

Наследование I



Наследование

Наследование - свойство, позволяющее создавать новый тип данных на основе уже существующих с полным или частичным заимствованием функционала.

Наследование является одним из основных понятий, на которых зиждется ООП.



Пример и синтаксис наследования

Допустим пишем простую компьютерную игру с некоторым набором персонажей, которые могут сражаться друг с другом и, возможно, выполнять другие действия:

```
struct Archer {  
    int strength;    // сила  
    int hp;          // здоровье  
    int xp;          // опыт  
  
    void Shoot();    // выстрелить  
    // ...  
};  
  
struct Soldier {  
    int strength;    // сила  
    int hp;          // здоровье  
    int xp;          // опыт  
  
    void Melee();    // ближний бой  
    // ...  
};
```

Пример и синтаксис наследования

Ясно, что с увеличением числа персонажей и их возможностей код может разрастаться очень сильно.

```
class Hero;  
  
class Archer;  
class Melee;  
class Cavalry;  
class Imposter;
```

Таким образом, возникает две ключевые проблемы:

1. Дублирование кода.
2. Отсутствие ясной структуры и иерархии классов.

Пример и синтаксис наследования

Для решения проблем дублирования кода и указания связи между классами можно воспользоваться следующим синтаксисом:

```
struct Hero {  
    int strength;    // сила  
    int hp;          // здоровье  
    int xp;          // опыт  
  
    void Heal();     // лечиться  
    // ... (другие общие методы и поля)  
};  
  
struct Archer : public Hero {    // лучник унаследован от класса "персонаж"  
    void Shoot();  
    // ... (другие специфичные методы и поля)  
};
```

Hero - базовый класс, Archer - производный класс (класс-наследник).

Пример и синтаксис наследования

Теперь все поля и методы `Hero` также являются полями и методами класса

`Archer` :

```
Hero hero;  
Archer archer;  
  
hero.hp = 90;    // hp - поле hero  
archer.hp = 80;  // у archer тоже есть такое поле  
  
hero.Heal();     // аналогично  
archer.Heal();  
  
archer.Shoot();  // Archer - более специфичный класс  
// hero.Shoot();  CE
```

Модификаторы доступа и наследование

public и private

public и private модификаторы работают как и раньше - public открывает доступ всем, private закрывает доступ для всех, даже для наследников

```
struct Hero {
public:
    void Heal() { UpdateHp(20); }
private:
    void UpdateHp(int delta) { hp += delta; }
};
////////////////////////////////////
struct Archer : public Hero {
    void Rest() {
        Heal();           // Ok (public)
        // UpdateHp(10);  // CE (private)
    }
};
////////////////////////////////////
Archer archer;
archer.Heal();           // Ok (public)
// archer.UpdateHp(10);  // CE (private)
```


public и private

`public` и `private` могут задавать и *режим наследования*, то есть задавать уровень доступа к полям и методам базового класса.

public наследование - все знают о том, что класс унаследован от базового, и внешний код имеет полный доступ к открытым полям и методам базового класса.

private наследование - никто не должен знать о наследовании, доступ к открытым полям и методам базового класса имеет только наследник.

```
struct Hero {  
    // ...  
};  
  
struct Archer : public Hero { // или struct Archer : private Hero  
    // ...  
};
```

public и private

```
struct A {  
    public: int x;  
    private: void f();  
};  
struct B : public A {  
    void h() {  
        x = 0;    // ???  
        f();      // ???  
    }  
};  
struct C : private A {  
    void h() {  
        x = 0;    // ???  
        f();      // ???  
    }  
};
```

```
A a;  
a.x = 11;    // ???  
a.f();      // ???
```

```
B b;  
b.x = 11;    // ???  
b.f();      // ???
```

```
C c;  
c.x = 11;    // ???  
c.f();      // ???
```

public и private

```
struct A {  
    public: int x;  
    private: void f();  
};  
struct B : public A {  
    void h() {  
        x = 0;    // Ok  
        f();      // CE  
    }  
};  
struct C : private A {  
    void h() {  
        x = 0;    // Ok  
        f();      // CE  
    }  
};
```

```
A a;  
a.x = 11;    // Ok  
a.f();      // CE
```

```
B b;  
b.x = 11;    // Ok  
b.f();      // CE
```

```
C c;  
c.x = 11;    // CE  
c.f();      // CE
```

Модификатор доступа `protected`

Существует третий модификатор доступа - `protected`.

