C++, II семестр

Весна 2015, лектор: Линский Евгений Михайлович

Авторы: Ольга Черникова, Глеб Валин, Юрий Кравченко, Дмитрий Лапшин, Денис Галеев

Собрано: 6 сентября 2015 г. 04:52

Оглавление

Глава 1

Шаблоны

1.1. Дальнейший план действий

1.1.1. Отчетность:

Баллы ставятся за домашние задания и практики, домашних заданий будет больше. Что бы быть допущенным к экзамену нужно сдать все домашние задания не на ноль.

Чем позже язык был придуман, тем больше у него стандартная библиотека.

В C стандартная библиотека тоже есть, но в ней много чего не хватало. Прежде всего, структур данных.

Но перед тем как изучать STL, нужно изучить еще две темы: шаблоны и исключения.

На следующем занятии (20.02.2015) большой тест на по темам прошлого семестра. Надо пересмотреть конспекты.

1.2. Зачем нужны шаблоны?

Когда мы делали классы, мы сокрушались, что явно должны были указывать тип.

```
1 class Array{
2      int *p;
3 };
```

И чтобы хранить `int` и `double` нужно было сделать два различных класса: `ArrayInt` и `ArrayDouble`. И нам бы пришлось два раза скопировать код.

Копировать код — это не очень хорошо, поскольку надо читать в два раза больше кода и если нашлась ошибка в первом классе, то нужно исправлять ее и во втором.

1.3. Как люди пытались это решить в языке C?

Это делалось с помощью *#define*

array.h

```
1 class Array {
2     type *a;
3     ...
4 };
5
6 inline Type Array::get(...) {
```

```
    ...
}
```

Реализация методов должна быть в заголовочном файле, соответственно нужен `inline`. В отдельном файле не получится это реализовать.

`main.c`

```
#define Type int;
#include "array.h"

int main() {
    Array a;
}
```

В данном примере можно использовать **typedef** и здесь это даже лучше, поскольку *#define* обрабатывается в режиме препроцессора и работает со строками, ничего не зная о типах и так далее.

Теперь пусть в одном и том же файле хотим иметь и массив элементов типа **int** и **double**.

`main.c`

```
#define Type int
#include "array.h"
#undef Type
#define Type double
#include "array.h"
#undef Type

int main() {
    ...
}
```

В результате в нас получилось что-то вроде:

```
class Array{
    int *a;
};

inline int Array::get...

class Array{
    double *a;
};

inline double Array::get...

int main() {
    ...
}
```

Теперь у нас два объявления одинаковых классов. Такого быть не может. `Array ar;` — компилятор не сможет понять, что выбрать.

Нужно писать в `array.h`

```
class Array#Type{
    Type *a;
}

inline Array#Type::get(...) {

}
```

Тогда `#Type` заменится на тип и получится `Arrayint`. Синтаксис позволяет заменить подслово. `My#Type#Array`, если хотим заменить слово посередине строки.

```
int main() {
    Arrayint a;
    Arraydouble b;
}
```

В `#define` проблема состоит еще в том, что мы видим не то, что видит компилятор, и ошибки выдаются уже в измененном коде.

1.4. Еще одна попытка сделать шаблоны в С

`array.h`

```
#define DefineArray(name, type) \
class name{ \
   type *date; \
};
```

Многострочный *#define*. В многострочном макросе строчки разделяются \.

Дальше идет полный код класса. `main.c`

```
#include "array.h"
DefineArray(ArrayInt, int);

main() {
    ArrayInt a;
}
```

Подставится полный текст класса с заменой имени и типа.

1.5. Шаблоны в С++. Синтаксис

```
template <typename T>
class A {
    T* data;
    T get(size_t index) {
        return data[index];
    }
};
```

```
template <typename T>
class A {
    T* data;
    T get(size_t index);
};

template <typename T>
T Array<T>::get(size_t index) {

}
```

`template` обозначает, что следующая единица зависит от параметров. Параметров может быть несколько.

Это должен быть один файл. Это важно. В разных файлах это быть не может, поскольку это не может быть скомпилированно. Этот код находится в заголовочном (`*.h`) файле, но не в `*.cpp` файле.

Иногда, чтобы код не разбухал, пишут несколько `*.h` файлов и в конце первого пишут *`#include`*.

`main.cpp`

```
#include "Array.h"
Array <int> a;
Array <double> b;
Array < Array <int> > aa;

main() {
    int c = a.get(3);
    b.get(5);
}
```

`Array< Array <int> > aa;` — здесь компилятор начинает путать шаблоны и сдвиг вправо, поэтому здесь лучше ставить пробел.

Препроцессор вставит весь кусок, потом компилятор создаст новое имя класса. Разница в том, что замена имени произойдет на другом этапе. Нам ничто не мешает написать на место `T` любой тип.

1.6. Требования к типу

`Array< Array <int> > aa;` — здесь у класса `Array` есть неявное требование.

```
Array(size_t size) {
    data = new T[size];
}
```

У `T` должен быть конструктор по умолчанию. Такие требования описаны только в документации или надо смотреть исходники.

1.7. Функции с template

```
template <typename V>
void reverse(Array<V> &a) {
    for (size_t i = 0; i < a.size()/2; ++i) {
```

```
        V t = a.get(i);
        a.set(i, a.get(a.size() - i - 1));
        a.set(a.size() - i - 1, t);
    }
}

Array <int> a;
reverse <int>(a);
```

При вызове функции надо указать параметры шаблона. Так делать муторно, если у функции много параметров, достаточно написать `reverse(a)` — компилятор сам попытается вывести нужный тип. Задание на подумать к следующему тесту: Приведите пример, когда компилятор не может вывести тип и его нужно указывать явно.

```
template <typename V, typename T>
void copy(Array <V>& a, Array <T>& b);

Copy <int, double> (c,d);
Copy (c d);
```

Компилятор может вывести тип, если о них можно догадаться по параметрам.

1.8. Нужно написать функцию сортировки, которая может сортировать массивы любых типов

```
sort(/*?*/ array, int size)
```

Раньше эта сортировка выглядела так:

```
sort(void *array, size_t elem_size, size_t size, int (*comp)(void *, void *));
```

Теперь как сделать эту сортировку в стили ООП?

```
class compar{
    virtal int cmp(const compar &) const = 0;
};

class Complex: public compar{
    int cmp(...) {
        if (....
    }
}
```

То есть мы можем сортировать только те объекты, которые от наследованы от compar.

```
compar* array;

class Complex:public compar{
    int re;
    int im;
    int cmp(const compar * c) {
```

```
        Complex * comp = (Complex)c;
        if (re < comp -> re)
    }
}

array[i] = new Cat();
array[i + 1] = new Dog();
```

Классы dog и cat наследуются от compar. Теперь у нас в массиве лежат разные типы, для которых по-разному определен оператор сравнения. Проблема.

compar** array; Тип должен быть **. Мы сделали базовый класс compar (или интерфейс) и в итоге выяснили, что тип в sort должен быть compar **, что бы в памяти объекты занимали 4 байта и можно было менять элементы местами. Потенциальная ошибка: наличие в массиве разных типов.

```
template <typename T>
void sort(T* array, size_t size) {
}
```

Здесь можно писать указатель, а не **, поскольку она будет скомпилирована, только когда у нас будет уже конкретный массив собак.

```
Dog *d = new Dog[100];
sort <Dog> (d, 100);

int *i = new int[100];
sort <int> (i, 100);
```

Неявное требование к типам: у типа должен быть operator <.

1.9. Константы в шаблонах

В шаблонную переменную можно передавать константу.

```
template <typename T, size_t Size>
class Array{
    public:
        T arr[Size];
};
```

Array<int, 100> a; — значит a' будет массив размера 100.

1.10. Специализация

Если мы хотим, например, сэкономить память и bool хранить в char.

```
template <typename T>
class Array{

}
```

Отдельно выделяем класс `bool` и его специализацию нужно написать целиком. То есть, если другой тип, будет использован первый вариант, второй специально для `bool`.

```
template <>
class Array<bool>{
    set(){
    }
    char *data;
}
```

Глава 2

Исключения

Какие бывают ошибки? Выделяют 2 типа:

1. Происходят в момент исполнения по вине окружения (нехватка памяти, неправильное имя файла, некорректный ввод данных)

2. По вине программиста, то есть некорректность программы (например передача нулевого указателя и попытка получения доступа к его полям, нехватка `delete`)

Требования к программам отличаются, но всегда нужно учитывать ошибки первого типа. Если ставить везде проверки, то это замедляет выполение программы (например вектор с безопасным доступом, вместо массива — накладные расходы на проверку прикаждом обращении по индексу). Язык C++ тратит некоторые ресурсы для проверки ошибок.

2.1. assert

Существует макрос `assert(bool)`. Если выражение `true` — ничего, если `false`, то вызывается функция авариного завершения прграммы. Например мы пишем функцию `strlen(char* p)`, пусть мы договорились, что никогда не передаём нулевые указатели:

```
int strlen(char* p) {
    assert(p != NULL);
    ...
}
```

В макросе написано что-то похожее на:

```
#define assert(bool) \
#ifdef DEBUG \
if (!bool) exit(); \
#else \
; \
#endif
```

То есть в готовом продукте на месте `assert` проверок не будет.

Замечание 2.1.1. `assert` лежит в `assert.h` и `cassert`.

