Лекция 3: Типы данных, переменные и операции. Говорим на языке компьютера.

От абстракции к конкретике

На прошлых занятиях мы познакомились с миром C# и .NET, написали первую программу и обсудили общие принципы программирования. Сегодня мы переходим к фундаментальным кирпичикам, из которых строится любая программа: **типам данных, переменным и операциям** над ними.

«В информатике всегда есть две сложные проблемы: инвалидация кеша, именование вещей и ошибка на единицу.» — Леон Бамбирик (шутливая вариация на тему Филла Карлтона).

1. Переменные: Контейнеры для данных

Представьте себе коробку. Вы можете положить в нее что угодно: книгу, яблоко, игрушку. Чтобы не путаться, вы подписываете коробку. **Переменная** в программировании — это такая же "коробка" (ячейка в памяти компьютера) с "именем" (идентификатором), в которой хранится какое-то значение.

2. Типы данных: Какие бывают "коробки"?

Нельзя положить суп в бумажный пакет, а гвозди — в стеклянную банку без последствий. Так и в программировании: для разных данных нужны разные "контейнеры" — **типы данных**. C# — это язык со **строгой статической типизацией**. Это значит, что если вы объявили переменную как `int`, она до конца своих дней будет хранить только целые числа.

Как это устроенно?

Давайте представим, что компьютерная память — это огромная стена с пронумерованными ящиками для хранения. Каждый такой ящик — это **байт**. Когда вы создаете в программе переменную, вы, по сути, просите компьютер: "Выдели мне один такой ящик (или несколько), чтобы я положил туда своё число".

Но здесь возникает первый важный вопрос: **сколько места нужно выделить?** Ведь числа бывают разные: маленькие (например, 5 или 100) и очень большие (например, 5 миллиардов). Если под все числа выделять ящики одного размера, это будет неэффективно.

Компьютер должен точно знать, где ваше число **начинается** и где **заканчивается**, иначе он может перепутать его с данными другой программы.

Пример с числом 9

Допустим, мы хотим сохранить число 9. Компьютер не понимает запись "9" в привычном нам виде. Внутри него всё хранится в виде нулей и единиц — в двоичной системе. Давайте переведем 9 в двоичный код:

9 (в десятичной) = 1001 (в двоичной)

Теоретически, чтобы записать 1001, достаточно 4 бит (каждый 0 или 1 — это один бит). Но вот ключевой момент: **компьютер не умеет работать с отдельными битами при выделении памяти.** Минимальная "порция", которую он может выдать, — это один **байт**, который состоит из 8 бит.

Поэтому нашему числу 1001 выделят целый байт. Чтобы заполнить все 8 бит, его дополнят незначащими нулями слева. В итоге в памяти оно будет выглядеть так:

0000 1001

А что с отрицательными числами?

Следующая задача: как в этой же системе записать отрицательное число, например, -9?

Первая мысль — выделить отдельный байт (или хотя бы бит) под знак "+" или "-". Но выделять целый байт (8 бит!) под одну лишь галочку — это огромная расточительность.

Разработчики нашли остроумное решение: они договорились использовать **первый, самый старший бит** в байте (или в группе байтов) в качестве знакового.

- Если первый бит равен 0 — число **положительное**.
- Если первый бит равен 1 — число **отрицательное**.

Такой подход называется **представление числа со знаком**.

Давайте применим это к нашему числу 9 (0000 1001). Первый бит у нас 0, всё верно, число положительное.

А как будет выглядеть -9? По этой логике, мы просто меняем первый бит на единицу:

1000 1001

Именно так (в упрощенном виде) компьютер и отличает положительные числа от отрицательных, не выделяя под знак отдельную ячейку памяти.

Вывод:
Когда вы в C# объявляете переменную, например, int number = 9;, вы не просто сохраняете число. Вы говорите компьютеру:

1. Выделить память определенного размера (для int это 4 байта).
2. Интерпретировать эту память как число со знаком (то есть первый бит будет указывать на знак).
3. Записать число 9 в двоичном виде в эти 4 байта, соблюдая все соглашения.