Он работает так же как и `private`, но доступ дополнительно получают наследники класса:

```
struct A {  
    protected:  
        int x;  
        void f();  
};  
  
struct B : public A {  
    void h() { x = 0; f(); } // ok  
};  
  
A a;  
// a.x = 0; a.f(); // CE  
B b;  
b.h(); // b.x = 0; b.f(); // CE
```

Модификатор доступа `protected`

`protected` можно использовать для изменения режима наследования.

protected наследование - никто не имеет доступ к базовому классу, кроме наследников

```
struct A {  
    int x;  
    void f();  
};  
  
struct B : protected A {  
};  
  
struct C : public B {  
    void h() { x = 0; f(); } // ok  
};  
  
A a; a.x = 0; a.f();  
B b; // b.x = 0; b.f() // CE
```

Модификатор доступа **protected**

```
struct A {  
    public: int x;  
    private: void f();  
    protected: void g();  
};  
  
struct B : protected A {  
    void h() {  
        x = 0;    // ??  
        f();      // ??  
        g();      // ??  
    }  
};  
  
struct C : public B {};
```

```
A a;  
a.g();    // ??  
  
B b;  
b.x = 11; // ??  
b.f();    // ??  
b.g();    // ??  
b.h();    // ??  
  
C c;  
c.x = 11; // ??  
c.f();    // ??  
c.g();    // ??  
c.h();    // ??
```

Модификатор доступа **protected**

```
struct A {  
    public: int x;  
    private: void f();  
    protected: void g();  
};  
  
struct B : protected A {  
    void h() {  
        x = 0;    // Ok  
        f();      // CE  
        g();      // Ok  
    }  
};  
  
struct C : public B {};
```

```
A a;  
a.g();    // CE  
  
B b;  
b.x = 11; // CE  
b.f();    // CE  
b.g();    // CE  
b.h();    // Ok  
  
C c;  
c.x = 11; // CE  
c.f();    // CE  
c.g();    // CE  
c.h();    // Ok
```

TL;DR

- `public` режим наследования позволяет обращаться к полям и методам базового класса напрямую (то есть пользоваться наследником в точности как базовым классом). Реализует семантику "является".

```
struct A { /* ... */ };  
struct B : public A { /* ... */ }; // B является A
```

- `private` и `protected` режимы запрещают внешнему коду (кроме друзей) каким-либо образом использовать знание о том, что что-то от чего-то унаследовано. `protected` дополнительно разрешает доступ для наследников класса (для остальных работает как `private`). Реализует семантику "содержит".

```
struct A { /* ... */ };  
struct B : protected A { /* ... */ }; // B содержит A
```


Замечание о `private` и `protected` наследовании

`private` и `protected` наследование почти всегда можно заменить на композицию (введение поля нужного типа в класс):

```
struct A { /* ... */ };

struct B : private A { /* ... */ };

struct C {
private:
    A a;
    // ...
};
```

Классы `C` и `B` практически эквивалентны (с точки зрения внешнего кода эквивалентны, так как он не использует факт наследования от `A` или наличия поля `a`).

Empty Base Optimization (EBO)

Размер в байтах любого, даже пустого, объекта в C++ обязан быть > 0 .

Однако при наследовании от пустого класса размер наследника не увеличивается

```
struct A {}; // sizeof(A) == 1

struct B : private A { // sizeof(B) == sizeof(int)
    int x;
};

struct C { // sizeof(C) > sizeof(int)
    int x;
private:
    A a;
};
```

Это бывает полезно, если класс реализован с помощью другого класса, у которого нет нестатических полей. В этом случае наследование лучше композиции.

Еще одно отличие `class` от `struct`

Знаем: в классах модификатор доступа по умолчанию - `private`, в структурах - `public`

С наследованием аналогично: классы по умолчанию наследуют приватно, а структуры - публично.

```
class A { /* ... */ };  
  
struct S : A { /* ... */ };  
// <=>  
struct S : public A { /* ... */ };  
  
class C : A { /* ... */ };  
// <=>  
class C : private A { /* ... */ };
```

На этом список отличий заканчивается.

Порядок вызова конструкторов и деструкторов при наследовании

Пример

```
class StackMax : public Stack {
    int* max_buffer_ = nullptr;

public:
    StackMax() = default;
    StackMax(const StackMax& other) { max_buffer = /* ... */ }
    // ...
};

class StackMin : public Stack {
    int* min_buffer_ = nullptr;

public:
    StackMin() = default;
    StackMin(const StackMin& other) = default;
    // ...
}
```

Как в StackMax/StackMin будет проинициализирована часть относящаяся к Stack?

Пример

Перед входом в тело конструктора все поля и *базовые классы* должны быть проинициализированы.

```
class StackMax : public Stack {
public:
    StackMax() = default;    // вызывается Stack()
    StackMax(const StackMax& other) { /* ... */ } // 1: вызывается Stack()
    // ...
};

class StackMin : public Stack {
public:
    StackMin() = default;    // вызывается Stack()
    StackMin(const StackMin& other) = default; // 2: вызывается Stack(const Stack&)
    // ...
}
```

В случае 1 в Stack ничего не копируется; в случае 2 Stack скопируется, но часть, относящаяся к StackMin проинициализируется по умолчанию.

