

Информатика. Семинар №8 + №9

Сколько раз выполнится тело цикла?

```
for (int i(0); i < 100; ++i)
{
    std::cout << i << " ";
    i++;
}
```

Сколько раз выполнится тело цикла?

```
for (int i(0); i < 100; ++i)
{
    std::cout << i << " ";
    i++;
}
```

Ответ: 50

Каким будет значение переменной k
после вызова ф-и f?

```
void f(int& a)
{
    a++;
}

int main()
{
    int k = 0;
    f(k);
}
```

Каким будет значение переменной k
после вызова ф-и f?

```
void f(int& a)
{
    a++;
}

int main()
{
    int k = 0;
    f(k);
}
```

Ответ: 1

Чему эквивалентна (не в объявлении) запись `a[5]` ? где `a` - это указатель или массив и `operator[]` не перегружен

☐

`*(a + 5)`

☐

`5[a]`

☐

`*a + 5`

☐

`(a + 3)[2]`

☐

`*a[5]`

☐

`(a + 2)[3]`

Чему эквивалентна (не в объявлении) запись `a[5]` ? где `a` - это указатель или массив и `operator[]` не перегружен



`*(a + 5)`



`5[a]`



`*a + 5`



`(a + 3)[2]`



`*a[5]`



`(a + 2)[3]`



Скомпилируется ли следующий код?

```
class X {};  
class Y : X {};  
  
int main()  
{  
    X* x = new Y;  
}
```


Скомпилируется ли следующий код?


```
class X {};  
class Y : X {};  
  
int main()  
{  
    X* x = new Y;  
}
```

Нет: наследование private по умолчанию для class'ов, поэтому для конструктора Y конструктор X оказывается private ⇔ недоступен

Что выведет программа?

```
struct A { A() { cout << "A"; } };  
struct B { B() { cout << "B"; } };  
struct C { C() { cout << "C"; } };
```


```
A a;
```

```
int main() {  
    B b;  
}  
C c;
```

Что выведет программа?

```
struct A { A() { cout << "A"; } };  
struct B { B() { cout << "B"; } };  
struct C { C() { cout << "C"; } };
```

```
A a;
```

```
int main() {  
    B b;  
}  
C c;
```

Ответ: ACB (глобальные
переменные до main)

Что выведет программа?

```
struct A {  
    std::string s;  
    auto p;  
  
    A() { cout << "A"; }  
    A(const std::string& s_, auto p_) :  
        p(p_), s(s_) {}  
    ~A() { cout << "~A"; }  
};  
  
int main() {  
    A("file", "c:/tmp/file.txt");  
}
```

- A
- ~A
- A~A
- Ничего
- Не скомпилируется

Что выведет программа?

```
struct A {  
    std::string s;  
    auto p;  
  
    A() { cout << "A"; }  
    A(const std::string& s_, auto p_) :  
        p(p_), s(s_) {}  
    ~A() { cout << "~A"; }  
};  
  
int main() {  
    A("file", "c:/tmp/file.txt");  
}
```

- A
- ~A
- A~A
- Ничего
- **Не скомпилируется**

1) Не смогли
вывести тип для
auto

2) Список
инициализации:
можно было бы
назвать p(p), s(s)

Что выведет программа?

```
struct A {  
    virtual void f(float x) { cout << x; }  
};  
  
struct B : A {  
    void f(double x) { cout << x + 1; }  
};  
  
int main() {  
    A* x = new B;  
    x->f(1);  
}
```

Что выведет программа?

```
struct A {  
    virtual void f(float x) { cout << x; }  
};
```

```
struct B : A {  
    void f(float x) override { cout << x + 1; }  
};
```

```
int main() {  
    A* x = new B;  
    x->f(1);  
}
```

Старая версия: 1, новая – 2.
override

Что выведет программа?

```
struct A {  
    A() { f(); }  
    virtual void f() { cout << "A"; }  
};
```

```
struct B : A {  
    B() { f(); }  
    void f() { cout << "B"; }  
};
```

```
int main() {  
    A* x = new B;  
    x->f();  
}
```

- ABB
- BBB
- BBA
- AAA
- Не скомпилируется
- «Упадет» при исполнении

Что выведет программа?

```
struct A {  
    A() { f(); }  
    virtual void f() { cout << "A"; }  
};
```

```
struct B : A {  
    B() { f(); }  
    void f() { cout << "B"; }  
};
```

```
int main() {  
    A* x = new B;  
    x->f();  
}
```

