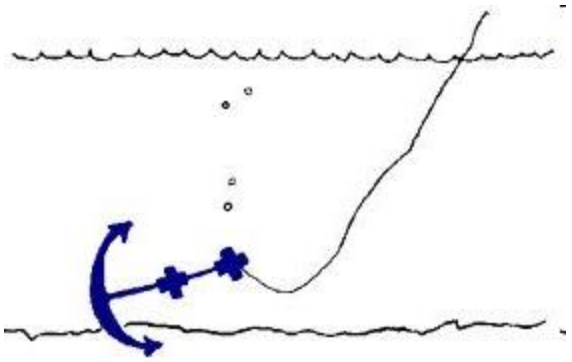
Карпов В.Э.

Объектно-ориентированное программирование

C++. Лекция 6



ОБЪЕКТЫ

Структура в C++ является объектом, объединяя в описании данные и процедуры, реализуя механизм наследования и т.п.

ИНКАПСУЛЯЦИЯ – НАСЛЕДОВАНИЕ – ПОЛИМОРФИЗМ

Инкапсуляция. Возможность определения методов в теле структуры.

```
struct A
{       int x, y;
        void f(void);
};
void A:: f(void)
{   printf("%d %d",x,y); }

void f(void)
{   printf("I am 'f'"); }

void main(void)
{
  A a;
  a.x = 1;
  a.y = 10;
  a.f();
}
```

Вопрос о размере объекта

sizeof(A) = ?

Может,

sizeof(A) = sizeof(x)+ sizeof(y)+ sizeof(f) =
sizeof(int)+ sizeof(int)+ sizeof(void*) =
4+4+4?

На самом деле:

sizeof(A) = sizeof(x)+ sizeof(y) = 4+4 = 8

«Поле» f не хранится в объекте

Обращение a.f() – это лишь вызов функции.

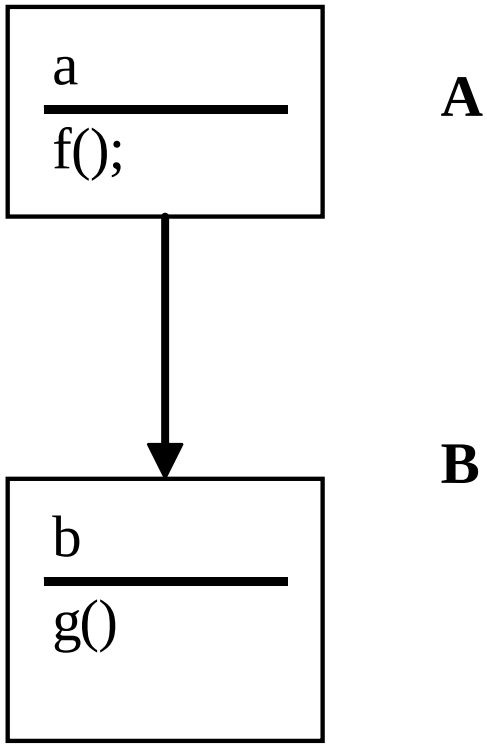
Наследование

```
struct A
{
  int a;
  void f(void);
};
struct B:A
{
  int b;
  void g(void);
};

void A::f(void)
{   printf(“a=%d”,a); }

void B::g(void)
{
  f();
  printf(“a=%d, b=%b”, a, b);
}

void main(void)
{
  B x;
  x.f();
}
```



Методы, как функции-подстановки

```
struct A
{
  int a;
  void g(void);
  void f(void) {  printf("a=%d",a); }
};
void A::g(void) {  … }
```