Понимание этой связи между типом переменной, выделяемой памятью и способом хранения — ключ к глубокому пониманию программирования.

Целочисленные типы

Используются для хранения целых чисел. Различаются по **диапазону значений** и по **объему занимаемой памяти**.

Тип C#	Размер (байт)	Диапазон	Пример использования
`sbyte`	1	-128 до 127	Уровень громкости (-100%, 0%, 100%)
`byte`	1	0 до 255	Цвет канала (RGB, от 0 до 255)
`short`	2	-32,768 до 32,767	Год рождения (до нашей эры/нашей эры)
`ushort`	2	0 до 65,535	Количество пикселей по ширине (малое изображение)

Тип C#	Размер (байт)	Диапазон	Пример использования
`int`	4	~ -2.1 млрд до ~ 2.1 млрд	**Самый часто используемый.** Счетчики, возраст, идентификаторы.
`uint`	4	0 до ~ 4.2 млрд	Большие неотрицательные значения.
`long`	8	Огромный диапазон	Число жителей Земли, большие финансовые операции.
`ulong`	8	Огромный неотрицательный диапазон	Астрономические расчеты.

Как это устроено в памяти? Переменная типа `int age = 17;`:

1. Компьютер находит в стеке свободную ячейку размером 4 байта.
2. Преобразует число 17 в двоичную систему: `00000000 00000000 00000000 00010001`.
3. Записывает эти 4 байта в выделенную ячейку.
4. С этого момента имя `age` является **ссылкой** на эту ячейку.

Примечание: Как можно было понять, тип с буквой u – это беззнаковый тип. (unsigned). Интересный факт – изначально не было “вторых” типов, т.е изначально int и выше задумывались как знаковые типы, а уже потом придумали беззнаковые варианты. Но с byte ситуация обратная - Тип byte изначально и по своей сути был задуман как беззнаковый. Он предназначен для работы с сырыми данными, кодами символов (вроде ASCII), где отрицательные значения не нужны.

Вещественные типы (с плавающей запятой)

Используются для хранения дробных чисел. Хранятся в формате IEEE 754.

Научная нотация

Вспомним, как мы записываем очень большие или очень маленькие числа:

- **6 020 000 000 000 000 000 000 000** = **6.02×10^{23}**
- **0.000 000 001** = **1.0×10^{-9}**

Здесь есть три важные части:

1. **Знак:** Плюс или минус (здесь везде плюс).
2. **Мантисса (significand):** Основные значащие цифры числа (**6.02**, **1.0**).
3. **Экспонента (порядок):** Степень десятки (**23**, **-9**), которая показывает, насколько далеко сдвинута запятая.

Компьютер делает то же самое, но в двоичной системе (только 0 и 1) и хранит всё в виде битов.

Перевод в двоичную систему и "плавающая запятая"

Представьте, что у нас есть число **5.25**.

1. Переведем его в двоичный вид:
    - Целая часть (5) — это `101` в двоичной системе.
    - Дробная часть (0.25) — это `01` (потому что 0.25 = 1/4 = $1/2^2$).
    - Итого: **5.25 = 101.01_2** (где $_2$ — обозначение двоичной системы).
2. Теперь применим "научную нотацию" к двоичному числу. Запятая должна стоять **после первой значащей единицы**.
    - `101.01` = `1.0101 × 2²`
    - Почему на 2^2? Потому что мы сдвинули запятую на 2 разряда влево (как в десятичной системе сдвиг на 2 разряда влево — это умножение на 10^2).

Теперь у нас есть три компонента для хранения, как и в научной нотации:

- **Знак:** `0` (потому что число положительное). Если бы было отрицательное — `1`.
- **Экспонента (порядок):** `2` (это наша степень двойки).
- **Мантисса:** `0101` (это то, что было после запятой, `1.0101` — единицу мы убрали, она подразумевается, об этом ниже).