Ошибки первого типа нельзя обрабатывать `assert`, т.к. `assert` в realise-ной версии не сохранится, а ошибки всё равно могут возникнуть, кроме того если бы он работал всегда, то при проблемах программа просто бы падала — это не то, что ожидает пользователь.

Обработка ошибки:

1. Сообщение об ошибке
2. Освободить ресурсы (память, открытые файлы)
3. Возобновить работу (например — браузер не вываливается, когда не получает ответа и делает ещё несколько обращений, или другой пример: если неверный ввод, то спросить ещё раз)

2.2. Обработка ошибок в языке C

Рассмотрим следующий пример:

```
FILE *fp;
fp = fopen(''a.txt'', ''+'');
if (!fp) {
    // вывод: произошла ошибка
}
```

Так обрабатывались ошибки в C. Какие проблемы могли возникнуть? Файл мог не открыться по двум причинам:

1. Не существует файл
2. Нет прав доступа

Мы судим по значениям, которые возвращает `fopen()`, а указатель может принять только одно некорректное значение `NULL`, то есть 2 ошибки никак не обозначить, да и не хорошо это указателем шифровать ошибку.

Или другой пример:

```
class Array;
int Array::get(index);
```

Не понять получили ли мы значение или произошла ошибка.

Ещё пример: функция `atoi()` — аналогично, во время ошибки вернёт 0, но и значение при аргументе `''0''` — 0.

Как с этим разбирались в языке C? Существовала глобальная переменная `errno`, в которой хранился код ошибки. Но и этот способ тоже не лишён проблем:

```
f();
// если забыть здесь проверить errno, то вызов g() перезапишет значение
g();
```

Ещё один недостаток постоянных проверок — загромождедние кода, рядом с каждой функцией возникает `if`.

Возникают вопросы:

1. Нельзя ли разделить обработку ошибок и код возврата?
2. Нельзя ли из блока с ошибкой переходить в отдельный блок с обработкой?

Вспомним шаблон Model-View на примере игры крестики-нолики:

```
//Model
move(int x, int y, ...) {
    if (/*на (x, y) уже стоит фигура*/) {
        printf(''плохой ход'');
    }
}
// модель не знает как выводить на экран, поэтому мы должны
// как-то сообщить в view об ошибке, где она будет обрабатываться.
// Model
int move(int x, int y, ...) { // добавили int как возвращаемое значение
    if (/*на x, y уже стоит фигура*/) {
        printf(''плохой ход'');
        return -1;
    }
}
```

Пример: `f()` — обработка ⇒ `g()` ⇒ `h()` — ошибка. Тогда `g()` выглядит так:

```
void g() {
    int ret = h();
    if (ret < 0)
        return ret;
}
```

Хочется писать поменьше такого кода, чтобы если функция не могла обработать ошибку, то выбрасывала её выше.

Предположим мы решили нашё проблему через `goto`, а `g()` вглядит так:

```
void g() {
    MyArray a;
    h();
}
```

Возникает проблема: мы не освобождаем ресурсы, в коде выше у объекта `a` не будет вызван деструктор. То есть нам нужен механизм, двигающийся по стеку и вызывающий во всех фреймах деструкторы.

В итоге, что мы теперь хотим от обработки ошибок:

1. Нужно, чтобы информация об ошибке хранилась в классе, чтобы можно было указать достаточно информации
2. Разделять код выброса ошибки и код обработки
3. Передавать управление к блоку обработки, чтобы по пути особождались все стек-фреймы.

2.2.1. Обработка исключительных ситуаций в C++

Рассмотрим синтаксис на примере ниже:

```
int f() {
    if (...) {
        throw w; // здесь выбрасываем исключение: throw <любой тип>
    }
```

```
}

int g() {
    try {   // блок, из которого могут выброситься исключения
        f();
    }
    catch(/*тип ошибки*/ except) {  // обработка исключений
        ...
    }
    catch(/*тип ошибки*/ except) {  // catch-блоков сожет быть несколько
        ...
        throw except2;  // из catch-блока тоже можно выбрасывать исключения
    }
}
```

Так выглядит стек программы:

```
f()
g()
main()
```

При выбросе исключения мы раскручиваем стек и освобождаем ресурсы. Если нигде в стеке нет обработчика исключений, программа аварийно завршается. Для каждого типа ошибок можно создать свой класс и обрабатывать, только их. Можно отловить все ошибки:

```
catch(...) {
    throw;  // т.к. нет переменной исключения,
            // то чтобы бросить выше исполбзуем throw без параметров.
}
```

2.3. Исключения в C++ (лекция Кудинкина)

Некоторый материал ниже повторяется с тем, что написано выше, но для целостности конспекта, пусть будет

C — errno. В C++ exceptions — обработать ошибку не там, где она образовалась.

```
void foo(...) {
    /* ... */
}

void bar(...) {
    foo(/* ... */);
}
```

Как узнать об ошибке на C? Возвращаем код ошибки. Если мы в данной функции не можем обработать значение, то возвращаем его выше, если возращаемое значение используется, то вводим дополнительный аргумент через указатель для ошибки.

`try-catch` в C++:

```
int foo(...) {
    throw 1;
}
```

```
4
5  int bar () {
6      try {
7          foo;
8      }
9      catch (int e) {
10         throw e;
11     }
12 }
```

2.4. Где нельзя выбрасывать исключения

Где нельзя выбрасывать исключения?

2.4.1. try-function block

`try-function` блок подменяет тело функции на try-блок

```
1  void foo() try {
2          // ...
3      } catch (...) {
4          // ...
5  }
```

Есть ли разница между функциями, приведёнными ниже?

```
1  void foo() try {
2          // ...
3      } catch (...) {
4          // ...
5      }
6
7  void foo() {
8      try {
9          // ...
10     } catch (...) {
11         // ...
12     }
13 }
```

Для обычных функций (не конструкторов) разницы нет.

В конструкторах список инициализации идёт после слова try:

```
1  X::X try : i(0), l(1) {
2          // ...
3      } catch (...) {
4          // ...
5  }
```

Разница для конструкторов, что ошибка могла быть ещё на стадии инициализации полей, поэтому обычный `try-catch` внутри тела их не поймает.

```
X::X() try : i(0) {
        int *i = new in[1ll << 63];
    } catch(bad_alloc x) {
        printf(...);
}
```

Заметим, что если ошибка произошла в списке инициализации, то какое-то поле объекта непроинициализированно и мы не сможем установить состояние объекта, нельзя будет вызвать деструктор. На данном уровне мы не сможем обработать ошибку, поэтому нужно выбросить наверх, вместо завершения программы, так как наверху кто-нибудь сможет его обработать.

Если внутри `catch` нет `throw`, то пойманное исключение будет проброшенно наверх.

2.4.2. Вернёмся к исходному вопросу...

Вернёмся к исходному: где мы всё-таки не сможем выбрасывать исключения? — В деструкторе. Пусть где-то произошло исключение, исключение пробрасывается вверх по стеку, все локакльные объекты должны быть уничтожены, будут вызваны деструкторы (stack unwining). Пусть в деструкторе какого-то объекта произошло исключение, теперь 2 исключения, что делать?

Глава 3

STL

STL (Standart Template Library) состоит из:

- Контейнеры (`vector`, `map`, ...) — структуры для хранения динамического количества объектов
- Алгоритмы
- Ввод/вывод

Также в C++ было принято называть заголовочные файлы без `.h`, например: `#include <vector>`. Тогда, подключение стандартной библиотеки C выглядит странно: `#include <stdlib.h>`. При этом просто откинуть для них `.h` нельзя — тогда `string` и `string.h` совпадут. Поэтому сделано было так: `#include <cstdlib>`.

3.1. Пространства имён (namespace)

Как мы знаем, имена разных сущностей не должны совпадать. В связи с ростом количества слов гарантировать их уникальность уже очень сложно, особенно при использовании библиотек. Как с этим бороться?

Можно объявлять некоторые элементы внутри других. Например, вспомним `shared_ptr`. Ему нужен был класс `stoarge` — довольно распространённое имя. Мы могли бы объявить его внутри класса `shared_ptr`, не добавляя его в глобальное пространство имён:

```
class shared_ptr {
    class stoarge {
        //...
    };

    //...
};

//...

shared_ptr::stoarge a;
```

Как видно, класс теперь имеет полное имя `shared_ptr::stoarge`. Теперь `stoarge` без уточнений виден только внутри `shared_ptr`.

Исторически сначала сложился компилятор C++, а потом стандартная библиотека. Из-за этого у многих были свои реализации структур данных. Поэтому возникло обобщение идеи такого сокрытия не только для классов — создание пространств имён:

```
// Complex.h

namespace au {
    class Complex {
        //...
    };
}

//Comlex.cpp

namespace au {
    Complex::Complex(int Re = 0, int Im = 0) {
        //...
    }
}

// или

au::Complex::Complex(int Re = 0, int Im = 0) {
    //...
}
```

Есть правила хорошего тона при использовании пространств имён. Пусть мы хотим написать двумерную матрицу комплексных чисел:

```
#include "Complex.h"

class Matrix {
    au::Complex **data;
}
```

И в конструкторе тоже придётся писать

```
Matrix::Matrix(...) {
    data = new au::Complex[...];
}
```

Как видно, если пространство имён длинное, то писать придётся много. Часть людей считает, что так и нужно, часть считает, что можно использовать более короткий синтаксис:

```
using namespace au; //Всё из пространства имён стало доступно

Matrix::Matrix(...) {
    data = new Complex[...];
}
```

Если в файлах с кодом на это можно согласиться, то в заголовочных файлах такое не поддерживается. Представим, что в проекте появился ещё один класс `Complex`. Тогда при использовании заголовочного файлов вашей матрицы возникнет ошибка (двойное определение), при которой разбираться придётся не автору.

