Функции и их параметры. Рекурсия

Владислав Хорев
Ведущий программист в Palta People Ltd

Владислав Хорев

О спикере:

- Ведущий программист в компании Palta People Ltd
- Работает в IT с 2011 года
- Опыт разработки на C++ более 10 лет

Вспоминаем прошлое занятие

Вопрос: как создать массив целых чисел на 50 элементов, который называется arr?

Вспоминаем прошлое занятие

Вопрос: как создать массив целых чисел на 50 элементов, который называется arr?

Ответ: int arr[50];

Вспоминаем прошлое занятие

Вопрос: с помощью чего можно получить доступ к каждому элементу массива по очереди (пробежаться по массиву)?

Вспоминаем прошлое занятие

Вопрос: с помощью чего можно получить доступ к каждому элементу массива по очереди (пробежаться по массиву)?

Ответ: с помощью цикла (обычно for)

Вспоминаем прошлое занятие

Вопрос: «изменение порядка элементов массива так, чтобы этот порядок соответствовал определённому правилу (обычно от меньшего к большему или от большего к меньшему)» — что это?

Вспоминаем прошлое занятие

Вопрос: «изменение порядка элементов массива так, чтобы этот порядок соответствовал определённому правилу (обычно от меньшего к большему или от большего к меньшему)» — что это?

Ответ: сортировка массива

Вспоминаем прошлое занятие

Вопрос: какой метод сортировки массива мы уже знаем?

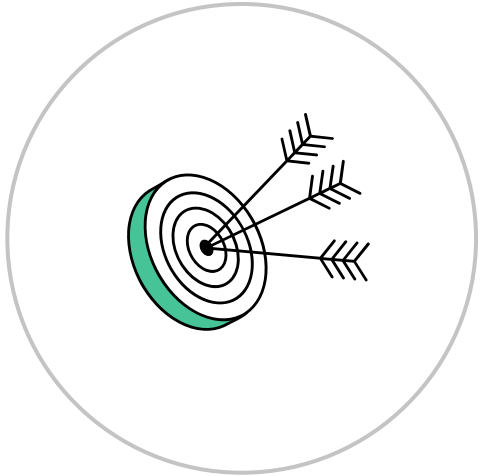
Вспоминаем прошлое занятие

Вопрос: какой метод сортировки массива мы уже знаем?

Ответ: сортировка «пузырьком»

Цели занятия

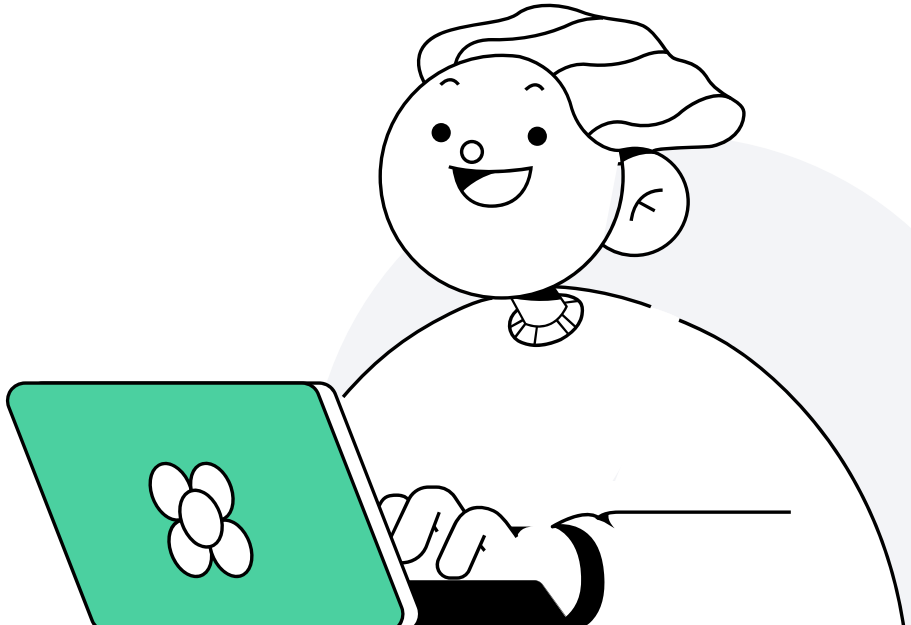
- Научимся создавать и вызывать функции
- Узнаем, что такое параметры функции
- Разберёмся, что такое возвращаемое значение функции
- Затронем приведение типов
- Научимся работать с рекурсивными функциями



