Язык C++

Мещерин Илья

Лекция 8

//Оффтоп от меня - переопределить можно виртуальный метод, а перегрузить любой метод.

5.6) Приведение типов между наследниками

```
1  struct Base{
2      int a;
3      void f();
4  };
5
6  struct Derived : public Base{
7      int b;
8      void f();
9  };
10
11 int main(){
12     Base b;
13     Derived d;
14     Base &g = d;
15     g.f();
16     //g.b;
17 }
```

Вызовется *Base::f()*, а к полю *Derived::b* вообще обратиться нельзя.

```
Base *g = &d;
g->f();
//g->b;
```

Аналогично.

```
Base &g = d;
static_cast<Derived&>(g).f();
```

Вызовется *Derived::f()*.

```
Base b;
Derived d;
b = d;
static_cast<Derived&>(b).f();
```

Бред.

```
static_cast<Base&>(d).f();
```

Вызовется *Base::f()*.
Если наследование приватное, то это все делать нельзя, т.к. *static_cast<>* это проверяет.

```
reinterpret_cast<Base&>(d).f();
```

А вот *reinterpret_cast<>* этого не проверяет.

```
1  struct A{
2  };
3
4  struct B : A{
5  };
6
7  struct C : A{
8  };
9
10 struct D : B, C{
11 };
12
13 int main(){
14     D d;
15     static_cast<A&>(d); //incorrect
16 }
```

При ромбовидном наследовании кастовать D к A нельзя, т.к. возникает неоднозначность.

Оператор *dynamic_cast* является частью механизма динамической идентификации типа данных, который позволяет выполнять приведение типа данных. Проверка корректности приведения типов производится во время выполнения программы. Оператор *dynamic_cast* может быть применён к указателям или ссылкам. В случае если осуществляется преобразование указателя, который содержит адрес объекта-родителя, к указателю типа объекта-потомка, то в результате преобразования будет получен нулевой указатель. При работе со ссылками при невозможности преобразования типа будет сгенерировано исключение *std::bad_cast*.

```
1  struct A{
2      int x;
3      virtual void foo();
4  };
5
6  struct B : A{
7      void g(){
8          cout << "YEAH!\n";
9      }
10 };
11
12 void f(A &a){
13     try{
14         B &b = dynamic_cast<B&>(a);
15         b.g();
16     } catch(const std::bad_cast &e){
17         std::cout << e.what();
18     }
19 }
20
```

```
21 void s(A *a){
22     B *b = dynamic_cast<B*>(a);
23     if (b) b->g();
24     else std::cout << "nullptr\n";
25 }
26
27 int main(){
28     A a;                                    stdout:
29     B b;                                        YEAH!
30     f(b);                                       std::bad_cast
31     f(a);                                       YEAH!
32     s(&a);                                      nullptr
33     s(&b);
34 }
```

В классе *A* функция *foo()* необходима, т.к. механизм динамической идентификации типа данных доступен только для полиморфных классов (т.е. классов, содержащих хотя бы одну виртуальную функцию-член).

5.7) Виртуальные функции (одно из проявлений полиморфизма)

```
1  struct Base{
2      int a;
3      void f();
4  };
5
6  struct Derived : public Base{
7      int b;
8      void f();
9  };
10
11 int main(){
12     Derived d;
13     Base &b = d;
14     b.f();
15 }
```

Вызовется *Base::f()*, но логичнее было бы использовать *Derived::f()* как более частный случай.

```
struct Base{
    int a;
    virtual void f();
};
```

Если дописать слово *virtual*, то в примере выше будет вызываться *Derived::f()*.

5.8) Виртуальный деструктор

```
D *d = new D();
B *b = d;
delete b;
```

В данном случае будет утечка памяти. Чтобы решить эту проблему нужно объявить виртуальный деструктор.

```
virtual ~B() {}
```

5.9) Абстрактные классы

```
1  struct Shape{
2      virtual double area() const = 0;
3  };
4
5  struct Square : public Shape{
6      double a, b;
7      double area() const {
8          return a * b;
9      }
10 };
11
12 int main(){
13     Square sq;
14     Shape &sh = sq;
15     sh.area();
16 }
```

Метод *area()* называется чисто виртуальным (pure virtual). В данном случае класс *Shape* будет абстрактным классом и запрещено создавать объекты этого класса, но можно создавать ссылки и указатели на этот класс, а также наследоваться от него. Если при наследовании не определить чисто виртуальные методы, то новый класс также становится абстрактным.

```
double Shape::area() const {}
```

Можно написать реализацию чисто виртуального метода за областью класса, но класс останется абстрактным. Если при определении метода *area()* в наследнике забыть написать *const*, то переопределения не произойдет, а только создастся новая функция и новый класс станет абстрактным.

5.10) Слова *override* и *final*

```
struct Base{
    virtual void f();
};

struct Derived : Base{
    void f() override;
};
```

Слово *override* гарантирует, что функция переопределяет виртуальную функцию базового класса. Если это не так, то получится ошибка компиляции.

```
1  struct Base{
2      virtual void foo();
3  };
4
5  struct A : Base{
6      void foo() final; // A::foo is overridden and it is the final override
7      void bar() final; // Error: non-virtual function cannot be overridden or be final
8  };
9
10 struct B final : A{ // struct B is final
11     void foo() override; // Error: foo cannot be overridden as it's final in A
12 };
13
14 struct C : B{ // Error: B is final
15 };
```

Слово *final* позволяет запретить наследоваться от класса или переопределять виртуальный метод класса.

5.11) Оператор *typeid*

```
#include <typeinfo>
int x;
typeid(x).name();
```

Функция *typeid()* возвращает объект класса *std::type_ info*. С помощью этой функции можно проверять типы объектов на равенство.

```
1  class Base{
2  public:
3      virtual void vvfunc() {}
4  };
5
6  class Derived : public Base {};
7
8  using namespace std;
9  int main(){
10     Derived* pd = new Derived;
11     Base* pb = pd;
12     cout << typeid( pb ).name() << endl; //prints "class Base *"
13     cout << typeid( *pb ).name() << endl; //prints "class Derived"
14     cout << typeid( pd ).name() << endl; //prints "class Derived *"
15     cout << typeid( *pd ).name() << endl; //prints "class Derived"
16     delete pd;
17 }
```

Кроме того с помощью этой функции при наследовании можно правильно определить, что лежит под указателем.

```
1  class Base {
2  public:
3      void vfunc() {}
4  };
```

```
5
6  cout << typeid( pb ).name() << endl; //prints "class Base *"
7  cout << typeid( *pb ).name() << endl; //prints "class Base"
8  cout << typeid( pd ).name() << endl; //prints "class Derived *"
9  cout << typeid( *pd ).name() << endl; //prints "class Derived"
```

Если класс не полиморфный, то реально определить, что лежит под указателем не удается.

Компилятор должен поддерживать *runtime type information*, поэтому все это работает долго.

5.12) Таблицы виртуальных функций

Размер полиморфного класса это не просто суммарный размер всех полей, а еще и указатель на место в памяти, где компилятор хранит информацию о том, какая версия функции предназначена для какого объекта.