Семинар 3 (наследование)

МГТУ им. Н.Э. Баумана

March 11, 2016

Объяснение без синтаксиса наследования

Person, Student, Prefect

```
1  struct Person {
2    string name;
3    uint32_t age;
4  };
5  struct Student {
6    Person person; // student's record as a Person
7    uint32_t grade;
8  };
9  struct Prefect {
10   Student student; // prefect's record as a Student
11   vector<Student> group; // students prefect is responsible for
12 };
```

Использование

```
1 void printName(const Person& p) {cout << p.name;}
2 void printName(const Student& s) {cout << s.person.name;}
3 void printName(const Prefect& p) {cout << p.student.person.name;}
```

Синтаксис наследования

Person, Student, Prefect

```
1  struct Person {
2    string name;
3    uint32_t age;
4  };
5  struct Student : public Person {
6    uint32_t grade;
7  };
8  class Prefect : public Student {
9    vector<Student> group; // students prefect is responsible for
10 };
11 void printName(const Person& p) {cout << p.name << '\n';}
```

Синтаксис наследования

Person, Student, Prefect

```
1  struct Person {
2    string name;
3    uint32_t age;
4  };
5  struct Student : public Person {
6    uint32_t grade;
7  };
8  class Prefect : public Student {
9    vector<Student> group; // students prefect is responsible for
10 };
11 void printName(const Person& p) {cout << p.name << '\n';}
```

Person	Student	Prefect
name age	name age	name age
	grade	grade
		group

Prefect -> Student -> Person

Использование

```
1  void f(const Person& p, const Student& s, const Prefect& pr) {
2    printName(p); printName(s); printName(pr);
3  }
```

Приведение типов

Указатели и ссылки

```
1 Student s("Name", 19, 2);
2 Person& p = s; // OK
3 Person* pp = &s; // OK
4 cout << p.grade; // ERROR: Person has no field 'grade'
5 cout << static_cast<Student*>(pp)->grade; // ok
```

По значению: срезка

```
1 Student s("Name", 19, 2);
2 Person p = s; // OK: person fields copied
3 cout << p.grade; // ERROR: Person has no field 'grade'
```

Типы наследования

public, protected, private – доступ к членам

```
1  class A
2  {
3  public: int x;
4  protected: int y;
5  private: int z;
6  };
7  class B : public A {
8      // x is public
9      // y is protected
10     // z is not accessible from B
11 };
12 class C : protected A {
13     // x is protected
14     // y is protected
15     // z is not accessible from C
16 };
17 class D : private A { // 'private' is default for classes
18     // x is private
19     // y is private
20     // z is not accessible from D
21 };
```

Наследование методов

Правила доступа к методам такие же, как к полям.

```
class A
{
public: void e() {cout << "A::e()";}
protected: void f() {cout << "A::f()";}
private: void g() {cout << "A::g()";}
};
class B : public A {
public: void h() { f(); } // OK, f is protected
// protected: void k() { g(); } // ERROR, B has no access to A::g()
};
```

```
B b;
b.e(); // OK
// b.f(); // ERROR: f is protected
```

Конструкторы и деструкторы

Зовется конструктор по умолчанию

```
class A {
public:
  A() { cout << "A ctor; "; }
  A(int i) { cout << "A(int i) ctor; "; }
};
class B : public A {
public:
  B() { cout << "B ctor; "; }
};
```

```
B b; // A ctor; B ctor;
```

Конструкторы и деструкторы

Зовется конструктор по умолчанию

```
class A {
public:
  A() { cout << "A ctor; "; }
  A(int i) { cout << "A(int i) ctor; "; }
};
class B : public A {
public:
  B() { cout << "B ctor; "; }
};
```

```
B b; // A ctor; B ctor;
```

Можно вызывать другой конструктор в списке инициализаторов

```
class C : public A {
public:
  C() : A(3) {
    cout << "C ctor; ";
  }
  C(int j) { cout << "C(int j) ctor; "; }
};
```

```
C c; // A(int i) ctor; C ctor;
C c1(5); // A() ctor; C(int j) ctor;
```

Деструктор базового класса вызовется после деструктора наследуемого.

