Основы языков C/C++

Кафедра ЭО

Петрухин О.М.

Хранение переменных в памяти

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

```
int main()
{
    int number;
    return 0;
}
```

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.

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.

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).

```
int main()
{
    int number = 57;
    std::cout << &number << std::endl;
    return 0;
}
```

В результате выполнения программы, вы увидите, что адрес переменной `number` равен например **0x0054F7DC**.

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.

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.

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.

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

Особенности вызова функций

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.

Сделаем это, например для переменной `number`:

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

Хранение переменных при вызове функций

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

int main()
{
    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 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

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

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

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 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»

Как же получить доступ к переменной?

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

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

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

Передача параметра по ссылке

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

Чтобы передать переменную по ссылке, используется знак &, как для операции взятия адреса. Для передачи переменной по ссылке **нужно указать знак & сразу после названия типа в параметрах функции.**

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

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

Адреса переменных

Вернёмся к адресам. Проведём эксперимент. & (амперсанд) — это оператор взятия адреса. Он позволяет нам узнать адрес ячейки памяти, на которую «смотрит» переменная. Раньше мы всё время просто выводили адрес на консоль. Давайте попробуем сохранить его (т. е. результат работы оператора &) в переменную.

```
int main()
{
    int number = 5;
    int address = &number; // Ошибка компиляции. Несоответствие типа значения
                           // типу переменной
    return 0;
}
```

Указатель

Чтобы узнать тип значения адреса, используем оператор `typeid`.

```
int main()
{
    int number = 5;
    std::cout << "type of &number: " << typeid(&number).name() << std::endl;
    return 0;
}
```

```
type of &number: Pi
```

Оператор взятия адреса при применении к типу `int` возвращает значение типа `int*`.

`int*` — это один из типов в C++, который используется для указателей.

Указатель — это переменная, значение которой — адрес ячейки памяти.

Объявляется указатель так: **<тип данных> * <имя указателя>;**

Тип указателя содержит информацию о том, на значение какого типа он указывает.

Указатель типа `int*` будет содержать в себе адрес значения типа `int`, указатель типа

`double*` будет указывать на значение типа `double`.

Указатель

Тип указателя содержит информацию о том, на значение какого типа он указывает.

Указатель типа `int*` будет содержать в себе адрес значения типа `int`, указатель типа `float*` будет указывать на значение типа `float`.

Теперь сохраним адрес нашей целочисленной переменной в другую переменную:

```
int main()
{
    int number = 5;
    int * p_number = &number; // Инициализируем указатель сразу при объявлении
    int * p_number2; // Сначала объявляем
    p_number2 = &number; // Потом инициализируем
    std::cout << "p_number: " << p_number << std::endl;
    std::cout << "p_number2: " << p_number2 << std::endl;
    return 0;
}
```

Зачем это нужно?

Зачем же нам нужны все эти указатели? Какой смысл в том, чтобы сохранять адрес в переменных?

С помощью указателей мы можем не только передавать адрес в функции, но и получать **доступ к содержимому ячейки**, в том числе **изменять это содержимое**.

Оператор разыменования

Получить доступ к ячейке, адрес которой хранится в указателе, можно с помощью **оператора разыменования** — * (звёздочка).

Чтобы разыменовать указатель, необходимо написать оператор разыменования **перед** указателем: так же, как мы берём адрес переменной.

Взятие адреса (&) и разыменование (*) — это обратные операции:

```
int main()
{
    int number = 5;
    std::cout << *&number << std::endl; // 5
    return 0;
}
```

Разыменование. Пример

Изменим содержимое ячейки памяти с помощью указателя:

```
int main()
{
    int number = 5;
    int * p_number = &number;
    std::cout << "number: " << number << std::endl; // 5
    std::cout << "*p_number: " << *p_number << std::endl; // 5
    *p_number = 10;
    std::cout << "number: " << number << std::endl; // 10
    std::cout << "*p_number: " << *p_number << std::endl; // 10
    return 0;
}
```

Мы изменили значение переменной без её участия. Такие возможности даёт нам указатель

Передача параметра в функцию по указателю

```
void changeByPointer(int* ptr) {
    *ptr = 10; // Увеличиваем значение, на которое указывает ptr
}

int main(){
    int num = 5;

    std::cout << "До вызова функции: " << num << std::endl;

    changeByPointer(&num);  // Передаем адрес переменной num в функцию

    std::cout << "После вызова функции: " << num << std::endl;
    return 0;
}
```

КАИ
Крылья для жизни!

Спасибо за внимание!