- ABB
- BBB
- BBA
- AAA
- Не скомпилируется
- «Упадет» при исполнении

Полиморфизм не работает в конструкторах/деструкторах

Что выведет программа?

```
struct A {  
    int f() { return 1; }  
    int g() { return f() + 1; }  
};
```

```
struct B : A {  
    int f() { return 3; }  
};
```

```
int main() {  
    A a;  
    B b;  
    cout << a.g() << b.g();  
}
```

- 24
- 22
- «упадет»
- не скомпилируется

p.s. Считаем, что выше
написан `#include` и
`using namespace`
нужные.

Что выведет программа?

```
struct A {  
    int f() { return 1; }  
    int g() { return f() + 1; }  
};
```

```
struct B : A {  
    int f() { return 3; }  
};
```

```
int main() {  
    A a;  
    B b;  
    cout << a.g() << b.g();  
}
```

- 24
- 22
- «упадет»
- не скомпилируется

p.s. Считаю, что выше
написан #include и
using namespace
нужные.

Как обратиться к private'ным переменным класса?

```
class A {  
    int x = 0;  
    friend struct B;  
    friend void g(A& a);  
};
```

```
struct B {  
    void f(A& a) { a.x++; }  
};
```

```
void g(A& a) { a.x++; }
```

Лямбда-функции

```
struct Comparator  
{  
    // TODO  
};
```

```
int main() {  
    std::vector<int> v;  
    // ...  
    std::sort(v.begin(), v.end(), Comparator());  
}
```

Лямбда-функции

```
int count = 0;
```

```
auto comparator = [&](auto a, auto b) {  
    count++;  
    return a > b;  
};
```

```
std::sort(v.begin(), v.end(), comparator);
```

1. Компилятор вместо auto сам «подставит» имя нужного типа.
2. Здесь comparator – функтор (класс, у которого определен оператор «круглые скобки» ())
3. [&] – значит, что ко всем переменным, переданным в качестве аргументов и используемым, обращение будет вестись по ссылке; [=] – по значению; [&a, =b] – первый аргумент по ссылке, второй – по значению.

Лямбда-функции

```
int count = 0;  
std::sort(v.begin(), v.end(), [&](auto a, auto b) {  
    {  
        count++;  
        return a > b;  
    });  
});
```

Где ещё можно использовать лямбда-функции?

```
• std::for_each(v.begin(), v.end(), [&](int x) {  
•   std::cout << x << ' ' ;  
• });
```

```
int count = std::count_if(v.begin(), v.end(), [](int x)  
• {  
•   return x < 5;  
• });
```


Где ещё можно использовать лямбда-функции?

```
1 // copy_if example
2 #include <iostream>      // std::cout
3 #include <algorithm>     // std::copy_if, std::distance
4 #include <vector>        // std::vector
5
6 int main () {
7     std::vector<int> foo = {25,15,5,-5,-15};
8     std::vector<int> bar (foo.size());
9
10    // copy only positive numbers:
11    auto it = std::copy_if (foo.begin(), foo.end(), bar.begin(), [](int i){return !(i<0);} );
12    bar.resize(std::distance(bar.begin(),it)); // shrink container to new size
13
14    std::cout << "bar contains:";
15    for (int& x: bar) std::cout << ' ' << x;
16    std::cout << '\n';
17
18    return 0;
19 }
```

<http://www.cplusplus.com/reference/algorithm/> - если можно воспользоваться какой-то готовой функцией из стандартной библиотеки, то лучше так и сделать, а не писать свою (ещё одну) реализацию.

Упражнение 1

Дано число N и далее N строк. Найти количество уникальных *эквивалентных* строк и вывести их.

P.S. Две строки назовём *эквивалентными*, если после удаления пробелов и перевода символов в нижний регистр они совпадают.

P.P.S. Размер программы должен быть как можно меньше - активно используйте лямбда-функции. Обратите внимание на ф-и

<http://www.cplusplus.com/reference/algorithm/transform/>,

`std::sort`,

<http://www.cplusplus.com/reference/algorithm/unique/>

Для работы со строками используйте класс `std::string`.

