Конспект по C++

Черепанов Валерий

15 апреля 2016 г.

1 25 февраля 2016 г.

1.1 Универсальная сортировка

Сортировка с стиле C

```
{void nsort(void *array, size_t n, size_t elem_size, int (*fcmp)(void*, void*));}
```

Сортировка с стиле C++

```
// Interface
class Comparable {
    virtual int compare(const Coparable*) = 0 const;
}

// ** потому что мы не знаем размер наследников ‘Comparable‘.
void nsort(Comparable **c, size_t n);
```

Поддержка встроенных типов

```
template<typename T>
int nsort(T* array, size_t size) {
  if (array[i] < array[i+1])
      T t = array[i]; // swap ...
}

// OK
int a[100];
nsort(0, 100);

Worker a[100]; // Требуется operator <, конструктор копий, operator =
Worker **w; // Это работает?


template<typename T, size_t size>
class Array {
  T array[size];
```

```
  size_t getSize() {
      return size;
  }
}

Array<int, 100> a;
f(a); // Как выглядит сигнатура f?
```

Проблема с инстанцированными шаблонами

```
template<class T, class Container>
class Stack {
    Container C;
    void push(const T&);
    T pop();
};

Stack<int, Vector<int>>;
Stack<int, list<int>>;
Stack<double, vector<int>>; // Пользователь ошибся
s.push(3.5);
double d = s.pop(); // Возвратит 3, но не факт, что будет хотя бы warning
```

Решение:

```
// vector - значение по умолчанию
template<class T, template <typename V> class Container=vector>
stack<int, list>
class Stack {
    Container<T> c;
    ...
};
```

Альтернативная реализация для определенных типов

```
tempalte<class T>
class MyArray {
    T *array;
};
template<>
class MyArray<bool> { // Используется только для bool
};
```

2 3 марта 2016 г.

2.1 Исключения

```
    class BookEntry {
        Image *myImage;
```

```
    char *name;
    BookEntry() {
        name = new char[256];
        myImage = new myImage;
        myImage.load(...);
    }
    ~BookEntry() {
        delete [] name;
        delete myImage;
    }
}
```

Проблема — если конструктор получит exception, то деструктор не будет вызван.
Варианты решения:

1. RAII, то есть, например,

   ```
   shared_ptr<Image>
   ```

2. ```
   NetworkConnection {
       connect() {
           // работа с сетью, при ошибке Exception
       }
       ~NetworkConnection {
           logger log;
           log.print("Destructor NC");
       }
   }
   void main() {
       try {
           NetworkConnection nc;
           nc.connect(...);
       } catch (...) {

       }
   }
   ```

   Неприятная ситуация — допустим, было вызвано какое-то исключение, оно попало в деструктор, а в деструкторе log.print тоже бросил исключение. Тогда исходное исключение потеряется. Поэтому в C++ запрещено бросать исключение в деструкторе. При появлении оно сразу летит в main и кладет программу.

2.1.1 Гарантия исключений

No except просто не бросаем исключения.

Base guarantee даже при исключении класс остается валидным.

Strong guarantee если произошло исключение, то класс возвращается в исходное состояние. Обычно для этого делается копия изначального состояния класса.
   ```

2.2 STL

2.2.1 Конфликт заголовков

Некоторые заголовки C и C++ называются одинаково. Поэтому к названиям заголовков C в начале добавляется буква “c”.

2.2.2 Конфликт имен

Чтобы не засорять лишний раз scope можно объявлять классы внутри классов.

Развитие этой идеи — namespaces.

Замечание 1. *По очевидным причинам не стоит писать using в заголовочных файлах.*

У ifstream есть 4 флага:

good Все ок.

fail Неправильный формат.

bad Что-то не то с файлом.

eof Конец файла

```
ifs.exception(std::ifstream::badbit | ...); // При выставлении флага вылетит except
```

3 18 марта 2016 г.

3.1 Исправление некоторых неточностей

3.1.1 reserve

Для вызова `vector.reserve` не требуется вызвать какие-либо конструкторы (даже по умолчанию). Там используется placement new, то есть просто выделяется кусок памяти с помощью new и sizeof, а потом пишется `new(adress) Obj` где `adress` — это выделенная память.

3.1.2 string

Раньше string старались быть ленивыми и при создании старались сделать ссылку на уже созданное. Но плохо было то, что мы могли с легкостью получить линию при обращении с помощью `[]` (строка копировалась в новую память, чтобы ее можно было модифицировать), да к тому же мы могли инвалидировать итератор.

3.1.3 Манипуляторы потоков

Манипуляторы вроде `hex` как правило просто выставляют необходимый флаг с помощью `setf`. Для их использования перегружен оператор <<, он принимает поток и указатель на функцию.

3.2 STL

3.2.1 Кое-что про map

Как уже говорилось, `map` — это по сути `set` пар. Оператор `[]` у `map` создает новый элемент (вызывая стандартный конструктор), если мы обратились по несуществующему ключу, потому что он не знает, хотим ли мы лишь получить значение по ключу или изменить его.

