

# Вопросы

- 1 На примере умножения матриц напишите код дружелюбный к кэшу.
- 2 Что такое висячий указатель / висячая ссылка? Приведите примеры.
- Что такое умный указатель? Какие умные указатели есть в стандартной библиотеке?
- 4 Как работает std::unique\_ptr?
- 5 Что произойдёт? Приведите пример с сырыми указателями и объясните.
- 6 Что произойдёт в коде?
- **7** Как работает std::shared\_ptr?
- 8 Что такое std::weak\_ptr и концепция слабых ссылок?



1 На примере умножения матриц напишите код дружелюбный к кэшу.

```
1 // Размерность a[l][m], b[m][n], c[l][n].

2 for (std::size_t i = 0; i < l; i++) {

3    for (std::size_t j = 0; j < n; j++) {

4        int cij = 0;

5        for (std::size_t k = 0; k < m; k++) {

6            cij += a[i][k] * b[k][j];

7        }

8        c[i][j] = cij;

9    }

10 }
```

В коде есть одна неэффективность, это выражение B[k][j] в самом внутреннем цикле. Мы обходим матрицу В по столбцам.

```
1 // Размерность a[l][m], b[m][n], c[l][n].

2 for (std::size_t i = 0; i < l; i++) {

3    for (std::size_t k = 0; k < m; k++) {

4        int aik = a[i][k];

5        for (std::size_t j = 0; j < n; j++) {

6            c[i][j] += aik * b[k][j];

7        }

8    }

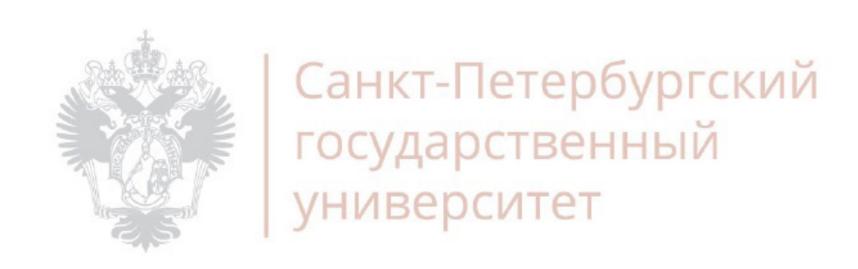
9 }
```

Мы не вычисляем каждый элемент матрицы с за раз. Мы вычисляем элементы частично на каждой итерации. Главное — это то, что во внутреннем цикле мы обходим обе матрицы построчно.

Программисты должны стараться писать код дружелюбный к кэшу. Как правило, основной объём вычислений производится в нескольких местах программы. Если есть вложенные циклы, то внимание надо сосредоточить на самом внутреннем, потому что код там выполняется чаще всего. Эти места программы и нужно оптимизировать, стараясь улучшить локальность их данных.

#### Рекомендации:

- 1. Сконцентрируйте внимание на внутренних циклах. Т.к. там происходит наибольший объём вычислений и обращений к памяти.
- 2. Постарайтесь максимизировать пространственную локальность, читая объекты из памяти последовательно, в том порядке, в котором они в ней расположены.
- 3. Постарайтесь максимизировать временную локальность, используя объекты данных как можно чаще после того, как они были прочитаны из памяти.



2 Что такое висячий указатель / висячая ссылка? Приведите примеры.

Висячий указатель или висячая ссылка — это указатель, указывающий на область памяти, где ранее хранились данные.

