Введение в язык программирования C.

Лекция 3
Поток управления

Сергей Леонидович Бабичев

Поток управления

- Исполнитель алгоритма исполняет действия одно за другим, образуя *поток исполнения*.
- Пока мы пишем программы, исполняющиеся в один поток.
- Существуют программы, исполняющиеся в несколько потоков — многопоточные, это пока не для нас.
- 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 очень 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++;
}
```

Оператор do-while

- В операторе `while` условие цикла проверяется *до того*, как войти в блок.
- В цикле, начинающемся ключевым словом `do` проверка происходит *после* исполнения всего блока.
- Задачу по суммированию ряда можно было бы записать и через цикл `do`:

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

Оператор do-while

- Ключевое слово `while` всё ещё присутствует в операторе.
- Перенос проверки в конец меняет смысл алгоритма.
- В цикле `while` мы сначала вычисляли новое значение элемента ряда `el`, и если убеждались, что он больше границы, то прибавляли его к сумме.
- В цикле `do-while` мы прибавляем к сумме вычисленное значение `el` и это происходит независимо от того, больше или меньше границы оказывался вычисленный элемент.
- Конструкция `do-while` встречается намного реже «обычного» `while`.

Оператор `for` — самый мощный оператор цикла

Задача. Найти сумму кубов всех чисел от 1 до заданного n.

```
// Input: n
// Output: sum
int sum = 0;
for (int i = 0; i < n; i++) {
   sum += i*i*i;
}
```

- Необходимое условие: *две* точки с запятой, которые делят пространство внутри скобок на *три* выражения.
- Первое выражение исполняется однократно, как только начинается цикл.
- Если второе выражение истинно, итерация исполняется. Здесь может находиться любое выражение. Оно истинно, если отлично от нуля.
- Третье выражение исполняется после, исполнения *тела* цикла. После чего управления передаётся на *второе* выражение.

Цикл for

Перепишем задачу о степени тройки:

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

- Первое выражение — два присвоения начальных значений. Здесь применяется *операция запятая*. Её результат — последнее выражение.
- Второе выражение — проверка, продолжать ли попытки поиска.
- Третье выражение — опять операция запятая.

При использовании цикла `for` уменьшается вероятность допустить ошибку, если переменная, за значением которой мы следим, помещена в заголовок цикла. Таким образом мы явно показываем её продвижение к цели. Программу легче и писать, и читать.

Цикл for: детали

Все выражения могут быть опушены.

- Если опущено первое выражение — в инициализации не было нужды.
- Если опущено второе выражение — оно полагается истинным.
- Если опущено третье выражение — продвижение к цели где-то внутри тела цикла.

Цикл, эквивалентный while.

```
for ( ; pow3 < n; ) {
    /// ...
}
```

Бесконечный цикл.

```
for ( ; ; ) {
    /// ...
}
```

Оператор for

- Цикл в теле другого цикла называют *вложенным*.
- Уровней вложенности может быть много.

```
#include <stdio.h>

int main() {
    int count = 0, i1, i2, i3, i4, i5, i6;
    for (i1 = 0; i1 < 10; i1++)
        for (i2 = 0; i2 < 10; i2++)
            for (i3 = 0; i3 < 10; i3++)
                for (i4 = 0; i4 < 10; i4++)
                    for (i5 = 0; i5 < 10; i5++)
                        for (i6 = 0; i6 < 10; i6++)
                            count += i1+i2+i3 == i4+i5+i6;
    printf("Total %d lucky tickets\n", count);
}
```

Оператор break

Форма записи проста:

```
break;
```

- В боксе это слово говорит рефери говорит это слово, чтобы разнять боксёров.
- В Си его действие похоже на действие рефери: прекращается какое-то действие, например, прекращается очередная итерация любого цикла и управление передаётся на первый оператор, следующий за телом цикла.