Пространства имён могут быть вложенны:

```cpp
namespace ru {
    namespace spb {
        namespace au {
            class stack;
        }
    }
}

ru::spb::au::stack;
```

Важно отметить, что если внутри пространства имён `ru.spb.au` нужно использовать что-то из глобалного пространства имён, перекрытое чем-то внутри пространства имён, можно сделать `::stack`.

Собственно, вся стандартная библиотека хранится в пространстве имён `std`.

3.2. Линейно упорядоченные контейнеры (`vector`, `list`)

Линейно упорядоченные контейнеры — контейнеры, которые, не внедряясь в реализацию, хранят элементы в линейном постоянном порядке добаления. Требования к типу данных внутри контейнера (должны быть написаны программистом или компилятором следующие публичные элементы):

1. Конструктор копирования
2. Опреатор присваивания
3. Конструктор по умолчанию

3.2.1. `vector`

`vector` хранит элементы в динамическом массиве подряд в одном куске памяти. Его поле `capacity()` описывает текущий размер выделенной памяти, `size()` — количество хранимых элементов. Они не совпадают для избежания частого расширения массива, массив увеличивается редко, но в сколько-то раз (вспоминаем курс алгоритмов).

Интерфейс:

1. Добавить в конец новый элемент за $O(1)$ (амортизированно)
2. Удалить из конца за $O(1)$
3. Вставка и удаление из середины за $O(n)$
4. Доступ к i-му элементу за $O(1)$

```cpp
vector<int> a;
a.push_back(1);
a[0] = 2; // Не проверяет границы индекса, очень быстрый
x = a.at(15); // Проверяет границы, может бросить исключение
std::cout << a.size() << '' '' << a.capacity() << std::endl;
// a.pop_back();

a.reserve(200); // capacity = 200, size = 1
a.resize(500); // capacity = size = 500
```

Задачка: что делает эта строчка?

```
vector<int>(v).swap(v);
```

Ответ: уменьшает capacity до минимально возможного.

3.2.2. Итераторы

Итератор — некоторое обобщение указателя для контейнеров. Они определены внутри соответствующего им класса и поэтому создаются так:

```
class_name::iterator name;
```

Чтобы не писать такую длинную строку, можно использовать `typedef`:

```
typedef class_name::iterator new_name;
```

или auto (начиная с C++11):

```
auto name = other_name.begin();
```

D каждом контейнере, для которого определён итератор, есть методы `begin()` и `end()`, которые возвращают итераторы на начало и послеконец соответственно. У любого уважающего себя итератра должны быть:

1. `operator ++` (инкремент)
2. `operator --` (декремент)
3. `operator *`
4. `operator !=`

В качестве примера можно привести вывод зачений вектора

```
for (auto i = v.begin(); i != v.end(); ++i) {
    std::cout << *i;
}
```

Часто многим методам контейнера можно/нужно передавать в качестве параметра итератор или метод возвращвет итератор на что-либо. Например так:

```
auto it = m.find(4);
m.erace(it);
m.insert(it, 5);
```

для удобства использования есть `reverse_iterator`. В них операторы `++` и `--` инвертированы, а методы контейнера `rbegin()` и `rend()` возвращают `reverse_iterator` на конец и доначало соответственно.

3.2.3. Инвалидация указателей

Стоит помнить, что иногда при добавлении/удалении элемента или иных методах, меняющих расположение элементов в памяти, может пройти инвалидация итераторов. Это может стать причиной баги.

Хотим перед каждым чётным числом вставить нолик.

```
vector<int> v;
v.push_back();

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

Тут возникает вопрос про инвалидацию итераторов. Итератор — некоторая абстракция над указателем, как мы помним. `insert` вставляет элемент ровно перед тем элементом, на который передали итератор.

Было	0	2	1	3	
Стало	0	0	2	1	3

Как видно, после итерации цикла итератор всё ещё указывает на чётный элемент. Это не то, чего мы хотели.

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

Теперь работает, но инвалидация указателей всё ещё сломана. Если было перевыделение памяти, то итератор стал сломан и указывает на некорректную память. Поэтому стоит делать так:

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

3.2.4. string

Есть строки. Есть строки с ASCII— кодировкой

```
typedef basic_string<char> string;
```

Можно объявить свою строку с Unicode через тип **wchar_t**.

Строки очень удобно обрабатывать благодаря перегруженным операторам.

```
string s1 = "Hello, ", s2 = "World!";
string s3 = s1 + s2;
```

Строки работают по принципу Copy-on-Write (CoW). Это означает, что при копировании из одного контейнера в другой будет скопирован только указатель, и он будет так храниться до тех пор, пока контейнер не будет изменён. Только тогда произойдёт выделение новой памяти.

```
string s1 = ''Hello''; // в памяти один раз ''Hello''
string s2 = s1; // в памяти один раз ''Hello''
s2 += ''!''; // в памяти ''Hello'' и ''Hello!''
```

Но, например, такое не сможет сэкономтить память (хотя в Java так работает):

```
string s1 = ''Hello!''; // в памяти один раз ''Hello!''
string s2 = ''Hello!''; // в памяти два раза ''Hello!''
```

3.2.5. deque

deque — по-сути, такой же vector (добавление в конец, доступ к элементу), только ещё добавление и удаление из начала (push_front и pop_front) за амортизированное $O(1)$.

Типовая реализация такая: есть массив указателей на области памяти одинакого размера, а также указатель на начало и конец данных. Если нам нужно перевыделить память, то мы только копируем массив указателей и выделяем новые фрагменты, а старые не трогаем.

3.3. Работа с итераторами в шаблонах

```
template <typename T>
f(T* a, size_t n);

vector<int> a;
f(&(*T.begin()), v.size());
```

Выглядит не очень. У string есть решение — метод c_str() вернёт чистую строчку char *.

3.4. pair

pair — шаблонный класс, позволяющий хранить два элеметна любого типа:

```
pair<int, string> a;
a.first = 20;
a.second = ''Meaow!'';
```

3.5. Ассоциативные контейнеры (set, multiset, map)

3.5.1. set

set представляет собой реализацию массива с поиком. Внутри он реализован как сбалансированное дерево поиска. Именно поэтому почти все операции работают за $O(\log n)$:

- insert
- erase
- lower_bound

- upper_bound
- find

Метод insert возвращает не просто итератор на добавленный элемент. set хранит каждый элемент не более одного раза. Поэтому insert возвразщает pair<set<T>::iterator, bool>, где второй элемент пары указывает, был ли добавлен элемент, а не найден уже находящийся в контейнере.

Итератор у set ассоциативен в том смысле, что идёт не в порядке добаления элементов, а в порядке их возрастания.

```
set<int> s;
s.insert(10);
s.insert(5);
s.insert(1);
// Здесь it++ гораздо менее эффективен
for (set<int>::iterator it = s.begin(); it != s.end(); ++it)
    cout << *it << '' ''; //1 5 10
```

С точки зрения инвалидации, итераторы у set аналогичны list: итератор на элемент, лежащий в структуре, всегда корректен. Именно поэтому просто нельзя делать так (не компилируется, это бы нарушало свойства дерева):

```
for (set<int>::iterator it = s.begin(); it != s.end(); ++it)
    if (*it == 5)
        *it = 6;
```

set<T>::iterator::operator * возвращает T const &. Если всё же нужно поменять значение в множестве, делаем erase() и insert().

Если нужно потроить set() от нашей структуры, в ней необходим оператор сравнения

```
bool operator <(T const &b) const
```

Принято, что достаточно его одного. set сам реализует остальные сравнения через <, хотя это не так эффективно: a = b через !(a < b) && !(b < a).

3.5.2. multiset

multiset аналогичен set, но может хранить равные значения сколь угодно много раз.

Важное отличие: операция multiset<T>::erase может принимать как итератор, так и значение. Первый вариант удалит только одно значение, второй — все равные.

3.5.3. map

map реализует ассоциативный массив. На самом деле он является надстройкой над set<key, value> — мы ищем значение по ключу и храним некоторое значение:

```
map<string, int> age;
age.insert(pair<string, int>(''vaysa'', 18));
```

Чего-то грустно. В utility есть крутая шаблонная функция, и компилятор попытается вывести типы:

```
age.insert(make_pair(''vasya'', 18));
```

Как она устроена?