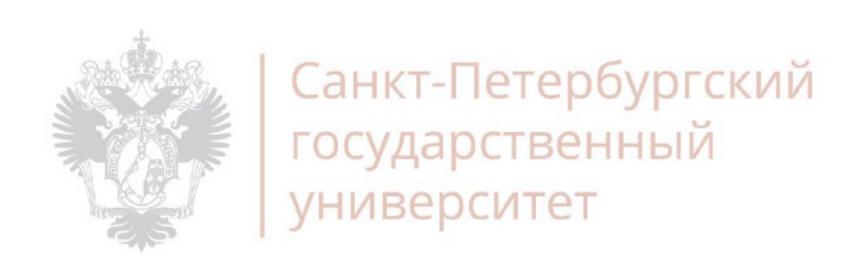
Т.к. ОС может перераспределить ранее освобождённую память (в том числе в другой процесс), то оборванный указатель приводит к неопределённому поведению программы (UB — undefined behavior). Этот вид ошибок очень опасен, и наряду с утечками памяти.

```
1 // Примеры 1.
2 char* f() {
3 char s[100];
4 // Код
5 return s;
6 }
7 char* s = f();
```

```
1 // Пример 2.
2 s = std::malloc(...);
3 // Код
4 std::free(s);
5 // Область видимости s не закончилась
6 // -> могут быть обращения к s.
7
8 // -> Правильный код:
9 s = std::malloc(...);
10 // Код
11 std::free(s);
12 s = nullptr;
13 // Область видимости s продолжается.
```

```
1 // Пример 3.
2 {
3          char* dp = nullptr;
4          // код
5          {
6                char c;
7                dp = &c;
8          } // с выпадает из области видимости
9          // теперь dp является висящим указателем
10          // Код
11 }
```

```
1 // Пример 4.
2 int& f() {
3    int a = 101;
4    int &b = a;
5    return b;
6 }
```





### Что такое умный указатель? Какие умные указатели есть в стандартной библиотеке?

Все базовые функции выделения и освобождения памяти являются небезопасными. Наиболее часто можно столкнуться со следующими ситуациями:

- 1. Память была выделена, но не была освобождена (утечка памяти).
- 2. Память была освобождена, но работа с ней продолжается так, как будто она остаётся выделенной (после free или delete память, на которую указывал указатель считается освобождённой, но указатель при этом не равен nullptr).
- 3. Память не была выделена, но в неё выполняется запись данных (int\* ptr; // ptr имеет значение не nullptr, а произвольный адрес).
- 4. Попыка несколько раз осовободить одну и ту же память.

В языках java, python, c# для этого разработаны сборщики мусора. В C++11 была создана система умных указателей. Они на примитивном уровне подобны сборщику мусора и сами контролиуют процесс освобождения памяти.

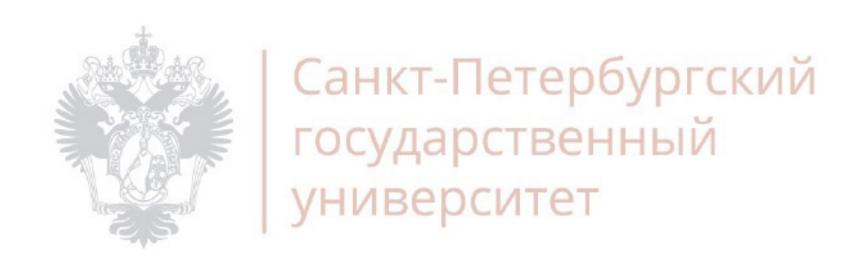
**Умный указатель** — это класс-шаблон пространства имён std из библиотеки <memory>, который умеет владеть объектом, управлять им через указатель и контролировать особождение выделенной ему памяти.

И, соответственно, разработчику не надо заботиться о вызове delete / delete[] для освобождения памяти.

B <memory> существует 3 умных указателя:

- 1. unique\_ptr УУ, владеющий данной памятью в единственном числе (на одну память ссылается только один unique\_ptr).
- 2. shared\_ptr УУ, позволяет владеть данной памятью во множественном числе. (на одну память могут ссылаться много shared\_ptr).
- 3. weak\_ptr УУ, являющийся "слабой ссылкой" на объект shared\_ptr (временное владение). Другими словами, weak\_ptr используется, когда нужен УУ, который имеет доступ к ресурсу, но не считается его владельцем.

С УУ нельзя делать операции адресной арифметики как это было возможно с сырыми указателями.