Конструкторы и деструкторы

Еще один пример

```
class A {
  int m_i;
public:
  A() : m_i(0) { cout << "A ctor; "; }
  A(int i) : m_i(i) { cout << "A(int i) ctor; "; }
  ~A() { cout << "A(" << m_i << ") dtor; "; }
};

class C : public A {
  int m_j;
public:
  C() : A(3), m_j(0) {
    cout << "C ctor; ";
  }
  C(int j) : m_j(j) { cout << "C(int j) ctor; "; }
  ~C() { cout << "C(" << m_j << ") dtor; "; }
};
```

```
{
  C c; // ???
  C c1(5); // ???
} // ???
```

Конструкторы и деструкторы

Еще один пример

```
class A {
  int m_i;
public:
  A() : m_i(0) { cout << "A ctor; "; }
  A(int i) : m_i(i) { cout << "A(int i) ctor; "; }
  ~A() { cout << "A(" << m_i << ") dtor; "; }
};

class C : public A {
  int m_j;
public:
  C() : A(3), m_j(0) {
    cout << "C ctor; ";
  }
  C(int j) : m_j(j) { cout << "C(int j) ctor; "; }
  ~C() { cout << "C(" << m_j << ") dtor; "; }
};
```

```
{
  C c; // A(int i) ctor; C ctor;
  C c1(5); // A() ctor; C(int j) ctor;
} // C(5) dtor; A(0) dtor; C(0) dtor; A(3) dtor;
```

Переопределение методов при наследовании

Определение метода в наследуемом перекрывает определение в базовом

```
class A
{
public: void e() {cout << "A::e()";}
};
class B : public A {
public:
  void e() {cout << "B::e()";}
  void g() {
    cout << "B::g()";
    A::e(); // outputs A::e()
  }
};
```

```
B b;
b.e(); // B::e()
b.g(); // B::g() A::e()
```

Переопределение методов при наследовании

Определение метода в наследуемом перекрывает определение в базовом

```
class A
{
public: void e() {cout << "A::e()";}
};
class B : public A {
public:
  void e() {cout << "B::e()";}
  void g() {
    cout << "B::g()";
    A::e(); // outputs A::e()
  }
};
```

```
B b;
b.e(); // B::e()
b.g(); // B::g() A::e()
```

Вызов метода по ссылке на базовый класс

```
B b;
A& a = b;

b.e(); // B::e()
a.e(); // A::e()
```

Виртуальные методы

Модификатор virtual

```
class A
{
public:
  virtual void e() { cout << "A::e()"; }
  void f() { cout << "A::f()"; }
};
class B : public A {
public:
  void e() { cout << "B::e()"; }
  void f() { cout << "B::f()"; }
};
```

```
B b;
A* a = &b;

a->e(); // B::e()
a->f(); // A::f()
b.f(); // B::f()
a = new A(); a->e(); // A::e();
```

Динамический полиморфизм (времени выполнения)

Выбор вызываемого метода происходит в момент выполнения и зависит от типа объекта, на который указывает указатель (ссылка).

Обеспечение динамического полиморфизма

Для каждого класса с виртуальными методами

- Создается таблица vftable с адресами виртуальных методов
- Объект класса содержит указатель на vftable
- Вызов метода происходит по адресу из vftable

```
1 struct A {
2   int i;
3   virtual void e() { cout << "A::e()"; }
4   virtual void f() { cout << "A::f()"; }
5 };
6 struct B : public A {
7   int j;
8   void e() { cout << "B::e()"; }
9 };
```

struct A:

void* A::vfptr	i

struct B:

void* B::vfptr	i	j

A::vfptr:

void e();	0x45e4
void f();	0x4700

B::vfptr:

void e();	0x4910
void f();	0x4700

```
1 A* a = new B();
2 a->e(); // <=> a->vfptr->e();
```