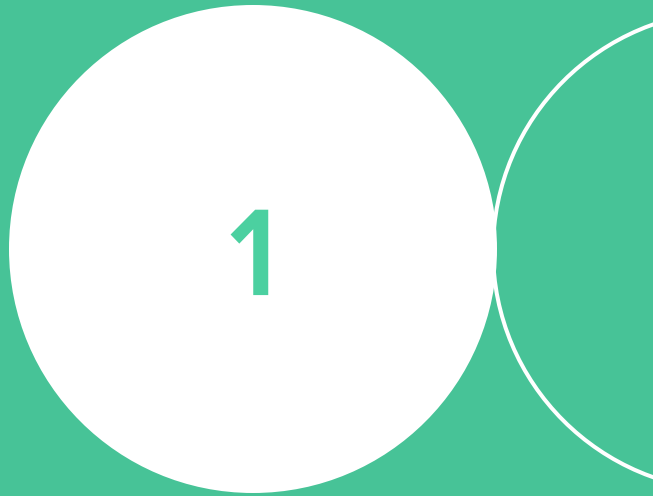
План занятия

1. Функции
2. Параметры по умолчанию
3. Возвращаемое значение
4. Приведение типов
5. Рекурсия
6. Итоги
7. Домашнее задание

*Нажми на нужный раздел для перехода



Функции



Принцип DRY и дублирование кода

В программировании существует принцип DRY - Don’t Repeat Yourself (Не повторяй себя, не повторяйся). Это значит, что нужно стараться как можно больше переиспользовать уже написанный код

Для этого любой код с уникальной логикой нужно выделять в отдельные переиспользуемые кусочки - самым простым вариантом являются функции

Зачем переиспользовать код? Для того, чтобы если в будущем понадобилось изменить поведение программы, то нужно было внести изменение всего в одно место, а не в несколько (так будет, если код, подлежащий изменению, дублируется)

Дублирование кода

```
#include <iostream>

int main(int argc, char** argv)
{
     std::cout << "Hello, Dima" << std::endl;
     std::cout << "Hello, Vova" << std::endl;
     std::cout << "Hello, Alena" << std::endl;
     std::cout << "Hello, Natasha" << std::endl;
     std::cout << "Hello, Igor" << std::endl;
     std::cout << "Hello, Svetlana" << std::endl;

     return 0;
}
```

Нет дублирования кода

```
#include <iostream>

int main(int argc, char** argv)
{
     hello("Dima");
     hello("Vova");
     hello("Alena");
     hello("Natasha");
     hello("Igor");
     hello("Svetlana");

     return 0;
}
```

Функция

Переиспользуемая часть кода, которая решает какую-либо задачу:

```
<тип возвращаемого значения> <название функции>(
     [<тип параметра 1> <название параметра 1>,
      <тип параметра 2> <название параметра 2>, ...]
)
{
     [тело функции]
}
```

В [] — необязательная часть, можно совсем не писать

В <> — обязательная часть, надо что-то написать

Функция

```
<тип возвращаемого значения> <название функции>(
     [<тип параметра 1> <название параметра 1>,
      <тип параметра 2> <название параметра 2>, ...]
)
{
     [тело функции]
}
```

Тип возвращаемого значения — любой тип данных + void

Название функции — допустимое имя в C++

Тип параметра N — любой тип данных (void нельзя)

Название параметра N — допустимое имя в C++ (snake_case)