```
template <typename T1, typename T2>
pair<T1, T2> make_pair(T1 a, T2 b);
```

Ещё немного пар:

```
map<string, int> age;
cout << age.begin()->first; // string const &
```

Всё же, хочется работать удобнее. Есть такой оператор:

```
age[''petya''] = 32;
```

Он, если находит элемент, позволяет его изменить. При этом, если такого ключа не было, он его создаст:

```
int x = age[''peter'']; // 0
```

Поэтому `map::operator []` недоступен у `const map`.

3.6. Comparator-ы

У нас может возникнуть необходимость строить `set` или `map` от нашей струтуры, но с разными порядками. В Си мы это делали бы, передавая указатель на оракул сравнения (компаратор). В ООП мы бы создали наследника класса `Comparator`, и через передачу его экзмепляра и виртуальные методы сравнивать объекты.

С шаблонами придумали несколько свой подход. Мы передаём функтор (объект-функция с перегруженным `operator ()`):

```
template <typename T>
class less {
    bool operator ()(T const &a, T const &b) const {
        return a < b;
    }
}
```

Глава 4

Исключения. Больше исключений

Мы сейчас ещё поговорим о исключениях. Такая же лекция будет про шаблоны. К STL мы вернёмся позже.

4.1. Вспоминим пройденный материал

Вспомним, за что люди любят исключениях. Пусть есть класс

```
class NetworkConnection {
    int* buffer;

    NetworkConnection() {
        buffer = new int[1024];
    }

    ~NetworkConnection() {
        delete[] buffer;
    }

    void Connect(/*...*/);

    //...
};

void startConnection() {
    NetworkConnection nc();
    nc.Connect(/*...*/);
    //...
}

int main() {
    try {
        startConnection();
    }
    catch (NetworkException &e) {
        //...
    }
}
```

Как видим, есть класс `NetworkConnection`, обслуживающий общение с сетью и содержащий буфер. Есть метод `startConnection()`, создающий соединение.

У метода `NetworkConnection::Connect()` могут возникнуть ошибки: адрес компьютера указан неверно или нужный компьютер недоступен. Он при ошибке кидает (`throw`) `NetworkException`. Мы бросаем объект, так как тогда можно передать подробную информацию об ошибке, чтобы пользователь иди программа могла узнать, что произошло, и исправить ситуацию.

Почему это любят?

1. Ошибка передаётся с подробными сведениями
2. Ошибка может быть обработана не там, где произошла.

При создании исключения начинается разкручивание стека (stack unwinding) до тех пор, пока не встретится обработчик (`try`):

`NetworkConnection::Connect()`
`startConnection()`
`main()`

Так `NetworkConnection` может передавать сообщения об ошибке, не зная, где, кто и как его использует и как обрабатывает ошибки.

Важно отметить, что при разкручивании стека вызываются деструкторы всех объектов на фреймах стека. Это поможет нам, например, избежать утечки памяти в `NetworkConnection::buffer`.

Отметим, что блоки у `try catch finally` являются новыми областями видимости, и в них можно создавать локальные переменные. Также, если исключение объект, его лучше в `catch` ловить по ссылке, чтобы не копировать объект исключения лишний раз.

4.2. Разделение исключений

Если у нас большой проект (например, игра), то его часто полезно разделять на отдельные подсистемы, например:

1. Сеть (Network)
2. Модель (Model)
3. Отображение (View)
4. Искусственный интеллект (AI)

В каждой подсистеме логично создать свой тип исключения. Так можно разделить ошибки и их обработку. Сетевую ошибку можно попытаться разрешить, ошибку в модели чаще всего критическая.

Где-то высоко в `main()` есть главный блок:

```
try {
    connect();
    //...
    makeMove();
    //...
}
catch (NetworkException &ne) {
    //вывод сообщения об ошибке, переподключение, ...
}
```

```
catch (ModelException &me) {
    //Послать BugReport
    //Завершить работу
}
```

Но иногда мы хотим получить любую ошибку. Тогда можно сделать так: есть главный родитель `GameException`, а остальных сделать его наследниками.

```
catch (GameException &ge) {
    Log.Write(/* message */);
}
```

Важно заметить, что из всех блоков `catch` выбирается первый подходящий! Если написать `catch (GameException &ge)`, первым, он перехватит все исключения!

4.3. Стандартная библиотека

В C++ есть базовый класс `std::exception`, являющийся предком всех исключений, бросаемых стандартной библиотекой. У него всего один метод `virtual const char* std::exception::what();`, который должен вернуть описание. Собственно, сам `std::exception` не кидают никогда, а кидают его различных наследников (`std::ios::failure` у ввода-вывода, `std::runtime_exception` для критических ошибок, `std::logic_error` для логических и т. д.).

```
try {
    vector.push_back(/*...*/);
    file.open();
}
catch (failure &fe) {
    //...
}
catch (exception &e) {
    //...
}
```

В некоторых случаях решают, что пользовательская система исключений тоже наследуется от `std::exception`.

4.4. RAII

RAII (Resource Acquisition Is Initialization) — идеология, при которой для использования каждого ресурса создаётся объект.

```
while (/*...*/)
    try {
        Animal *pA = read();
        pA->process();
        delete pa;
    }
    catch (ios::failure &fe) {
        //проблема с вводом
    }
```

Пусть мы обрабатываем два класса животных — кошечки и собачки. Виртуальная функция `Animal::process` решает, нужна ли вакцинация, и проводит её, если нужно. У каждого вида животного понимает она по-своему.

Метод `read()` при чтении опознаёт, информацию о каком животном она сейчас читает, и возвращает указатель на соотвествующий новый объект.

Есть проблема — если ошибка при чтении произошла, а объект был создан, то мы его не удалим!

```
while (/*...*/)
    try {
        shared_ptr<Animal> pA;
        read(pA);
        pA->process();
    }
    catch (ios::failure &fe) {
        //проблема с вводом
    }
```

В этом и идея RAII — если ресурс внезапно не нужен, будет вызван его деструктор.

4.5. Исключения в деструкторах

```
class NetworkConnection {
    int* buffer;
    NetworkConnection() {
        buffer = new /*...*/;
    }

    void Connect() {
        //...
        throw NetworkException(/*...*/);
        //...
    }

    ~NetworkConnection() {
        delete[] Buffer;
        ofstream file;
        file.exceptions(ios::badbit | ios::failbit);
        file.open(''log.txt'');
        //запись в файл
    }
}

//...

try {
    NetworkConnection nc();
    nc.Connect();
}
```

В чём скрывается проблема? Пусть произошло исключение. Нужно удалить объект `NetworkConnectio` поскольку он на стеке. Но его деструктор может создать исключение (из `ostream`). Тогда, если бы

возникло два исклчения, не ясно, что делать. В C++ просто запрещено наличие более одного исключения, и поэтому из деструктора запрещено создавать исключения.

4.6. Исключения в конструкторе

```
class BookEntry {
    string name;
    string phone;
    Image *image;
    Audio *track;
    BookEntry();
    ~BookEntry();
};

BookEntry(/*...*/) {
    image = new Image(fname);
    image->Load();
    track = new Track(tname);
    track->Load();
}

~BookEntry() {
    delete image;
    delete track;
}
```

Пусть в конструкторе произошло исключение (в `Image::Load`, например). Тогда деструктор не вызывается, так как объект не считается достроенным. Вот так не прокатит:

```
try {
    BookEntry be(''Vasya'', ''vasya.jpg'', ''vasya.mp3'');
}
catch (/*...*/) {
    //...
}
```

Необходимо в конструкторе освободить ресурсы обратно.

```
BookEntry(/*...*/) {
    try {
        image = new Image(fname);
        image->Load();
        track = new Track(tname);
        track->Load();
    }
    catch (/*...*/) {
        delete image;
        delete track;
    }
}
```

4.7. Объявление возможных исключений

Были предприняты попытки указывать, какие исключения бросает метод

```
void C() throw (A, B) {
//...
}
```

Оно просто проверяет, нужного ли типа исключения, и если нет, аварийно завершает работу. Имеенно из-за такого поведения эта практика не очень рекомендуется.

4.8. Гарантии стандарта

У стандарта есть три типа заявления о исключениях:

No throws: Исключения никогда не бросаются.

Strong: Если исключение и будет создано, объект останется в корректном состояни, в каком был после последней транзакции (последнее цельное состояние объекта).

```
template <typename T>
T stack::pop() {
    if (count == 0)
        throw logic_error();
    else
        return data[--count];
}
```

У этого кода нет гарантии Strong: если копирование при возвращении не удалось, состояние стека будет испорчено. Для этого метод разделили на два: `stack<T>::top()` и `stack<T>::pop()`.

Basic: Если исключение и будет создано, объект останется в **некотором** корректном состояни.

Глава 5

Функторы и typename

5.1. Функторы

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

Для того, чтобы построить set<Person>, нужен порядок. Мы могли бы написать:

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

В чём неприятность? Мы теперь можем сравнивать структуру только по имени. Для избежания этого мы бы могли создать две структуры, но это неинтересно. Вместо этого мы передадим функтор — объект с перегруженным operator ():

```
class by_age {
    bool operator ()(Person const& a, Person const &b) const {
        return a.age < b.age;
    }
}

//...