Стандарт IEEE 754: "Упаковка" в биты

Стандарт IEEE 754 — это как ГОСТ для чисел. Он говорит, как именно упаковать эти три части в байты памяти. Самый популярный формат — **одинарной точности (float, 32 бита)**.

Давайте "упакуем" наше число 5.25 в 32 бита:

Знак (1 бит)	Экспонента (8 бит)	Мантисса (23 бита)
0	10000001	01010000000000000000000

Теперь разберем каждое поле:

1. Знак (1 бит):

- `0` — положительное
- `1` — отрицательное

2. Экспонента (8 бит):

- Здесь есть хитрость. Экспонента может быть отрицательной (для очень маленьких чисел). Чтобы не хранить знак отдельно, используют **смещение (bias)**.
- Для 32-битного формата смещение = **127**.
- Наша реальная экспонента была `2`. Чтобы получить хранимое значение, делаем: `2 + 127 = 129`.
- Переводим 129 в двоичную систему: `10000001`.
- Такой трюк позволяет легко сравнивать числа побитово.

3. Мантисса (23 бита):

- Мы помним, что наше число было `1.0101 × 2²`.

- **Ведущая единица (`1` перед точкой) — это "секретный ингредиент"!** Её **не хранят** в памяти, чтобы сэкономить один бит и получить чуть большую точность. Это называется **"скрытый бит"** или **"implied bit"**.
- В мантиссу мы записываем **только дробную часть** после точки, то есть `0101`.
- Остальные биты до 23 заполняем нулями справа.

Что в итоге получается?

Когда компьютер видит эти 32 бита:

1. Он смотрит на знак: `0` — значит, число положительное.
2. Он берет экспоненту (`10000001` = 129) и вычитает смещение: `129 - 127 = 2`.
3. Он берет мантиссу (`0101000...`) и **добавляет обратно ту самую ведущую единицу**, получая `1.0101000...`.
4. Он собирает число: **± (1 + мантисса) × 2^(экспонента-смещение)**
    - В нашем случае: `+ (1.0101) × 2²` = 101.01_2 = **5.25**.

Особые случаи и почему бывают "глюки" с дробями"

1. **Точность:** У нас всего 23 бита под мантиссу. Это как листок бумаги, на котором можно записать только ~7 значащих десятичных цифр. Если число очень длинное, его хвост обрежется. Отсюда ошибки вычислений.
    - Классический пример: `0.1 + 0.2 != 0.3`. Потому что десятичная дробь 0.1 в двоичной системе — это бесконечная периодическая дробь (`0.0001100110011...`), как 1/3 в десятичной (`0.333...`). Её нельзя точно представить в точное число бит, происходит округление.
2. **Ноль:** Чтобы представить ноль, экспонента и мантисса состоят из одних нулей.
3. **Бесконечность:** Если экспонента состоит из всех единиц (`11111111`), а мантисса — из нулей, это `±∞`. Возникает при делении на ноль.
4. **Не число (NaN - Not a Number):** Если экспонента из всех единиц, а мантисса не нулевая — это `NaN`. Получается при операциях вроде `0/0` или `√(-1)`. Это как компьютер говорит: "Результат этой операции не имеет смысла!".

Итог простыми словами:

Представь, что число с плавающей точкой — это **координата на линейке**, где:

- **Знак** — говоришь "влево" от нуля или "вправо".
- **Экспонента** — выбираешь **грубый масштаб** (сантиметры, миллиметры, километры).
- **Мантисса** — ставишь **точную метку** в выбранном масштабе.

Эта система гениальна, потому что позволяет одним и тем же способом хранить и атомы (очень маленькие числа), и расстояния до галактик (очень большие числа), но за это приходится платить небольшой потерей точности.

