GeekBrains

Основы C++

Условия и циклы.

На этом уроке

1. Подробно рассмотрим понятия условного и безусловного перехода
2. Научимся писать условия и поймём принципы ветвления программы
3. Изучим особенности оператора множественного выбора
4. На практике столкнёмся с понятием области видимости

Оглавление

Операторы языка

В языке Си можно выделить три основных типа операторов:

- элементарные операторы;
- операторы ветвления;
- операторы цикла.

К каждому типу относятся несколько операторов.

Элементарные операторы

Представляют собой любое выражение, за которым следует `;` (точка с запятой). В качестве такого выражения обычно выступает строка из операций и операндов. Но эта строка может быть и пустой (состоять из одной только `;`). Такой оператор называется пустым оператором. Оператор может быть и составным, т.е. состоящим из нескольких взаимосвязанных операций. Составной оператор также называют блоковым оператором, или просто блоком. При этом транслятор рассматривает весь блок как один оператор.

Вызов функции также рассматривается в C++ как оператор, но это особый оператор передачи управления. После завершения этого оператора происходит возврат управления в то выражение, из которого был вызов функции, на следующую за вызовом функции операцию. Как мы узнаем на следующем занятии, функция имеет свой тип и может возвращать значение. Поэтому функция может рассматриваться в выражении и в качестве операнда. Поэтому вызов функции - это особый оператор - операнд.

Операторы ветвления

Известные также под именами "операторы перехода", "операторы передачи управления", предназначены для изменения естественного в языке C++, построчного (т.е. строка за строкой), порядка выполнения программы. В языке C++ определены два типа операторов ветвления: **условный** и **безусловный**.

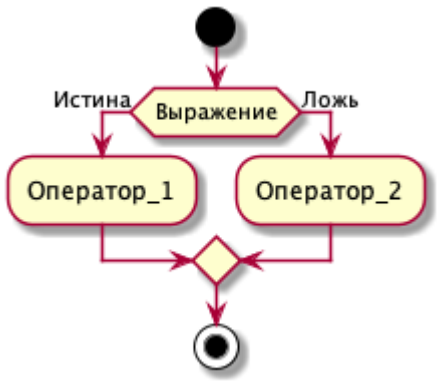
Условные переходы позволяют выбрать одну из двух (простое ветвление) или из многих (множественное ветвление) последовательностей выражений в программе для исполнения, в зависимости от указанных в операторе условий. Оператор простого ветвления `if`/`else`, оператор множественного ветвления (множественного выбора) `switch`/`case`.

Операторами безусловного перехода в языке Си являются: `goto`, `break`, `continue`. Первый из этого списка оператор параметризованный (т.е. в качестве аргумента этого оператора указывается метка адреса (ярлык, лейбл), на которую осуществляется передача управления), два последних оператора из списка не предоставляют в этом плане никакого выбора. Сразу *закроем тему* оператора `goto`. Оператор `goto` и соответствующий лейбл должны находиться в одной и той же функции. Существуют некоторые ограничения на использование операторов `goto`. Например, вы не сможете перепрыгнуть

вперед через переменную, которая инициализирована в том же блоке, что и `goto`. В целом, программисты избегают использования оператора `goto` в языке C++ (и в большинстве других высокоуровневых языков программирования). Основная **проблема** с ним заключается в том, что он позволяет программисту управлять выполнением кода так, что точка выполнения может прыгать по коду произвольно. А это, в свою очередь, создает то, что опытные программисты называют «спагетти-кодом». *Спагетти-код* — это код, порядок выполнения которого напоминает тарелку со спагетти (всё запутано и закручено), что крайне затрудняет следование порядку и понимание логики выполнения такого кода. Как говорил один известный специалист в информатике и программировании, Эдсгер Дейкстра: «Качество программистов — это уменьшающаяся функция плотности использования операторов goto в программах, которые они пишут».

