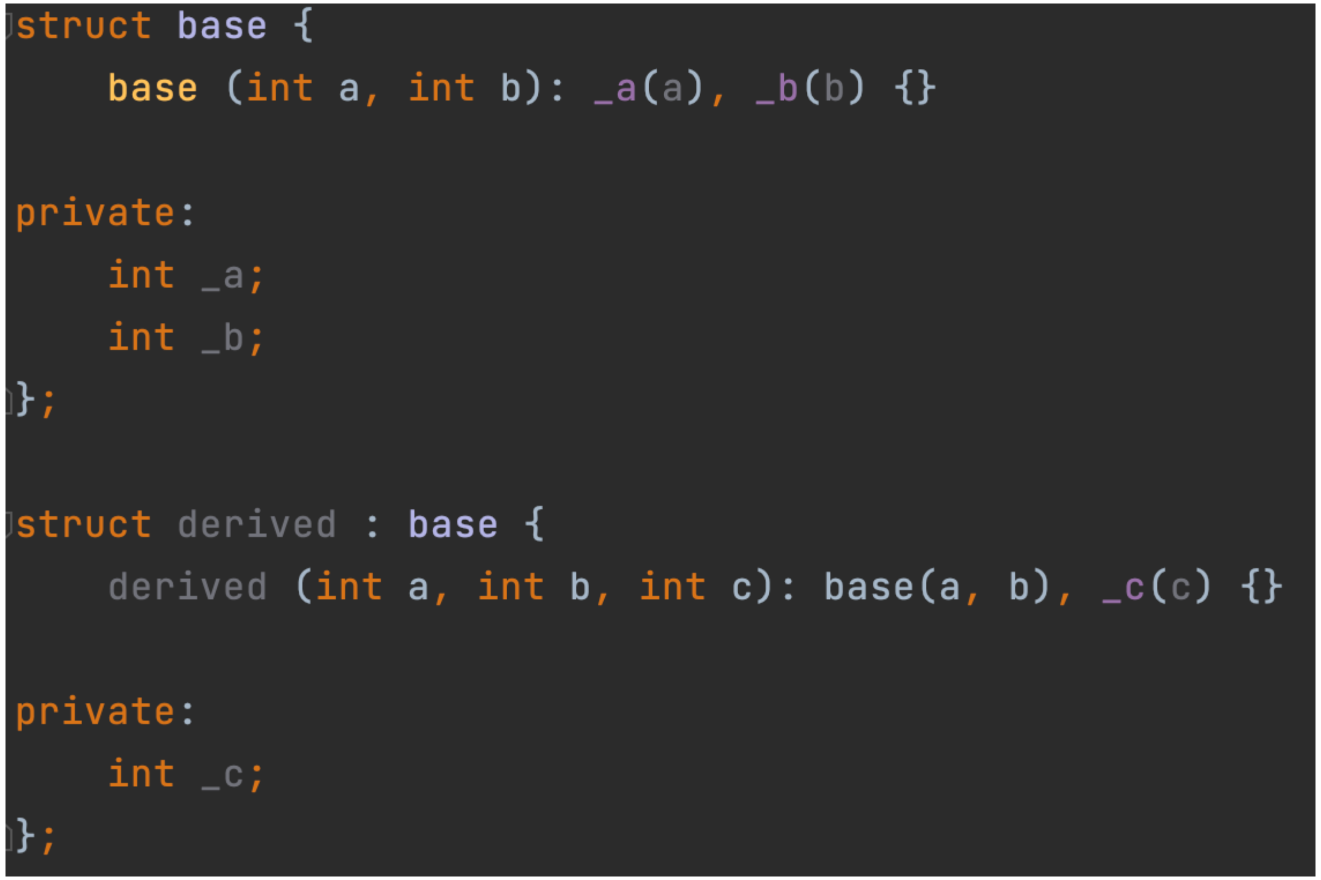
Solution: let’s use delegate constructors in order to initialize the base.



```cpp
struct animal {
    explicit animal (const std::string& name): name(name) {}
    std::string name;
};

struct dog : animal {
    dog (const std::string name, const double bark_volume_db):
        animal(name), bark_volume_db(bark_volume_db)
    {}

    double bark_volume_db;
};
```