Вызов конструктора базового класса

```
class StackMax : public Stack {
    int max_buffer_ = nullptr;

public:
    StackMax() = default;    // Stack()

    StackMax(const StackMax& other)
        : Stack(other)
        , max_buffer_(new int[kCapacity]) {

        if (size_ > 0) {
            max_buffer_[0] = buffer_[0];
        }
        for (size_t i = 1; i < size; ++i) {
            max_buffer_[i] = std::max(buffer_[i], max_buffer_[i - 1]);
        }
    }
    // ...
};
```

Вызов конструктора базового класса

- Класс инициализируется в следующем порядке: сначала инициализируются базовые классы, затем поля класса-наследника в порядке объявления (список инициализации не может повлиять на порядок!)

```
struct A {  
    int x;  
    int y;  
  
    A(int);  
    A(int, int);  
};  
  
struct B : public A {  
    int z = 0;  
  
    // B() {} - СЕ, у А нет конструктора по умолчанию  
    B(int x) : A(x) {}  
    B(int x, int y) : A(x, y) {} // сначала A(x, y), затем z = 0  
    B(int x, int y, int z) : A(x, y), z(z) {}  
}
```


Вызов конструктора базового класса

- Чтобы не дублировать конструкторы базового класса (как в прошлом примере) можно воспользоваться конструкцией `using A::A` (теперь B можно создавать так же как и A)

```
struct A {  
    int x;  
    int y;  
  
    A(int);  
    A(int, int);  
};  
  
struct B : public A {  
    int z = 0;  
  
    using A::A;  
    B(int x, int y, int z) : A(x, y), z(z) {}  
}
```

Вызов конструктора базового класса

- Нельзя проинициализировать отдельное поле базового класса, только всю базовую часть целиком

```
struct A {  
    int x;  
    int y;  
  
    A(int);  
    A(int, int);  
};  
  
struct B : public A {  
    int z = 0;  
  
    B(int x, int y, int z) : x(x), y(y), z(z) {} // CE  
}
```

```
error: class 'B' does not have any field named 'x'  
error: class 'B' does not have any field named 'y'
```

Порядок вызова деструкторов

При уничтожении объекта сначала выполняется тело деструктора, затем деструктурируются поля класса-наследника в порядке обратном объявлению и в самом конце уничтожаются части базового класса

```
class A {  
    // ...  
};  
  
class B : public A {  
    Stack s1;  
    Stack s2;  
  
    // ...  
    ~B() {  
        // (1) ...  
    } // <- (2) s2.~Stack(), (3) s1.~Stack(), (4) ~A()  
};
```

Срезка (Slicing)

Срезка

Так как публичное наследование реализует отношение "является", можно инициализировать объекты предка объектами потомков, а также присваивать предкам потомков. Но не наоборот!

```
struct A {  
    int x;  
    void f();  
};  
  
struct B : public A { // B является A (но A не является B)  
    int y;  
    void g();  
};  
  
B b;  
A a = B();  
a = b;  
// b = a; // CE
```

Срезка

Данная возможность называется "срезкой" (при присваивании используются только часть класса, относящаяся к базовому классу).

```
struct A { /* ... */ };  
  
struct B : public A { /* ... */ };  
  
A a = B(); // от правой части остается только кусок, содержащий A
```

Наоборот нельзя, так как в **B** могут быть элементы, которых нет в **A**:

```
// B b = A(); // СЕ
```

Стоит отметить, что если в базовом классе есть копирующий/перемещающий конструктор/присваивание, то при срезке будут использованы именно они.

Срезка

При приватном и защищенном наследовании внешний код не имеет права использовать факт наличия связи между классами, поэтому срезка запрещена.

```
struct A { /* ... */ };  
struct B : protected A { /* ... */ };  
  
// A a = B(); // CE
```

Исключение 1: внутри класса-наследника срезку делать можно (он "знает" про то, что он от чего-то унаследован)

```
struct B : protected A {  
    void f() { A a = B(); } // ok  
};
```

Исключение 2: друзья наследника тоже имеют право знать (и использовать) факт наследования

Настройка поведения при срезке

Рассмотрим пример

```
struct A {  
    std::string name;  
    A() { name = "A"; }  
    // ...  
};  
  
struct B : public A {  
    B() { name = "B"; }  
    // ...  
};
```

```
A a; a.name;           // "A"  
B b; b.name;           // "B"  
A aa = b; aa.name;     // "B" - но, возможно, хочется "A"
```


Настройка поведения при срезке

Для задания конкретного поведения нужно определить конструктор **A** от **B**:

```
struct B; // forward declaration

struct A {
    std::string name = "A";
    A(const B& other);
    // ...
};

struct B : public A {
    // ...
};

A::A(const B& other) : name("A") { /* ... */ }
```

```
A aa = b; aa.name; // "A"
```

