Билеты к экзамену C++
МКН, Современное программирование
семестр II

Тамарин Вячеслав

June 17, 2020

Contents

Вопрос 1 Шаблоны

- решение в стиле C (`#define`)
- шаблонные классы
- шаблонные функции
- специализация шаблонов (частичные и полные; в т.ч. для функций)
- шаблонный параметр, не являющийся типом

i Решение в стиле *C*

Пусть есть класс массива для целых чисел или умный указатель

```
class MyArray {
private:
    int *array;
};

class scoped_ptr {
private:
    GaussNumber *ptr;
}
```

Эти классы рассчитаны только для одного типа данных и для каждого типа придется вручную создавать новый тип.

Решить проблему можно с помощью `#define`. Классы для каждого нового типа будет генерировать препроцессор с помощью макросов.

MyArray.h

```
#define MyArray(TYPE) class MyArray_##TYPE {\
private: \
    TYPE *array; \
    size_t size; \
public: \
    TYPE get(size_t index) { \
        return array[index]; \
    } \
};
```

main.c

```
#include "MyArray.h"
MyArray(int);
MyArray(double);

int main() {
    MyArray_int a; // вместо MyArray(int) будет полный текст макроса
    MyArray_double b;
}
```

Проблема: Программист и компилятор видят разный исходный текст, разные сообщения од ошибках, препрепроцессор заменит любое подходящее слово на данный код.

ii Шаблонные классы

MyArray.h

```
template <typename T>
class MyArray {
private:
    T *array;
    size_t size;
```

```
public:
    T& get(size_t index) {
        return array[i];
    }
};
```

Можно вынести определение методов за пределы объявления класса

MyArray.h

```
template<class T> // синоним template<typename T>
T& MyArray<T>::operator[] (size_t index) {
    return array[i];
}
```

main.cpp

```
#include "MyArray.h"
int main() {
    MyArray<int> a;
    MyArray<double> b;
    MyArray<MyArray<int>> c; // лучше не писать до c++11
}
```

Особенности:

1. Подстановку делает компилятор, а не препроцессор
2. Код шаблонного класса всегда в заголовочном файле
3. Иногда помещают в `MyArray_impl.h`
4. Увеличивается время компиляции
5. Методы шаблонного класса всегда `inline`

iii Шаблонные функции

```
template <class T>
void swap(T &a, T &b) {
    T t(a);
    a = b;
    b = t;
}
int i = 10, j = 20;
swap<int>(i, j);
```

```
template <typename V>
void reverse(MyArray<V> &a) {
    V t;
    for (size_t i = 0; i < a.size()/2; ++i) {
        t = a.get(i);
        a.set(i, a.get(a.size() - i - 1);
        a.set(a.size() - i - 1, t);
    }
}
// Вызов
reverse<int>(a);
```

Вывод шаблонных параметров

Компилятор может понять, какие аргументы у шаблона функции, если это однозначно определяется.

```
MyArray<int> a;
MyArray<double> b;
reverse(a);
reverse(b);
```

iv Специализация

Идея: оптимизация для конкретного класса.

Общая версия

```
template<typename T>
class Array {
private:
    T *a;
public:
    Array (size_t size) {
        a = new T [size];
        ...
    }
};
```

Полная специализация

Для bool

```
template<>
class Array<bool> {
private:
    char *a;
public:
    Array (size_t size) {
        a = new char [(size-1)/8 + 2];
        ...
    }
};
```

Частичная специализация

Для массивов

```
template<class T>
class Array <Array<t>> {
    T **a;
};
```

v Шаблонный параметр, не являющийся типом

```
template<size_t Size>
class Bitset {
private:
    char m[(Size-1)/8 + 1];
public:
    bool get(size_t index) { ... }
};

Bitset<128> b1;
```

Вопрос 2 Исключения

- обработка ошибок в стиле C
- try/catch/throw
- исключения в конструкторах и деструкторах
- идиома RAII: использование и примеры классов
- гарантии исключений

Виды ошибок:

1. По вине программиста

Примеры

```
char *s = NULL;
size_t l = strlen(s);
Array a(-1);
```

Обработка этих ошибок

- Лучше выявить на стадии тестирования
- Если программа идеальна, не происходят
- Библиотека C такие ошибки не обрабатывает
- Библиотека C++ по-разному: `vector.at(i)`, `vector.operator[i]`
- Обрабатывать или нет — на усмотрение программиста