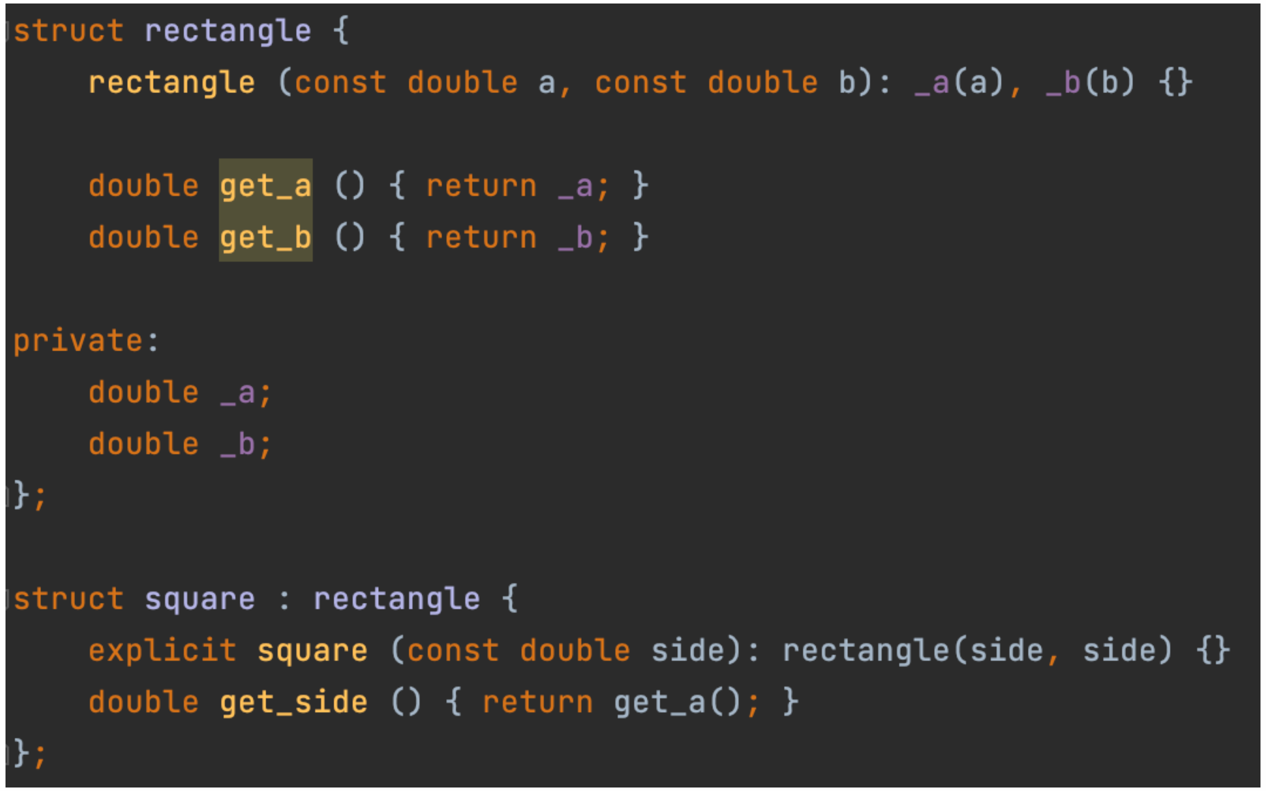
How to initialize base class: solution

Another example:

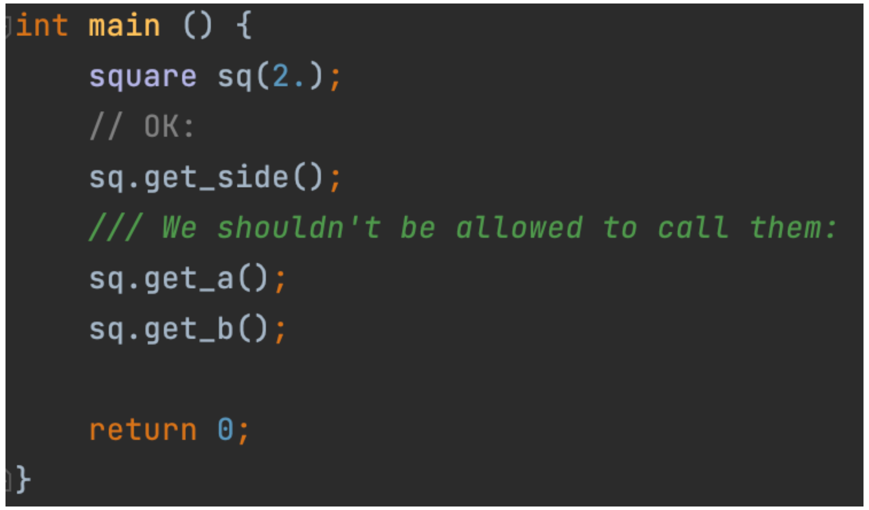


```cpp
struct base {
    base (int a, int b): _a(a), _b(b) {}

private:
    int _a;
    int _b;
};

struct derived : base {
    derived (int a, int b, int c): base(a, b), _c(c) {}

private:
    int _c;
};
```

Inheritance modifiers, motivation

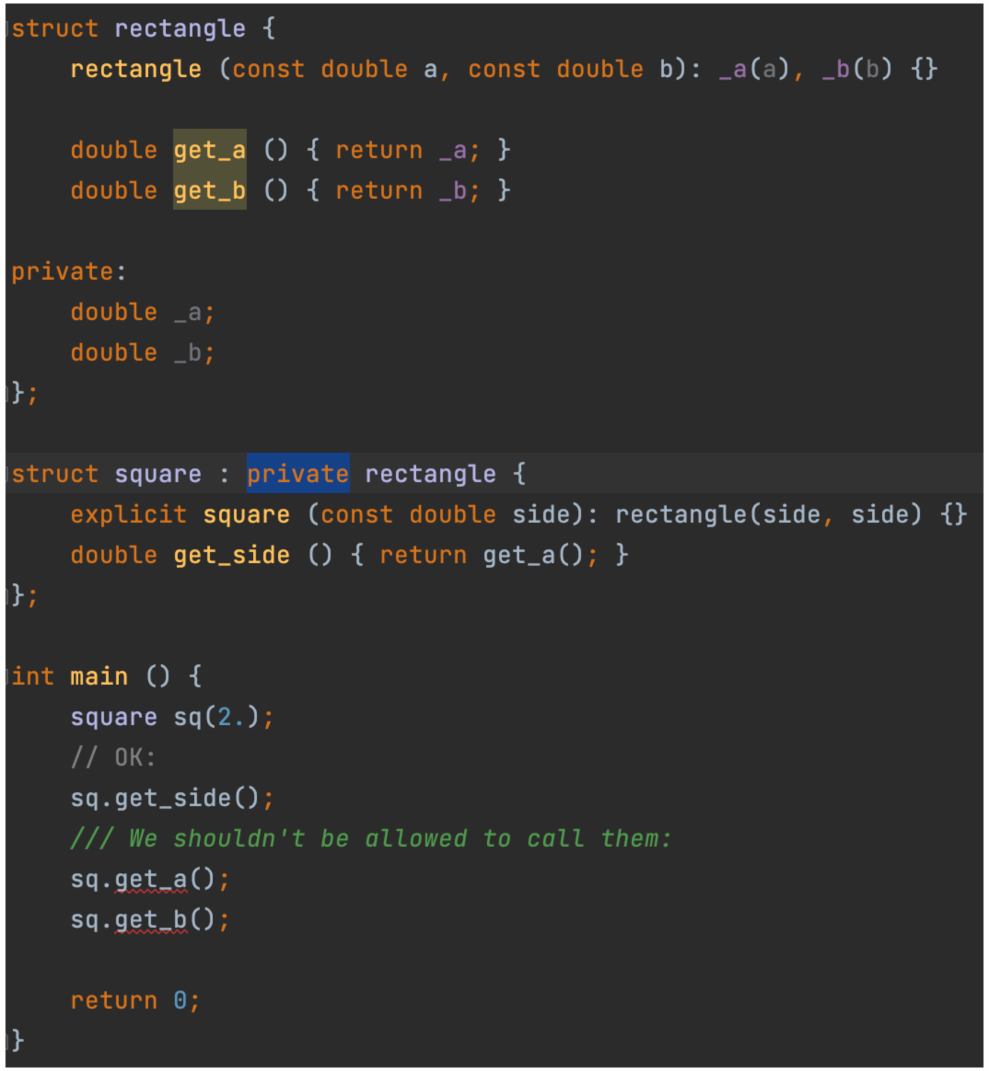


```cpp
struct rectangle {
    rectangle (const double a, const double b): _a(a), _b(b) {}

    double get_a () { return _a; }
    double get_b () { return _b; }

private:
    double _a;
    double _b;
};

struct square : rectangle {
    explicit square (const double side): rectangle(side, side) {}
    double get_side () { return get_a(); }
};
```



```cpp
int main () {
    square sq(2.);
    // OK:
    sq.get_side();
    /// We shouldn't be allowed to call them:
    sq.get_a();
    sq.get_b();

    return 0;
}
```