Оператор `if`/`else`

Имеет следующий алгоритм исполнения:



и может иметь один из следующих форматов:

```
if (выражение)
{ // Этот блок операторов выполняется, если "выражение" истинно (T)
    оператор_1;
    ....
    оператор_n;
}
```

или:

```
if (выражение)
{ // Этот блок операторов выполняется, если "выражение" истинно (T)
    оператор_1;
    ....
    оператор_n;
}
else  // Соответствует пунктирной ветке "F" на алгоритме этого оператора
{ // Этот блок операторов выполняется, если "выражение" ложно (F)
    оператор_n + 1;
    оператор_n + 2;
    ....
```

```
    оператор_n + m;
}
```

Можно также использовать и такую (сокращенную) форму записи оператора if/else:

```
if (x == 3)
    y = 5;
else
    y = 8;
```

В последнем примере и выражением, и единственным оператором в случае как истинности, так и ложности выражения, являются простейшие операции сравнения и присваивания, поэтому фигурные скобки `{}` в оформлении блоков операторов можно опустить.

Операторы `if`/`else` допускают вложение одного оператора в другой. Например:

```
if (x == 3)
    if (y == 5)
        оператор_1;
    else
        оператор_2;
else
    оператор_3;
```

В этом примере ветка `else оператор_2;` относится к невыполнению условия `if (y == 5)`, а ветка `else оператор_3;` относится к невыполнению условия `if (x == 3)`. Если же для условия `if (y == 5)` нет ветки `else`, этот пример необходимо записать так:

```
if (x == 3)
    if (y == 5)
        оператор_1;
else
    оператор_2;
```

Однако читабельность (удобство чтения) большого количества вложенных операторов `if`/`else` зачастую оставляет желать лучшего, поэтому альтернативой глубокому вложению `if`/`else` является применение оператора множественного ветвления switch/case.

Оператор `switch`/`case`

Известен также под именем “оператор множественного выбора”. Алгоритм исполнения:

Для пояснения синтаксиса, ключевых слов и вложенных операторов, сразу приведем пример использования оператора `switch`/`case`:

```
#include <iostream>
main()
{
    char op;
    cout << "Введите символ арифметической бинарной операции: ";
    cin >> op;
    switch(op) {
        case('+'):
            cout << "Это символ операции сложения." << endl;
            break;
        case('-'):
            cout << "Это символ операции вычитания." << endl;
            break;
        case('*'):
            cout << "Это символ операции умножения." << endl;
            break;
        case('/'):
            cout << "Это символ операции деления." << endl;
            break;
        case('%'):
            cout << "Это символ взятия остатка от деления." << endl;
            break;
        default:
            cout << "Это не символ бинарной арифметической операции." << endl;
    }
```

```
}
```

Выражение `op` вычисляется, при необходимости, но в нашем примере это символьная переменная, полученная от функции - оператора `cin >> op;` и последовательно сравнивается с константами, в нашем примере это символьные константы `'+'`, `'-'`, `'*'`... При совпадении с какой-либо из констант выполняются все операторы, следующие за ключевым словом `case(константа):` соответствующим этой константе, пока не встретится оператор `break;` (оператор безусловного завершения блока кода) или операция `}` завершения оператора `switch`/`case`. Если ни в одном из `case(...):` константа не совпадает с аргументом `(op)` оператора `switch(op)`, выполняется ветка "прочее", т.е. блок операторов, следующих за ключевым словом `default`. Но ветки `default:` может и не быть, на усмотрение программиста.

Константы - аргументы `case(константа):` могут представлять собой только константные выражения, т.е. выражения, не содержащие переменных.

Ветка `default:` может присутствовать в `switch`/`case` только один раз.

