

# Модель памяти и хранение данных

Владислав Хорев  
Ведущий программист, Luxoft



# Владислав Хорев

О спикере:

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



# Вспоминаем прошрое занятие

**Вопрос:** что такое функция?



# Вспоминаем прошрое занятие

**Вопрос:** что такое функция?

**Ответ:** блок кода, который имеет своё имя,  
и который можно вызывать из других  
функций



# Вспоминаем прошрое занятие

**Вопрос:** как вернуть значение из функции?



# Вспоминаем прошрое занятие

**Вопрос:** как вернуть значение из функции?

**Ответ:** использовать ключевое слово `return` и указать правильный возвращаемый тип у функции



# Вспоминаем прошрое занятие

**Вопрос:** как привести переменную double  
к типу int?



# Вспоминаем прошрое занятие

**Вопрос:** как привести переменную double к типу int?

**Ответ:** использовать оператор `static_cast<double>()`





# Вспоминаем прошрое занятие

**Вопрос:** что такое рекурсия?



# Вспоминаем прошрое занятие

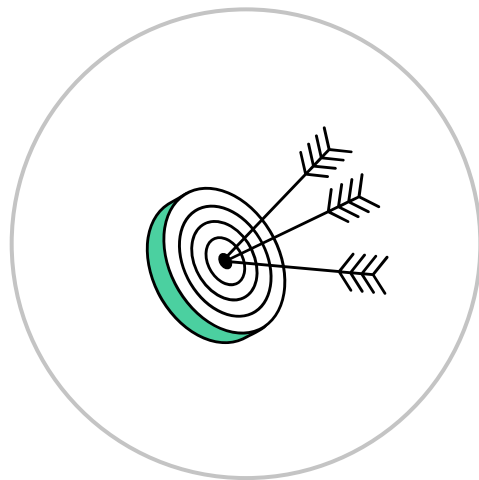
**Вопрос:** что такое рекурсия?

**Ответ:** это вызов функцией самой себя



# Цели занятия

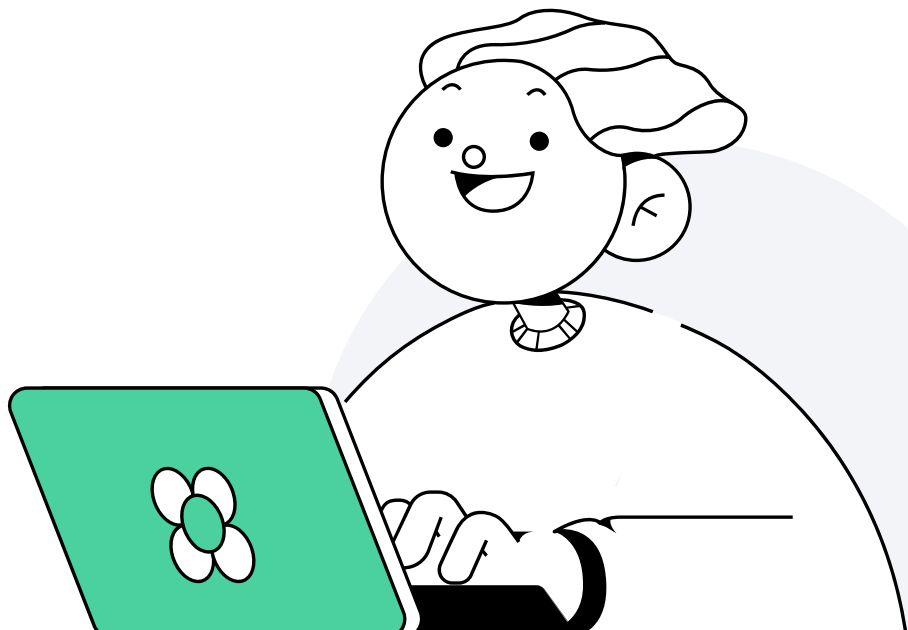
- Разберёмся с моделью памяти в C++
- Посмотрим, как можно узнать адрес и размер переменных
- Узнаем, что происходит с переменными при вызове функций
- Выясним, как передавать переменные по ссылке