Inheritance modifiers, motivation



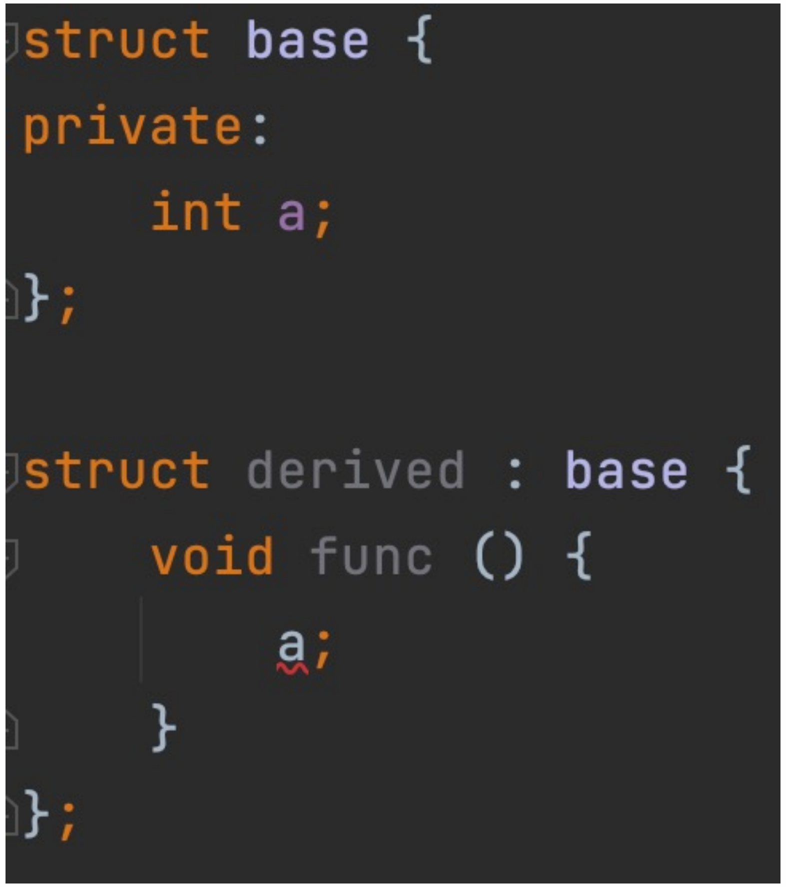
```
struct rectangle {
    rectangle (const double a, const double b): _a(a), _b(b) {}

    double get_a () { return _a; }
    double get_b () { return _b; }

private:
    double _a;
    double _b;
};

struct square : private rectangle {
    explicit square (const double side): rectangle(side, side) {}
    double get_side () { return get_a(); }
};

int main () {
    square sq(2.);
    // OK:
    sq.get_side();
    /// We shouldn't be allowed to call them:
    sq.get_a();
    sq.get_b();

    return 0;
}
```

Private inheritance allows you to hide all inherited fields and methods from the user.