Person p1, p2;
by_age ba;
ba(p1, p2); // == ba.operator ()
```

У std::set есть дополнительный параметр — класс функтора:

```
// STL
template<typename T, typename Comparator = typename std::less<T>> set;

// Наш код
std::set<Person, by_age> s;
```

5.2. typename

Иногда, при написании шаблонов, внезапно компилятор требует слово **typename**:

```
1 template <typename T>
2 void f(T& container) {
3     typename T::iterator *it;
4 }
```

Для того, чтобы компилятор не подумал, что `T::iterator` — статическая переменная в классе `T`, нужно писать это слово. Сам догадаться он, в большинстве случаев, не может.

Глава 6

Алгоритмы и приведение типов

6.1. Алгоритмы

```
#include <algorithm>
```

Вся идеология STL построена на обобщенном программировании.

Все алгоритмы, которые работают с контейнерами, построены на итераторах (итератор — черная коробочка над указателем).

Базовые требования от итератора:

1. Наличие операций: перемещение вправо/влево `--`, `++`, разыменование `*->`, `*`.
2. `begin()` на начало.
3. `end()` на следующий после конца.

Почему `end()` должен указывать на следующий после конца? Если будет один элемент или ни одного, то для того, чтобы узнать количество элементов в контейнере, потребуется дополнительная проверка, что увеличит время работы.

6.1.1. Постановка задач

Рассмотрим следующий код:

```
1 void foo (It q){
2     T t = *q;
3 }
4
5 int main () {
6     vector<int> v;
7     foo (v.begin ());
8     return 0;
9 }
```

Он не скомпилируется, так как компилятор не знает, что должно быть вместо `T`.

Задача 1: достать тип `It`.

Задача 2: разная реализация в зависимости от типа итератора. Спрашивается зачем нужна разная реализация. Например, нам захотелось написать свой бинпоиск на итераторах. Понятно, что нам не хватит стандартных операций интеретора. Но например для `std::list` мы ничего быстрее, чем просто перебирать все элементы контейнера сделать не можем, а для обычного вектора, в котором все элементы упорядочены по возрастанию/убыванию, хочется сделать быстрее, чем просто тупой перебор.

Начнем решать задачу с контейнеров.

```
// В стандартных контейнерах два шаблонных параметра - тип и аллокатор
// Нужно зафиксировать оба
template<template<class, class> class V, class T, class A>
void foo (const V<T, A> &v) {
    T temp = *v.begin ();
}

int main () {
    list<int> l;
    vector<int> v;
    foo (l);
    foo (v);
}
```

Но такая реализация фиксирует количество шаблонных параметров. Если у нас свой вектор с одним шаблонным параметром — не скомпилируется. Но как же сделанно в STL:

```
template<class T, class allocator = std::allocator>
class vector {
public:
    typedef T value_type;
...
}


template <class Container>
foo (Container& C) {
    typename Container::value_type
        t = *C.back ();
}
```

Сейчас требования для контейнера это, `typedef` и наличие `back()`. Внимание! без `typename` не скомпилируется, компилятор может подумать, что это статическая переменная.

6.1.2. Итераторы

Вернемся к итераторам. Для начала рассмотрим различные типы итераторов в стандартной библиотеке:

1. Random Access Iterator

   Наличие операций: `*->`, `++`, `--`, `-`, `+=`, `-=`.

   Примеры: `vector`, `deque`.

2. Bidirectional Iterator

   Наличие операций: `*->`, `++`, `--`.

   Примеры: `list`, `map`, `multimap`, `multiset`.

3. Forward Iterator

   (a) Output Iterator

   Наличие операций: `++`, `write`.

(b) Inptut Iterator
Наличие операций: `++`, `read`.

А теперь поймем, что мы хотим получить. На примере функции `advance` из стандартной библиотеки.

```
template<class It>
It advance (It q, size_t n) {
    /* хотим :*/
    /* если q --- RA, то */
        q += n;
    /* если q --- Bi, Fi, то */
        for (size_t i = 0; i < n; i++)
            q++;
    return q;
}

int main () {
    std::list<int> l (10, 1);
    std::list<int>::iterator    it1 = advance (l.begin (), 5); // за линию

    std::vector<int> v (10, 2);
    std::vector<int>::iterator  it2 = advance (v.begin (), 5); // за const

    return 0;
}
```

Рассмотрим, что для этого есть в STL. Там написано примерно следующее:

```
struct bidirectional_iterator_tag {};
struct random_access_iterator_tag {};
struct input_iterator_tag {};
struct output_iterator_tag {};
struct forward_iterator_tag {};

template<class T>
class myContainer {
public:
    class iterator {
        typedef bidirectional_iterator_tag iterator_category;
        /* ... */
    }
    /* ... */
}
```

Используя все тоже самое, только из стандартной библиотеки, перепишем нашу функцию. Покрутому будет так:

```
template<class Iter>
Iter advance (Iter q, size_t n, std::random_access_iterator_tag) {
    return q + n;
}
```

```
5
6  template<class Iter>
7  Iter advance (Iter q, size_t n, std::bidirectional_iterator_tag) {
8      for (int i = 0; i < n; ++i)
9          q++;
10     return q;
11 }
12
13 template<class Iter>
14 Iter advance (Iter q, size_t n) {
15     typename Iter::iterator_category tag;
16     advance (q, n, tag);
17     /* можно просто: advance (q, n, Iter::iterator_category ()); */
18 }
```

6.1.3. iterator_traits

Мы написали код, и вроде все работает. Но давайте вспомним про простые указатели ведь код `sort (a, a + n)`, где `a` — массив, тоже работает. В нашей же реализации это получит ошибку компиляции, попробуем это исправить.

Для начала вспомним про частичную специализацию, на примере `vector<bool>`.

```
1  /* основное описание */
2  template<class T>
3  class vector {
4      T* array;
5
6      /* ... */
7  };
8
9  /* частичная специализация */
10 template<>
11 class vector<bool> {
12     int* array;
13
14     /* ... */
15 };
```

Теперь создадим `class iterator_traits`, и у него сделаем частичную специализацию для указателей. Например это можно сделать так:

```
1  /* тут например будет работать iterator_traits<vector<int>::iterator> */
2  template<class Iter>
3  class iterator_traits {
4  public:
5      typedef typename Iter::value_type value_type;
6      typedef typename Iter::iterator_category iterator_category;
7  };
8
9  /* а тут будет работать iterator_traits<T*> */
10 template <class Iter>
11 class iterator_traits<Iter*> {
```

```
12 public:
13     typedef Iter value_type;
14     typedef std::random_access_iterator_tag iterator_category;
15 };
16
17 template<class Iter>
18 Iter advance (Iter q, size_t n) {
19     advance (q, n, typename iterator_traits<Iter>::iterator_category ());
20 }
```

В стандартной реализации кроме полей `value_type` и `iterator_category`, есть и другие например `pointer`.

6.2. Приведение типов

Новые операторы:

1. `const_cast`
2. `reinterpret_cast`
3. `static_cast`
4. `dynamic_cast`

Первые три происходят во время компиляции, четвёртый — в момент работы программы.
`const_cast` позволяет менять наличие `const` у типа, точнее, снимать его:

```
1 void f(const char* p)
2 {
3     char *p1 = const_cast<char*>(p);
4 }
```

Это, кстати, может приводить к падению программы:

```
1 const char s[] = ''Hello!'';
2 char s2* = const_cast<char*>(s);
3 s2[0] = 'h'; // падение, так как строчка лежит в защищённой области памяти.
```

Зачем это вообще? Предположим, что вы используете библиотеку, которая была написана ещё когда слово `const` не существовало. Для вызова такой функции из современного кода придётся использовать `const_cast`. Сам `const_cast` эквивалентен простому привидению типа указателя в Си, его легче видеть и находить в коде.

`reinterpret_cast` позволяет приводить один тип к абсолютно другому, просто перевоспринимая память, отведённую под первый тип, как второй. Если вспомнить интрузивные списки, то именно этот приём мы хотели там применить. Работает оно ровно так же, как в Си, но опять-таки это легче воспринимать и находить в коде.

`static_cast` используется для:

1. Приведения типов между примитивными типами:

```
1 double d = 1.5;
2 int i = static_cast<int>(d);
```

Между указателями на совсем разные типы это уже не работает.

2. Привидение указателей в/из `void *`
3. Привидения типов, описанные пользователем.
4. Приведение указателей на разные классы, отслеживая, можно ли приводить (является ли один класс предком другого). Пусть у класса `A` есть два наследника `B` и `C`, а также сторонний класс `D`.

```
B* pB = new B();
A* pA = static_cast<A*>(pB);
D* pD = new D();
pa = static_cast<D*>(pD); // Ошибка компиляции

C* pC = static_cast<C*>(pA); //Компилируется, но некорректно
```

Заметим, что в последнем случае действие могло быть корректным, если бы `pA` указывал на `C`, но на этапе компиляции, в общем случае, проверить это невозможно. Вообще говоря, это нормальная практика — мы передаём по указателю на родителя некоторого наследника, и, если мы уже знаем, кто он, то можем привести тип.

Перед тем, как займёмся `dynamic_cast`, рассмотрим его механизм — RTTI (Run-Time Type Information). Он позваляет проверить по указателю на некоторый класс, какой же он на самом деле, но только в одном случае — если у класса есть хотя бы один виртуальный метод. Тогда, как мы помним, у класса есть таблица виртуальных функций, которая хранится одна на класс. Тогда RTTI добавляет туда поле: структуру `type_info`:

```
#include <typeinfo>

const type_info& ti = typeid(*pB);
cout << ti.name();
```

Вы получите имя класса из объектного класса. Его точное имя не получите, так как работает манглинг (mangling). У структуры так же определены операторы `==` и `!=`:

```
if (ti == typeid(B))
    //...
```

Зачем это использовать? Ну пусть есть список элементов интерфейса и все отрисовать, причём код их отрисовки зависит от системы, и поэтому в модели (Model) лежать не должен. Мы просто для каждого выясняем его тип и обрабатываем. Получается немного убого, гораздо лучше эту задачу решает шаблон Посетитель (Visitor pattern).

Вернёмся к `dynamic_cast`:

```
A* pA = new B();
C* pC = static_cast<C*>(pA); // Некорректно
C* pC = dynamic_cast<C*>(pA); // Или отработает,
                              // Или запишет NULL
if (pC != NULL)
    //...
```

6.3. mutable

Представим, что у нас есть матрица, и у неё хотим считать определитель. Если мы будем много раз вызывать этот метод, то каждый раз он будет вычисляться, что затратненько. Предположим, что мы кешируем значение определителя:

```
class Matrix {
private:
    //...
    bool isUpdated; //менялась ли матрица?
    double det;
public:
    //...

    void set(int i, int j, double x) {
        //...
        isUpdated = true;
    }

    double determinant() const {
        if (isUpdated) {
            det = //...
            isUpdated = false;
        }
        return det;
    }
};
```

Всё бы хорошо, но... `const` у определителя уже ставить нельзя. Можно бы убрать, но тоже не хорошо: у константных объектов нельзя узнать определитель. Тогда можно просто указать `mutable` у наших полей: оно означает, что эти поля можно менять даже из константных методов:

```
class Matrix {
    //...
    mutable bool isUpdated;
    mutable double det;
    //...
};
```

Глава 7

c++11

7.1. Новые ключевые слова

7.1.1. Ключевое слово default

```
class A {
public:
    A (int);
    A () = default;
};
```

Разумное использование:

```
class B {
public:
    B (int);
protected:
    B (const &B) = default;
};
```

7.1.2. Ключевое слово delete

Предназаначение удалять что-то из класса, чтобы это нельзя было использовать. Следующий код запретит не все:

```
class A {
public:
    A (const &A);
    f ();

    friend g();
};
```

В функциях `g` и `f` все еще можно будет использовать приведение типов.

Поэтому надо использовать `= delete`:

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

7.1.3. Ключевое слово override

`override` строго говорит что функция будет перекрыта.

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

/* в Derived.h*/
class Derived: public Base {
public:
    void f (int) = override; // Ok
    virtual int g () = override; // Error
};
```

Функции `f` — одинаковые и вторая перекрывает первую, функции `g` — разные, так как одна реализована для `const` класса.

7.1.4. Ключевое слово `final`

Вспомним пример про рабочих и их зарплату. Слово `final` говорит что функцию уже нельзя будет перекрывать. Можно писать где первая строка описания класса.

```
class Worker {
    virtual int getSalary () = 0;
};

class Developer: public Worker {
    virtual int getSalary () = override, final {
        return 100;
    }
};

int main () {
    Worker *w;
    /* считываем n */
    /* динамическое связывание */

    if (n > 1)
        w = Developer ();
    else
        w = Tester ();
    w->getSalary ();

    /* статическое связывание */
    Developer *d = new Developer;
```

```
24      d->getSalary ();
25  }
```

7.1.5. Ключевое слово nullptr

```
1   f(long);
2   f(char *);
3
4   int main () {
5       f (NULL); //error
6       f (0l); /* будет вызывать f(long) */
7       f ((char*)NULL); /* будет вызывать f(char*) */
8       f (static_cast<char*>(NULL)); /* будет вызывать f(char*) */
9       f (nullptr); /* будет вызывать f(char*) */
10  }
```

7.1.6. Конструкторы

```
1   class Board {
2   public:
3       static const size_t BOAD_SIZE = 8;
4       int board [BOARD_SIZE];
5       int x = 10; // c++11
6       int y = 10; // c++11
7   };
```

7.1.7. constructor chaining

Позволяет использовать нормально в одном контрукторе другие.
Как могли бы написатью

```
1   class Complex {
2   private:
3       int myRe;
4       int myIm;
5   public:
6       Complex (int re = 0, int im = 0);
7       Complex (int* arr) {
8           Complex (arr[0], arr[1]);
9           /* будет создан новый Complex и удалится в той же строке*/
10      }
11  }
```

Как надо:

```
1   class Complex {
2   private:
3       int myRe;
4       int myIm;
5   public:
6       Complex (int re = 0, int im = 0);
```

```
    Complex (int* arr): Complex (arr[0], arr[1]) {}
}
```

Но конструктор может бросить исключение. Поэтому перепишем следующим образом:

```
class Complex {
private:
    int myRe;
    int myIm;
public:
    Complex (int re = 0, int im = 0);
    Complex (int* arr) try: Complex (arr[0], arr[1]) {
        //...
    }
    catch (){

    }
}
```

7.1.8. Списки инициализации

```
stuct Point {
    int x;
    int y;
};

int main () {
    int arr[] = {3, 6, 9};
    Point p {0, 1};
    //C++11
    vector<int> a {3, 5, 6};
}
```

`vector<int> a {3, 5, 6};` по этой штуке компилятор генерирует `std::inirializer_list`.

```
template<typename T>
class vector {
public:
    vector (std::initializer_list<T> l) {
        size_t size = l.size ();
        const T* it = l.begin ();
        for (int i = 0; i < size; ++i)
            cout << *(it++); //it[i]
    }
};
```

7.2. Еще нововведения

7.2.1. auto

Во время компиляции.

```
std::vector<int>::iterator it = v.begin ();
auto it = v.begin ();
int x = 3;
auto y = x;
```

7.2.2. decltype

```
template<typename T, typename V>
void multiply (const T& t, const V& v) {
    typedef decltype (t * v) res_t;
    res_t r = t * v;
}

//хотелось, но нельзя, так как t, v раньше объявления находятся
template<typename T, typename V>
decltype (t * v) multiply (const T& t, const V& v) {
    auto r = t * v;
    return r;
}

//поэтому так
auto multiply (const T& t, const V& v) -> decltype (t * v) {
    auto r = t * v;
    return r;
}
```

7.2.3. Различия auto и decltype

Пример 1:

```
int main () {
  int x = 5;
  int& rx = x;
  auto y1 = rx; // int
}
```

Будет непонятно, чем является y1, ссылкой или обычным переменной типа int. На самом деле auto снимает ссылку. Но если мы четко хотим указать, что хотим получить ссылку, то надо просто написать auto& y1 = rx.

Пример 2:

```
int main () {
    int x = 5;
    const int& crx = x;
    auto y1 = crx; // int
    auto &y2 = crx; // const int &
}
```

Компилятор всегда пробует снять константность.

7.2.4. rvalue references, move semantic

1. rvalue — может стоять только в правой части выражения
2. lvalue — нечто с именем, нечто, от чего можно ввзять ссылку, может стоять и в левой и в правой части выражения

```
int main () {
  int a = 3; // l = r
  int b = 5; // l = r
  a = b; // l = l
  b = a; // l = l
  a = a * b; // l = r
}
```

Так же `rvalue` — являются все временные значения Но все же `const int` — `lvalue`.

Рассмотрим некоторый класс `X`, у него есть:

1. конструктор копий
2. оператор присваивания
3. деструктор
4. `X::pRes` — некоторый ресурс, которым владет объект.

Как работает следующее выражение `a = b`:

```
operator=(const X& rhs) {
    // освободить this->pRes
    // выделить память
    // скопировать из rhs.pRes в this->pRes
}
```

Как работает следующий код:

```
X foo () {
    //...
}

int main () {
    X a = foo (); // copy constructor
}
```

Здесь неявно создаётся временная переменная (назовём её `tmp`), в которую записывается результат `foo`, а потом из неё копируется в `a`. Но можно сделать быстрее, просто поменяв указатели на память у `a` и `tmp`. Научимся так делать.

Теперь у нас будет два конструктора копий и два оператора присваивания: первые когда в правой части выражения стоит `lvalue`, а вторые когда в правой части стоит `rvalue`.

Чтобы ловить `rvalue` используем `rvalue reference`, выглядит так: `X&&`

Синтаксис оператора присваивания для `rvalue`:

```
X& operator= (X&& rhs) {
  T* tmp = this->pRes;
  this->pRes = this->pRes;
  rhs.pRes = tmp;
  /* или swap (this->pRes, rhs->pRes) */
  return *this;
}
```

Итак, мы написали сейчас следующие методы:

```
  X (const X&);
  X (X&&);
  X& operator= (const X&);
  X& operator= (X&&);
```

Теперь вспомним про обычный `swap`.

```
template<class T>
void swap (T& a, T& b) {
  T tmp (a);
  a = b;
  b = tmp;
}
```

Но этот swap не будет использовать наши новые конструкторы (т.к. мы инициализируем именованной переменной — `lvalue`) Чтобы превратить `lvalue` в `rvalue` надо использовать `std::move`, которое возвращает `rvalue`.

Перепишем `swap`:

```
template<class T>
void swap (T& a, T& b) {
  T tmp (std::move (a));
  a = std::move (b);
  b = std::move (tmp);
}
```

Пример:

```
    vector<string> v;
    v.push_back (string (''Hello''));
    string s (''World'');
    v.push_back (std::move (v));
```

Этот приём, когда мы `lvalue` превращаем в `rvalue` начинаем по сути перемещать объект, а не копировать и называется `move semantic`.