# План занятия

- 1 Хранение переменных в памяти
- 2 Хранение переменных при вызове функций
- 3 Область видимости
- 4 Передача переменной по ссылке
- 5 Итоги
- 6 Домашнее задание

\*Нажми на нужный раздел для перехода



# Хранение переменных в памяти



1

# Переменные в C++

Мы с вами уже умеем создавать переменные. Вот простой пример: объявим переменную **number** типа **int**:

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

Казалось бы, ничего сложного. Но только до тех пор, пока мы не попытаемся ответить на вопрос: **что на самом деле происходит**, когда мы пишем «**int** number;»?

Чтобы ответить на этот вопрос, нам нужно узнать, как C++ хранит переменные в памяти компьютера

# Как устроена память

Память — это набор ячеек, в которые можно складывать какую-нибудь информацию.

Переменная	a	b	val		c1	c2	c3	c4	c5
Значение	54	22	4511		H	e	i	i	o
Адрес (HEX)	0xFE0	0xFE1	0xFE2	0xFE3	0xFE4	0xFE5	0xFE6	0xFE7	0xFE8
Адрес (DEC)	4064	4065	4066	4067	4068	4069	4070	4071	4072

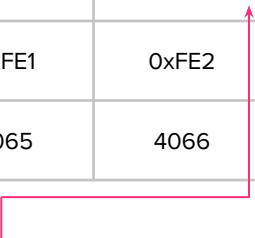
У каждой ячейки есть адрес. Размер каждой ячейки — 1 байт (8 бит).

Когда мы создаём переменную определённого типа, операционная система сообщает нам адрес, который свободен и которым можно пользоваться, чтобы складывать туда значение

# Как устроена память

Память — это набор ячеек, в которые можно складывать какую-нибудь информацию.

Переменная	a	b	val		c1	c2	c3	c4	c5
Значение	54	22	4511		H	e	i	i	o
Адрес (HEX)	0xFE0	0xFE1	0xFE2	0xFE3	0xFE4	0xFE5	0xFE6	0xFE7	0xFE8
Адрес (DEC)	4064	4065	4066	4067	4068	4069	4070	4071	4072



Некоторые типы большие, и их значения не влезают в одну ячейку, поэтому они занимают несколько ячеек подряд. Переменная в таком случае смотрит на первую ячейку из занятых.

Важно: у **каждой** переменной свой адрес



# Как узнать адрес переменной

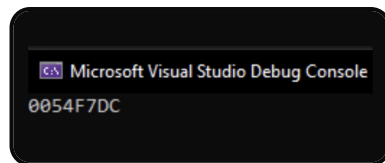
C++ позволяет нам узнать, какой адрес у той ячейки, на которую смотрит переменная.

Для этого в C++ существует специальный оператор **&** (амперсанд)

# Как узнать адрес переменной

Давайте создадим переменную и узнаем её адрес:

```
int main(int argc, char** argv)
{
    int number = 57;
    std::cout << &number << std::endl;
    return 0;
}
```



Мы видим, что адрес переменной `number` — **0054F7DC**. Это шестнадцатеричное (HEX) число.

В десятичной (DEC) форме оно записывается как **5568476**.

Это число — адрес ячейки памяти, которую нам выделила операционная система для хранения целочисленного значения. Именно на эту ячейку памяти смотрит переменная `number`, начиная с этой ячейки она хранит своё значение

# Запустим код несколько раз

```
int main(int argc, char** argv)
{
    int number;
    std::cout << &number << std::endl;
    return 0;
}
```

Microsoft Visual Studio Debug Console

010FFE04

Microsoft Visual Studio Debug Console

00D3FE94

Microsoft Visual Studio Debug Console

008DFCB4

Microsoft Visual Studio Debug Console

008FF754

Мы видим, что каждый раз у переменной number разный адрес.