In addition to private inheritance, there are public and protected inheritance.

We'll talk about them a little later. Well, now let’s talk about the third access modifier in C++.

Protected keyword, motivation



```
struct base {
private:
    int a;
};

struct derived : base {
    void func () {
        a;
    }
};
```

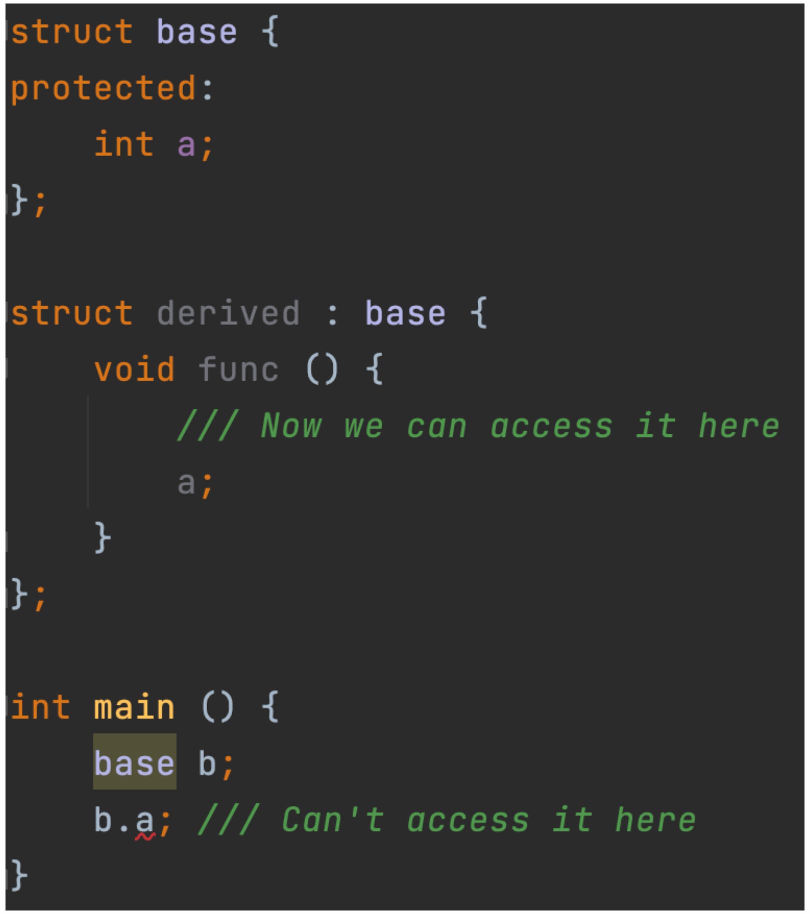
Imagine the situation: we have a base class that has a private member that we want to use somewhere inside the heir. This is where the problem arises: **private members and methods are not inherited under any circumstances**. Thus, we can’t refer to it in the derived classes.

A simple solution would be to make the field public, but in this case the user could access this field from outside (and we want to prevent this!)

This is where the protected access modifier comes handy. The thing is that protected fields can be inherited, but are **not** available to the user.