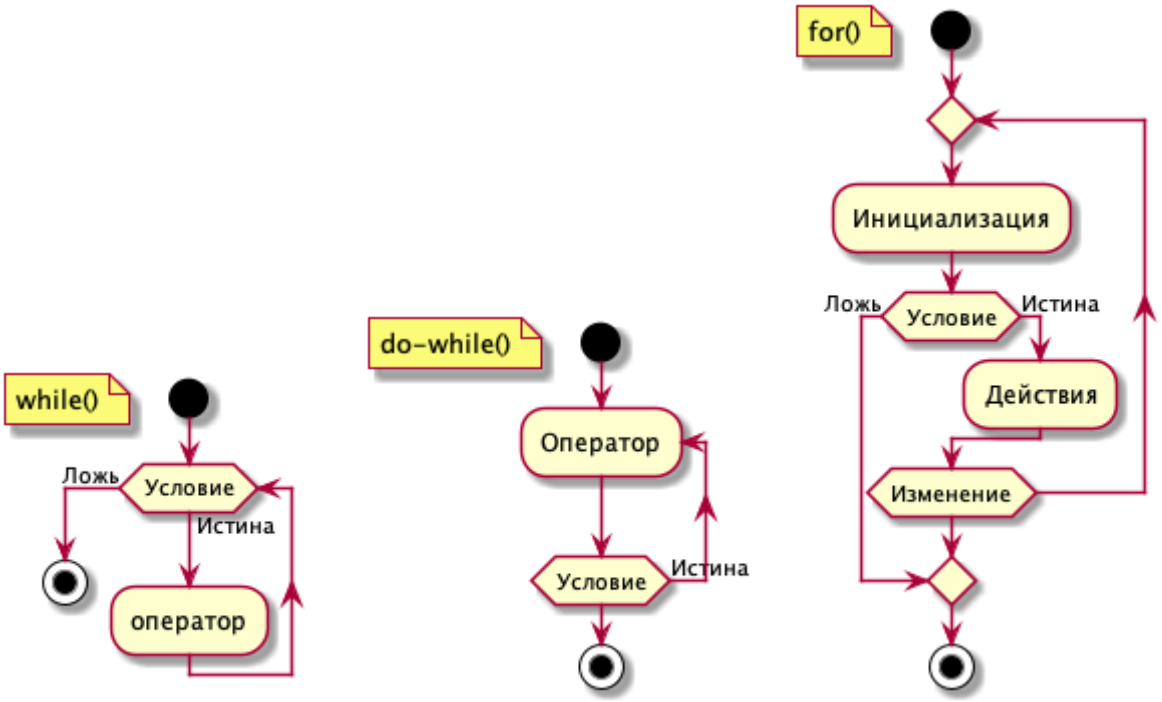
Операторы цикла

В языке Си определены два типа циклов: параметрический и итерационные.

Для параметрических циклов параметром является число повторений, которое известно, или может быть вычислено заранее. Параметрическим в Си является цикл `for`.

Для итерационных циклов характерно то, что цикл выполняется, или не выполняется, исходя из некоторого условия, которое проверяется внутри самого цикла. Условие завершения цикла может проверяться как в самом начале цикла (циклы `for` и `while`), так и в конце цикла (цикл `do`/`while`). В первом случае итерационный цикл может вообще не выполняться ни разу, во втором случае (цикл `do`/`while`) обязательно исполняется хотя бы один раз. Теоретически, все циклы взаимозаменяемы, вопрос о применении того или иного цикла зависит, по мнению авторов, только от опыта и личных симпатий программиста.

Вот диаграммы алгоритмов этих трёх циклов:



Оператор `while()`

Проверяет истинность "условия" каждый раз перед выполнением тела цикла. Под "условием" понимается понимается любая операция сравнения, любой оператор, или даже блок операторов. Если при первом же вычислении "условия" результатом оказывается ложь, тело цикла ("оператор") не выполняется ни разу. Вот пример цикла `while`:

```
#include <iostream>
main() {
  int num, sum = 0;
  cout << "Введите число от 1 до 10. Для завершения программы введите 0. > ";
  cin >> num;
  while (num != 0) // Это заголовок цикла. Можно было написать: while(!num)
  { // В фигурных скобках {} тело цикла
    sum += num;
    cout << "Введите число от 1 до 10. Для завершения программы введите 0. > ";
    cin >> num;
  }
  cout << "Сумма введенных чисел равна %d " << sum << endl;
}
```

Конечно, это простейший пример, без проверок на переполнение суммы и достоверность введенных данных, это всего лишь иллюстрация цикла `while`.