Что выведет программа?

```
class Base {  
    virtual void method() {std::cout << "from Base" << std::endl;  
public:  
    virtual ~Base() {method();}  
    void baseMethod() {method();}  
};  
  
class A : public Base {  
    void method() {std::cout << "from A" << std::endl;}  
public:  
    ~A() {method();}  
};  
  
int main(void) {  
    Base* base = new A;  
    base->baseMethod();  
    delete base;  
    return 0;  
}
```

Что выведет программа?

```
class Base {  
    virtual void method() {std::cout << "from Base" << std::endl;  
public:  
    virtual ~Base() {method();}  
    void baseMethod() {method();}  
};  
  
class A : public Base {  
    void method() {std::cout << "from A" << std::endl;}  
public:  
    ~A() {method();}  
};  
  
int main(void) {  
    Base* base = new A;  
    base->baseMethod();  
    delete base;  
    return 0;  
}
```

from A
from A
from Base

Что выведет программа?

```
int main(int argc, const char * argv[]) {  
    int a[] = {1, 2, 3, 4, 5, 6};  
    std::cout << (1 + 3)[a] - a[0] + (a + 1)[2];  
}
```

Что выведет программа?

```
int main(int argc, const char * argv[]) {  
    int a[] = {1, 2, 3, 4, 5, 6};  
    std::cout << (1 + 3)[a] - a[0] + (a + 1)[2];  
}
```

$$a[4] - a[0] + a[3] = 5 - 1 + 4 = 8$$

Где ещё могу быть полезны lambd'ы?

- В прошлом семестре говорили про указатели на ф-и

```
#include <iostream>
#include <functional>

//using FuncPtr = int (*)(double);
using FuncPtr = std::function<int(double)>;

int f(double x) {
    return std::ceil(x); // the smallest integral value that is not less than x
}

int main() {
    FuncPtr ptr = f;
    std::cout << ptr(3.2) << std::endl;
}
```

P.S. 1) using – более общий случай typedef'а. Лучше всегда использовать его. 2) std::function – C++ аналог указателей на ф-и ... можно также передавать в качестве аргумента в другие ф-и, методы

Где ещё могу быть полезны lambd'ы?

```
using FuncPtr = std::function<int(double)>;
```

```
int f(double x) { return x - 3.14; }
```

```
/* Кстати, в качестве std::function можно подставить lambda-функцию */  
int g(FuncPtr ptr) { return ptr(3.14); }
```

```
struct A {  
    int h(double x) { return 1; }  
};
```

```
int main() {  
    std::cout << g(&A::h) << std::endl;  
}
```

Так сделать не можем, т.к. в общем случае ф-я `h` может зависеть от значений полей структуры `A` ... должны как-то указать для какого именно объекта метод `h` нужно вызывать.

P.S. `std::function` удобно, например, использовать для параметризации стратегии поведения ботов в игре (оборонительная, наступательная и т.п.)

Где ещё могу быть полезны lambd'ы?

```
using FuncPtr = std::function<int(double)>;

int g(FuncPtr ptr) { return ptr(3.14); }

struct A {
    int h(double x) { return static_cast<int>(x) + p; }
    int p = 0;
};

int main() {
    A a;
    a.p = 1;
    std::cout << g([&a](double x) { return a.h(x); }) << std::endl;
}
```

Просто «заворачиваем» вызов метода h в lambda-функцию

Ключевое слово static

- Статическая константа в классе

```
struct A
{
    static const int MaxCount = 42;
    const int nonStatic = 5;
};

int main()
{
    /*
    1) статическую константу можно использовать без
       создания экземпляра самого объекта
    2) сколько бы ни было у нас экземпляров объекта A,
       память под эту константу будет выделена лишь 1 раз
    */
    A a;
    std::cout << A::MaxCount << a.nonStatic;
}
```

Ключевое слово static

- Статический метод класса – может обращаться только к статическим полям класса и вызывать статические методы. Можно вызвать «снаружи» без создания

```
struct Vector2 {  
    Vector2(float x, float y) : x(x), y(y) {}  
  
    /* если компилятор сможет однозначно по параметрам внутри  
       {} вызвать один из конструкторов, то название класса можно опускать */  
    static Vector2 i() { return { 1, 0 }; }  
  
    /* можно вызвать снаружи без экземпляра объекта */  
    static Vector2 j() { return { 0, 1 }; }  
  
    float x, y;  
};  
  
int main() {  
    std::cout << Vector2::i().x;  
}
```