Правила определения inline-функции – те же

Обращение к суперклассу

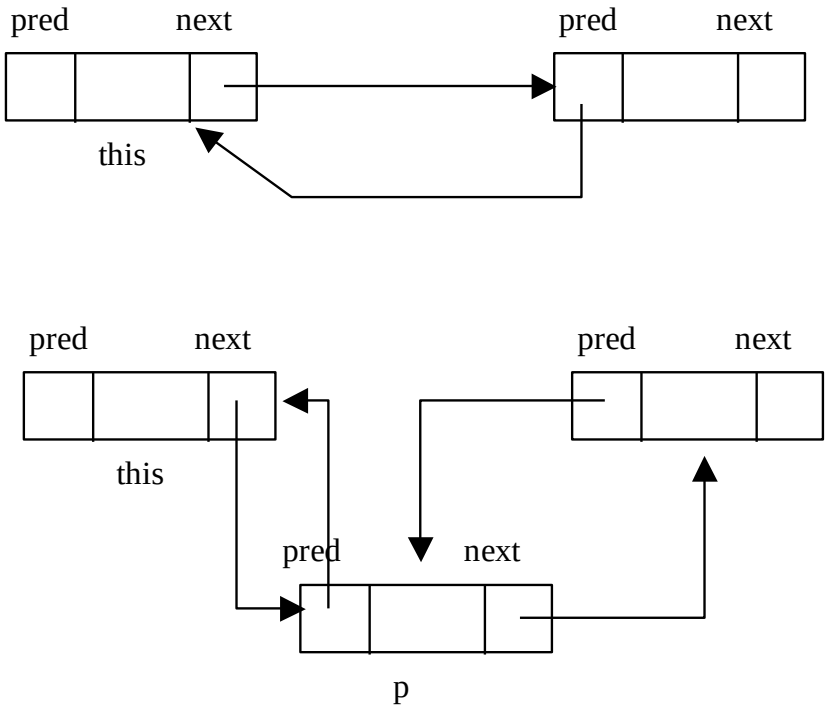
В C++ нет псевдопеременной *super*, как в Смолтоке. Вместо этого пользуются явным указанием имени родительского класса.

```
struct A
{
  int a;
  void f(void) {  printf(“a=%d”,a); }
};
struct B:A
{
  int a;
  int b;
  void g(void)
  {
    f();
    A::f();
    printf(“A::a=%d, a=%d, b=%b”, A::a, a, b);
  }
}
```

Указатель на объект-адресат. Переменная this

Аналогом псевдопеременной *self* является указатель на объект-адресат – псевдопеременная ***this***.

```
struct Tlink
{    Tlink *pred; // предыдущий элемент
     Tlink *next; // следующий элемент
     void insert(Tlink*);
}
void Tlink:: insert(Tlink *p)
{    p-> next = next;
     p->pred = this;
     next ->pred = p;
     next = p;
}
 . . . . . .
Tlink *lh, *a, *b;
lh->insert(a);
lh->insert(b);
```



КЛАССЫ

public и private – части

- При использовании декларации *struct* мы получаем объект, к переменным и методам которого имеется *свободное* обращение. Ограничить доступ к переменным и методам объекта можно введением специальных меток *public* и *private*.

```
struct C
{
private:
   int x, y;
public:
   void init(int, int);
};
…
C c;
c.init(10,10);
c.x = 1; // Сообщение об ошибке
a = c.y; // Сообщение об ошибке
```

Декларация *class*

- *struct* и *class* отличаются друг от друга исключительно действующими соглашениями по умолчанию.
- По умолчанию считается, что при определении типа *struct* мы получаем объект, у которого все составляющие **публичны** (определены как *public*).
- Декларация *class* подразумевает, что все составляющие определены по умолчанию как **приватные** (*private*).

Struct и class

struct A
{
  int a;
};
 	
struct B: A
{
  int b;
};
 	
void main(void)
{
  B x;
  x.b = 1;
  x.a = 1;
  x.A::a = 1;
}
	class A
	{
	  int a;
	};
	
	class B: A
	{
	  int b;
	};
	
	void main(void)
	{
	  B x;
	~~ x.b = 1;~~
	~~ x.a = 1;~~
	~~ x.A::a = 1;~~
	}
		class A
		{
		public:
		  int a;
		};
		
		class B: **public** A
		{
		public:
		  int b;
		};
		
		void main(void)
		{
		  B x;
		  x.b = 1;
		  x.a = 1;
		  x.A::a = 1;
		}

При объявлении *struct* все по умолчанию является публичным	При объявлении *class* все по умолчанию является защищенным. Доступ полностью запрещен.	Эквивалент определения через *struct*