Так происходит потому, что при каждом запуске программы операционная система выделяет нам разные участки оперативной памяти для работы. Поэтому переменная number при каждом новом запуске смотрит в новое место

# Важно

Оператор **&** нельзя использовать с литералами, такими как числа, а можно использовать только с тем, что хранится в памяти и имеет **ИМЯ**.

Например, это переменные и функции: функции тоже смотрят на определённый адрес.

Есть и другие, которые мы пока не знаем



# Вспоминаем оператор

Однако, чтобы нарисовать, как переменная хранится в памяти, нам недостаточно знать, на какой адрес она смотрит. Нам ещё нужно знать, сколько ячеек она использует для хранения значения. Для этого в C++ тоже существует оператор, и мы даже им пользовались.

**Кто вспомнит? Пишите в чат**

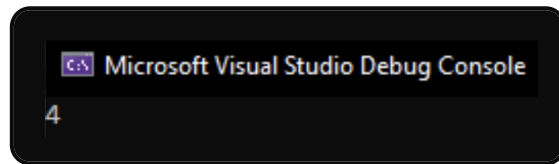


# Как узнать размер переменной

Это оператор `sizeof`.

Он возвращает размер переменной в байтах. Так как каждая ячейка памяти занимает 1 байт — `sizeof` возвращает количество ячеек, которое занимает переменная в памяти для хранения своего значения. Давайте узнаем размер переменной `number`:

```
int main(int argc, char** argv)
{
    int number;
    std::cout << sizeof(number) << std::endl;
    return 0;
}
```




Мы видим, что размер переменной `number` типа `int` составляет 4 байта, то есть переменная `number` занимает 4 ячейки памяти. В отличие от адреса, размер переменной не будет меняться от запуска к запуску. Он зависит исключительно от типа переменной, здесь это тип `int`

# Как узнать размер переменной

Оператор **sizeof** можно применять к переменным, а можно применять и к целому типу, чтобы узнать, сколько ячеек памяти будет занимать переменная этого типа. Посмотрим на примерах:

```
int main(int argc, char** argv)
{
    int number;
    double d1, d2;
    std::cout << "sizeof(number) = " << sizeof(number) << std::endl;
    std::cout << "sizeof(int) = " << sizeof(int) << std::endl;
    std::cout << "sizeof(d1) = " << sizeof(d1) << std::endl;
    std::cout << "sizeof(d2) = " << sizeof(d2) << std::endl;
    std::cout << "sizeof(double) = " << sizeof(double) << std::endl;
    return 0;
}
```

 Microsoft Visual Studio Debug Console

```
sizeof(number) = 4
sizeof(int) = 4
sizeof(d1) = 8
sizeof(d2) = 8
sizeof(double) = 8
```

# Как хранится переменная в памяти

Теперь мы можем узнать адрес ячейки, на которую смотрит переменная, и её размер — количество ячеек, которое она занимает в памяти. Значит, мы можем нарисовать, как переменная хранится в памяти.

Давайте сделаем это для переменной `number`. В качестве адреса возьмём первое полученное нами значение **0054F7DC**. Помним, что при каждом запуске адреса переменных будут меняться:


Переменная	???	number					???
Значение	???	57					???
Адрес (HEX)	0x0054F7DB	0x0054F7DC	0x0054F7DD	0x0054F7DE	0x0054F7DF	0x0054F7E0	
Адрес (DEC)	5 568 475	5 568 476	5 568 477	5 568 478	5 568 479	5 568 480	



# Как хранится переменная в памяти

Давайте посмотрим, как хранятся в памяти переменные, объявленные друг за другом:

```
int main(int argc, char** argv)
{
    int number_int_1 = 60;
    double number_double = 120.5;
    int number_int_2 = -70;
    std::cout << "number_int_1: " << &number_int_1 << " " << sizeof(number_int_1) << std::endl;
    std::cout << "number_double: " << &number_double << " " << sizeof(number_double) << std::endl;
    std::cout << "number_int_2: " << &number_int_2 << " " << sizeof(number_int_2) << std::endl;
    return 0;
}
```