Ключевое слово static

- Переменные класса

```
// следим за количеством экземпляров объектов A
struct A {
    A() { ++count; }
    ~A() { --count; }
    static int count;
};

// инициализация происходит вне класса в *.cpp файле
int A::count = 0;
```

Ключевое слово static

- Глобальные статические переменные -> область видимости ограничивается текущим файлом

Ключевое слово static

- Локальные статические переменные

```
[-] User createUniqueUser() {  
  [-] /* Подсчет числа вызовов ф-и createUniqueUser.  
      nextId будет инициализирована нулем только в первый вызов.  
      Так можно раздавать уникальные Id новым пользователям */  
    static int nextId = 0;  
    return User(nextId++);  
  }  
}
```

Ключевое слово static

- Паттерн «одиночка» - в программе должен быть объект ровно в одном экземпляре и жить до завершения работы программы: класс отвечающий за доступ к БД, система логгирования событий и т.п.

Ключевое слово static

- Паттерн «одиночка»

```
class Logger {  
public:  
    static Logger& getInstance();  
    void save(const std::string& message) {  
        file << message << std::endl;  
    }  
private:  
    Logger() { /* открываем файл log.txt */ }  
    std::fstream file;  
};  
  
Logger& Logger::getInstance() {  
    static Logger logger;  
    return logger;  
}  
  
int main() {  
    /* в любом месте программы можно написать подобную строчку в один и тот же файл */  
    Logger::getInstance().save("blah-blah");  
}
```


Множественное наследование

- Один класс может наследовать атрибуты двух и более классов одновременно.

```
class A
{
public:
    int a;
};
```

```
class B
{
public:
    int b;
};
```

```
class C : public A, public B
{
public:
    int getAB() { return a * b; }
}
```

Конструкторы вызываются
слева направо, деструкторы в
обратном порядке

Не должно быть двусмысленности

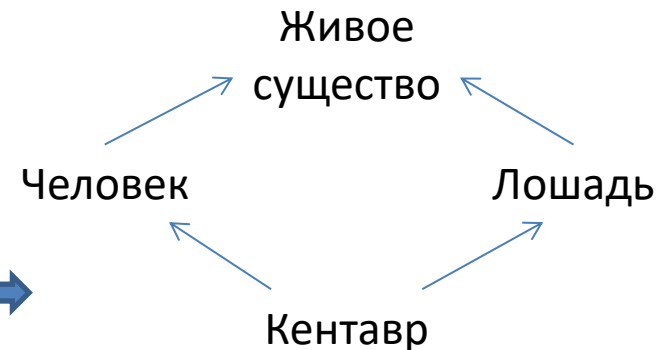
```
struct A
{
    void f() {}
};

struct B
{
    void f() {}
};

struct C: A, B
{
    void g() { A::f(); }
```

Явно указываем, какой именно метод
ХОТИМ ВЫЗВАТЬ

Для полноты картины почитайте дома про
то, как жить с ромбовидным наследованием
(русский вариант там тоже есть)
https://en.wikipedia.org/wiki/Virtual_inheritance



private наследование

- Реализация отношения «владеет» между классами ... в англоязычной литературе public наследование выражает отношение «является» = «is a...», а protected/private – «has a...», например,

```
class Car : private Engine  
{ }
```

private наследование

- Отношение «has a» в большинстве случаев можно выразить композицией

```
class Car
{
    Engine engine;
};
```

- Но если нам нужно переопределить виртуальные методы Engine или обратиться к protected методам/полям класса -> только наследование

«If in doubt, you should prefer composition over private inheritance»

private наследование

```
[-] /* хотим отнаследоваться от классов БЕЗ виртуального деструктора и
    |   запретить работать с полиморфно, чтобы избежать утечек памяти */
[-] struct Vec : private std::vector<int>
    | {
[-]   |   using vector<int>::push_back; /* если вызовут метод push_back у Vec,
    |   |                                   то вызови родительский */
    |   |
    |   | };

[-] int main() {
    |   Vec v;
    |   v.push_back(2);           // ok
    |   std::vector<int>* p = &v; // error
    | }

```

private наследование

- Реализация паттерна Адаптер

https://sourcemaking.com/design_patterns/adapter/cpp/1

https://sourcemaking.com/design_patterns/adapter

private наследование (ссылки)

