Лекция 12. Наследование: разное. C++11: разное.

Евгений Линский

Наследование: разное

Сортировка в стиле C

```
struct point_s {
  int x, y;
};

void qsort (void* base, size_t num, size_t size,
            int (*compar)(const void*,const void*)){}

int cmp_point1(const void* v1,const void* v2) {
  const struct point_s *p1 = (const struct point_s*)v1;
  const struct point_s *p2 = (const struct point_s*)v2;
  ...
}

struct point_s points[10];
qsort(points, 10, sizeof(points[0]), cmp_point1);
```

*void** – нет никакой проверки типов со стороны компилятора!

Сортировка в стиле C

Или так:

```
int cmp_point2(const void* v1,const void* v2) {
  const struct point_s *p1 = (const struct point_s*)*v1;
  const struct point_s *p2 = (const struct point_s*)*v2;
  ...
}
struct point_s **points;
//allocation:  points = new ...  and for() { points[i] = new }
qsort(points, N, sizeof(points[0]), cmp_point2);
```

Можно использовать и для структур (классов) и для примитивных типов.

```
int cmp_int(const void* v1,const void* v2) {
  const int *p1 = (const int*)v1;
  const int *p2 = (const int*)v2;
  ...
}
int numbers[10];
qsort(numbers, 10, sizeof(numbers[0]), cmp_int);
```

Сортировка в стиле ООП

ООП:

```
class Comparable{
    //or virtual bool operator<(const Comparable *v) = 0 const;
    virtual int compare(const Comparable* v) = 0 const;
};
void nsort(Comparable** m, size_t size){
  ...
  m[i] =
}

class Point: public Comparable {
private:
  int x, y;
public:
  virtual int compare(const Comparable* v) const {...};
};

Point** points;
//allocation:  points = new ...  and for() { points[i] = new }
nsort(points, N);
```

Сортировка в стиле ООП

А можно ли было сделать так?

```
void nsort(Comparable* m, size_t size){}

Point p[100];
nsort(p, 100);
```

Для примитивных типов надо написать “обертку” (наследник *Comparable*).

```
class Integer : public Comparable {
private:
  int value;
public:
  virtual int compare(const Comparable* v) const { ... };
};
```

Связный список в стиле C: неинтрузивный

```
struct node_s {
  void* user_data;
  struct node_s *next;
};
struct list_s {
  struct node_s *head;
};

void push_back(struct list_s *l, struct node_s *n);
```

```
struct node_s *n = malloc(sizeof(struct node_s));
n->user_data = malloc(sizeof(struct point_s));
push_back(&l, n);
```

- Два вызова malloc!
- *void** – нет никакой проверки типов со стороны компилятора!

Связный список в стиле C: интрузивный

```
struct node_s {
  struct node_s *next;
};
struct list_s {
  struct node_s *head;
}

void push_back(struct list_s *l, struct node_s *n);
```

```
struct point_s {
  int x, y;
  struct node_s node;
};
list_t l;
struct point_s *pn = malloc(sizeof(*pn));
push_back(&l, &pn->node);
```

- Один malloc!
- Но нужен трюк, чтобы, имея указатель на node, добраться до полей x и y (см. лабу про интрузивные списки)

Связный список в стиле ООП

Требуется сделать базовый класс, от которого должны быть отнаследованы все классы в языке C++.

```
class Object {
public:
  // for red-black tree/priority queue/etc
  virtual bool operator<(const Object* o)
    { return this < o; }
  virtual bool operator==(const Object* o)
    { return this == o; }
  // for hash table
  virtual int hash(const Object* o)
    { return (int)this; }
};
```

Связный список в стиле ООП

```
class Node {
private:
  Object *o;
  Node *next;
  ...
};

class List {
  Node *head;
public:
  void push_back(Node *n);
};
```

Связный список в стиле ООП

```
class Point: public Object {
  int x, y;
  virtual bool operator<(const Object* o) {
    Point *p = (Point*)o;
    ...
  }
};

List l;
Node *n = new Node;
n->setData(new Point(3,3));
l.push_back(n);
```

- Два вызова new!
- Необходимо сделать обертки для примитивных типов *class Integer: public Object*.
- Иногда придется использовать множественное наследование *class Developer: public Object, public Person*.
- Vector придется делать так: *Object** array*.