Тело функции — действия, которые необходимо выполнит

Где писать функции

В C++ функции могут существовать как внутри классов (пока не рассматриваем), так и совершенно отдельно:

```
void hello() { /* действия */ }
void calc() { /* действия */ }
void main(int argc, char** argv) { /* действия */ }
```

На заметку: void main(int argc, char** argv) {} тоже почти обычная функция.
Её особенность лишь в том, что программа всегда начинается именно с неё

Пара слов о void

void — специальный тип, который обозначает отсутствие типа и значения. Может использоваться только в качестве возвращаемого значения
в функциях. Говорит о том, что функция просто совершает какие-то действия и ничего обратно не отдаёт

Функция hello

```
#include <iostream>

void hello(std::string name)
{
     std::cout << "Hello, " << name << std::endl;
}

int main(int argc, char** argv)
{
     hello("Dima");
     hello("Vova");
     hello("Alena");
     hello("Natasha");
     hello("Igor");
     hello("Svetlana");

     return 0;
}
```

Примеры функций

```
// без параметров
void print_border()
{
   std::cout << "##########" << std::endl;
   std::cout << "#        #" << std::endl;
   std::cout << "#        #") << std::endl;
   std::cout << "#        #") << std::endl;
   std::cout << "#        #") << std::endl;
   std::cout << "#        #") << std::endl;
   std::cout << "##########") << std::endl;
}

// с несколькими параметрами
void print_volume(int a, int b, int c)
{
   int volume = a * b * c;
   std::cout << volume << std::endl;
}
```

Особенности вызова функций

Важно учесть: чтобы можно было вызвать функцию, о ней должно быть известно, то есть должна быть известна её **сигнатура** (имя функции, её параметры, возвращаемый тип). Для этого на момент вызова функции она уже должна быть объявлена **раньше**. Такой код не скомпилируется:

```
int main(int argc, char** argv)
{
     hello("Dima");

     return 0;
}
void hello(std::string name)
{
     std::cout << "Hello, " << name << std::endl;
}
```

Особенности вызова функций

Чтобы решить эту проблему, можно следить за правильным порядком объявления функций. Пример:

```
void hello(std::string name)
{
     std::cout << "Hello, " << name << std::endl;
}
int main(int argc, char** argv)
{
     hello("Dima");

     return 0;
}
```

Однако если функций много, это может стать сложной задачей

Особенности вызова функций

Поэтому в C++ существует механизм **объявления функции без тела**. То есть описывается только сигнатура функции, а тело функции может быть описано в другом месте. Скомпилируется такой код:

```
void hello(std::string name);
int main(int argc, char** argv)
{
     hello("Dima");

     return 0;
}
void hello(std::string name)
{
     std::cout << "Hello, " << name << std::endl;
}
```

Здесь функция hello сначала **объявляется**, а **реализация** приведена уже после **использования**

Параметры по умолчанию

2

Параметры по умолчанию

Для одного или нескольких параметров функции можно задать значение по умолчанию. Тогда при вызове функции можно будет не указывать значение этого параметра, а в теле функции этот параметр получит указанное значение по умолчанию:

```
#include <iostream>

void hello(std::string name = "Anonymous")
{
    std::cout << "Hello, " << name << std::endl;
}

int main(int argc, char** argv)
{
    hello("Dima");
    hello();
    return 0;
}
```