Настройка поведения при срезке

Аналогичным образом можно запретить срезку:

```
struct B; // forward declaration

struct A {
    std::string name = "A";
    A(const B& other) = delete;
    // ...
};

struct B : public A {
    // ...
};
```

```
A aa = b; // CE
```

Затенение методов базового класса (shadowing)

Пример 1

```
struct A {  
    void f(int) {  
        std::cout << "A::f(int)\n";  
    }  
};  
  
struct B : public A {  
    void f(int) {  
        std::cout << "B::f(int)\n";  
    }  
};
```

```
B b;  
b.f(0); // ok: B::f(int)
```

А что, если хочется вызвать метод `f` из `A`?

ЯВНЫЙ ВЫЗОВ МЕТОДА БАЗОВОГО КЛАССА

```
struct A {  
    void f(int) {  
        std::cout << "A::f(int)\n";  
    }  
};  
  
struct B : public A {  
    void f(int) {  
        std::cout << "B::f(int)\n";  
    }  
};
```

```
B b;  
b.f(0);      // Ok: B::f(int)  
b.A::f(0);   // Ok: A::f(int)
```

Пример 2

```
struct A {  
    void f() {  
        std::cout << "A::f()\n";  
    }  
};  
  
struct B : public A {  
    void f(int) {  
        std::cout << "B::f(int)\n";  
    }  
};
```

```
B b;  
b.f(0); // ok: B::f(int)  
// b.f(); // CE
```

error: no matching function for call to 'B::f()'

Shadowing

Если в производном классе присутствует метод с тем же именем, что и метод в базовом классе, то это имя затеняет базовые методы с тем же именем (shadowing).

Логично ли это?

```
struct A {  
    void f();  
};  
  
struct B : public A {  
    void f(int);  
};
```

```
B b;  
b.f(0);           // Ok: B::f(int)  
// b.f()         // CE  
b.A::f();         // Ok: A::f()
```

Shadowing

Есть несколько решений этой "проблемы" (не факт, что проблема):

1. Смириться.
2. Использовать полное имя метода (`b.A::f()`).
3. Ввести имена базового класса в область производного с помощью `using`:

```
struct A {  
    void f();  
    void f(double);  
};  
struct B : public A {  
    using A::f; // A::f(), A::f(double) (частично нельзя!)  
    void f(int);  
};
```

```
b.f(0); // Ok: B::f(int)  
b.f(); // Ok: A::f()  
b.f(0.0); // Ok: A::f(double)
```


**Работа с производным классом через
указатель или ссылку на базовый**

Указатель и ссылка на базовый класс

```
class B : public A { /* ... */ };
```

Если класс **B** публично унаследован от класса **A**, можно (подобно срезке) присваивать указатель/ссылку на **B** указателю/ссылке на **A**:

```
B b;  
A* b_ptr = &b;    // ok  
A& b_ref = b;     // ok
```

Но не наоборот:

```
A a;  
B* a_ptr = &a;    // CE  
B& a_ref = a;     // CE
```

В чем отличие от срезки?

Указатель или ссылка ссылаются на *тот же* объект и работают с его данными, в отличие от срезки, где мы работаем с копией:

```
struct A {  
    int x = 0;  
    void f() { ++x; }  
};  
  
struct B : public A {  
    int y = 0;  
};
```

```
B b;           // b.x == 0; b.y == 0  
A a = b;       // a.x == 0  
a.f();         // a.x == 1, b.x == 0  
// a.y;       // CE  
A& b_ref = b;  
b_ref.f();     // b.x = 1;  
// b_ref.y;   // CE: все-таки компилятор видит, что type(b_ref) == A&
```

Какие методы вызываются?

Компилятор выбирает версию метода, основываясь на *статическом типе* указателя/ссылки, не обращая внимания на реальный тип объекта, на который ссылаются:

```
struct A {  
    void f() { std::cout << "A::f()\n"; }  
};  
  
struct B() : public A {  
    void f() {std::cout << "B::f()\n"; }  
};
```

```
B b;  
A* b_ptr = &b;  
b_ptr->f(); // A::f()
```

Тизер

```
B b;  
A* b_ptr = &b;  
b_ptr->f(); // A::f()
```

Это поведение может смущать. Придется жить с этим чувством до следующей недели. А там встанем с колен и исправим этот пример.

Резюме

- Наследование позволяет задать связь между классами и способствует повторному использованию кода.
- `public` наследование реализует отношение "является", `protected` и `private` - "содержит" (часто можно заменить на композицию).
- Имена в производном классе могут затенять соответствующие имена в базовом классе.
- Объекты базового класса могут создаваться из объектов производного с помощью механизма срезки.
- Ссылки и указатели на производный класс свободно приводятся к ссылкам и указателям на открытый базовый класс.