```
for( ; ; ) {
   // ...
   if (x > 0) break;
   // Here x <= 0.
}
```

Оператор break

Ещё раз перепишем программу по вычислению экспоненты:

```
double sum = 1;
int n = 1;
double el = 1;
for (;;) {
    el *= x / n;
    if (el < 1e-7) break;
    sum += el;
    n++;
}
```

Оператор `break`: важное правило

Если имеется несколько циклов, вложенных друг в друга, то оператор `break` принудительно завершает только самый **внутренний** цикл.

Оператор continue

- Если `break` принудительно *завершает* цикл, то `continue` принудительно *продолжает* цикл.

Наивная задача. Вывести все числа от 1 до 100, которые делятся на 3, но не делятся на 5.

```
#include <stdio.h>

int main() {
    int x;
    for (x = 3; x <= 100; x += 3) {
        if (x % 5 == 0) continue;
        printf("%d ", x);
    }
}
```

Оператор switch

Задача. На вход алгоритма поступает число `10 <= n <= 30`. На выходе требуется получить истину, если число простое и ложь в противном случае.

- Первая идея — проверять число на делимость на простые множители.
- Вторая — воспользоваться ограниченностью множества решений.

Оператор switch: решение задачи

```
// input: n
// output: ans
int ans;
switch (n) {
case 11:
case 13:
case 17:
case 19:
case 23:
case 29:
    ans = 1;
    break;
default:
    ans = 0;
    break;
}
```

Оператор switch

Состоит из:

- *заголовка* (`switch (expr)`) в круглых скобках и *тела*.
- Выражение `expr` должно иметь перечислимое значение.
- Тело состоит из *случаев* `case`, с обязательным двоеточием. Ещё их называют *метки case*.
- Случаев может быть любое количество, в них должны фигурировать обязательно **константы**.
- Мы просматриваем случаи сверху вниз, и как только определилось, что значение `expr` совпадает с одной из констант, перечисленных в случаях, начинается исполняться последовательность операторов, следующая за данной меткой.
- Исполнение прекращается или когда мы доходим до самого конца оператора `switch`, или до оператора `break`, или исполняется оператор `return`.

Оператор switch: примеры

Что здесь происходит?

```
double p = 1.;
double q = 2.72;
switch (n) {
case 8: p *= q;
case 7: p *= q;
case 6: p *= q;
case 5: p *= q;
case 4: p *= q;
case 3: p *= q;
case 2: p *= q;
case 1: p *= q;
default:
   break;
}
```

Оператор switch: примеры

Вот ещё один пример:

```
switch (n) {
case 0:
   printf("Zero");
   break;
 case 1:
   printf("One");
   break;
 case 2:
   printf("Two");
   break;
 default:
   printf("What??? Unknown number!");
   break;
}
```

Функции

- Функция в Си — основной строительный блок.
- Функции расширяют возможности языка, добавлением новых *абстракций*.
- Всё взаимодействие любой программы с окружением происходит через функции.
- Используемая функция должна быть где-то *определена*.
- *Определение* функции задаёт её *имя*, *параметры* и *тип возвращаемого значения*.
- Обычно функция как-то обрабатывает свои *параметры* и *возвращает значение*.
- Одним из способов определения функции является помещение её *тела* в *исходном файле* программы где-нибудь *перед* её *использованием*.

Функции: пример определения

- В Си нет встроенной операции возведения числа в квадрат, которая есть в Pascal.
- Её очень легко дописать.

```
double sqr(double x) {
    double result = x * x;
    return result;
}
```

- x — параметр функции с объявленным типом double.
- В теле функции мы заводим любое количество *локальных* переменных, которые существуют только внутри функции.

Функции: пример использования

- Использовать её можно, например, присвоив какой-либо переменной результат её *вызова*:

```
double t2 = sqr(t);
```

- Чтобы вызов был успешным, компилятору надо знать как тип возвращаемого значения функции, так и типы всех её аргументов.
- Если функция *определена* выше *точки вызова*, то у компилятора достаточно информации для того, чтобы проверить правильность её вызова.
- Функция *определяется* где-то точки вызова, то компилятор может сделать собственные предположения о том, что это за функция.
- Эти предположения могут не совпасть с тем, что он увидит, как только доберётся до её определения.
- Можно помочь компилятору, поместив *объявление* или *прототип* функции перед точкой её вызова. Для нашей функции `sqr` объявление будет выглядеть так:

```
double sqr(double t);
```

Функция: полный пример использования

```
#include <stdio.h>

double sqr(double); // prototype or declaration

int main() {
    double a = 1234.567;
    double a2 = sqr(a); // usage
    printf("a^2=%lf\n", a2);
}

double sqr(double x) { // definition
    double result = x * x;
    return result;
}
```

Функции

- Функция может иметь любое число параметров.
- Они перечисляются через запятую, каждый со своим типом.
- Перед именем функции — тип её возвращаемого значения.
  Правильно:

```
int max3(int a, int b, int c);
```

  Неправильно:

```
int max3(int a,b,c);
```

Следующая функция принимает координаты двух точек на плоскости и возвращает расстояние между ними. Она использует *библиотечную* функцию `sqrt`.

```
double distance(double x1, double y1, double x2, double y2) {
    return sqrt((x2-x1)*(x2-x1)+(y2-y1)*(y2-y1));
}
```

Функции

- При объявлении функций допустимо опустить имена переменных, оставив только их типы:

  ```
  double sqr(double);
  ```

- Имена переменных лучше всё же писать, выбирая что-то осмысленное.
- Какое объявление функции `distance` более понятно?

  ```
  double distance(double x1, double y1, double x2, double y2);
  double distance(double, double, double, double);
  ```

Передача аргументов в функцию

- То, что приходит в параметры функций, считаются переменными, которым присвоены начальные значения. Они приходят из *точки вызова*.
- В месте вызова функции *аргументы* вызова должны соответствовать *параметрам*.

```
int max3(int a, int b, int c) {
  // ...
}


  ...
  int a = 33;
  int b = 77;
  int m = max3(10, a+5, b);
```

- Здесь значением параметра a в функции max3 будет 10, параметра b будет 38 (сумма значения переменной a в точке вызова и 5), а значением параметра c — 77.

Параметры функции как переменные

- Функции могут распоряжаться аргументами, полученными из точки вызова, как им заблагорассудится.
- Они могут их изменять, при этом то, что передано в функцию в точке вызова, не изменится.

```
void foo(int x) {
  x = 10;
}


...
int a = 33;
foo(a);
// Here a still equal 33.
```

Если мы хотим, чтобы функция изменила значение переменной, которую мы в неё передали, нужно воспользоваться *указателями*.

Функции и ключевое слово *static*

- Большие проекты состоят из сотен и тысяч файлов.
- Если определить функцию в одном файле, во всех местах, где она используется, требуется её описание.
- Определять функции с одним именем и разными именами в Си нельзя.
- Некоторые функции в проекте могут использоваться исключительно в одном исходном файле.
- Если заранее известно, что именно этой функцией в других файлах пользоваться не будут, добавляют слово *static*.
- Тогда имя этой функции не будет конфликтовать с другими такими же именами в проекте.

Несколько слов о проектах

- Большие проекты пишутся несколькими людьми или группами.
- Каждая группа работает над своим набором файлов.
- Мы разделяем функции на те, которыми мы будем пользоваться только в нашей части проекта, в нашем файле (*внутренние*), и на те, которые мы специально пишем для того, чтобы другие смогли ими воспользоваться (*внешние* или *интерфейсные*).
- Большое количество внешних функций в программе может сделать невозможным стыковку двух частей большого проекта, созданных в разных файлах разными людьми из-за *коллизий* имён.
- Ключевое слово `static`, перед именем функции уменьшает эту проблему.

```
static int func(int n) {
    ...
}
```

Теперь имя `func` можно будет использовать в других файлах проекта.

Рекомендации к использованию слова `static`

Всегда используйте аттрибут `static` для тех функций, которые не планируется делать общими в проекте.

Это поможет вам, так как компилятор сможет оптимизировать использование этой функции, зная, что нигде, кроме данного файла она не доступна.

Это поможет другим, так как вы не *загрязняете пространство глобальных имён*.

Функции: несколько советов

- Используйте хорошо читаемые и осмысленные имена функций.
- Используйте хорошо читаемые и осмысленные имена параметров.
- Не пишите очень больших функций: постарайтесь, чтобы она поместилась на экране.
- Постарайтесь не использовать более 5-6 параметров в функции.
- Используйте `static` для тех функций, которые не должны быть видимы в других файлах.