4

#### Как работает std::unique\_ptr?

 $unique\_ptr$  — УУ, владеющий данной памятью в единственном числе. (на одну память ссылается только один  $unique\_ptr$ ).

T.e.  $ptr_1 = ptr_2$  и unique\_ptr<int>  $ptr_2$  { $ptr_1$ }; — ошибка компиляции. Потому, что ОПК и КК у unique\_ptr удалены.

unique\_ptr полезен, когда нужен указатель на объект, на который НЕ будет других указателей и который будет удален после удаления указателя.

По умолчанию unique\_ptr инициализируется nullptr в отличие от сырых указателей: std::unique\_ptr<int> ptr;

Выделить память и создать в ней объект можно с помощью функции std::make\_unique<int>(1): std::unique\_ptr<int> ptr {std::make\_unique<int>(1)}; std::unique\_ptr<int> ptr = std::make\_unique<int>(1); std::cout << \*ptr1 << std::endl;

unique\_ptr могут работать и с массивами: std::unique\_ptr<int[]> arr = std::make\_unique<int[]>(10);

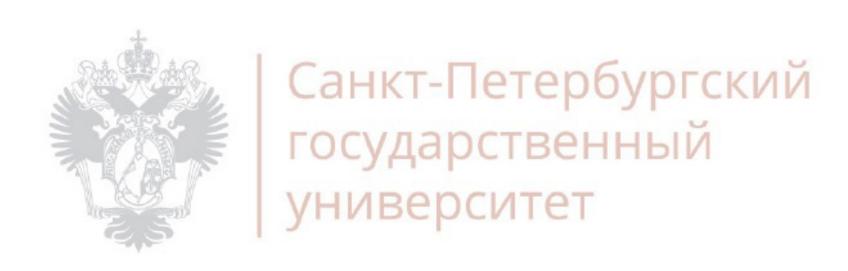
Уто произойдёт? Приведите пример с сырыми указателями и объясните.

```
1 std::unique_ptr<int> ptr;
2 ptr = std::make_unique<int>(10);
3 ptr = std::make_unique<int>(11);
```

ОПК в unique\_ptr удалён и поэтому ptr\_1 = ptr\_2 вызовет ошибку компиляции, а в случае ОПП строк 2 и 3 всё будет хорошо: память из-под текущего значения освободится, и будет записано новое значение для указателя.

```
1 int* iptr;
2 iptr = new int(10);
3 iptr = new int(11); // утечка памяти, где лежит 10.
```

Умность указателя заключается в том, что он контролирует ранее выделенную область памяти и когда надо, то сам её освобождает. А освобождение памяти сырым указателям должен контролировать разработчик.



6 Объясните работу кода.

Нет в мире таких умных указателей, которые нельзя было бы поломать, вооружившись хорошо теорией.

```
1 int* p = new int(10);
2 std::unique_ptr<int> ptr_1 {p};
3 std::unique_ptr<int> ptr_2 {p};
```

В данном случае оба умных указателя будут указывать на одну и ту же память.

Важно понимать, что std::unique\_ptr<int> ptr  $std::make_unique<int>(1)$ ; — это конструктор перемещения, а в коде в строках 2 и 3 написан конструктор с параметрами (сырого указателя). И в данной ситуации просто копируется значение указателя и вся ответственность за не/единственное владение лежит на разработчике.

И поэтому после освобождения одного из УУ произойдёт через какое-то время опять освобождение этой же памяти, что может привести к неопределённому поведению UB (undefined behavior).

## **7** Как работает std::shared\_ptr?

**shared\_ptr** — УУ, позволяющий владеть данной памятью во множественном числе. (на одну память могут ссылкать много shared\_ptr).