Виртуальный деструктор

Невиртуальный деструктор: что разрушать?

```
class A
{
public:
  ~A() { cout << "A dtor"; }
};
class B : public A {
public:
  ~B() { cout << "B dtor"; }
};
```

```
B* b = new B;
A* a = b;
```

```
delete a; // ???
```

Виртуальный деструктор

Невиртуальный деструктор: что разрушать?

```
class A
{
public:
  ~A() { cout << "A dtor"; }
};
class B : public A {
public:
  ~B() { cout << "B dtor"; }
};
```

```
B* b = new B;
A* a = b;
```

```
delete a; // A dtor
```

Виртуальный деструктор

Решение: виртуальный деструктор

```
class A
{
public:
  virtual ~A() { cout << "A dtor; "; }
};
class B : public A {
public:
  ~B() { cout << "B dtor; "; }
};
```

```
B* b = new B;
A* a = b;

delete a; // B dtor; A dtor;
```

Виртуальный деструктор

Решение: виртуальный деструктор

```
class A
{
public:
  virtual ~A() { cout << "A dtor; "; }
};
class B : public A {
public:
  ~B() { cout << "B dtor; "; }
};
```

```
B* b = new B;
A* a = b;

delete a; // B dtor; A dtor;
```

Почему так произошло?

Поведение аналогично вызову любого виртуального метода.

Herb Sutter

Конструктор базового класса должен быть либо public и virtual, либо protected и non-virtual.

Чисто виртуальные методы (pure virtual)

Pure virtual method

```
class Shape
{
public:
  virtual void rotate(double angle) = 0;
};
class Line2d : public Shape {
  double m_x, m_y;
  double m_angle;
public:
  void rotate(double angle) { m_angle = (m_angle + angle) % (2*3.14); }
};
class Line3d : public Shape {};
```

```
Shape* s = new Line2d;
s->rotate(); // Line2d::rotate()
```

Чисто виртуальные методы (pure virtual)

Pure virtual method

```
1  class Shape
2  {
3  public:
4    virtual void rotate(double angle) = 0;
5  };
6  class Line2d : public Shape {
7    double m_x, m_y;
8    double m_angle;
9  public:
10   void rotate(double angle) { m_angle = (m_angle + angle) % (2*3.14); }
11 };
12 class Line3d : public Shape {};
```

```
1  Shape* s = new Line2d;
2  s->rotate(); // Line2d::rotate()
```

Абстрактный класс

```
1  // Shape* s = new Shape; // ERROR: Shape is abstract
2  // Shape* s = new Line3d; // ERROR: Line3d is abstract
```

Класс, имеющий хотя бы один чисто виртуальный метод, является абстрактным. Объекты абстрактного класса создать нельзя.
Абстрактный класс описывает интерфейс взаимодействия с объектами

Типы наследования

public, protected, private – информация о наследовании

```
1  class A {
2  public:
3    int a;
4  };
5
6  class B : public A {};
7
8  class C : protected A {
9  public:
10   void f() {};
11 };
12
13 class D : private A {
14 public:
15   void g() {};
16 };
```

Использование

```
1  B b1; C c1; D d1;
2  A& a1 = b1; // OK, everyone knows of inheritance
3  A& a2 = c1; // ERROR, it's known to C and derived of C
4  A& a3 = d1; // ERROR, private inheritance
```

Типы наследования

public, protected, private – информация о наследовании

```
1  class A {
2  public: void f() { cout << "A"; }
3  };
4  class C : protected A {
5  public: void f() { cout << "C"; A::f(); }
6  };
7  class D : private A {
8  public: void f() { cout << "D"; A::f(); }
9  };
10 class E : public D {
11 public:
12   void f() {
13     cout << "E";
14     A::f(); // ERROR: E does not know of A
15   }
16 };
17 class F : public C {
18 public:
19   void f() {
20     cout << "F";
21     A::f(); // OK
22   }
23 };
```