3.2.2 Кое-что про set

```
class Person {
    string name;
    int age;

    bool operator <(const Person& p) {
        return name < p.name;
    }
}
```

Если мы создадим `set<Person>`, то элементы будут отсортированы по полю `name`. Но мы можем захотеть сортировать по `age`. Как этого добиться? На помощь приходят функторы!

```
struct by_age {
    bool operator()(const Person& p1, const Person& p2) {
        return p1.age < p2.age;
    }
}

set<Person, by_age> s; // Используем так
```

Внутри это устроено примерно так:

```
template<typename T, class comparator>
class set {
    insert(...) {
        if (comparator()(n1, n2)) { ... } // Анонимный объект
    }
}
```

Если мы не передаем в `set` второй шаблонный параметр, то используется стандартный функтор `less`. Он выглядит примерно так:

```
template<typename T>
struct less {
    bool operator()(const T& t1, const T& t2) {
        return t1 < t2;
    }
}
```

Замечание 2. *`multimap` и `multiset` — это тоже деревья, но обычно в узле хранится список элементов.*

3.2.3 `algorithm`

Итераторы — обертки над указателями. Их идея заключается в том, что алгоритм может работать с разными структурами данных, поддерживающими одинаковые операции (от структуры требуется лишь предоставить итераторы и методы работы с ними).

Некоторые алгоритмы STL:

1. `swap(T& a, T& b)`, `max(T& a, T& b)`
2. `count[_if](It a, It b, const T& x)`
   `x` — значение в `count`, функция или функтор в `count_if` (можно считать сложные функции или экономить время на сравнение).
3. `equal(It a, It b, It it)` *`// Сравнивает [a, b) с [it, ...)`*
4. `sort, min_element, nth_element, reverse, ...`

3.2.4 Чем хороши функторы?

Ну и чем же?

3.2.5 Чем плохи итераторы?

У них слишком маленький базовый интерфейс (`++`, `--`, `*`, `->`), поэтому они "из коробки" не подходят для адекватной реализации многих алгоритмов (даже для бинарного поиска). Кроме того, мы не знаем тип объекта итератора, а он может нам понадобиться. Для решения этих проблем были придуманты `iterator_traits`.

Как часть решения первой проблемы алгоритмы обарачивают некоторые обращения к итераторам используя функции `distance` и `advance`.

С точки зрения перемещения итераторы бывают:

1. Forward
2. Bidirectional
3. Random Access

4 25 марта 2016 г.

4.1 Итераторы внутри STL

```
template<class Iter>
void sort(Iter p, Iter q);

list<int> l;
vector<int> v;
sort(l.begin(), l.end());
sort(v.begin(), v.end());
```

Проблемы

1. Знаем итератор, но не знаем, например, тип элементов вектора.
2. Не знаем, что умеет итератор (например, может ли он в random access?). Поэтому большинство операций с итератором обарачиваем в библиотечные функции

```
advance(Iter& it, int n);
distance(Iter& it1, Iter& it2);
```

Решения

Как решена проблемы в STL?

```
template<class T>
class vector {
    T *array;
    class Iterator {
        typedef value_type T; // Решение первой проблемы
        // В sort пишем typename Iter::value_type var;
        typedef interator_category ra_iterator; // Решение второй проблемы
    };
};
```

Как делать “if” по типу? Перегрузкой!

```
template<class Iter>
void advance (Iter it, int n) {
    typename Iter::iterator_category ite;
    advance_impl(it, n, ine);
}
template<class Iter>
void advance_impl(Iter& it, int n, ra_iterator it) {
    it += n;
}
template <class Iter>
void advance_impl(Iter int n, int n, bidi_iterator it) {
    int i = 0;
    if (n > 0) {
        while(i < n) {
            ++it;
            ++i;
        }
    }
    if (n < 0) {
        while(i > n) {
            --it;
            --i;
        }
```

```
    }
}
```

Используется полиморфизм времени компиляции.

4.2 Iterator traits

Хотим делать примерно то же самое, но не для итераторов, а для указателей. Проблема:

```
template<class Iter>
void sort(Iter p, Iter q) {
    Iter::value_type;
}
```

Если вызовем sort от двух указателей, то получим compilation error.

Решение проблемы:

```
template<class Iter>
class iter_traits {
    typedef value_type Iter::value_type;
    typedef iterator_category Iter::iterator_category;
}
vector<int>;
typename iter_traits<vector<int>::iterator>::value_type a;
```

Кажется, мы ничего на самом деле не решили, а просто написали какую-то чушь. Но на самом деле это не так, нужно лишь воспользоваться специализацией шаблонов!

```
template<typename Iter*> // специализация для указателей
class iter_type {
    typedef value_type Iter;
};
```

```
Iter:iterator_category → iter_traits<Iter>::iterator_category
```

5 08 апреля 2016 г.