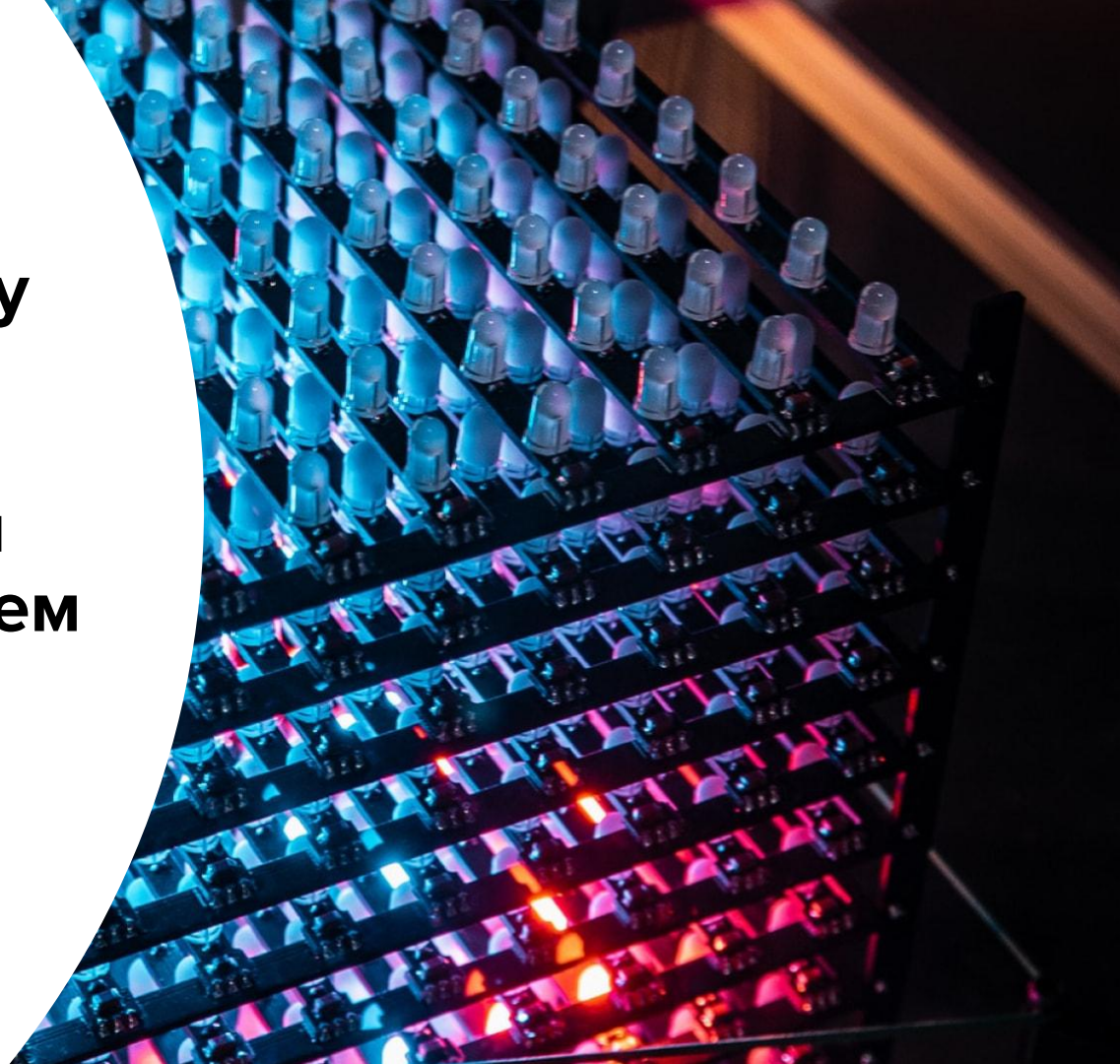
Правила параметров по умолчанию

- Параметров со значением по умолчанию может быть сколько угодно
- Все параметры со значениями по умолчанию должны идти в конце, после параметров без значений по умолчанию

```
void calc_sum_correct(int x = 1, int y = 2, int z = 3)
{
     return x + y + z;
}
void calc_sum_incorrect(int x = 1, int y = 2, int z)
{
     return x + y + z;
}
```

Напишем программу для отрисовки прямоугольников с разной границей и разным заполнителем

Готовый пример кода



Возвращаемое значение

Возвращаемое значение

До этого мы постоянно использовали void, то есть ничего не возвращали.

Чаще всего функции что-то возвращают — результат проделанных действий.