Инициализация. Конструкторы

```
class S
{    private:          int x, y;
  public:
     void sput(char *s) { puts(s); }
     S(int newx, int newy)
        { x = newx; y = newy; }
  S(void) { }
};
```

Варианты вызова конструктора:

```
S x;
S x(1,2);
S *ptr;
ptr = new S;
ptr = new S(1,2);
```

Продолжение примера

```
class V
{   private:      int val[2];
    public:
      V(int a = 0) { cout << "\nV-constructor:"<< a << "\n";
                     val[0] = val[1] = a; };
};
.......
V x = 10; // вызов конструктора с параметрами.
          // Аналогично V x(10)
x = 9;    // вызов конструктора с параметром a=9
```

Деструкторы

- Деструктор – это некоторая специальная функция (метод).
- Деструктор выполняется **автоматически** при выходе из области действия объекта.
- Обозначается как имя класса с добавленным символом '~' в начале имени.
- Деструктор не должен иметь аргументов.

```
struct Sampl
{
  int x, y;
  void say() {printf("[%d %d]",x,y);}
  Sampl(int newx, int newy) { x = newx; y = newy; }
  Sampl(void) { }
  ~Sampl(void) { x = 0; y = 0; }
};
void main(void)
{
  Sampl s(10,8);
  Sampl a;
  Sampl b = s;
  Sampl *p;
  s.say();
  a.say(); // будет выведено то же самое
  b.say();
  p = new Sampl(123,456);
  p->say();
  delete p; //Здесь будет выполнен деструктор
} // И здесь тоже будет выполнен деструктор
```

Деструкторы

Деструктор выполняется

- при вызове *delete*
- при выходе за пределы области действия переменной.

```
void f()
{   Sampl *p, z;
    p = new Sampl();
    ...
    delete p;
} //Здесь будет выполнен деструктор для объекта z
```

Примеры

```
class C
{   private:
        float re,im;
      public:
        C(float r, float i) {re = r; im = i;}
       C(float a) {re = im = a;}
       C() {re = im = 0;}
       ~C() {}
};
void main(void)
{   C c;
    C c1(10,20);
    C c2 = 2.1;
    C c3[2] = {1.0,2.0};
   C *c5[2] = {new C(1), new C(9,10)};
}
```

Массивы объектов

- Для их инициализации необходим конструктор без аргументов ().

  *C *clist = new C[10];*

  // Объявление массива из

  // 10 элементов
- При удалении требуется указать размерность уничтожаемого массива.

  delete [10] clist;
- Если использовать просто

  delete clist,

  то функция-деструктор будет вызвана только для первого элемента.

```
class TA1
{ int a, b; };
class TA2
{
  int a, b;
public:
  TA2(_a) {a=_a;}
  TA2() {a=b=0;}
};
class TA3
{
  int a, b;
public:
  TA3(_a) {a=_a;}
};
void main(void)
{
  TA1 *n1 = new TA1[10];
  TA2 *n2 = new TA2[10];
  TA3 *n3 = new TA3[10]; // Здесь компилятор выдаст ошибку
}
```

Последовательный вызов конструкторов

class A { A(int);… };
class B:A { B(int);… };
class C:B { C(int, int);… };

C::C(int a, int b): A(a), B(b) { …};

- Запись

B::B(int n):(n)

эквивалентна

B::B(int n):A(n)

Но лучше этим не злоупотреблять (хуже читается).