2. По вине окружения
   - Файл не существует
   - Сервер разорвал соединение
   - Вместо числа ввели букву

   Обработка ошибок
   - Могут происходить и при работе идеальной программы
   - Обязательно обрабатывать

i Обработка ошибок в стиле C

Для обработки ошибок:

- Проверка на наличие ошибки в if
- Освобождение ресурсов

```
delete [] array;
close(f);
```

- Сообщить пользователю или вызывающей функции

```
FILE *f = fopen("a.txt", "r");
if (f == NULL) {
    printf("File a.txt not found\n");
}

if (f == NULL) {
    return -1;
}
```

- Предпринять действия по восстановлению (попробовать соединиться еще раз)

В стиле C информация об ошибке передается через возвращаемое значение и через глобальную переменную:

```c
FILE* fopen(...) {
    if (file not found) {
        errno = 666;
        return NULL;
    }
    if (permission denied) {
        errno = 777;
        return NULL;
    }
    ...
}
```

По возвращаемому значению не знаем причину ошибки, глобальная переменная хранит код ошибки, можно получить оттуда сообщение (`strerror(code)`) .

Не всегда хватает диапазона возвращаемых значений функции

```c
class Array {
    int *a;
public:
    // возвращает -1, когда индекс выходит за границу
    int get(size_t index);
};
int r1 = atoi("0");
int r2 = atoi("a"); // ?
```

Также код логики и обработка ошибок перемешаны

```c
r = fread(...);
if (r < ...) {
    // error
}
r = fseek(...);
if (r != 0) {
    // error
}
```

ii try/catch/throw

Структура исключений

```c
class MyException {
private:
    char message[256];
    // filename, line, function name ...
public:
    const char* get();
};

double divide (int a, int b) {
    if (b == 0) {
        throw MyException("Division by zero");
    }
    return a/b;
}
```

Структура исключений

```c
try {
    x = divide(c, d);
}
catch (MyException& e) {
    std::cout << e.get(); // сообщаем пользователю
    // можем освободить ресурсы
    // throw e; проинформировать вызвавшую функцию
```

iii Идиома RAII

Взятие ресурса должно «инкапсулировать» в класс, чтобы в случае исключения вызвался деструктор и освободил ресурс.

```
void f() {
    MyArray buffer(n);
    if ( ... ) throw MyException( ... );
}
```

```
void g() {
    autoptr p(new Person("Jenya", 36, true));
    divide(c, e); // может быть исключение
```

Если в divide произойдет исключение, объект еще не будет «достроен», поэтому деструктор не вызовется:

```
class PhoneBookItem {
    PhoneBookItem (const char *audio, const char *pic) {
        af = fopen(audio, "r");
        pf = fopen(pic, "r");
        divide(c, e); // исключение
        f();
    }
    ~PhoneBookItem() {
        fclose(af);
        fclose(pf);
    }
```

Поэтому нужно предусмотреть такую ситуацию и обернуть конструктор в try/catch:

```
class PhoneBookItem {
    PhoneBookItem (const char *audio, const char *pic) {
        try {
            af = fopen(audio, "r");
            pf = fopen(pic, "r");
            divide(c, e); // исключение
            f();
        }
        catch(MyException& e) {
            fclose(af);
            fclose(pf);
            throw e;
        }
    }
}
```

Исключения в деструкторе бросать нельзя, так как они могут подменить реальную причину ошибки. Если так происходит, программа аварийно завершается. В такой ситуации можно поступить так:

```
class PersonDatabase {
    ~PersonDatabase() {
        try {
            // брошена серверная ошибка
            networkLogger.log("Database is closed.");
        }
        catch (...) {} // поймать все
    }
};

f() {
    PersonDatabase db;
    if (...) throw MyException("Error: disk is full.")
}
```

iv Гарантии

Гарантии:

1. обязательства функции (метода) с точки зрения работы с исключениями
2. документация для программиста, работающего с функцией (методом)