- <http://stackoverflow.com/questions/656224/when-should-i-use-c-private-inheritance>
- http://www.bogotobogo.com/cplusplus/private_inheritance.php
- <https://isocpp.org/wiki/faq/private-inheritance>

ПОТОКИ В C++11

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

void func0(){
    std::cout << "Hi! I'm function 0\n";
}

void func1(int x){
    std::cout << "Hi! I'm function 1." << x << "\n";
}

int main(){
    std::thread t0(func0);
    std::thread t1(func1, 42);

    t0.join();
    t1.join();
}
```


Потоки + лямбда-функции

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

int main() {
    std::thread t0([]() {
        std::cout << "Hi! I'm function 0\n";
    });

    std::thread t1([](int x) {
        std::cout << "Hi! I'm function 1." << x << "\n";
    }, 42);

    t0.join();
    t1.join();
}
```

Async + future

- <https://solarianprogrammer.com/2012/10/17/cpp-11-async-tutorial/>
- Высокоуровневая обёртка над потоками
- Отдельные потоки запускаются после создания future. Если на момент, когда результат работы потока потребуется, он ещё не завершился, то ожидаемся завершения

Async + future

```
1 #include <future>
2 #include <iostream>
3
4 void called_from_async() {
5     std::cout << "Async call" << std::endl;
6 }
7
8 int main() {
9     //called_from_async launched in a separate thread if possible
10    std::future<void> result( std::async(called_from_async));
11
12    std::cout << "Message from main." << std::endl;
13
14    //ensure that called_from_async is launched synchronously
15    //if it wasn't already launched
16    result.get();
17
18    return 0;
19 }
```

Async + future

```
1 #include <future>
2 #include <iostream>
3
4 int main() {
5     //called_from_async launched in a separate thread if possible
6     std::future<int> result( std::async([](int m, int n) { return m + n; } , 2, 4));
7
8     std::cout << "Message from main." << std::endl;
9
10    //retrive and print the value stored in the future
11    std::cout << result.get() << std::endl;
12
13    return 0;
14 }
```

Async + future

```
1  #include <future>
2  #include <iostream>
3  #include <vector>
4
5  int twice(int m) {
6      return 2 * m;
7  }
8
9  int main() {
10     std::vector<std::future<int>> futures;
11
12     for(int i = 0; i < 10; ++i) {
13         futures.push_back (std::async(twice, i));
14     }
15
16     //retrive and print the value stored in the future
17     for(auto &e : futures) {
18         std::cout << e.get() << std::endl;
19     }
20
21     return 0;
22 }
```

Async + future

```
1 #include <future>
2 #include <iostream>
3 #include <vector>
4
5 int main() {
6     std::vector<std::future<int>> futures;
7
8     for(int i = 0; i < 10; ++i) {
9         futures.push_back (std::async([](int m) {return 2 * m;} , i));
10    }
11    ...
12 }
```

Сумма элементов массива в N потоков + замер времени работы

```
#include <future>
#include <numeric> // std::accumulate
#include <functional> // std::plus

const int N = 8; const int Size = 1000000;

int main() {
    std::vector<int> v(N * Size);
    for (int& x : v) x = rand() % 10;

    std::chrono::high_resolution_clock::time_point start = std::chrono::high_resolution_clock::now();

    std::vector<std::future<int>> f;
    for (int i = 0; i < N; ++i) {
        f.push_back(std::async([&v, i]() {
            int sum = std::accumulate(v.begin() + i * Size, v.begin() + (i + 1) * Size,
            0, std::plus<int>());
            return sum;
        }));
    }

    for (auto& result : f) std::cout << result.get() << " ";
    std::cout << std::endl;

    std::chrono::duration<double> diff = std::chrono::high_resolution_clock::now() - start;
    std::cout << diff.count() << "s";
}
```

auto

auto

Аналог progress bar`a

```
• std::chrono::milliseconds::span(10);  
• while (f.back().wait_for(span) == std::future_status::timeout)  
• • std::cout << "." << std::flush;
```


Упражнение 2

Методом Монте-Карло посчитать объём сферы в несколько потоков + измерить ускорение.

P.S. Можно и n -мерной:

<https://en.wikipedia.org/wiki/N-sphere>