private/protected/public наследование

```
class A {
public:
    int x;
protected:
    int y;
private:
    int z;
};
```

private/protected/public наследование

```
class B : public A {
    // x is public
    // y is protected
    // z is not accessible from B
};

class C : protected A {
    // x is protected
    // y is protected
    // z is not accessible from C
};

class D : private A   { // 'private' is default for classes
    // x is private
    // y is private
    // z is not accessible from D
};
```

private/protected/public наследование

- Обычно при дизайне иерархии классов стараются сохранить между классами отношения из “реального мира”.
- Два типа отношений:
  - “has a”: у машины есть двигатель, у машины есть тормоза.

```
class Car {
  Engine e;
  Brakes b;
};
```

  - “is a”: грузовик это машина, автобус это машина

```
class Track : public Car { };
class Bus : public Car { };
```

private/protected/public наследование

Иногда в реализации надо сделать 'хак': у поля класса вызвать protected метод или перекрыть виртуальную функции и т.п. В этом случае можно применить private/protected наследование.

```
class Engine {
protected:
  void maintenanceCheck() { ... };
...
};

class Car : private Engine {
  void reset() {
    maintenanceCheck();
  }
};
```

Хотим подчеркнуть, что все-таки машина не является двигателем и это просто хак.

Разное

Параметры по умолчанию

Player.h

```
class  Player {
private:
  double life; int damage;
public:
  Player(double life = 100.0, int damage = 10);
  // Player(double life, int damage = 10); ok
  // Player(double life = 100.0, int damage); error
}
```

main.cpp

```
#include "Player.h"
Player p1(2, 3);
Player p2(2);
Player p2();
```

- Почему параметры по умолчанию должны быть указаны в header файле?

Параметры по умолчанию

Player.h

```
class  Player {
private:
  double life; int damage;
public:
  Player(double life = 100.0, int damage = 10);
  // Player(double life, int damage = 10); ok
  // Player(double life = 100.0, int damage); error
}
```

main.cpp

```
#include "Player.h"
Player p1(2, 3);
Player p2(2);
Player p2();
```

- Почему параметры по умолчанию должны быть указаны в header файле?
- Их необходимо знать при компиляции *main.cpp*. Другие cpp файлы в этот момент не используются.

C++11: разное

nullptr

```
libc: #define NULL 0
```

```
void f(int* ptr) { ... }
void f(int n) { ... }

f(NULL); // error:  call of overloaded ‘f(NULL)’ is ambiguous
f(nullptr); // OK
```

Explicitly defaulted and deleted special member functions

Было:

```
class A {
public:
  A(int);
  A() {}; // 1.  автоматически не сгенерируется из-за A(int)
          // 2.  нужен для array = new A [100]
private:
  // сделать приватным --> запретить копирование
  A(const A&);
  A& operator=(const A&);
};
```

Приватные copy constructor и assignment operator:

- Использование: Singletone, scoped_ptr
- Проблемы: все-таки можно копировать в friend классе или в самом классе

Explicitly (явно!) defaulted and deleted special member functions

В C++11 стало явно:

```
class A {
public:
  A(int);
  A() = default;
  A(const A&) = delete;
  A& operator=(const A&) = delete;
};
```

Explicit overrides and final

Было:

```
class Base {
public:
  virtual void f(int);
  virtual int g() const;
  void h(int);
};
```

```
class Derived: public Base {
public:
  void f(int);
  int g();
  void h(int);
};
```

Проблемы:

Explicit overrides and final

Было:

```
class Base {
public:
  virtual void f(int);
  virtual int g() const;
  void h(int);
};
```

```
class Derived: public Base {
public:
  void f(int);
  int g();
  void h(int);
};
```

Проблемы:

- чтобы понять является ли f виртуальной, надо смотреть в базовый класс.
- для функции g случайно сделали перегрузку вместо перекрытия.
- функция h тоже не перекрывается.

Explicit overrides and final

В C++11 — явно “перекрывается или не перекрывается”:

```
class Derived: public Base {
public:
  void f(int) override; // ok
  int g() override; // compilation error
  void h(int) override; // compilation error
};
```

Explicit overrides and final

```
struct Base {
    virtual void f();
};
struct Derived : public base {
    void f() final;    // virtual as it overrides base::f
    void g() { f(); } // compiler optimization:  direct call
                       // without virt.  func.  table
};
struct DerivedDerived : public derived {
    void f(); // error:  cannot override!
};
```

```
struct Base1 final { };
struct Derived1 : Base1 { }; // error
```

Object construction improvement

```
C++11:
class A {
    int avg = 1; // initial value
    A(int a1, a2) { avg = (a1 + a2) / 2; }
    //constructor chaining
    A(int *array) : A(array[0], array[1]) { }
};
```

Не нужно делать функцию init(...), которую будет вызывать каждый из конструкторов.