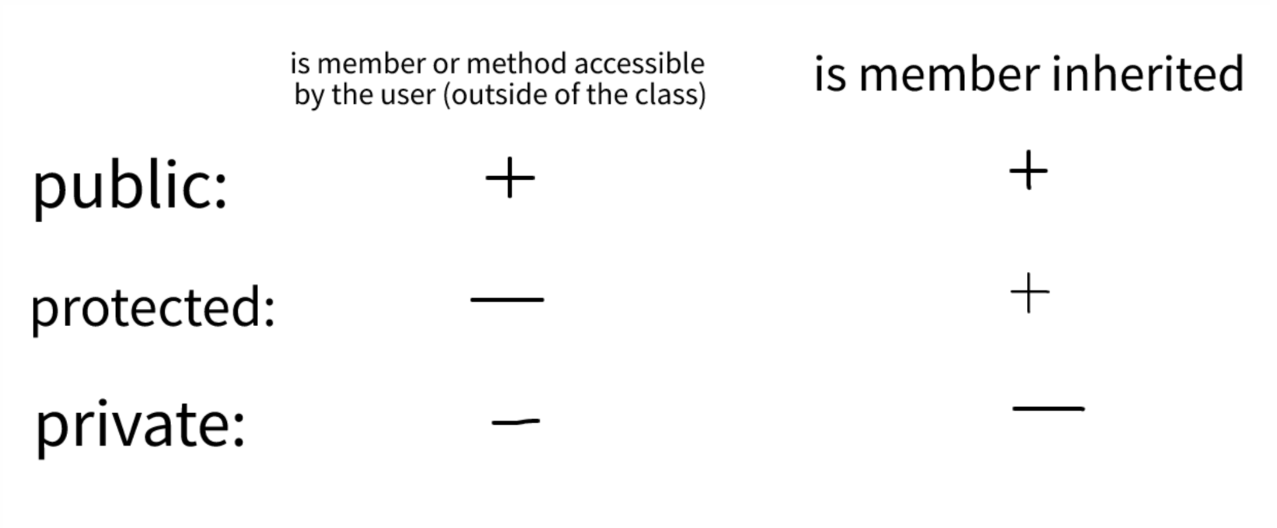
Protected keyword

Full solution:



```
struct base {
protected:
    int a;
};

struct derived : base {
    void func () {
        /// Now we can access it here
        a;
    }
};

int main () {
    base b;
    b.a; /// Can't access it here
}
```

Also, here I provide a table that shows the differences between the three access modifiers.