Обсуждение предыдущего

```
#include <iostream>
using namespace std;
class A
{
public:
  A(int n) { cout << "A: " << n << "\n"; }
};
class B:A
{
public:
  B(int n) { cout << "B: " << n << "\n"; }
};
class C:B
{
public:
  C(int a, int b): A(a+1), B(b+1) { cout << "C: a="
<< a << " b=" << b << "\n"; };
};
main(void)
{
  C c(1,2);
}
```

```
1.cpp: In constructor ‘B::B(int)’:
1.cpp:16:12: error: no matching function for call to
‘A::A()’
   B(int n) { cout << "B: " << n << "\n"; }
            ^
1.cpp:16:12: note: candidates are:
1.cpp:10:3: note: A::A(int)
   A(int n) { cout << "A: " << n << "\n"; }
   ^
1.cpp:10:3: note:   candidate expects 1 argument, 0
provided
1.cpp:7:7: note: A::A(const A&)
 class A
       ^
1.cpp:7:7: note:   candidate expects 1 argument, 0
provided
1.cpp: In constructor ‘C::C(int, int)’:
1.cpp:8:1: error: ‘class A A::A’ is inaccessible
 {
 ^
1.cpp:23:20: error: within this context
   C(int a, int b): A(a+1), B(b+1) { cout << "C: a=" << a
<< " b=" << b << "\n"; };
                    ^
1.cpp:23:20: error: type ‘A’ is not a direct base of ‘C’
```

Обсуждение предыдущего - 2

```
class A
{
public:
  A(int n) { cout << "A: " << n << "\n"; }
  A() { cout << "A: noarg constructor ()\n"; }
};

class B:A
{
public:
  B(int n) { cout << "B: " << n << "\n"; }
};
class C:B
{
public:
  C(int a, int b): A(a+1), B(b+1) { cout << "C: a="
<< a << " b=" << b << "\n"; };
};
main(void)
{
  C c(1,2);
}
```

```
1.cpp: In constructor ‘C::C(int, int)’:
1.cpp:61:1: error: ‘class A A::A’ is inaccessible
 {
 ^
1.cpp:77:20: error: within this context
   C(int a, int b): A(a+1), B(b+1) { cout << "C: a=" << a
<< " b=" << b << "\n"; };
                    ^
1.cpp:77:20: error: type ‘A’ is not a direct base of ‘C’
```

Обсуждение предыдущего - 3

```
class A
{
public:
  A(int n) { cout << "A: " << n << "\n"; }
  A() { cout << "A: noarg constructor ()\n"; }
};
class B:A
{
public:
  B(int n) { cout << "B: " << n << "\n"; }
};
class C: A, B
{
public:
  C(int a, int b): A(a+1), B(b+1) { cout << "C: a="
<< a << " b=" << b << "\n"; };
};

main(void)
{
  C c(1,2);
}
```

```
1.cpp:114:7: warning: direct base ‘A’
inaccessible in ‘C’ due to ambiguity [enabled by
default]
 class C:A,B
       ^
```

OUTPUT:

A: 2

A: noarg constructor ()

B: 3

C: a=1 b=2

Обсуждение предыдущего - 4

```
class A
{
public:
  A(int n) { cout << "A: " << n << "\n"; }
  A() { cout << "A: noarg constructor ()\n"; }
};
class B: virtual A
{
public:
  B(int n) { cout << "B: " << n << "\n"; }
};
class C: virtual A, B
{
public:
  C(int a, int b): A(a+1), B(b+1) { cout << "C: a=" << a <<
" b=" << b << "\n"; };
};
main(void)
{
  C c(1,2);
}
```

OUTPUT:

A: 2

B: 3

C: a=1 b=2

Статические переменные

- Основное назначение - уменьшить необходимость в **глобальных** переменных. Статический член не связан с отдельным экземпляром (и не копируется в каждом объекте). Он существует вне зависимости от того, существуют ли объекты этого класса
- Обратиться к нему можно явно: **имя_класса::статич_член = значение**).
- Это чем–то напоминает *переменные класса* языка Смолток.

```
class Sampl
{
public:
  static int common;
  static const int C;
  void say() { printf("\ncommon
     %d",common); }
  Sampl(void) {}
};
// инициализация в ГЛОБАЛЬНОЙ
// области
int Sampl::common = 10;
const int Sampl::C = 20;

void main(void)
{    Sampl a,b;
      a.say();
     Sampl::common = -1;
     b.say();
}
```