Виды гарантий:

no throw guarantee не бросает исключений вообще

```
void strlen(const char *s) {
    int count = 0;
    while (*s != 0) {
        s++; count++;
    }
    return count;
}
```

```
void f() {
    try {
        strlen(s);
        divide(a, b);
    }
    catch (...) { }
}
```

basic guarantee в случае возникновения исключения ресурсы не утекают

Если произойдет исключение, то память «течь» не будет, но измененные элементы array свои значения не восстановят:

```
class PersonDatabase {
    MyVector<Person> array;
    void process() {
        auto_ptr<Person> p(new Person());
        for (int i = 0; i < array.length; i++) {
            int a = divide(rand(), rand())
            // может быть исключение
            array[i]->setAge(a);
            std::cout << p;
        }
    }
};
```

strong guarantee переменные принимают те же значения, что были до возникновения ошибки

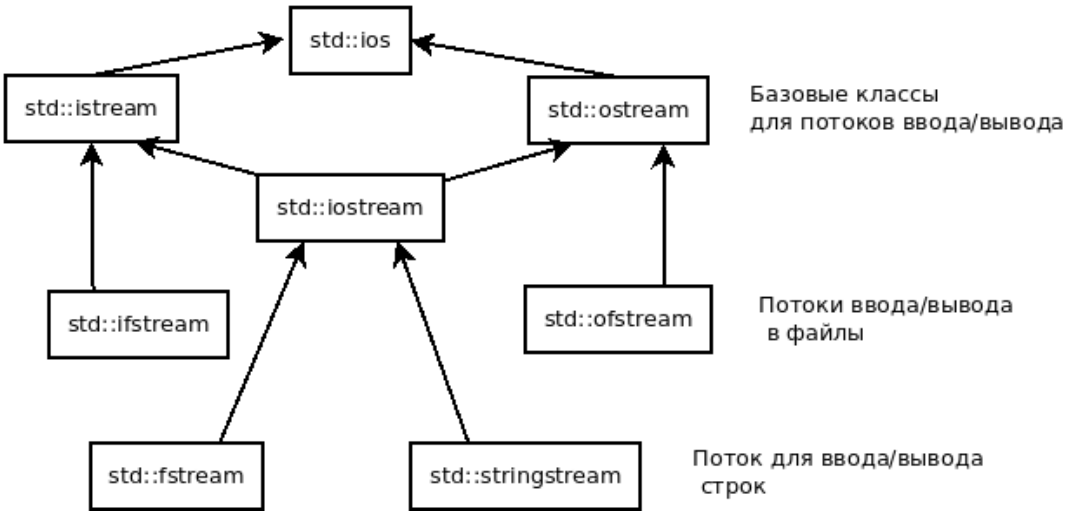
Идиома copy-and-swap

```
class PersonDatabase {
    MyVector<Person> array;
    void process() {
        auto_ptr<Person> p(new Person());
        MyVector<Person> copy(array);
        for (int i = 0; i < array.length; i++) {
            int a = divide(rand(), rand())
            // может быть исключение
            copy[i]->setAge(a);
        }
        array = copy;
    }
};
```

Вопрос 3 Ввод-вывод в C++

- иерархия классов
- методы/флаги/манипуляторы
- обработка ошибок
- ввод-вывод пользовательских типов

i Иерархия классов



Глобальные переенные `cin` (istream), `cout` (ostream), `cerr` (ostream).

Стандартная библиотека содержит перегруженные операторы `operator »`, `operator «` для примитивных типов и строк.

```
std::ostream& operator << (std::ostream& os, int v) {
    // convert int to bytes, write bytes
    return os;
}
```

```
std::istream& operator >> (std::istream& is, int& v) {
    // read bytes, convert to int
    return is;
}
```

Такой код приводит к очистке буфера fflush потока, что замедляет вывод:

```
std::cout << x << std::endl;
```

Оператор читает строку до пробела, getline до конца строки:

```
std::ifstream ifstream if("in.txt");
std::string word;
if >> word;
std::string line;
getline(if, line);
```

Побайтовый ввод/вывод: read, write, seekg, tellg.

ii Обработка ошибок

rdstate() — чем закончиласть последняя операция: eofbit, goodbit, failbit (считываем другим типом), badbit (не существует файла)

iii Формат вывода

```
int x = 255;
std::cout.setf(std::ios::hex, std::ios::basefield);
std::cout << x; // после вывода флаг очистится
```

Через манипуляторы

```
ostream& operator << (ostream& (*pf)(ostream&));
```

iv Ввод-вывод пользовательских типов

```
class Point {
private:
    int x;
    int y;
public:
    friend ostream& operator << (ostream& os, const Point& p);
    friend istream& operator << (istream& is, Point& p);
    // friend функция может иметь доступ к приватным членам
};

ostream& operator << (ostream& os, const Point& p) {
    os << p.x << " " << p.y << "\n";
    return os;
}
istream& operator << (istream& is, Point& p) {
    is >> p.x >> p.y;
    return is;
}
```

Так как ostream — базовый класс, можно использовать один и тот же оператор для вывода на экран, строку и файл: (cout, ofstream of("file"), stringstream ss).

