Введение в язык программирования C.

Лекция 2

Сергей Леонидович Бабичев

Операции

Операция присваивания

- Имеется *левая часть* операции присваивания
- *l-значения* — то, что может находиться в левой части.
- *переменные* относятся к *l-значениям*.
- *r-значения* — то, что может находиться в правой части.

```
12345 = i;  // No passaran
123 + a = b // No passaran
b = 123 + a // OK;
```

Приведение типов

Преобразование из более длинного целочисленного типа в более короткий производится *отбрасыванием* «лишних» битов.

1. Вещественные типы старше целочисленных.
2. Беззнаковые типы старше знаковых.
3. Длинные типы старше коротких.

Особенности операций

Результаты операций рассматриваются просто как числа с тем же количеством бит. Лишние биты просто отбрасываются.

```
int x = 2000000000, y = 2000000000;
int z = x + y; // z < 0 here.
```

Особенности операций деления / и остатка %

- Если делимое и делитель — неотрицательны, то всё традиционно, `7 / 3 = 2`, а `7 % 3 = 1`.
- Если или делимое или делитель отрицательны, то знак остатка совпадает со знаком делителя. Так в математике и `Python`.
- `Си` использует аппаратные особенности компьютеров, на которых он исполняется, а они могут эти правила не соблюдать. На `Intel/AMD/ARM`

```
int d = -17 / 10; // d <- -1
int e = -17 % 10; // e <- -7
```

- Это не `Python` и не `Pascal`! Операция деления — одна, `/`.
- Для вещественных чисел операции остатка нет.

Побитовые операции

Их немного:

- `&` — побитовое *и*;
- `|` — побитовое *или*;
- `^` — *побитовое исключающее или*;
- `~` — побитовое *не*;
- `<<` — *сдвиг влево*;
- `>>` — *сдвиг вправо*;

Они применяются к каждому из битов в операнде. Всё происходит строго по законам алгебры логики. Если один из операндов короче другого, он расширяется по правилам приведения типов.

Побитовые операции

Операция *побитовое и* имеет следующую таблицу истинности:

```
0 & 0 = 0
0 & 1 = 0
1 & 0 = 0
1 & 1 = 1
```

Операция *побитовое или* имеет следующую таблицу истинности:

```
0 | 0 = 0
0 | 1 = 1
1 | 0 = 1
1 | 1 = 1
```

Примеры побитовых *и* и *или*

```
int ia = 3;        //       00000000 00000000 00000000 00000011
char cb = 5;       //                                  00000101
int ic = -3;       //       11111111 11111111 11111111 11111101
char cd = -5;      //                                  11111011
int ie = ia & cb; // cb -> 00000000 00000000 00000000 00000101
                  // ie  = 00000000 00000000 00000000 00000001
int ig = ic & cd; // cd -> 11111111 11111111 11111111 11111011
```

Для побитовых операций старайтесь использовать беззнаковые типы данных. Избегайте отрицательных значений до тех пор, пока вы точно не будете понимать, как они представляются в компьютере.

Операция исключающего или

```
0 ^ 0 = 0
0 ^ 1 = 1
1 ^ 0 = 1
1 ^ 1 = 0
```

Она обладает потрясающим свойством:

```
int a = 123, b = 555;
int c = a ^ b;
// c^a -> b, c^b -> a.
```

Операция ~(тильда)

Операция `тильда ~`, *побитовое не* для одиночных битов.

```
~0 = 1
~1 = 0
```

Однако не думайте, что если в программе вы напишете ~0, то получите единицу! А что?

Операции сравнения

- Их шесть: >, <, >=, <=, ==, !=
- Два операнда, всё преобразуется к старшему из типов. Результат — 1, если условие истинно или 0, если оно ложно.