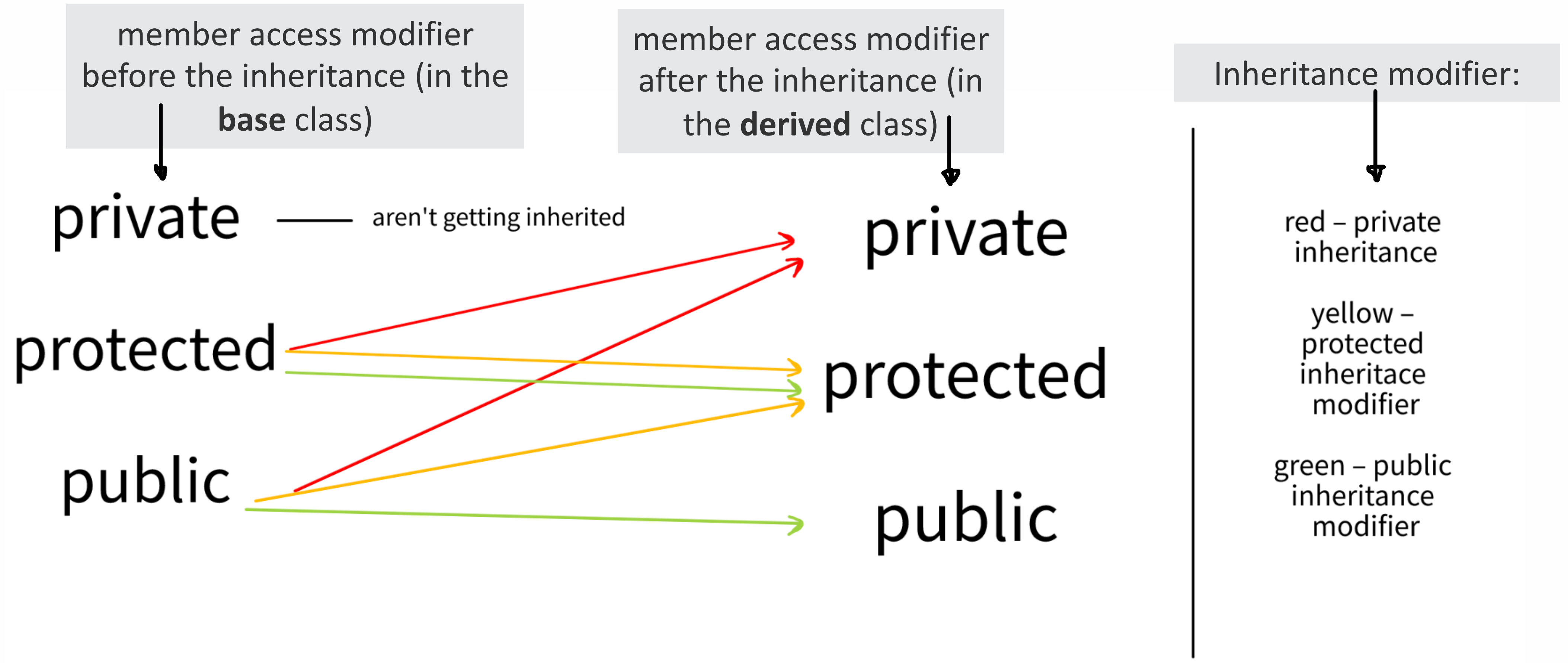


	is member or method accessible by the user (outside of the class)	is member inherited
public:	+	+
protected:	—	+
private:	—	—

Now let's get back to inheritance modifiers.

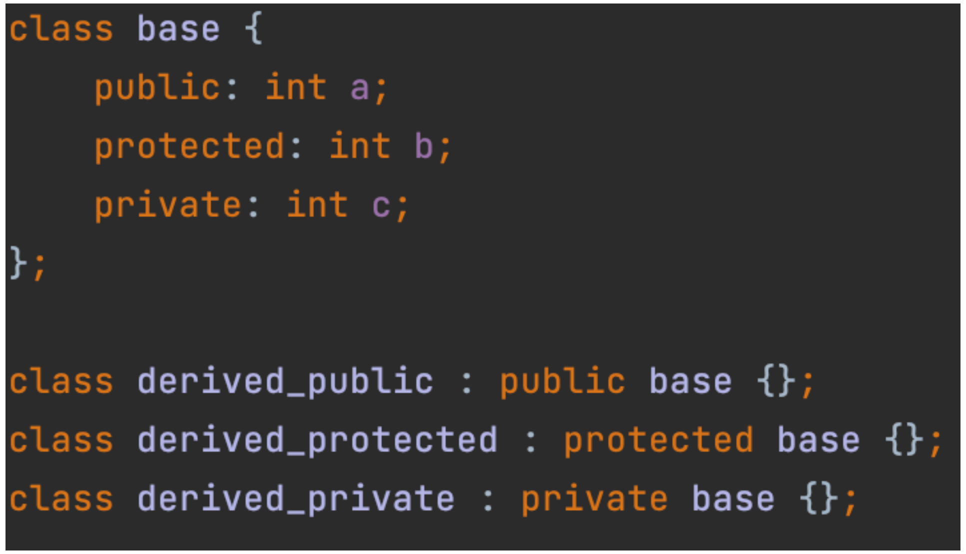
Inheritance modifiers

Inheritance modifiers allow you to change the access modifiers of fields and methods when inheriting (this explains the rectangle example, namely why adding a **private** access modifier solved our problem)


member access modifier before the inheritance (in the base class)
member access modifier after the inheritance (in the derived class)
Inheritance modifier:
private
aren't getting inherited
protected
public
private
protected
public
red – private inheritance
yellow – protected inheritace modifier
green – public inheritance modifier

