Язык C++. Лекция 7

Мещерин Илья

29 октября 2018 г.

5.2б) Явный вызов методов предка

```
1  struct Base{
2      void f();
3  };
4
5  struct Derived : public Base{
6      void f(int x);
7  };
8
9  int main(){
10     Derived d;
11     d.Base::f();
12 }
```

Явно пишем откуда взять нужную функцию. Если наследование приватное, то все равно ошибка компиляции.

```
struct Derived : public Base{
    using Base::f;
    void f(int x);
};
```

Второй способ решения вопроса. В таком случае вызывать функцию *f()* можно как обычно.

5.2в) Пример

Оффтоп - если при наследовании не написать тип наследования, то по умолчанию *private* (у *struct* он *public*)

```
1  struct Base{
2  public:
3      void f();
4  };
5
6  struct Derived : public Base{
7  private:
8      void f(int x);
9  };
10
11 int main(){
12     Derived d;
13     d.f();
14 }
```

Ошибка компиляции. **Проверка доступа происходит после поиска имен.**
5.2г) Пример

```
1  struct Granny{
2      int a;
3  };
4
5  struct Mom : private Granny{
6      int b;
7  };
8
9  struct Son : public Mom{
10     int c;
11     void f(Granny &g){
12
13     }
14 }
```

Ошибка компиляции. Название типа *Granny* запрещено внутри класса *Son*.

```
struct Son : public Mom{
    int c;
    void f(:: Granny &g){

    }
}
```

А так уже писать можно, т.к. в глобальной области видимости запретов нет.

```
struct Granny{
    int a;
    friend struct Son;
};
```

Все равно ошибка компиляции, т.к. ошибка в выражении *s.Granny::a* возникает после точки, а не после двух двоеточий.

```
struct Mom : private Granny{
    int b;
    friend struct Son;
};
```

Такой способ уже решает проблему.
5.2д) Пример

```
1  struct Granny{
2  protected:
3      int a;
4  };
5
6  struct Mom : public Granny{
7      int b;
8      friend void f();
9  };
```

```
10
11 void f(){
12     Mom m;
13     m.a;
14 }
```

Все OK, но вообще отношение дружбы не транзитивно.

5.3) Порядок вызова конструкторов и деструкторов при наследовании

```
1  struct Granny{
2      int a;
3      Granny(int a): a(a) {}
4  };
5
6  struct Mom : public Granny{
7      int b;
8      Mom(int b): b(b) {}
9  };
10
11 int main(){
12
13 }
```

Ошибка компиляции, т.к. нечем проинициализировать класс *Granny*. Если в классе *Granny* был бы конструктор по умолчанию, то все сработало бы.

```
struct Mom : public Granny{
    int b;
    Mom(int b): Granny(...), b(b) {}
};
```

Чтобы решить проблему, можно явно вызвать конструктор для класса *Granny*.

```
struct Mom : public Granny{
    int b;
    Mom(int a, int b): a(a), b(b) {}
};
```

Ошибка компиляции по двум причинам: таким способом можно инициализировать поля только своего класса, а также таким способом мы так и не проинициализровали класс *Granny*.

```
1  struct Granny{
2      int a;
3      Granny(int a): a(a) {}
4  };
5
6  struct Mom : public Granny{
7      int b;
8      Mom(): Granny(0), b(0) {}
9  };
10
```

```
11 struct Son : public Mom{
12     int c;
13     Son(int c): Granny(0), c(c) {}
14 };
15
16 int main(){
17
18 }
```

Ошибка компиляции. A constructor of a child will always call the constructor of it's parent.

При вызове конструктора сначала конструируются предки, а только потом сам класс. При вызове деструктора сначала разрушается сам класс, а только потом его предки.

5.4) Множественное наследование

5.4a) Примеры

```
 1 struct Parallelogram{
 2     int x;
 3 };
 4
 5 struct Rectangle : public Parallelogram{
 6     int y;
 7 };
 8
 9 struct Rhombus : public Parallelogram{
10     int z;
11 };
12
13 struct Square : public Rectangle, public Rhombus{
14     int t;
15 };
16
17 int main(){
18     Square s;
19     cout << sizeof(s) / sizeof(int);
20 }
```

На выходе получим число 5 (*P, Re, P, Rh, S*).

```
Square s;
s.Parallelogram::x;
```

Ошибка компиляции из-за неоднозначности.

```
s.Rectangle::x;
s.Rhombus::x;
```

Решение проблемы.

Это все называется проблемой **ромбовидного наследования (diamond problem)**.

```
struct Rectangle : private Parallelogram{
    int y;
}
```

```
int main(){
    Square s;
    s.x;
}
```

Даже если одно из наследований приватное, то все равно ошибки компиляции, т.к. сначала происходит поиск имен, а только потом проверка доступа.

5.4б) Примеры

```
1  struct Granny{
2      int x;
3  };
4
5  struct Mom : public Granny{
6      int y;
7  };
8
9  struct Son : public Mom, public Granny{
10     int z;
11 };
12
13 int main(){
14     Son s;
15 }
```

В таком примере возникает проблема в том, что к одному $s.Mom::x$ мы можем обратиться, а к другому x никак не получится обратиться.

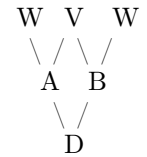
5.5) Виртуальное наследование

```
1  struct Parallelogram{
2      int x;
3  };
4
5  struct Rectangle : virtual public Parallelogram{
6      int y;
7  };
8
9  struct Rhombus : virtual public Parallelogram{
10     int z;
11 };
12
13 struct Square : public Rectangle, public Rhombus{
14     int t;
15 };
16
17 int main(){
18     Square s;
19     s.x;
20 }
```

Компилятор обязан создать только одну виртуальную версию класса *Parallelogram*.

Если одновременно наследовать один и тот же класс виртуально и невиртуально, то на все виртуальные наследования создастся одна версия класса, а на все невиртуальные наследования каждый раз будет создаваться отдельная версия класса.

```
1  struct W{
2      int y;
3  };
4
5  struct V{
6      int x;
7  };
8
9  struct A : public W, virtual public V{
10     int x, y;
11 };
12
13 struct B : public W, virtual public V{
14
15 };
16
17 struct D : public A, public B{
18
19 };
20
21 int main(){
22     D d;
23     d.x;
24     //d.y;
25 }
```



Обратиться к $d.y$ нельзя т.к. существует два пути $D{->}A.y$ и $D{->}B{->}W.y$. Для поля x также есть два пути, но в данном случае путь $D{->}A.x$ замещает путь $D{->}B{->}V.x$ и неоднозначности нет.

5.6) Приведение типов между наследниками

```
Base b;
Derived d;
b = d;
//d = b;
```

В первом случае произойдет *срезка при копировании*. А проинициализировать потомка с помощью предка нельзя, т.к. не хватает информации.

```
Base &b = d;
Base *b = d;
```

Правильно писать так, чтобы не терять информацию и при необходимости обратиться к полям класса *Derived* с помощью каста.
Но если наследование было приватным, то так писать нельзя (разве что только с помощью *reinterpret_cast<>*).