Тип C#	Размер (байт)	Диапазон и точность	Пример
`float`	4	~ $\pm 1.5 \times 10^{-45}$ до $\pm 3.4 \times 10^{38}$ (7 значащих цифр)	Простые расчеты, координаты в играх.
`double`	8	~ $\pm 5.0 \times 10^{-324}$ до $\pm 1.7 \times 10^{308}$ (15-16 значащих цифр)	**Самый часто используемый.** Научные расчеты, финансовые операции (с осторожностью).
`decimal`	16	~ $\pm 1.0 \times 10^{-28}$ до $\pm 7.9 \times 10^{28}$ (28-29 значащих цифр)	Точные финансовые расчеты, где важна каждая копейка.

Для `float` и `decimal` нужно добавлять суффиксы:

```
float gravity = 9.81f;   // Суффикс 'f' обязателен!
double pi = 3.14159;    // Можно без суффикса, по умолчанию double
decimal money = 100.5m; // Суффикс 'm' для decimal
```

ВАЖНО!

C# допускает неявное преобразование из float в double. То есть

```
double a = 9.81f;  // Корректно!
```

Это будет работать без ошибок, потому что в C# существует неявное преобразование из `float` в `double`.

Почему это разрешено:

1. **Безопасность преобразования**: `double` имеет большую точность (64 бита) и диапазон, чем `float` (32 бита). При преобразовании из `float` в `double` не происходит потери данных - наоборот, число получает больше "пространства" для хранения.
2. **Автоматическое расширение**: Компилятор C# разрешает неявные преобразования, когда происходит "расширение" типа (widening conversion), то есть переход от типа с меньшей точностью/диапазоном к типу с большей.

Что происходит на практике:

```
float floatValue = 9.81f;      // 32-битное число
double doubleValue = floatValue; // 64-битное число (расширение)

Console.WriteLine(floatValue);   // Выведет: 9,81
```

```
Console.WriteLine(doubleValue);  // Выведет: 9,8100004196167
```

Обратите внимание, что при выводе вы можете увидеть небольшую разницу в младших разрядах. Это происходит потому, что `float` имеет ограниченную точность (около 7 значащих цифр), и когда мы преобразуем его в `double`, мы видим "артефакты" двоичного представления числа с плавающей точкой.

Обратная ситуация НЕ разрешена:

```
// float b = 9.81;  // ОШИБКА! Нельзя неявно преобразовать double в float
float c = (float)9.81;  // Явное преобразование (но возможна потеря точности)
```

Кстати, `decimal c = 105.5` не сработает, так как это не расширение типа, а преобразование между принципиально разными системами:

- double - двоичная арифметика с плавающей точкой
- decimal - десятичная арифметика с фиксированной точкой

Логический тип

Тип C#	Размер	Значения	Пример
`bool`	~1 байт	`true` (истина, 1) или `false` (ложь, 0)	Флаги состояния, ответы "да/нет".

```
bool isActive = true;
bool isEmpty = false;
```

Использование `bool` вместо `byte` в C# для логических операций важно по нескольким причинам:

1. **Память**:
    - `bool` занимает **1 байт**, но семантически представляет только два значения: `true`/`false`.
    - `byte` также занимает 1 байт, но может хранить значения от 0 до 255, что избыточно для логики.
2. **Оптимизации**:
    - Компилятор и JIT-оптимизации учитывают, что `bool` имеет только два значения. Например, при работе с массивами `bool[]` может использоваться более эффективное хранение (например, побитовое сжатие в некоторых контекстах).
    - Для `byte` такие оптимизации не применяются, так как он treated как число.

3. **Читаемость и безопасность**:
    - `bool` явно указывает на логическую семантику, предотвращая ошибки (например, случайное использование `2` вместо `0`/`1`).
    - Пример:

```
o bool isActive = true; // Ясно и безопасно
o byte isActive = 1;    // Неочевидно, допускает неверные значения (например, 5)
```

4. **Производительность**:
    - Логические операции с `bool` (например, `&&`, `||`) компилируются в эффективные IL-инструкции (например, `brtrue`/`brfalse`), тогда для `byte` требуются сравнения (например, `!= 0`).