Замечание 7.2.1. Хотим оптимизировать возврат, но писать return `std::move(x)` не нужно, т.к. компиляторы давно делают *return value optimization* (Возвращаемое значение создаётся в кадре стека вызывающей функции)

```
X foo() {
    X x;
    return x;
}
```

7.2.5. placement new

Это особоя форма **new**, которая позволяет в выделенную память размещать объект с помощью конструктора. Так же полезно, когда у класса не существует конструктора по умолчанию и при этом требуется создать массив объектов. Синтаксис следующий:

```
#include <new>

class Packet {
private:
    int info;
public:
    Packet (int info = 0) : info (info) {}
};

int main () {
    char* buffer = new char[sizeof (Packet)];
    Packet *p = new (buffer) Packet();

    p->~Packet ();
    delete [] buffer;
}
```

Теперь чтобы удалить выделенные объекты, сначала нужно от каждого выделенного объекта вызвать деструктор, а затем удалить выделенную для них память, то что и происходит в последних строках, приведенного выше примера. На самом деле и без строки `p->~Packet ();` будет все работать, так как в внутри объекта не выделяется память, но так делать не рекомендуется.

7.3. lambda calculus

7.3.1. for_each

```
#include <algorithm>

template<class InputIt, class Function>
Function for_each (InputIt first, InputIt last, Function fn) {
    while (first!=last) {
        fn (*first);
        ++first;
    }
    return std::move (fn);
}
```

Как видно по коду `for_each` применяет функцию `fn` ко всем элементам, находящимся в диапазоне $[first, last)$. Примерно такая реализация этой функции в стандартной библиотеке.

Пример использования:

```
#include <algorithm>
#include <vector>

void f (int &n) {
```

```
    n = n * n;
}

class Func {
public:
    void operator () (int &n) {
        n = n * n;
    }
};

int main () {
    std::vector<int> v = {1, 3, 5, 7};
    for_each (v.begin (), v.end (), f);
    for_each (v.begin (), v.end (), Func ());
    return 0;
}
```

Два вызова `for_each` делают абсолютно одинаковые вещи, просто реализованы по-разному, за исключением того, что у функтора есть свои преимущества.

7.3.2. transform

```
template<class InputIt, class OutputIt, class UnaryOperation>
OutputIt transform (InputIt first1, InputIt last1, OutputIt d_first,
                    UnaryOperation unary_op) {
    while (first1 != last1) {
        *d_first++ = unary_op(*first1++);
    }
    return d_first;
}
```

`transform` применяет оператор `unarry_op` ко всем элементам, находящимися в диапазоне $[first1, last1$ и соохраняет результат в диапазоне, начинающимся с d_first.

Пример использования:

```
#include <algorithm>
#include <vector>

double f (const double &d) {
    if (d < 0.001)
        return 0;
    return d;
}

class Func {
    double eps;
public:
    Func (double eps) : eps (eps) {}

    double operator () (const double &d) {
        if (d < eps)
```

```
            return 0;
        return d;
    }
};

int main () {
    std::vector<double> v = {1, 0.1, 0.01, 0.001, 0.0001};
    transform (v.begin (), v.end (), v.begin (), f);
    transform (v.begin (), v.end (), v.begin (), Func (0.01));
    return 0;
}
```

В данном случае функтор получился функциональнее обычной функции. Так же константные ссылки не обязательны и можно спокойно менять значения, которые передаются в функтор или функцию.

Бонус. `transform` так же определен и для `BinaryOperation`.

Примерная релизация:

```
template<class InputIt1, class InputIt2,
         class OutputIt, class BinaryOperation>
OutputIt transform(InputIt1 first1, InputIt1 last1, InputIt2 first2,
                   OutputIt d_first, BinaryOperation binary_op) {
    while (first1 != last1) {
        *d_first++ = binary_op(*first1++, *first2++);
    }
    return d_first;
}
```

7.3.3. lambda function

Лямбда функции призваны уменьшить количество символов, так как позволяют записывать функторы более кратко.

Посмотрим синтаксис:

```
#include <vector>
#include <algorithm>

int main () {
    std::vector<double> v = {0.1, 0.01, 0.001, 0.0001};

    for_each (v.begin (), v.end (),
        [] (double &d) {
            if (d < 0.001)
                d = 0;
        }
    );

    transform (v.begin (), v.end (), v.begin (),
        [] (const double &d) -> double {
            if (d < 0.001)
                return 0;
```

```
            return d;
        }
    );
    return 0;
}
```

Описанные лямбда функции фактически соответствуют тем функторам, которые были написаны в двух предыдущих пунктах про `for_each` и `transform`.

Разберемся теперь в синтаксисе. В квадратных скобках `[]` указан `capture_list`. Он позволяет передавать какие-то переменные в функтор. Рассмотрим на примерах:

1. `[a, &b]` — в этом случае `a` захватывается по значению, а `b` по ссылке.
2. `[&]` — захватывает все переменные по ссылке.
3. `[=]` — захватывает все переменные по значению.
4. `[]` — ничего не захватывает.

В круглых скобках, описываются такие же параметры как и в функторах. Между круглой закрывающейся скобкой и `->` могут быть ключевые слова. После `->` указывается возвращаемое значение лямбда функции.

Важное замечание, если захватывать по значению, то переменные будут доступны как `const`. Чтобы это исправить следует использовать `mutable`, такой синтаксис выглядит так:

`[=] (const double &d) mutable -> double`.

Еще пример:

```
#include <vector>
#include <algorithm>

int main () {
    std::vector<double> v = {0.1, 0.01, 0.001, 0.0001};

    int epsilon = 0.001;
    for_each (v.begin (), v.end (),
        [epsilon](double &d) -> void {
            if (d < epsilon)
                d = 0;
        }
    );

    return 0;
}
```

7.4. std::array

```
std::array<int> a (10);
```

7.5. Функции begin и end

```
int res = accamulate (begin (a), end (a), 0, sum);
```

7.6. foreach

Синтаксический сахар.

```
int main () {
  vector <int> {1, 2, 3};
  for (auto x : v) {
    cout << x << ''\n'';
  }
  for (auto it = v.begin (); it != v.end (); ++it) {
   cout << x << ''\n'';
  }

}
```

Первый цикл развернеться во второй.

7.7. Новые контейнеры

7.7.1. forward_list

Односвязный список.

7.7.2. unordered_set и unordered̀map

Как сделать хеш, для своего объекта.

```
#include <unordered_set>

class A {
  int x;
  string s;
  bool operator=(const A& rhs) {
    return rhs.a = a;
  };
  friend size_t std::hash <A>;
};

unordered_set<class T, class hash_func = std::hash<T>, class pred = equal_to<T>>;

namespace std {
  template<>
  struct hash<A> {
    size_t operator ()(const A& rhs ) const {
      size_t val = 0;
      val += std::hash<int> () (rhs.x) * 31;
      val += std::hash<string> () rhs.s * 59;
    }
  };
};

```

```
25 int main () {
26
27
28 }
```

7.8. VARIADIC TEMPLATES

Хочется сделать функцию с произвольным числом параметров.
Хочется делать:

```
1 int main() {
2         int r = sum(1, 2, 3);
3         r = sum(1, 2);
4 }
```

Так же, как у крутых парней типа printf, scanf.
Механизм будет напоминать рекурсию.

```
1  template<typename T, typename ... Args>
2  // '...' before Args means that the number of arguments may be any
3  T sum(T n, Args ... rest) {
4      return n + sum(rest ...);
5  }
6  /* macros PRETTY_FUNCTION gives us the name
7  of the function with arguments and its return value */
8
9  template<typename T>
10 T sum(T n) {
11     return n;
12 }
```

Как сделано в printf:

```
1  void printf(const char* fmt, ...) {
2      va_list args;
3      va_start(args, fmt); // макрос, настраивает args на
4                           // смещение, равное размеру fmt
5                           // (fmt здесь в том смысле, что это
6                           // последний аргумент перед '...')
7      while(...) {
8          int i = va_arg(args, int); //тут вторым параметром может быть любой
9                                     // примитивный тип в зависимости от args кастуется
10                                    // к нужному типу и на сколько нужно смещается по стеку
11     }
12     va_end(args);
13 }
```

Написать таким механизмом правильно работающий код достаточно сложно, конечно, особенно с учетом того, что пользователь умеет радостно ошибаться, и это надо как-то отслеживать.

Глава 8

Метапрограммирование

Вспомним traits.
Что такое метапрограммирование?

1. Программа при компиляции может создать свою новую часть.
2. Программа при работе меняет саму себя.

В C++ реализуема только первая часть, вторая касается больше полиморфных вирусов и Java.

Как мы могли уже написать один код, чтобы был доступен разный функционал без дублирования кода?

Указатель на фунцию Так, как в Си делали.

Виртуальные функции Наследники реализуют конкретный функционал.

У обоих подходов есть проблема производительности: идёт вызов функции, идёт расчёт, что вызывать.

Обобщённое программирование (шаблоны) Тут всё быстрее, так как всё происходит в момент компиляции. Вспомним прошлые лекции: мы написали код, порождающий функции и структуры:

```
int sum();
int sum(int);
int sum(int, int);
// ...
```

Или вот в момент компиляции считаем факториал:

```
template<int n>
struct fact {
    static const int value = n * fact<n - 1>::value;
};
template<>
struct fact<0> {
    static const int value = 1;
};
```

Что такое trait? trait — объект, хранящий информацию о деталях реализации другого типа. Это исползуется для различных реализации для различных типов.

Например, шаблон принимает параметр: численный тип. Файл `limits.h` содержит макросы, объявляющие константы. Пока нам нужно просто писать специализации, что грустно.