 Microsoft Visual Studio Debug Console


```
number_int_1: 001AF6E4 4
number_double: 001AF6D4 8
number_int_2: 001AF6C8 4
```

Операционная система выделяет адреса переменным не как попало, а примерно в одном и том же месте

# Как хранится массив в памяти

Теперь давайте посмотрим на то, как хранится массив в памяти:

```
int main(int argc, char** argv)
{
    const int size = 3;
    int arr[size] = { 1, 2, 3 };
    std::cout << "arr: " << &arr << " " << sizeof(arr) << std::endl;
    for(int i = 0; i < size; i++)
    {
        std::cout << "arr[" << i << "]: " << &arr[i] << " " << sizeof(arr[i]) << std::endl;
    }
    return 0;
}
```

 Microsoft Visual Studio Debug Console

```
arr: 006FFE6C 12
arr[0]: 006FFE6C 4
arr[1]: 006FFE70 4
arr[2]: 006FFE74 4
```

# Как хранится массив в памяти

Теперь давайте посмотрим на то, как хранится массив в памяти:

Переменная	arr											
	arr[0]				arr[1]				arr[2]			
Значение	1				2				3			
Адрес (HEX)	0x006F FE6C	0x006F FE6D	0x006F FE6E	0x006F FE6F	0x006F FE70	0x006F FE71	0x006F FE72	0x006F FE73	0x006F FE74	0x006F FE75	0x006F FE76	0x006F FE77
Адрес (DEC)	7 339 628	7 339 629	7 339 630	7 339 631	7 339 632	7 339 633	7 339 634	7 339 635	7 339 636	7 339 637	7 339 638	7 339 639


При создании массива память под ячейки выделяется последовательно друг за другом

# Как хранятся переменные в памяти

Что происходит, когда мы присваиваем одной переменной значение другой переменной.

Давайте проверим:

```
int main(int argc, char** argv)
{
    int num1 = 5;
    int num2 = num1;
    std::cout << "num1: " << &num1 << std::endl;
    std::cout << "num2: " << &num2 << std::endl;
    return 0;
}
```

 Microsoft Visual Studio Debug Console

num1: 004FFB08

num2: 004FFAFC

Переменные имеют разные адреса. Значение, хранящееся в ячейке, на которую смотрит переменная num1, при выполнении операции присвоения копируется в ячейку, на которую смотрит переменная num2, поэтому переменные могут менять своё значение независимо друг от друга

# Хранение переменных при вызове функций



2

# Параметры в функции

```
void calc(int x)
{
    x *= 10;
    std::cout << x << std::endl; // ???
}

int main(int argc, char** argv)
{
    int value = 5;
    calc(value);
    std::cout << value << std::endl; // ???
    return 0;
}
```

Что выведет?



# Параметры в функции

```
void calc(int x)
{
    x *= 10;
    std::cout << x << std::endl; // 50
}

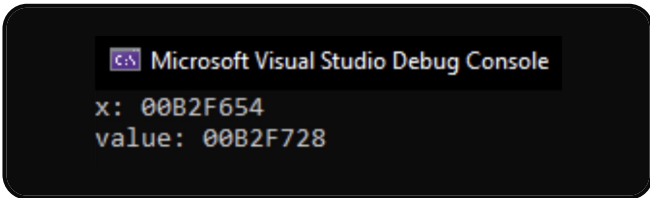
int main(int argc, char** argv)
{
    int value = 5;
    calc(value);
    std::cout << value << std::endl; // 5
    return 0;
}
```

# Параметры в функции

Почему так происходит: давайте посмотрим на адреса.

```
void calc(int x)
{
    std::cout << "x: " << &x << std::endl;
}

int main(int argc, char** argv)
{
    int value = 5;
    calc(value);
    std::cout << "value: " << &value << std::endl;
    return 0;
}
```



Адреса разные.