> Не путайте операции сравнения (==) и присваивания (=).

```
3 < 5                    // 1
5 == 4                   // 0
```

Сравнение вещественных чисел

- Вещественные числа неточны.

```
double d = 1.0;
int r = (d / 3.0) * 3.0 == d; // r = 1? r = 0?
float f = 1000000000; // f != 10^9
float g = f + 1; //  g == f
```

Сравнение вещественных чисел на равенство принципиально неверно! Все вещественные числа имеют *относительную погрешность*, зависящую от типа. Значения типа `float` имеют погрешность `FLT_EPSILON`, значения типа `double` — `DBL_EPSILON`. Забегая вперёд скажем, что эти константы располагаются в заголовочном файле `math.h`.

Логические операции

Их немного. Это:

- логическое И (`&&`)
- логическое ИЛИ (`||`)
- логическое НЕ (`!`)

Не путайте побитовые операции `&`, `|` и логические `&&`, `||`.
`5 | 3 == 7`, а `5 || 3 == 1`
`3 & 4 == 0`, а `3 && 4 == 1`

Эти операции — *ленивые*. Вычисляется левый операнд, и если оказывается, что результат не зависит от правого операнда, то правый операнд не вычисляется.

Тернарная операция. Приоритеты операций

Она имеет ровно три операнда.

```
x = a > b ? a : b;
```

Приоритеты операций

1. `() [] -> .`
2. `! ~ ++ -- u+ u- u& u* (cast) sizeof`
3. `* / %`
4. `+ -`
5. `<< >>`
6. `< <= >= >`
7. `== !=`
8. `&`
9. `^`
10. `|`
11. `&&`
12. `||`
13. `?:`
14. `= += -= *= /= %= &= |= ^= <<= >>=`
15. `,`

Приоритеты операций

Для группировки операций в выражении в нужном порядке используйте круглые скобки. Используйте их также в случаях, когда имеются хоть малейшие сомнения в порядке исполнения операций в выражении. Операции присваивания исполняются в последнюю очередь.

Первые программы.

Первая завершённая программа

```
01  #include <stdio.h>
02
03  int main() {
04    printf("Hi again\n");
05    return 0;
06  }
```

01. Включить файл из *стандартной библиотеки*.
02. Разделение фрагментов.
03. Определение функции `main`, начало *блока*.
04. Вызов *библиотечной функции* `printf` для вывода текста.
05. Возврат в *вызывающую функцию* с кодом возврата 0.
06. Завершение блока, начатого в строке 03.

Поток управления

- Мы пишем программы, исполняющиеся в один поток.
- Существуют программы, исполняющиеся в несколько потоков — многопоточные, это не для нас пока.
- Исполнитель совершает действия по изменению значения переменных и изменению порядка вычислений, в зависимости от условий.
- Поток управления состоит из более мелких единиц, *операторов*.

Оператор декларации

- В современном C переменные можно *определять* или *декларировать* в произвольном месте программы.
- Удобнее всего *декларацию* совмещать с *инициализацией*.
- Помещать объявления переменных в начале программы это или плохой стиль, или требование использования старых компиляторов.

```
int a = 1,b = 2,c = 3;
c = a * 10 + 16;
double d = c * 1.5, e = 3.1415, f = d * e;
```

Блок

- *Блок* — группа операторов, заключённая в фигурные скобки.
- Блок может располагаться в том месте, где допустим единичный оператор и его чаще всего применяют именно для того, чтобы сгруппировать операторы в единое целое.
- Все переменные, описанные внутри блока, существуют только внутри блока.
- Не думайте, что создание и уничтожение переменных потребует серьёзного расхода времени — это не так.
- Модель памяти, применяемая в C позволяет это делать чрезвычайно быстро.

```
int x = 10;
{
    int y = x+5;
    ...
}
// Here x exists, y does not exist.
```

Операция присваивания и оператор присваивания

- Мы написали

  `a = 5`
- Точки с запятой нет? Тогда это *операция присваивания*.
- Каждая операция имеет значение — здесь это 5.
- Если поставим точку с запятой, *операция* превратится в *оператор* — законченную конструкцию языка.
- Если нет, то можно использовать дальше:

  `b = (a=5) * 3`

  Здесь результат операции равен 15 и он присваивается `b`.
- Поставим точку с запятой — превратим всё в оператор. И.т.д.