Например, объём не печатают на экран, а обычно возвращают, чтобы дальше его использовать. Можно и на экран вывести, а можно и что-то ещё с этим значением сделать

Возвращаемое значение

Чтобы функция вернула какое-либо значение, нужно выполнить два условия:

1. Указать возвращаемый тип данных (не void)
2. В функции написать слово return и через пробел значение или переменную. Тип значения должен совпадать с типом из пункта 1

Значение, которое возвращает функция, можно записать в переменную соответствующего типа

Пример

```
int calc_volume(int a, int b, int c)
{
    int volume = a * b * c;
    return volume;
}

int main(int argc, char** argv)
{
    int vol = calc_volume(4, 2, 7);
    std::cout << vol << std::endl;
    return 0;
}
```

Возвращаемое значение и void

Если функция не имеет возвращаемого значения, то есть её тип void, то слово return в функции можно не писать (как мы ранее и делали).

Но можно и написать, только после него сразу должна идти точка с запятой, так как никакого типа нет, следовательно, никакое значение не возвращается.

Это будет означать завершение работы функции. Ключевое слово return говорит, что результат получен и дальнейшие действия не требуются, можно завершать функцию.

Возвращаемое значение всегда только одно. Нельзя вернуть несколько значений из функции

Пример

```
int calc_volume(int a, int b, int c)
{
    int volume = a * b * c;
    return volume;
    // следующие строки никогда не выполнятся,
    // так как return прекращает выполнение функции
    return a + b + c;
}

int main(int argc, char** argv)
{
    int vol = calc_volume(4, 2, 7);
    std::cout << vol << std::endl;
    return 0;
}
```

Пример

```
void print_data(int a, int b, int c)
{
    std::cout << a << std::endl;
    std::cout << a << std::endl;
    return;
    // следующая строка не выполнится
    std::cout << c << std::endl;
}

int main(int argc, char** argv)
{
    print_data(3, 4, 5);
    return 0;
}
```

Несколько return

```
int get_min(int a, int b)
{
    if (a < b) {
        return a;
    } else {
        return b;
    }  // как это можно проще написать?
}

int main(int argc, char** argv)
{
    std::cout << get_min(5, 8) << std::endl;
}
```

Несколько return

```
int get_min(int a, int b)
{
    return a < b ? a : b;  // вот так
}

int main(int argc, char** argv)
{
    std::cout << get_min(5, 8) << std::endl;
}
```

Приведение типов

4

Проблема

Нужно посчитать процент от целого числа.

Вероятнее всего, результат будет дробным: зависит от исходных значений, но в общем случае он будет дробным.

Мы хотим отбросить дробную часть и получить лишь целую часть.

Пример: Посчитать 17% от числа 49 и оставить только целую часть

Проблема

Нужно посчитать процент от целого числа.

Вероятнее всего, результат будет дробным: зависит от исходных значений, но в общем случае он будет дробным.

Мы хотим отбросить дробную часть и получить лишь целую часть.

Пример: Посчитать 17% от числа 49 и оставить только целую часть

Решение:

0,17 * 49 = 8,33

Ответ: 8

Реализация

```
int calc_percent(int value, int percent)
{
    return percent / 100 * value;
}

void main(int argc, char** argv)
{
    std::cout << calc_percent(49, 17) << std::endl;  // 0 ←——  Почему ?
}
```

Неявное приведение типов

percent / 100 в результате даёт 0.

Дело в том, что деление целочисленное, так как percent — целое число,
и 100 — целое число.

Выход: какой-либо из операндов (percent или 100 или оба) должен стать дробным.

Тогда мы получим **неявное** приведение типов.

Все операции должны происходить между одинаковыми типами.

Если типы разные, совершится попытка **автоматически** уравнять типы

Неявное приведение типов

Неявное приведение работает правильно только в больший тип.

Например, **int -> long** или **int -> double** или **long -> double**

Направленное в обратную сторону неявное приведение может вызвать **неопределённое поведение**.

Поэтому его стоит избегать и вместо него использовать **явное приведение**

Реализация

```
int calc_percent(int value, int percent)
{
    // неявное приведение из double в int, дробная часть отбрасывается
    return percent / 100.0 * value;
}

void main(int argc, char** argv)
{
    std::cout << calc_percent(49, 17) << std::endl;  // 8
}
```

Явное приведение типов

Если мы уверены в своих действиях и не боимся потерять часть информации, можно явно привести типы. В этом случае может произойти потеря некоторой информации.