Вопрос 4 Приведение типов

- C-style cast, static_cast, const_cast, reinterpret_cast - поведение и преимущества
- RTTI и dynamic_cast

i Приведение типов в C

```
void f(char *p);
int *pi = malloc(100 * sizeof(int));
f(pi); // неявное приведение типов

int a = 65535;
char b = a; // неявное приведение типов

int c = 3.5; // тоже неявное

int a = 5; int b = 6;
double c = a / (double)b; // явное, действительно хотим
```

Неявное приведение может вызвать warning, если указать -Wall -Werror, станет ошибкой.

ii Приведение типов в C++

Явное приведение требуется для указателей (кроме к void* и приведения у базовому классу).

```
void *f();
int* pi = (int*) f();

// class B : public class A
void print(const A* p);
B b;
print(& b);
```

Приведение для классов

```
Class BigInt {
    BigInt(int a);  // from int
    operator int(); // to int

    BigInt(const Complex&); // from Complex
    operator Complex();     // to Complex
```

Явное приведение упрощает поиск в коде и более точно выражает намерение программиста:

- Разные cast'ы для разных случаев
- Компилятор делает более точную проверку

explicit запрещает неявное приведение.

static_cast

Примитивные типы; классы, связанные с наследованием; приведение к void*; пользовательские преобразования BigInt → int

```
int a = 65535;
char b = static_cast<char>(a);

// class B: public class A
void f(B *b);
A *a = new B();
f(static_cast<B*>(a));
```

reinterpret_cast

Указатели разных типов

```
void* f();
int* pi = reinterpret_cast<int*>(f());

char *pc =  ...; int *pi = ...;
pc = reinterpret_cast<char*> pi;
```

const_cast

Добавление и удаление const

```
char const *p1 = "Hello";
char *p2 = const_cast<char*>(p1);
p2[0] = 'h'; // undefined behavior
```

iii Технология RTTI (Run-Time Type Information)

- Оператор dynamic_cast осуществляет безопасное преобразование указателя на базовый класс в указатель на производный (ссылки).
- Оператор typeid возвращает фактический тип объекта для указателя (ссылки).

dynamic_cast

```
// class B: public class A;
// class C: public class A;
void f(B *b);

A *a = new C();
f(static_cast<B*>(a)); // при компиляции без ошибок, но undefined behavior во время работы

if (dynamic_cast<B*>(a) != 0) {
    f(static_cast<B*>(a));
}
```

Фактический тип

```
#include <typeinfo>
// class C: public class A
A *a = new C();
type_info ti = typeid(*a); // требуется ссылка
ti.name(); // "C"
```

- RTTI работает для классов с виртуальными функциями, информация о типе хранится в таблице виртуальных функций
- Чаще всего используется, когда нужно сделать костыль для существующего кода, который нельзя переделать.

```
class Shape {
    virtual draw() = 0;
};

draw_all(Shape* shapes, size_t n) {
    for (int i = 0; i < n; i++) {
        Shape *p = shapes[i];
        p->draw();
        if (dynamic_cast<AnimatedShape*>(p) != 0)
            p->animated_draw();
    }
}
```

Вопрос 5 Последовательные контейнеры

- string, vector, list
- array
- внутреннее устройство и основные операции
- итераторы и их инвалидация

Требования к хранимым в контейнере объектам:

1. Корректно работает конструктор копий (copy-constructable)
2. Корректно работает оператор присваивания

Методы всех контейнеров:

1. Конструктор по умолчанию, конструктор копирования, оператор присваивания, деструктор
2. Операторы сравнения: `==, !=, >, >=, <, <=`
3. `size(), empty()`
4. `swap(obj2)`
5. `insert(), erase()`
6. `clear()`
7. `begin(), end()`