Постоянные члены класса

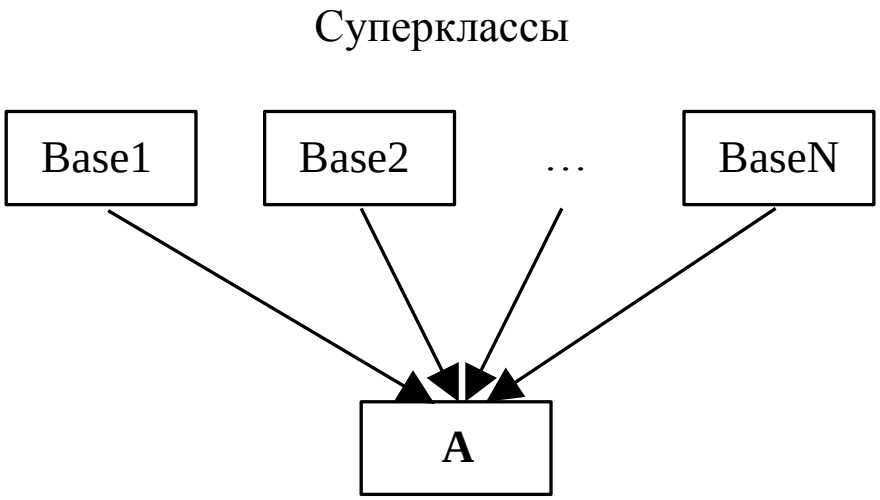
```
class C
{
  public:
    const double rp;
    const double ip;
    C(double a, double b): rp(a), ip(b) {…}
}
```

Иного способа проинициализировать константу нет. Поэтому приходится пользоваться записью, похожей на вызов последовательности конструкторов.

Наследование

- Наследование в языке C++ *множественное.* Это означает, что класс может иметь более чем одного предка (в отличие от языков Smalltalk и ObjectPascal).
- В C++ суперкласс обычно называется базовым классом. Объявление суперкласса выглядит так:

  class A:[public | *private] Base1, [public* | *private] Base2..*



Наследование

class A: public B
{ .... }
class A: private B или class A: protected B
{ .... }

По умолчанию наследуемый класс имеет тип ***private***.
Т.е. объявление
class A: B
эквивалентно
class A: private B

Защищенные (protected) члены класса

- Защищенные (*protected*) члены класса - то же, что и private-члены. Разница лишь в том, что *к ним возможен доступ из производных классов*.
- Внешне же private-члены являются недоступными.

Пример 1

```
class A
{
  public:    void pub() {puts("tA.pub()");}
  private:   void priv() {puts("tA.priv()");}
  protected:              void prot() {puts("tA.prot()");}
};
class B:public A
{
  public:    void pub()   {  puts("B.pub()");   A::pub();   A::prot();   }
  private:   void priv() { puts("B.priv()"); }
  protected:              void prot() { puts("B.prot()"); }
};


…
  A a;
  B b;
  a.pub();
  a.priv();  // ошибка: этот метод недоступен
  a.prot();  // ошибка: этот метод недоступен
  b.pub();
  b.A::pub(); // Если объявить B:A (вместо B:public A), то этот метод станет недоступным
…
```

Пример 2

```
class A
{ public:
    void fpub()  { printf("A.fpub()\n"); }
  protected:
    void fprot() { printf("A.fprot()\n"); }
  private:
    void fpriv() { printf("A.fpriv()\n"); }
};
class B:public A
{ public:
   void fpub()
   { printf("B.fpub()\n"); A::fprot(); }
  protected:
   void fprot() { printf("B.fprot()\n"); }
  private:
   void fpriv() { printf("B.fpriv()\n"); }
};
```

```
class C:B //По умолчанию - C:private B
{
  public:
    void fpub()  {
      printf("C.fpub()\n");
      A::fpub();
      A::fprot();  //если объявить класс
        // class B:A,  то этот вызов будет
        // недоступным
      }
  protected: void fprot() {printf("C.fprot()\n");}
  private: void fpriv() {printf("C.fpriv()\n");}
};
```

```
B b;
C c;
b.fprot(); // Ошибка - нет доступа
b.fpriv(); // Ошибка - нет доступа
b.fpub();
b.A::fpub(); // Если объявить B:A, то этот вызов будет недоступным
c.fpub();
c.A::fpub(); // Ошибка - нет доступа. Надо было бы объявить так: C:public B
```