Операции присваивания

- Имеется много вариантов операции присваивания.
- Они часто удобнее обычных.
- Обычная операция:

```
abra_shvabra_cadabra = abra_shvabra_cadabra * 3;
```

- Мы говорит так: *умножим abra_shvabra_cadabra на три.*
- Запись

```
abra_shvabra_cadabra *= 3;
```

более точно отражает алгоритмическую сущность происходящего.

```
a += 4;
b = (c *= 2) + 7;
d <<= 1;
e &= 0xFF;
```

- Все операции присваивания «имеют значение», равное присвоенной величине.

Снова про *l-значения*

Сделаем следующую итерацию понятий:

> `l-значение` — нечто, существующее в том числе и вне выражения.
> `r-значение` — нечто, существующее только в выражении.

Нельзя написать:

```
3 += a; // Литералы существуют только в выражениях
a + 2 = 4; // Значение a+2 существует только в данном выражении
```

так как в левой части операций присваивания находятся не `l-значения`.

Операции инкремента и декремента

- Это — операции ++ и - -
- Каждая имеет два варианта — слева и справа от l-значения.
- Операции преинкремента и предекремента просты.
- То, что было по l-значению, увеличивается или уменьшается на 1.
- Значением операции является *полученное* значение.
- Пусть имеется целая переменная a и её значение равно 5; `a = a + 1` — операция присваивания, значение которой равно новому значению a, то есть 6.
  `a += 1` — другая запись той же операции.
  `++a` — третья запись той же операции. Результат является `l-value`.

Постинкремент и постдекремент

- Пост-операции несколько сложнее.
- Пусть a=5.
- После `c = a++;` значение переменной a станет 6.
- Но переменная c станет 5.
- Переменная a увеличивается на 1.
- Значением любой пост-операции является *старое* значение изменяемой переменной.

Значение пре-операций есть `l-значение`.
Значение пост-операций есть `r-значение`.

Не стоит использовать одну и ту же переменную в выражении несколько раз, если хотя бы одна из операций над переменной есть операция инкремента и декремента.

Операторы изменения порядка действий.

Оператор if

- Язык C — *императивный*.
- Мы должны явно указывать, *как* решать задачу, какие действия должны быть выполнены.
- Для этого есть такие операторы, как `if`, `while`, `do`, `for`, `switch`, `break`, `continue` и `return`.
- Присвоим переменной `max` наибольшее значение из `a` и `b`, используя *простой* или *безальтернативный* оператор `if`:

```
max = b;
if (a > b)
    max = a;
```

- Другой вариант — *альтернативный* `if`.

```
if (a > b)
    max = a;
else
    max = b;
```

Особенности оператора if

- Посмотрим внимательнее.

```
if (a > b)
    max = a;
else
    max = b;
```

- После ключевого слова `if` обязательно выражение в круглых скобках. Это не `Python` и не `Pascal`.
- Выражение вычисляется, и если оно не равно нулю (истинно), то исполняется оператор, следующий за ним.
- В этом случае часть `else`, если она есть, пропускается.
- Если выражение ложно, то при отсутствии части `else` ничего не происходит и оператор пропускается, иначе исполняется оператор, следующий за `else`.

Особенности оператора if

- Если требуется исполнить несколько операторов, их надо заключить в *блок* — фигурные скобки.

```
min = a;
max = b;
if (a > b) {
    min = b;
    max = a;
}
```

или

```
if (a > b) {
    min = b;
    max = a;
} else {
    min = a;
    max = b;
}
```

Стили программирования

- Мы используем *отступы* для того, чтобы показать тому, кто читает программу, что данный блок или единичный оператор исполняется только при соблюдении определённых условий.
- Сам язык Си не заставляет этого делать.
- Использование отступов — часть *дисциплины программирования*, *codestyle*.
- Каждая фирма придерживается каких-то правил в стилях кода.