Для данных указателей применяется механизм подсчета ссылок. Каждый раз, когда создается объект shared\_ptr<T>, увеличивается счетчик объектов shared\_ptr<T>, которые содержат определенный адрес. Когда объект shared\_ptr<T> удаляется или ему присваивается другой адрес, счетчик ссылок уменьшается на единицу. Когда больше нет объектов shared\_ptr<T>, которые ссылаются на определенный адрес, счетчик ссылок сбрасывается в ноль.

```
Синтаксис инициализации аналогичный unique_ptr: std::shared_ptr<int> ptr1 {std::make_shared<int>(22)}; std::shared_ptr<int> ptr2 = ptr1; — а здесь КК и ОПК работают. shared_ptr нельзя проинициализировать через unique_ptr: std::unique_ptr<int> ptr1 { std::make_unique<int>(1) }; std::shared_ptr<int> ptr2 { ptr1 }; // ошибка компиляции ptr2 = ptr1; // ошибка компиляции
```



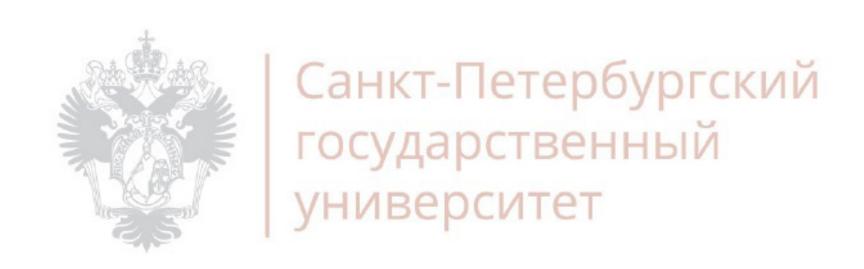
8 Что такое std::weak\_ptr и концепция слабых ссылок?

weak\_ptr — УУ, являющийся "слабой ссылкой" на объект shared\_ptr (временное владение). Другими словами, weak\_ptr используется, когда нужен УУ, который имеет доступ к ресурсу, но не считается его владельцем.

У weak\_ptr объект доступен только до тех пор, пока он существует (т.е. пока не была освобождена "сильная ссылка"). Пример использования:

```
1 std::weak_ptr<int> gw;
2 void observe() {
3     std::cout << "gw.use_count() == " << gw.use_count() << "; ";
4     if (std::shared_ptr<int> spt = gw.lock()) {
5         std::cout << "*spt == " << *spt << '\n';
6     } else {
7         std::cout << "gw истёк\n";
8     }
9 }
10 int main() {
11     {
12         std::shared_ptr<int> sp = std::make_shared<int>(42);
13         gw = sp;
14         observe();
15     }
16     observe();
17 }
18 // Вывод программы:
19 // gw.use_count() == 1; *spt == 42
20 // gw.use_count() == 0; gw истёк
```

gw.lock() проверяет, что "сильная ссылка" не освобождена и если так, то возвращает копию shared\_ptr.



8

### Что такое std::weak\_ptr и концепция слабых ссылок?

#### продолжение

Концепция слабых ссылок — это особый вид ссылки на объект из динамической памяти, связь которых с объектом, на который они ссылаются, не учитывается сборщиком мусора.

В каких задачах они используются:

- 1. Циклические ссылки
- 2. Кэширование
- 3. Другие

Пример циклических ссылок и утечки памяти, т.к. деструкторы объектов вызваны не будут →

```
1 class Person {
           std::string name;
           std::shared_ptr<Person> myfriend;
      public:
           Person(const std::string &name) : name(name) {
               std::cout << "Конструктор " << name << std::endl;
           ~Person() {
               std::cout << "Деструктор " << name << std::endl;
          friend void set_friends(
             std::shared_ptr<Person> &p1, std::shared_ptr<Person> &p2
               p1->myfriend = p2;
               p2->myfriend = p1;
17 };
19 int main() {
      std::shared_ptr<Person> masha { std::make_shared<Person>("Маша") };
      std::shared_ptr<Person> misha { std::make_shared<Person>("Миша") };
      set_friends(masha, misha);
      return 0;
```