Оператор do/while()

Аналогичен оператору while(), но проверка условия завершения выполняется в конце цикла. Вот пример вышеприведенной программы, но с использованием оператора do/while и использованием языка С. Потому что мы можем:

```
#include <stdio.h>
main() {
  int num, sum = 0;
  do { // В фигурных скобках {} тело цикла
    printf("Введите число от 1 до 10. Для завершения программы введите 0. > ");
    scanf("%d", &num);
    sum += num;
  } while(num != 0); // Проверка условия в конце, однако это заголовок цикла
  printf("Сумма введенных чисел равна %d ", sum);
}
```

Как видим, применение цикла do/while для нашего примера оказалось немного выгоднее, чем применение цикла while. Отсутствуют двойное кодирование подсказки и функции - оператора буферизованного ввода числа scanf и его “младшего брата” cin. Не повторять одни и те же операторы при реализации одного и того же алгоритма - пример хорошего стиля кодирования.

> *Don't repeat yourself.*
> *Акроним dry (англ. сухой, чистый). Не повторяйся. Любой код, который программист вынужден повторять можно убрать в цикл, метод, класс, подпрограмму, модуль, сервис. Если в Вашей программе есть повторяющийся (скопипастченный) код - это не очень хорошая программа, потому что если и когда Вам нужно будет её исправлять - Вы потратите значительное количество времени только на поиск всех мест, содержащих нужный код. Программы и компьютеры придумали как раз для того, чтобы не нужно было самому выполнять повторяющиеся действия. Поэтому, не повторяйте себя. Антагонист этого принципа - WET. Write everything twice. Акроним wet (англ. влажный). Пиши всё дважды. Экономьте собственное время, не пишите по два раза.*

Оператор цикла for

В начале этой темы мы уже указали разницу между параметрическими и итерационными циклами. Оператор for применим как для организации итерационного цикла, так и для параметрических циклов, для которых неприменимы операторы while и do/while (точнее, можно, конечно, приспособить и их, но путем никому не нужных на практике усложнений).

Итак, на схеме алгоритма цикла for:

- инициализация - есть указание начальных условий цикла, выполняется это выражение только один раз, перед началом выполнения цикла;
- условие является условием выполнения цикла, вычисляется и проверяется перед каждой итерацией выполнения цикла. Если результат вычисления условия не равен лжи, тело цикла выполняется, в противном случае цикл завершается, и управление передается на следующий за циклом оператор;

- `изменение` - это модификатор начальных условий цикла, выполняется это выражение после каждой итерации выполнения цикла. Таким образом, модификации могут подвергаться, по желанию программиста, как исходные начальные условия, установленные условием, так и (что происходит гораздо чаще) те же начальные условия, но уже модифицированные в предыдущих итерациях цикла.

В качестве инициализации, условия и изменений допустимы любые корректные выражения языка C++, в том числе и пустые операции `(;)`. Но если опущено условие, то считается, что условие выполнения очередной итерации такого цикла всегда истина. Иными словами, в языке C++ допустим такой цикл:

```
for(;;); // Это пример определения бесконечного цикла
```

На первый взгляд, бессмысленная конструкция, программа подвисает "навечно". Но, на самом деле, из такого цикла есть выходы - сброс аппаратной платформы по питанию и внешние прерывания. Поэтому такой цикл очень часто применяется в так называемых событийно управляемых программах, известных также как "программы реального времени". Бесконечный цикл можно организовать и на операторах `while` и `do`/`while`. Применяются разные описания бесконечных циклов равнозначно по желанию программиста.

Вот пример той же программы, что мы давали при разговоре про циклы `while` и `do`/`while`, но с использованием оператора for:

```
#include <stdio.h>
main() {
  int num, sum;
  printf("Введите число от 1 до 10. Для завершения программы введите 0. > ");
  scanf("%d", &num);
  for (sum = 0; num != 0; sum += num) { // В фигурных скобках {} тело цикла
    printf("Введите число от 1 до 10. Для завершения программы введите 0. > ");
    scanf("%d", &num);
  }
  printf("Сумма введенных чисел равна %d ", sum);
}
```

Обратите внимание, где в этом примере была произведена инициализация переменной `sum = 0`. В некоторых случаях, если область видимости переменной не выходит за пределы оператора `for`, допускается и объявлять такую переменную внутри оператора `for`. Работает с версиями языка C99 и выше. Например, если бы в нашем случае не нужно было выводить в терминал результат последней строкой программы, заголовок цикла мог - бы выглядеть так:

```
for (int sum = 0; num != 0; sum += num)
```

Области видимости переменных.