Для явного приведения необходимо использовать оператор **static_cast<тип>** — это рекомендуемый способ явного приведения типов. Иногда можно встретить старый вариант явного приведения типов — перед переменной в круглых скобках указан тип, в который надо перевести значение:

```
double d = 47.9;
int new_i = static_cast<int>(d);      // new_i = 47, потеряли 0.9
int old_i = (int)d;                   // old_i = 47, потеряли 0.9
```

Правильная реализация

```
int calc_percent(int value, int percent)
{
    double res = percent / 100.0 * value;
    return static_cast<int>(res);
}

void main(int argc, char** argv)
{
    std::cout << calc_percent(49, 17) << std::endl;  // 8
}
```

Правильная реализация. Вариант 2

```
int calc_percent(int value, int percent)
{
    double res = static_cast<double>(percent) / 100 * value;
    return static_cast<int>(res);
}

void main(int argc, char** argv)
{
    std::cout << calc_percent(49, 17) << std::endl;  // 8
}
```

Рекурсия

5

Рекурсия

Как видите, функции вызываются из других функций.

Возникает закономерный вопрос: что будет, если функция вызовет сама себя?

Этот механизм называется **рекурсия**, или **рекурсивный вызов**.

Давайте рассмотрим простой пример

Что будет, если вызвать эту функцию?

```
void print_and_increase(int start)
{
    std::cout << start << std::endl;
    print_and_increase(start + 1);
}

int main(int argc, char** argv)
{
    print_and_increase(1);
    return 0;
}
```

Выйдет ошибка

Правильный ответ: в течение короткого времени на экран будут выводиться увеличивающиеся на единицу числа, а затем программа завершится с ошибкой **Stack Overflow exception** (ошибка переполнения стека)

Почему так происходит

Дело в том, что при вызове функции информация об этом вызове помещается в специальное хранилище — **стек**, а когда работа функции завершится информация об этом вызове вынимается из стека.

У стека ограниченный размер.