Но в C++ есть стандартная структура-шаблон `std::numeric_limits`, которая принимает тип-число и содержит методы:

```
template<typename T>
struct std::numeric_limits<class T> {
    static T max();
    //...
    static const bool is_integer = false;
    //...
};

template<>
struct std::numeric_limits<char> {
    static T max() {
        return CHAR_MAX;
    };
};
```

Чуда не произошло, эти специализации всё ещё нужно писать, но теперь не в каждом алгоритме:

```
T largest = std::numeric_limits<T>::max();
```

Вопрос на подумать: а зачем `max()` функция?

```
template<typename T>
struct is_pointer {
    static const bool value = false;
};

template<typename T>
struct is_pointer<T*> {
    static const bool value = true;
};

//...

bool b = is_pointer<T>::value;
if (b) {
    //...
}
else {

}
```

Мы хотим, чтобы тут даже не доходило до компиляции ненужной ветки `if`-а. В Си это выглядело бы так:

```
void f(int a) {
    //...
}

void f_optimized(int a) {
```

```
    //...
}

void real_f(int a) {
    #ifdef _SUPPORT_MMX
        f_optimized(a);
    #else
        f(a);
    #endif
}
```

Мы не хотим препроцессор.

Как это делает старый стандарт:

```
template<bool b> // включать ли быструю версию
struct alg_selector {
    template<typename T>
    static void implementation(T& obj) {
        // медленная реализация
    }
};

template<>
struct alg_selector<true> {
    template<typename T>
    static void implementation(T& obj) {
        // быстрая реализация
    }
};

template<>
void algorithm(T& obj) {
    alg_selector<is_optimized<T>::value>::implementation(obj);
}

// struct ObjectB оптимальная, struct ObjectA нет

template<typename T>
struct is_optimized {
    static const bool value = false;
}

template<>
struct is_optimized<ObjectB> {
    static const bool value = true;
}

int main() {
    ObjectA a;
    ObjectB b;
    algorithm(a);
```

```
38      algorithm(b);
39  }
```

На экзамене попросят написать реализацию выше с чем-то содержательным. Кстати, в стандартной библиотеке `#include <type_traits>` уже есть много подобных фич.

Есть механизм SFINAE (Suptitiution Failure Is Not An Error).

```
1   template<typename T>
2   struct has_iterator {
3       template <typename U>
4       static char test(typename U::iterator x);
5
6       template <typename U>
7       static long test(U* x);
8
9       static const bool value = sizeof(test<T>(0)) == 1;
10  }
11
12  int main() {
13      int a = 128;
14
15  }
```

Там выше есть бага, но мы позже разберёмся. Суть в том, что можно на уровне языка проверять наличие каких-то типов, методов и полей у структур и прочего.

Теперь новый стандарт. На принципе выше есть конструкция `std::enable_if`:

```
1   template<typename T>
2   typename enable_if<!has_iterator<T>::value, void>::type show(T& x) {
3       cout << x << ''\n'';
4   }
5
6   template<typename T>
7   typename enable_if<has_iterator<T>::value, void>::type show(T& x) {
8       for (auto& i: x)
9           cout << i;
10  }
```

Что делает `std::enable_if`? Оно принимает два параметра — `bool` и некий тип. Если выражение итинно, то `std::enable_if::type` просто равен второму параметру. Иначе, подстановка не удаётся через SFINAE (подстановка не удаласть, но нет ошибки компиляции), и метод не создаётся.

В любом случае, будет создана только одна интанация метода `show` — только та, у которой условие верно.

Глава 9

Threads

Программа - процесс. Внутри можно запустить несколько функций на праллельное исполнение - потоки. Внутри ОС есть планировщик. Внутри некотарая очередь, где каждому потоку предоставляется ограниченный период времени, в течение, которого он выполняется. Затем управление переходит к другому потоку.

Когда поток выполняется, он задействует регистр процессора. Содержание регистра называют контекстом. Когда происходит переключение потока, нужно сохранить контекст старого потока, остановить выполнение, и скопировать контекст нового и запустить его.

Если бы мы только ограничивались вычислительными задачами, то последоватьельное выполнение было бы выгоднее, т.к. на переключение потоквв тратится время. Однако, мы занимаемся не только счётом, многие задачи, тогда решить не возможно (например gui), однако есть моменты, когда процессор простаивает вовремя работы перефирийных устройств, когда работают их микроконтролеры, если переключаться в эти моменты, то проигрыш во времени будет незаметен

Есть некоторые классы задач, которые разумно параллелить и нет. Например разумно параллелить умножение матриц, но не разумно параллелить алгоритм Евклида.

9.1. thread

Рассмотрим синтаксис. Строка компиляции: `g++ -std=c++11 ex01.cpp -lpthread -o main`

```
#include <thread>
#include <iostream>
#include <vector>

void hello () {
  std::cout << ''Hello from '' << std::this_thread::get_id () << ''\n'';
}

int main () {
  std::vector<std::thread> threads;
  for (int i = 0; i < 10; ++i) {
    threads.push_back (std::thread (hello));
  }

  for (auto& t:threads) {
    t.join ();
  }

```

```
19    return 0;
20  }
```

1. в этом примере на самом деле 11 потоков - в 11-ом выполняется функция main();
2. порядок выполнения всех 11 потоков не определён.
3. Нужно дождаться окончания потоков. Если main завершится раньше - segfault.

```
1       for (auto& t : threads) {
2           t.join();
3       }
4       return 0;
```

В параметрах у `thread` можно передавать функтор, `void*` и лямбда-выражения. Сейчас мы можем увидеть вместо строчек `Hello from 139744734836480`, например `Hello from Hello from 139752011310848`. Это происходит из-за того, что поток может прервать выполнение в любой момент.

9.2. Атомарные операции

Существуют атомарные операции, которые на какой-то архитектуре не могут приостановиться посередине их выполнения. Например чтение из памяти или запись в память.

Пример прерывания инкремента int.

```
1   #include <thread>
2   #include <iostream>
3   #include <vector>
4
5   class Counter {
6     int value = 0;
7
8   public:
9     void increment() {
10      ++value;
11    }
12
13    int get_count () {
14      return value;
15    }
16  };
17
18  int main () {
19    Counter counter;
20    std::vector<std::thread> threads;
21    for (int i = 0; i < 10000 ; ++i) {
22      threads.push_back (std::thread ([&counter] () {
23          for (int i = 0; i < 10000; ++i) {
24            counter.increment ();
25          }
26      }));
```

```
27    }
28
29    for (auto& t:threads) {
30      t.join ();
31    }
32
33    printf (''%d'', counter.get_count ());
34    return 0;
35  }
```

На экране мы скорей всего увидим не 100000000, а что-то меньше.

Так как мы вряд ли хотим чтобы такое происходило, нучимся делать операцию `++value;` атомарной.

Чтобы это исправить используем `mutex`.

```
1   #include <thread>
2   #include <mutex>
3   #include <iostream>
4   #include <vector>
5
6
7   class Counter {
8     int value = 0;
9
10  public:
11    void increment() {
12      ++value;
13    }
14
15    int get_count () {
16      return value;
17    }
18  };
19
20  class CounterConcurrent {
21    Counter &c;
22    std::mutex m;
23
24  public:
25    CounterConcurrent (Counter &_c) : c(_c){
26    }
27
28    void increment() {
29      m.lock ();
30      c.increment ();
31      m.unlock ();
32    }
33  };
34
35
36  int main () {
```

```
  Counter counter;
  CounterConcurrent counterc(counter);
  std::vector<std::thread> threads;
  for (int i = 0; i < 10000 ; ++i) {
    threads.push_back (std::thread ([&counterc] () {
        for (int i = 0; i < 10000; ++i) {
          counterc.increment ();
        }
    }));
  }

  for (auto& t:threads) {
    t.join ();
  }

  printf (''%d'', counter.get_count ());
  return 0;
}
```

Отлично рассмотрим пример когда, `cpounter` бросает исключения.

```
#include <thread>
#include <mutex>
#include <iostream>
#include <vector>


class Counter {
  int value = 0;

public:
  void increment() {
    ++value;
  }

  void decrement() {
    --value;
    if (value == 0) {
      throw std::runtime_error (''decrement'');
    }
  }

  int get_count () {
    return value;
  }
};

class CounterConcurrent {
  Counter &c;
  std::mutex m;

```

```
public:
  CounterConcurrent (Counter &_c) : c(_c){
  }

  void increment() {
    m.lock ();
    c.increment ();
    m.unlock ();
  }

  void decrement() {
    m.lock ();
    c.decrement ();
    m.unlock ();
  }
};


int main () {
  Counter counter;
  CounterConcurrent counterc(counter);
  std::vector<std::thread> threads;
  for (int i = 0; i < 5000 ; ++i) {
    threads.push_back (std::thread ([&counterc] () {
        for (int i = 0; i < 10000; ++i) {
          counterc.increment ();
        }
    }));

    threads.push_back (std::thread ([&counterc] () {
        for (int i = 0; i < 10000; ++i) {
          counterc.decrement ();
        }
    }));
  }

  for (auto& t:threads) {
    t.join ();
  }

  printf (''%d'', counter.get_count ());
  return 0;
}
```

Беда в том, что такой код вероятнее всего кинет исключение. Чтобы это исправить надо применить идиому `RAII`. И обернуть `mutex` в `lock_guard`.

Пишется это так:

```
lock_guard<mutex> lg(m);
c.decrement ();
```

Другие виды `mutex`.

1. `time_mutex`

   `try_lock_for`

   `try_lock_until`

2. `recursive_mutex`

9.3. Так же сделать

1. `producer_consumer` `conditional_variable`
2. std::bind, ref, function