К понятию области видимости переменных мы еще вернемся чуть позже, когда будем рассматривать функции языка Си, сейчас дадим только частное пояснение для вышеприведенного оператора `for`.

Переменные (в нашем случае `int sum` и `int num`) мы изначально объявили за пределами заголовка и тела цикла `for`, в теле функции `main`. В таком варианте они будут доступны любому оператору внутри функции `main`. Значит, тело функции `main` (от скобки `{` до скобки `}`) и составляет их область видимости. Иногда это не очень удобно. Если в пределах функции `main` будет много операторов и вызовов функций, то для каждого из них придется объявлять свои переменные, каждую из которых обязательно с уникальным именем. Следовательно, длина имен вырастет, к тому- же вырастет вероятность ошибки типа обращения оператора к “чужой” переменной.

Чтобы избежать подобных неприятностей, в языке Си широко применяются принципы “сужения области видимости” и “наложения имен”. Объявляя переменную `int sum` внутри заголовка цикла `for` мы, тем самым, сужаем ее область видимости до размеров заголовка и тела цикла. Следовательно, в других циклах `for`, и внутри других операторов в пределах нашей функции могут быть объявлены переменные с теми же именами `sum` и `num`, и это будут совершенно независимые переменные с одинаковыми именами (принцип “наложения имен”). Но и доступны каждая из таких локальных переменных будет только в пределах ее собственной, суженной до размеров заголовка и тела оператора, области видимости.

Операторы break и continue

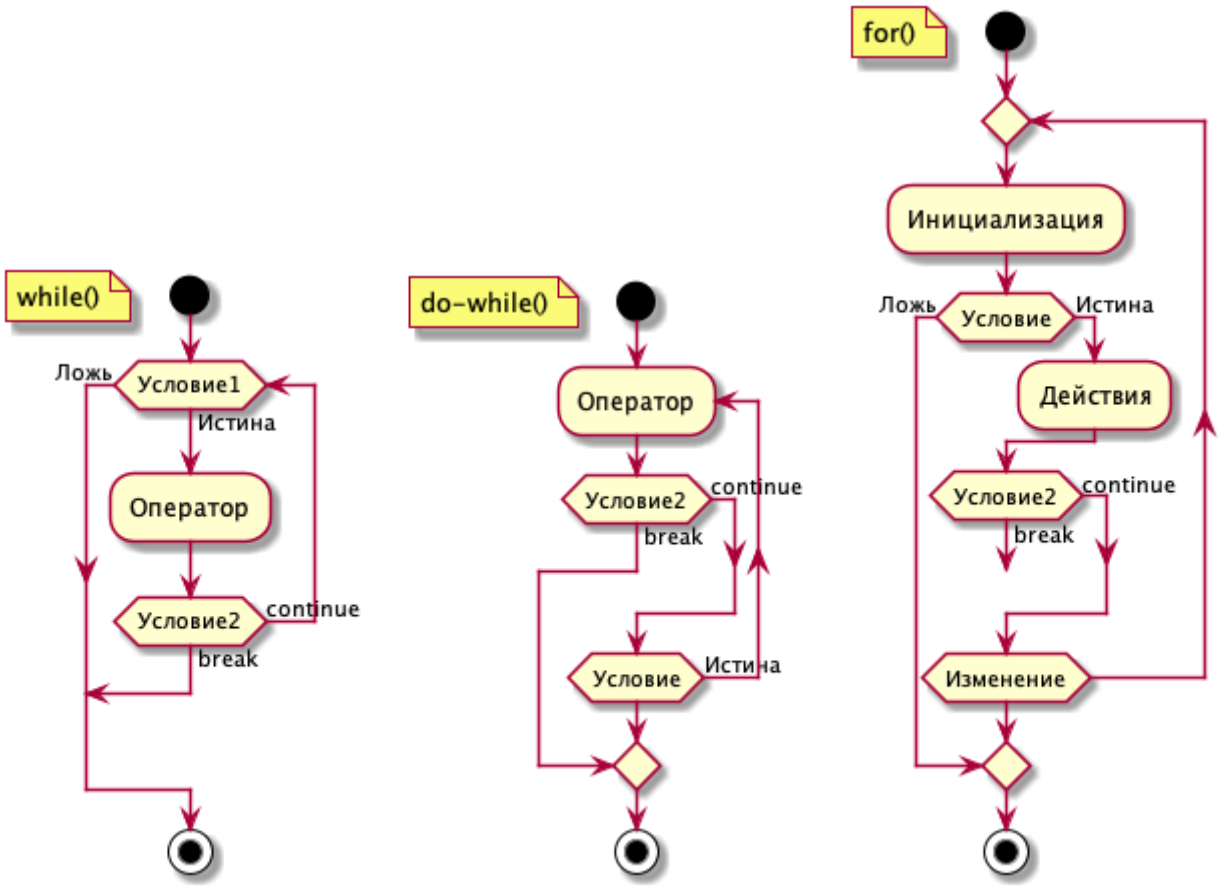
Используются, если необходимо изменить последовательное, оператор за оператором, исполнение тела цикла. Оператор `break` принудительно завершает выполнение цикла (или, как мы видели, оператора множественного выбора). Например, мы уже отмечали, что приведенные выше примеры использования операторов `for`, `while` и `do`/`while` не защищены от ошибочного ввода данных с клавиатуры, но возможно модифицировать эти примеры:

```
int num, sum = 0, err_code = 0;
for(sum = 0; num != 0; sum += num)    {
  printf("Введите число от 1 до 10. Для завершения программы введите 0. > ");
  scanf("%d %d ", &num, &err_code);
  if(err_code != 0) { // Если err_code не 0, цикл принудительно завершается
    printf("Ошибка ввода символа. Аварийное завершение программы.");
    break;
  }
  printf("Сумма введенных чисел равна %d ", sum);
}
```