Особенности последовательных контейнеров:

1. Сохраняют порядок, в котором были добавлены элементы
2. Есть добавление в конец: `push_back`

i string, vector, list

vector

Динамический массив с автоматическим изменением размера при добавлении элементов.

```
std::vector<int> v;
```

1. Для уменьшения числа вызовов new при добавлении элементов выделяется с запасом
2. `size, capacity`
3. Сложность добавления в конец вектора и удаления из конца — $\mathcal{O}^*(1)$
4. Сложность добавления и удаления элемента из начала или середины — $\mathcal{O}(n)$
5. Сложность доступа к элементу по индексу — $\mathcal{O}(1)$

Методы:

1. `size()`/`resize()` — получение/изменение размера вектора
2. `capacity()`/`reserve()` — получение/изменение зарезервированной памяти
3. `push_back()`/`pop_back()` — добавление/удаление последнего элемента
4. `operator[]`, `at()` — получение элемента по индексу
5. `data()` — указатель на массив

```
#include <vector>

int main() {
        std::vector<int> v;
        v.push_back(10);
        v.pop_back();
```

```
        v[0] = 13;
        v.at(0) = 14;
        size_t size = v.size();
        bool is_empty = v.empty();
        v.clear();
        v.resize(10); // меняем размер на 10 элементов, работает, если есть конструктор по умолчанию
        v.resize(20, 5); // новые 10 элементов будут равны 5

        int *dst = new ...
        memcpy(dst, v.data(), v.size());

        v.reserve(100); // резервируем память на 100 элементов, размер  не меняется
        v.reserve(v.size() + 100);
        v.clear(); // изменит размер до 0, но вместимость не изменится
        vector<int>(v).swap(v); // уменьшить размерность вектора до реально используемых элементов
        return 0;
}
```

list

Двусвязный список. Вставка и удаление в любом месте за $(O)(1)$. Нет обращения по индексу. Методы:

1. `size()`/`resize()` — получение/изменение размера вектора
2. `push_back()`/`pop_back()` — добавление/удаление последнего элемента
3. `push_front()`/`pop_front()` — добавление/удаление первого элемента
4. `merge()`/`splice()` — объединение/разделение

string

Контейнер для хранения символьных последовательностей.

1. Метод `c_str()` для совместимости со старым кодом:

```
std::string res = "Hello";
printf("%s", res.c_str());
```

2. Алгоритмы `substr()`, `find()`, `...`
3. Поддержка преобразований типа с C-строками

```
f(const std::string& s);
f("Hello");
```

4. `append`, `operator+`, `operator+=`
5. `string = basic_string<char>`
6. `wstring = basic_string<wchar_t>` — для работы с длинными символами

ii Итераторы

Объекты, которые синтаксически ведут себя как указатель (++, -, *, ->). Это универсальный способ перебора элементов контейнеров в STL. Итераторы реализованы как вложенные классы для контейнеров.

```
class vector {
        ...
        class iterator {
            operator ++() { T* ptr++; }
        }
```

```
}
vector::iterator it1;

class list {
    class iterator {
                operator ++() { Node<T> ptr = ptr->next; }
        }
}
list::iterator it2;
```

Все итераторы имеют функции, которые возвращают итераторы на первый и следующий за последним элемент.

Для vector и deque реализована арифметика, как для указателей:

```
int i = *(v.begin() + 5);
```

Проход по контейнеру:

```
list<int> l;
list<int>::iterator it = l.begin();
for (; it != l.end(); ++it) cout << *it;
```

Итератор, не позволяющий менять данные, на которые он указывает:

```
list<int>::const_iterator cit = l.begin();
```

Удаление элемента, на который указываем

```
it = v.erase(it);
```

Вставка элемента на данную позицию

```
it = v.insert(it, 5);
```

Инвалидация итераторов

После того, как произвели операцию с контейнером, итератор может указывать в неверное место.

```
std::vector<int> v;
it = v.erase(it); // возвращаем следующий элемент
std::vector<int>::iterator itb = v.begin();
v.push_back(3);
v.push_back(5);
```

Поставить перед каждым четным элементом вектора 0:

```
std::vector<int> v;
for (std::vector<int>:: iterator i = v.begin(); i != v.end(); ++i) {
    if (*i % 2 == 0) {
            i = v.insert(i, 0);
                ++i;
```