При вызове функции для каждого параметра создаётся новая переменная, в которую копируется значение, которое было указано для этого параметра — значение переменной или литерал



# Переменные и функции

Как сделать так, чтобы можно было внутри функции изменить значение переменной, которая находится в другой функции:

```
void calc()
{
    value *= 10;
}

int main(int argc, char** argv)
{
    int value = 5;
    calc();
    std::cout << value << std::endl;
    return 0;
}
```

К сожалению, такой код не скомпилируется.

**Как вы думаете почему?**

# Область видимости



3

# Область видимости переменных

Переменные доступны не везде и не всегда. Переменная доступна только в блоке, в котором она объявлена и во вложенных блоках и только с момента объявления.

Блок — то, что находится между { и }.

Параметрами функции можно пользоваться в любом месте функции

# Область видимости. Пример

```
int main(int argc, char** argv)
{
    // это блок, здесь можно пользоваться переменными argc и argv
    int x = 5;
    // а здесь можно пользоваться и переменной x
    // вплоть до конца блока, то есть до конца функции main
    {
        // это новый блок – вложенный
        int value = 7;
        std::cout << value << std::endl;
    } // конец вложенного блока
    // std::cout << value << std::endl; ошибка: предыдущий блок закончился,
    // поэтому переменная value больше недоступна
    return 0;
}
```

# Область видимости переменных

Если нам нужно получить доступ к одной переменной из разных мест, объявим её на самом верху

```
int value;
void calc()
{
    value *= 10;
}

int main(int argc, char** argv)
{
    value = 5;
    calc();
    std::cout << value << std::endl; // ???
    return 0;
}
```

Что выведет?

# Область видимости переменных

Если нам нужно получить доступ к одной переменной из разных мест, объявим её на самом верху

```
int value;
void calc()
{
    value *= 10;
}

int main(int argc, char** argv)
{
    value = 5;
    calc();
    std::cout << value << std::endl; // 50
    return 0;
}
```

# Область видимости переменных

Код действительно сработает и сделает то, что нам нужно. Этого мы добились, объявив переменную на самом верху, вне всяких функций. Такие переменные называются глобальными.

**Но** глобальные переменные — это почти всегда проблемы.

**Как вы думаете почему?**

# Глобальные переменные

Мы не можем контролировать доступ к переменной. Её может изменить кто угодно, как угодно и когда угодно. Когда мы захотим воспользоваться ей, нет гарантий, что её кто-то не испортил.

Бывают ситуации, когда глобальные переменные оправданы, но это крайне редкие ситуации, на которые стоит идти, хорошо обдумав свои действия.

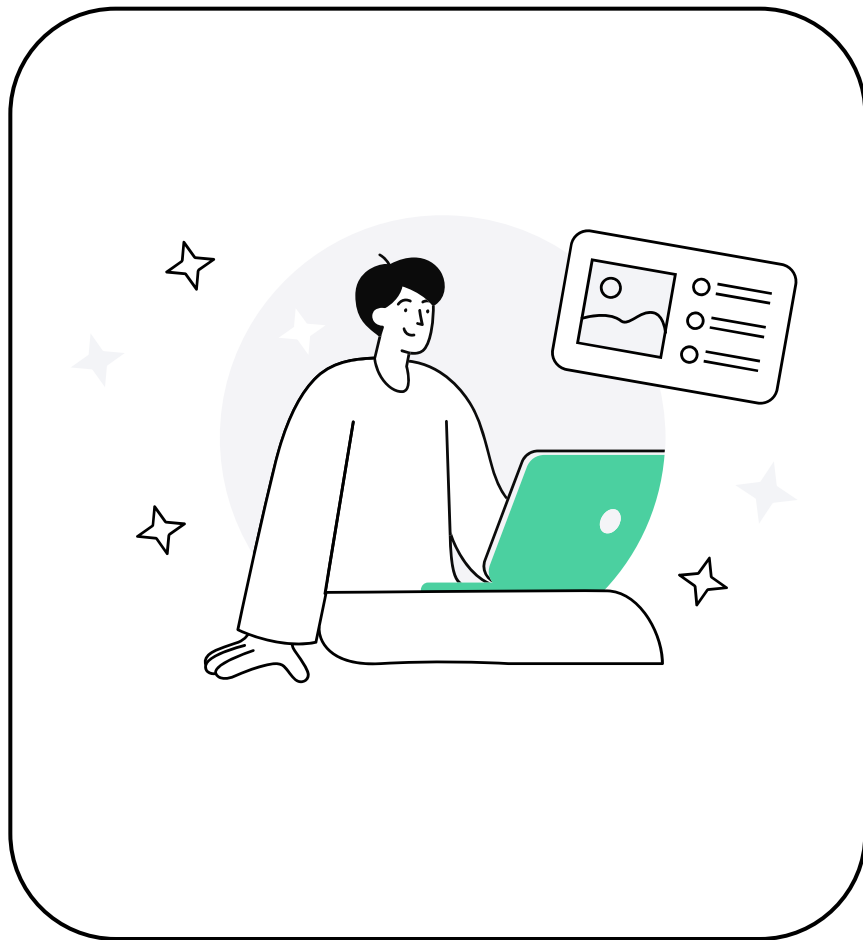
Общее правило при работе с глобальными переменными:

«Скорее всего вам не нужны глобальные переменные. Вероятнее всего была совершена ошибка при проектировании кода. Нужно найти и исправить ошибку»



# Избавляемся от глобальной переменной

Как можно избавиться  
от глобальных переменных



# Первый вариант

Мы можем возвращать из функции значение и присваивать его там, откуда вызывали функцию

```
int calc(int value)
{
    return value * 10;
}

int main(int argc, char** argv)
{
    value = 5;
    value = calc(value);
    std::cout << value << std::endl; // 50
    return 0;
}
```

# Второй вариант

Второй вариант предполагает использование нового механизма — **передачу**  
**переменной по ссылке**

# Передача переменной по ссылке



4

# Передача переменной по ссылке

Чтобы передать переменную по ссылке, используется знак **&**, как для операции взятия адреса, которую мы уже видели.

Однако применяется он по-другому. Если для взятия адреса необходимо было написать **&** сразу перед именем переменной, адрес которой мы хотим узнать, то для передачи переменной по ссылке **нужно указать знак & сразу после названия типа в параметрах функции**

# Передача переменной по ссылке

```
void calc(int& x)
{
    x *= 10;
    std::cout << x << std::endl; // 50
}

int main(int argc, char** argv)
{
    int value = 5;
    calc(value);
    std::cout << value << std::endl; // 50
    return 0;
}
```

# Передача переменной по ссылке

## Важно

Передача массива в функцию в качестве параметра имеет свои особенности, которые мы увидим в следующей лекции

# Итоги





# Итоги занятия

Сегодня мы

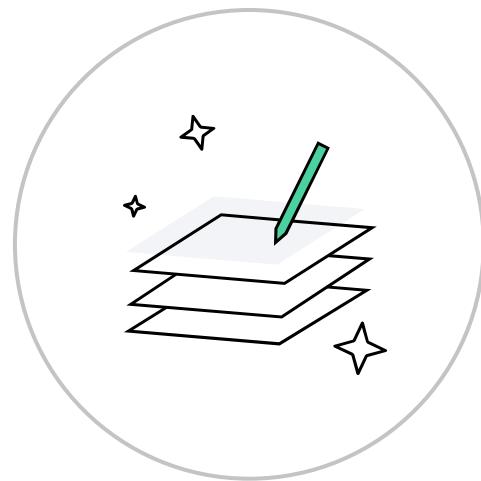
- 1 Разобрались с моделью памяти в C++
- 2 Посмотрели, как можно узнать адрес и размер переменных
- 3 Узнали, что происходит с переменными при вызове функций
- 4 Выяснили, как передавать переменные по ссылке



# 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



# Дополнительные материалы

- [Область видимости](#)



# Задавайте вопросы и пишите отзыв о лекции

Владислав Хорев  
Ведущий программист, Luxoft