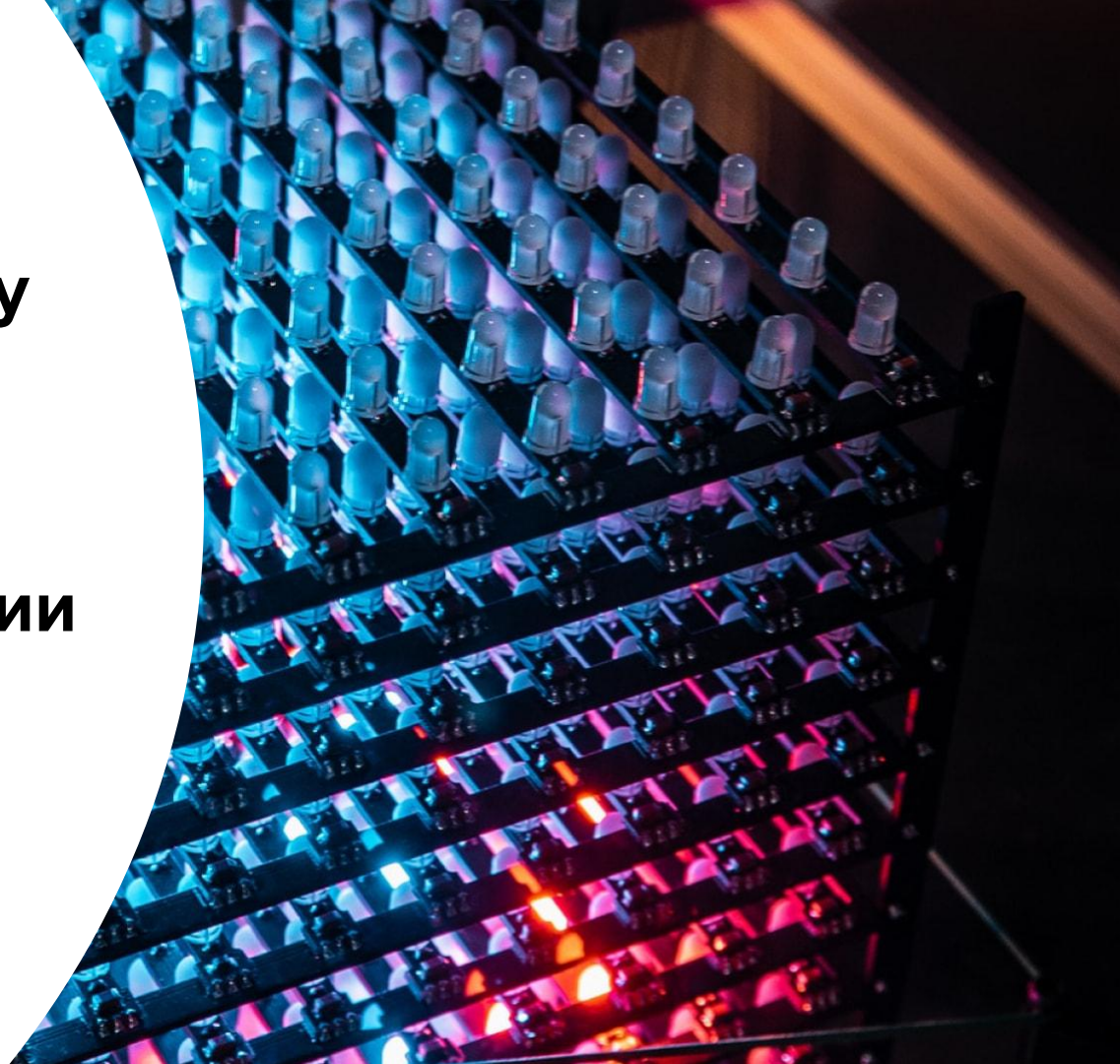
В обычной ситуации функции работают последовательно, но в ситуации с приведённым на прошлом слайде рекурсивным вызовом получается **бесконечное** дерево вызовов. Это приводит к переполнению стека,
и программа завершается.

Переполнение стека — один из главных минусов рекурсивных функций.