`bool` обеспечивает ясность, безопасность и потенциальные оптимизации, тогда как `byte` избыточен и error-prone для логических значений.

Символьный тип

Тип C#	Размер	Значения	Пример
`char`	2 байта	Один символ Unicode (от U+0000 до U+FFFF)	Буква, цифра, специальный символ.

```
char firstLetter = 'A'; // Одиночные кавычки!
char newLine = '\n';    // Управляющая последовательность для новой строки
char copyright = '\u00A9'; // Символ © через Unicode
```

Память под `char` занимает 2 байта (в отличие от 1 байта в старых языках), что позволяет работать с огромным количеством символов разных языков мира.

Объявление и инициализация

Чтобы начать пользоваться переменной, её нужно сначала объявить, а потом (или сразу) инициализировать.

```
// 1. Объявление (создаем "коробку" с именем 'age')
int age;

// 2. Инициализация (кладем в 'age' значение 17)
age = 17;
```

```
// 3. Объявление и инициализация одновременно (создаем и сразу наполняем)
string name = "Анна"; //Очевидно, что string – строка, но ее работу мы рассмотрим позже, по некоторым причинам.
double averageScore = 4.75;
bool isStudent = true;
```

Простые правила для имен переменных:

- Могут содержать буквы, цифры и символ подчеркивания `_`.
- Не могут начинаться с цифры.
- Регистрозависимы: `myVar` и `myvar` — это две разные переменные.
- Не могут совпадать с ключевыми словами языка (`int`, `class`, `void` и т.д.).

3. Операции: Что мы можем делать с "коробками"?

Операции позволяют манипулировать данными, хранящимися в переменных.

Арифметические операции

Оператор	Операция	Пример (`int a=10, b=3;`)
`+`	Сложение	`a + b = 13`
`-`	Вычитание	`a - b = 7`
`*`	Умножение	`a * b = 30`
`/`	Деление	`a / b = 3` (целочисленное деление!)
`%`	Остаток от деления	`a % b = 1` (остаток от 10/3)

Важно! Тип результата операции определяется типами операндов.

```
double result = 10 / 3;   // result = 3, т.к. оба операнда - int
double correctResult = 10.0 / 3; // correctResult ≈ 3.333... т.к. один операнд - double
```

Операции присваивания

Оператор	Пример	Эквивалент
`=`	`x = 5;`	`x = 5;`
`+=`	`x += 3;`	`x = x + 3;`

Оператор	Пример	Эквивалент
`-=`	`x -= 2;`	`x = x - 2;`
`*=`	`x *= 4;`	`x = x * 4;`
`/=`	`x /= 2;`	`x = x / 2;`
`%=`	`x %= 3;`	`x = x % 3;`

«Преждевременная оптимизация — корень всех зол.» — Дональд Кнут.

Операции сравнения

Возвращают значение типа `bool` (`true` или `false`).

Оператор	Проверка	Пример (`int a=5, b=10;`)
`==`	Равенство	`a == b` // false
`!=`	Неравенство	`a != b` // true
`>`	Больше	`a > b` // false
`<`	Меньше	`a < b` // true
`>=`	Больше или равно	`a >= b` // false
`<=`	Меньше или равно	`a <= b` // true

Логические операции

Работают с операндами типа `bool`. Используются для объединения условий.

Оператор	Операция	Пример
`!`	Логическое НЕ (унарный)	`!true` // false
`&&`	Логическое И (бинарный)	`true && false` // false
`||`	Логическое ИЛИ (бинарный)	`true && false` // true

Таблицы истинности: Логическое И (&&)

A	B	A && B
true	true	true
true	false	false
false	true	false
false	false	false

Логическое ИЛИ (||)

A	B	A || B
true	true	true
true	false	true
false	true	true
false	false	false

Унарные операции

Работают с одним операндом.