5.1 Преобразования типов

```
double dl = (double)3/4;
char *d = new double[100];
char *cd = (char)d; // Хотим вызвать send, поэтому cast

// В C часто использовали void*, char*.
int send(char *c, size_t size);
void *malloc(...);
sort(void *a, size_t n);
```

Новые возможности C++

`const_cast`
Убираем и добавляем константность.

```
void old_c_func(char *s);
f(const char *s) {
  old_c_func(s); // Не скомпилируется

  char *c1 = const_cast<char*>(s); // Аналог (char *)s из C
  old_c_func(c1);
}

// Проблема: похожие константные и не константные методы
class Vector {
  T& operator[](int id) {
    return const_cast<const Vector&>(*this)[id]; // Или не так?
    % FIXME (may be wrong)
  }
  const T& operator[](int id) const {

  }
}

// Просто примеры с const
void foo(string &s); // Если напишем const string, то все будет ок
foo("English"); // Не скомпилируется. Проблема в том, что вы передаем ссылку на временн
```

`static_cast`
Обычное приведение в духе C.

```
double d = static_cast<double>(3);
A *xa;
B *b = new B();
a = static_cast<A*>(b); // B к A
C *c = new c();
C *c1 = static_cast<C*>(a); // A (на самом деле B) к C. Компилятор ничего не понимает
```

`reinterpret_cast`
Говорит компилятору рассматривать что-то одного типа как что-то другого типа.
Не генерирует какой-либо ассемблерный код.

`dynamic_cast`
Выполняется только в рантайме. Провряет пристальнее, чем `static_cast`.

```
A* a;
B *b = new B();
// Если кажется, что можно сделать исходя из иерархии, то ок
```

```
a = static_cast<A*>(b);
// Хранит указатель на массив виртуальных функций и понимает больше
C *c = dynamic_cast<C*>(a); // исключение

Vector<A*> v;
v.add(new B());
v.add(new C());
v.add(new D());
v.add(new E()); // Есть какой-то метод g, которого нет у A, хотим вызывать его вместо g
for(int i = 0; i < n; i++) {
  v[i]->f();
}

#include <typeinfo>
E *pe;
if(pe=dynamic_cast<E*>(v[i])) { // Проверим, E ли это
  pe->g();
} else {
  v[i]->f();
}
```

`typeid` Выполняется в рантайме, действует как `dynamic_cast`

```
type_info ti = typeid(*c);
ti.name(); // "MyClass"
```

6 15 апреля 2016 г.

6.1 C++11

6.1.1 Избавляемся от define

nullptr Раньше было так:

```
// Было
int *x = 0;
int *p1 = NULL; // #define NULL (void*)0;

void f(long l) { }
void f(char *s) { }

int main() {
  f(0L); // OK
  char s[] = "Hello";
  f(s); // OK
  f(NULL);  // Ambiguity
  f(0);     // Ambiguity
}
```

В C++ эту проблему постарались решить с помощью нового ключевого слова `nullptr`. Такой “указатель” неявно кприводиться к указателю любого другого типа.

static_assert Раньше был только обычный `assert`.

```
// Если boolvar = false, то runtime error.
assert(boolvar);
```

Теперь появился `static_assert`, который выполняется на стадии компиляции.

```
// Второй аргумент - сообщение об ошибке
static_assert(sizeof(int)>=4, "This program could not work");

template<typename T, size_t size>
class static_array {
  T array[size];
  static_array() {
    static_assert(size > 0, "Empty arrays are prohibited");
  }
}
static_array<int, 100> a; // OK
static_array<int, 0> b;   // Error
```

Type alias Раньше:

```
// Так можно
typedef map<string, list<int>> ml_t;
f(ml_t a);

// А вот так уже нет
typedef map<string, list<T>> ml_t;
f(ml_t<int> a);
```

C++11 спешить на помощь!

```
template<typename T>
using mt_t=map<string, list<T>>
```

constexpr Было:

```
size_t square(int x) a {
  return x*x;
}
const int sz = cl;
int a[sz]; // OK
static_array<int, sz> sa; // OK
int a[square(3)]; // Error
static_array<int, square(3)> sa; // Тоже error
```

Решение:

```
// constexpr говорит, что функция при определенных
// условиях может использоваться в константных выражениях
constexpr size_t square(int x) a {
  return x*x;
}
```

6.1.2 Декораторы

default и delete

```
class T {
  int v = 0; // Такая инициализация - тоже фича C++11
public:
  // default говорит, что в любом случае нужно создать метод по умолчанию
  T() = default;
  // delete говорит, что не стоит создавать метод по-умолчанию
  T& operator= (const T&) = delete;
  T(const T&) = delete;
}
```

override Проблема:

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

class Derived:public Base {
  void f(int) const;
  virtual int g(int); // Забыли const
  void n(int) const;
}

const Base *b = new Derived();
b->f()
b->g(); // Base::g(), так как сигнатуры не совпадают
b->n();
```

Решение — `override`. Это слово говорит компилятору, что функция что-то перекрывает. “Под капотом” компилятор просто проверяет, есть ли такая виртуальная функция в базовом классе.

```
class Derived:public Base {
  void f(int) const override; // OK
  virtual int g(int) override; // Error
  void n(int) const override; // OK
}
```

Конструкторы

```
class MyNumber {
  MyNumber(char* s): A(atoi(s)) { } // Вызываем один конструктор из другого
  MyNumber(int i) { };
}

MyNumber("64");
MyNumber(64);
```