Второй минус рекурсивных функций — они могут быть сложны для понимания

Напишем программу для вычисления факториала числа с помощью рекурсивной функции

Готовый пример кода



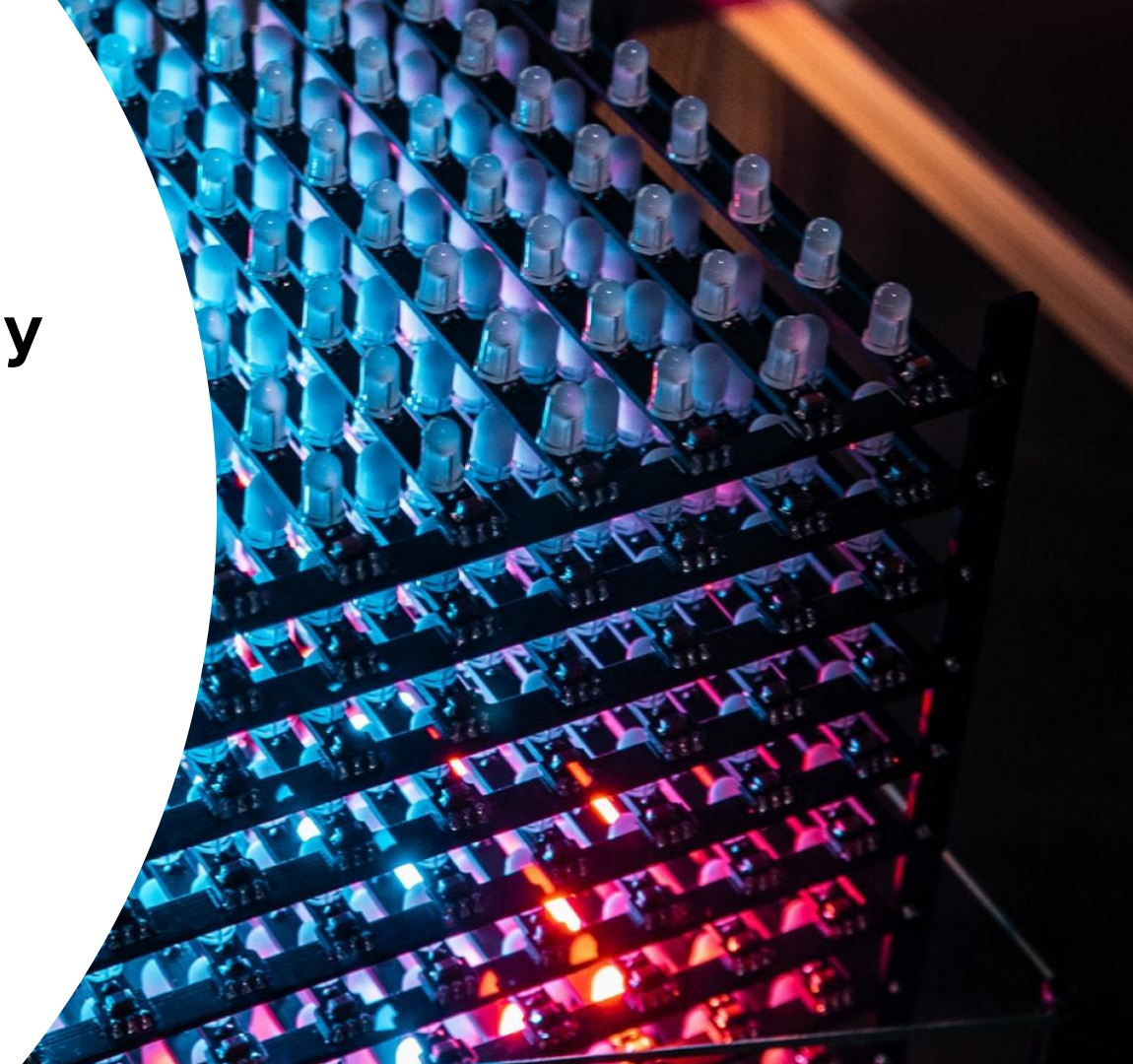
Рекурсия

При использовании рекурсивных функций важно предусмотреть **условие выхода**: когда функция должна перестать вызывать саму себя и вернуть что-то конкретное?

Хорошая новость — любую рекурсивную функцию можно переписать с использованием **циклов**

Напишем программу для вычисления факториала числа с помощью цикла

Готовый пример кода



Итоги

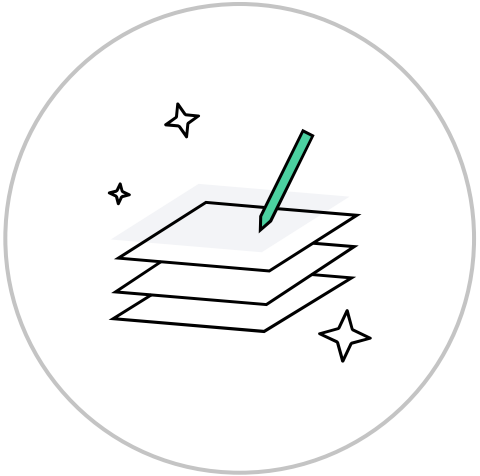
Сегодня мы

1. научились создавать и использовать (вызывать) функции на C++
2. разобрались с параметрами, в том числе с параметрами по умолчанию
3. изучили механизм возврата значений
4. научились применять приведение типов
5. научились работать с рекурсией

Домашнее задание

Давайте посмотрим ваше домашнее задание:

1. Вопросы по домашней работе задавайте в чате группы
2. Задачи можно сдавать по частям
3. Зачёт по домашней работе ставят после того, как приняты все задачи



Задавайте вопросы и пишите отзыв о лекции

Владислав Хорев
Ведущий программист