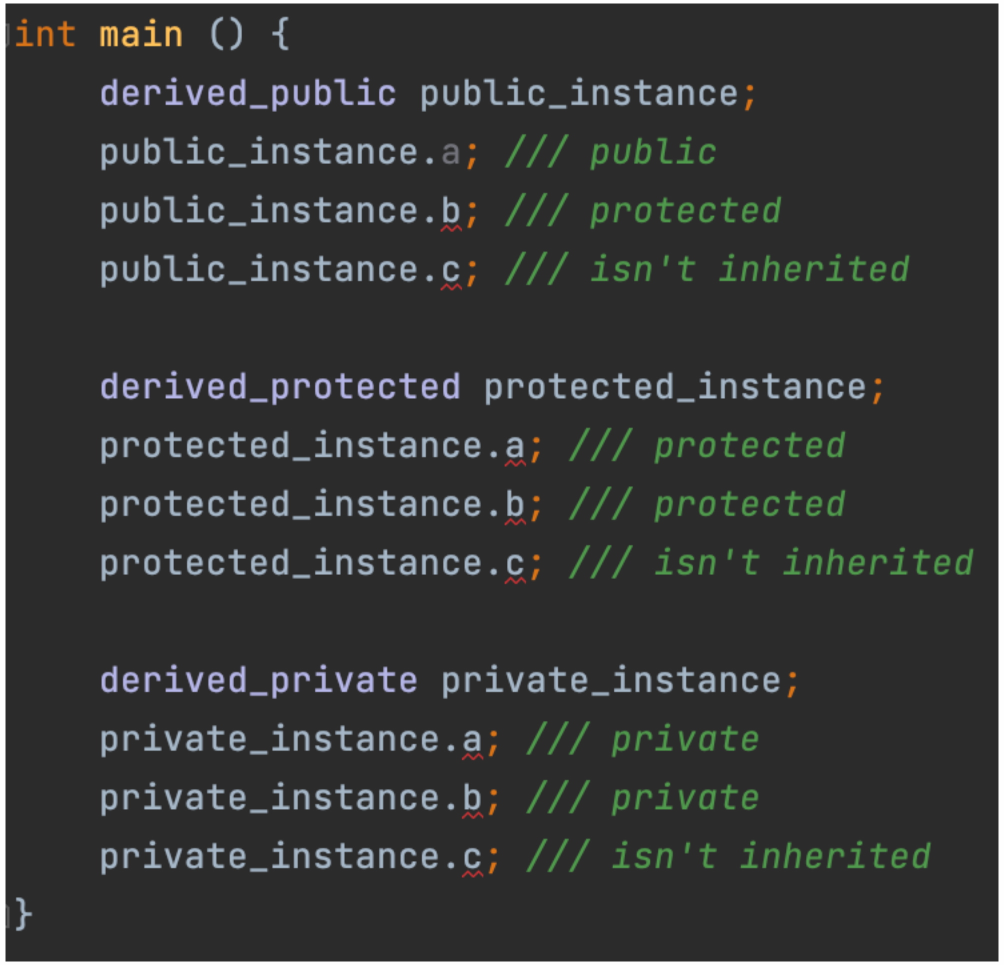
If we do not specify any modifier, then by default for the class, inheritance will be **private**, and for struct - **public**.

Inheritance modifiers, example



```cpp
class base {
    public: int a;
    protected: int b;
    private: int c;
};

class derived_public : public base {};
class derived_protected : protected base {};
class derived_private : private base {};
```

In this example, I have demonstrated all possible combinations of access modifiers and inheritance modifiers (previous table)



```cpp
int main () {
    derived_public public_instance;
    public_instance.a; /// public
    public_instance.b; /// protected
    public_instance.c; /// isn't inherited

    derived_protected protected_instance;
    protected_instance.a; /// protected
    protected_instance.b; /// protected
    protected_instance.c; /// isn't inherited

    derived_private private_instance;
    private_instance.a; /// private
    private_instance.b; /// private
    private_instance.c; /// isn't inherited
}
```

Lecture 15

Inheritance, pt.1

Konstantin Leladze

OOP in C++

DIHT MIPT 2021