Оператор	Операция	Пример (`int x=5;`)
`+`	Унарный плюс	`+x` // +5
`-`	Унарный минус	`-x` // -5
`++`	Инкремент (увеличение на 1)	`x++` // вернет 5, затем x=6
`--`	Декремент (уменьшение на 1)	`--x` // x=4, вернет 4

Разница между префиксной (`++x`) и постфиксной (`x++`) формой важна в выражениях:

```
int a = 5;
int b = a++; // b = 5, a = 6 (постфиксная: сначала присвоение, потом инкремент)
```

```
int c = 5;

int d = ++c; // d = 6, c = 6 (префиксная: сначала инкремент, потом присвоение)
```

Тернарная операция

Единственная операция, работающая с тремя операндами. Своего рода сокращенная запись `if-else`.

Синтаксис: `условие ? выражение_если_истина : выражение_если_ложь`

```
int age = 17;

string status = (age >= 18) ? "Совершеннолетний" : "Несовершеннолетний";

Console.WriteLine(status); // Выведет: "Несовершеннолетний"
```

Приоритет операций

Как в математике: умножение и деление выполняются раньше сложения и вычитания. Чтобы изменить порядок, используют **скобки** `()`.

1. `()` (скобки)
2. `++`, `--` (постфиксные/префиксные), `!`, `+`, `-` (унарные)
3. `*`, `/`, `%`
4. `+`, `-` (бинарные)
5. `<`, `>`, `<=`, `>=`
6. `==`, `!=`
7. `&&`
8. `||`
9. `=`, `+=`, `-=`, `*=` и т.д.

Пример:

```
int result = 10 + 2 * 3;        // result = 16 (2*3=6, затем 10+6)

int resultWithBrackets = (10 + 2) * 3; // resultWithBrackets = 36
```

4. Неявная типизация (`var`)

Иногда компилятор может сам "догадаться" о типе переменной по присваиваемому значению. Для этого используется ключевое слово `var`.

```
var number = 42;         // компилятор видит 42 и понимает, что это int

var greeting = "Hello"; // компилятор видит строку и понимает, что это string

var flag = true;         // компилятор понимает, что это bool
```

```
// var error; // ОШИБКА! Компилятор не может определить тип без значения.
```

Важно! `var` — это не "тип без типа". Это указание компилятору **самому вывести тип на этапе компиляции**. После этого переменная становится строго типизированной, как если бы вы объявили её через `int`, `string` и т.д.

«Программы должны быть написаны так, чтобы люди их читали, и лишь во вторую очередь — чтобы машины их исполняли.» — Джеральд Сассман и Гарольд Абельсон, авторы "Structure and Interpretation of Computer Programs"

ВАЖНОЕ ЗАМЕЧАНИЕ В КОНЦЕ ЛЕКЦИИ!

Ранее (во 2 лекции) мы рассматривали работу с консолью, и если быть точнее, работу ReadLine. Напомним, что ReadLine возвращает тип string (строка). Полноценную работу со строками мы рассмотрим в будущих лекциях, и там подробнее разберем преобразование строк в числа. В текущий момент, для выполнения заданий по курсу, пользуйтесь преобразованием Parse, и вводом по принципу "1 строка – 1 число".

Например, для ввода числа 17:

```
int a = int.Parse(Console.ReadLine());
```

Итог

Сегодня мы заложили мощный фундамент:

- Узнали, что такое **переменные** и как их создавать.
- Подробно разобрали **значимые типы данных** и то, как они хранятся в памяти компьютера.
- Изучили множество **операций**: от арифметических до логических и тернарных.
- Познакомились с неявной типизацией `var`.

Это основа, на которой будет строиться все дальнейшее обучение. На следующей лекции мы научимся управлять ходом выполнения программы с помощью условных операторов `if-else` и циклов.

«Лучше всего писать код так, как будто его сопровождать будет склонный к насилию психопат, который знает, где вы живете.» — Джон Вудс.