Стили программирования

- Рекомендуется всегда использовать блоки, даже если этот код состоит всего из одного оператора.

```
if (a > b) {
    max = a;
} else {
    max = b;
}
```

- Это тоже дисциплина программирования и следование такому правилу помогает предотвратить много потенциальных ошибок, связанных с тем, что при добавлении операторов в условную часть можно забыть оформить блок, исказив смысл замысла.
- Найти такую ошибку в большой программе может оказаться сложным делом. В моей практике мне запомнился случай, когда мой коллега (очень опытный системный программист) искал такую ошибку две недели и нашёл её только с чужой помощью.

Небольшая задача с if

Задача. Даны три числа, $a < b < c$, образующих на числовой прямой 4 интервала и число x. Нужно присвоить переменной r номер интервала, в который попадает x. Считаем, что интервалы нумеруются с единицы, левый конец отрезка принадлежит интервалу, а правый — не принадлежит, то есть интервал *открыт справа*.

Решаем задачу в лоб

- Если x попал в один интервал, то он не попал в остальные.
- Следовательно, нужно использовать *альтернативный* оператор `if`.

```
if (x < a) {
    r = 1;
} else if (x >= a && x < b) {
    r = 2;
} else if (x >= b && x < c) {
    r = 3;
} else if (x >= c) {
    r = 4;
}
```

- Конструкция `else if` обычно применяется для определения попадания некоторого значения в непересекающиеся множества, поэтому отступы всех условий равны.

Улучшаем решение

- Подумаем, нужно ли условие $x >= a$ во второй ветке.
- Если $x < a$, то мы во вторую ветку не попадём.
- Иначе эта проверка — лишняя, так как $x >= a$ в альтернативной ветке, а альтернативная и основные ветка всегда взаимоисключающиеся.
- Сокращаем алгоритм:

```
if (x < a) {
    r = 1;
} else if (x < b) {
    r = 2;
} else if (x < c) {
    r = 3;
} else {
    r = 4;
}
```

Оператор `while`

- `while` — первый пример того, как можно создавать программы, выполняющие много действий с помощью небольшого количества строк.
- `while` не имеет `else` и повторяет действия, *тело цикла*, до тех пор, пока условие остаётся истинным.
- Как только условие становится ложным — цикл завершается.

Небольшая задача с while

Задача: Найти такое наибольшее число m, что $3^m \leqslant n$.

- Мы видим маленький алгоритм, где входом является n, а выходом — m.
- Идея решения: вычислять очередную степень тройки до тех пор, пока она не станет больше `m`, после чего вернуться на единицу назад.
- Возражение: каждый раз, вычисляя очередную степень тройки, мы забываем все предыдущие.
- Улучшение: степень тройки 3^x вычислять можно по индукции, имея вычисленное 3^{x-1}.

```
int pow3 = 1, x = 0, result = 0;
while (pow3 < n) {
    result = x;
    pow3 *= 3;
    x++;
}
```

- После окончания алгоритма переменная `result` будет содержать нужное значение. Переменная `pow3` — промежуточные данные и больше не нужна.

Ещё одна задача с while

Задача: Вычислить число e^x с точностью до 6-го знака после запятой для $0 \leqslant x \leqslant 10$ по формуле разложения функции e^x в ряд:

$$e^x = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots$$

Идея решения: будем рекуррентно вычислять очередной член ряда до тех пор, пока он не станет меньше 10^{-7} (это математически нестрого, но достаточно для решения данной задачи).

```
double sum = 1;
int n = 1;
double el = 1;
while ( (el *= x / n) > 1e-7) {
    sum += el;
    n++;
}
```