Оператор `continue` прерывает выполнение тела цикла и передает управление на проверку условия выполнения цикла. Вот тот- же пример, с использованием оператора continue:

```
int num, sum = 0, err_code = 0;
for(sum = 0; num != 0; sum += num)    {
  printf("Введите число от 1 до 10. Для завершения программы введите 0. > ");
  scanf("%d %d ", &num, &err_code);
  if(err_code != 0) {  // Если err_code не равен 0, цикл возобновляется с
сообщением об ошибке
    printf("Ошибка ввода символа.\n");
    continue;
  }
  printf("Сумма введенных чисел равна %d ", sum);
}
```

Алгоритмы работы операторов break и continue представлены на изображениях ниже. Поскольку эти операторы изменяют последовательный характер выполнения программы, их по полному праву можно рассматривать как операторы ветвления.



Практическое задание

1. Написать программу, проверяющую что сумма двух чисел лежит в пределах от 10 до 20 (включительно), если да – вывести true, в противном случае – false;
2. Написать программу, проверяющую, является ли некоторое число - натуральным простым. Простое число - это число, которое делится без остатка только на единицу и себя само.

3. Написать программу, выводящую на экран “истину”, если две целочисленные константы, объявленные в её начале либо равны десяти сами по себе, либо их сумма равна десяти.
4. * Написать программу, которая заполняет диагональные элементы квадратной матрицы единицами. Размеры матрицы нужно задать константными числами, матрицу нужно инициализировать нулями.
5. * Написать программу, которая определяет является ли год високосным. Каждый 4-й год является високосным, кроме каждого 100-го, при этом каждый 400-й – високосный. Для проверки работы вывести результаты работы программы в консоль

Используемые источники

1. *Брайан Керниган, Деннис Ритчи.* Язык программирования C. — Москва: Вильямс, 2015. — 304 с. — ISBN 978-5-8459-1975-5.
2. *Stroustrup, Bjarne* The C++ Programming Language (Fourth Edition)