Цикл перечисления в C++

1 Синтаксис, назначение

При разработке алгоритмов часто возникает необходимость выполнить некоторое действие заданное количество раз. Кроме того, зачастую требуется не просто повторить какое-то действие, а, например, перебрать все элементы в каком-то множестве (представленном, как правило, каким-либо контейнером - о них поговорим позже). Необходимость эта встречается настолько часто, что циклы, используемые для таких целей, стали выделять в отдельный класс: **циклы перечисления**.

Строго говоря, всё, что реализуется с помощью циклов перечисления, можно реализовать и условным циклом; однако использование цикла перечисления более ясно показывает намерение программиста: **выполнить действие заданное количество раз/для каждого элемента**.

Синтаксически цикл реализуется с помощью ключевого слова **for**:

for ([<инициализация>];[<условие>];[<шаг цикла>]) <выражение>

Заметьте, что слова *инициализация, шаг цикла, выражение* взяты в квадратные скобки. Это означает, что при написании кода эти части можно опустить; самый простой цикл перечисления будет выглядеть так:

```
for (;;) {/* ...дальнейший код... */}
```

Такой цикл будет выполняться бесконечно (или пока его не прервёт какой-то код внутри тела цикла, но об этом позже). Тем не менее, чаще встречается иное написание:

```
for(int i = 0; i < 10; ++i) { /* тело цикла */ }
```

Здесь *инициализация* - это код **int** i = 0, *условие* - i < 10, *шаг цикла* - ++i. Рассмотрим подробнее каждое из этих понятий и укажем, что делает тот код, который представлен в качестве примера:

Инициализация - код, который будет выполнен **перед началом цикла** (перед запуском тела цикла первый раз). Как правило, здесь объявляют переменную, которую будут называть **переменной цикла**. В отличие от некоторых других языков (например, **Pascal** или **Basic**), в **C++** переменная цикла вполне может и не быть целочисленной; больше того, вполне допускается объявить сразу несколько переменных. Стоит учитывать, что переменная, объявленная в секции инициализации, **существует только внутри цикла**; так, следующий код вызовет ошибку компиляции:

```
1  #include <iostream>
2
3  int main()
4  {
5          int s = 0;
6          for(int i = 0; i < 15; ++i) {s = s + i;}
7          std::cout << s; /* Всё в порядке: s объявлена вне цикла */
8          std::cout << i; /* Ошибка: переменная i не объявлена в данной области
               видимости */
9          return 0;
10 }
```

Условие - код, который будет выполняться *перед* каждым запуском цикла. Данный код должен быть некоторым логическим или арифметическим выражением; если значение данного выражения равно нулю, то запуск цикла не происходит и управление переходит на код, написанный после него. Иначе тело цикла запускается; строго говоря, в языке **C/C++ цикл перечисления по сути является циклом с предусловием** с более богатым синтаксисом.

Шаг цикла - код, который выполняется **после каждого выполнения тела цикла** (то есть **перед первым выполнением** этот код **не** выполняется). Разумно в этом месте увеличивать переменную цикла; код ++i означает «префиксный инкремент» - это увеличивает значение переменной на 1. (С тем же успехом можно было бы писать i++ - в данном месте это *почти* то же самое). Впрочем, ничего не мешает написать, например, i = i + 2 или даже что-то более сложное; например, при использовании связных списков (структур данных, в которых каждый элемент содержит указатель на следующий) цикл перечисления выглядел бы примерно так:

```
for(node *cur = head; cur != nullptr; cur = cur->next) {/* ...дальнейший код... */}
```

(более подробно такой код будет рассмотрен при рассмотрении связных списков)

Стоит отметить, что цикл перечисления не налагает никаких ограничений на изменение переменной цикла внутри этого самого цикла; однако **так лучше не делать**, потому что это сильно затрудняет понимание кода. Если разрабатываемый вами алгоритм требует изменения условной переменной в нескольких местах цикла, то, возможно, лучше присмотреться к условному циклу?

К слову, в **C++11** появился альтернативный синтаксис для цикла перечисления; новый синтаксис куда больше раскрывает суть цикла как цикла **перечисления** (а не просто удобного условного цикла) и используется для обхода коллекций. Рассматривать его сейчас мы, конечно же, не будем - отложим это до появления массивов.

2 Элементарные алгоритмы

(альтернативное название: «Это должен знать каждый школьник»)

2.1 Сумма и среднее арифметическое последовательности

Здесь и далее будем предполагать, что длина последовательности нам задаётся с клавиатуры перед самой последовательностью.

Для вычисления суммы последовательности следует в цикле считывать очередной элемент последовательности и прибавлять его к уже имеющейся сумме. Для вычисления среднего арифметического следует эту сумму разделить на количество элементов, при этом не забывая привести хотя бы что-то из этого к вещественному типу:

```
1  #include <iostream>
2
3  int main()
4  {
5          int sum = 0; /* Накопленная сумма */
6          int n;       /* Длина последовательности */
7          std::cin >> n;
8          for(int i = 0; i < n; ++i)
9          {
10                 int a;
11                 std::cin >> a; /* Считываем элемент... */
12                 sum = sum + a; /* ...и добавляем его к сумме */
13         }
14         std::cout << "sum: " << sum << std::endl;
15     /* При вычислении среднего обязательно делаем хотя бы делимое вещественного
          типа */
16         std::cout << "mean: " << (double)sum / (double)n << std::endl;
17         return 0;
18 }
```

2.2 Минимум и максимум

Если надо найти наименьший/наибольший элемент последовательности (без дополнительных условий, вроде «наибольший чётный» или «наименьший положительный»), то просто принимаем первый элемент последовательности за минимум/максимум и каждый следующий сравниваем с ним:

```
1  #include <iostream>
2
3  int main()
4  {
5          int n;       /* Длина последовательности */
6          std::cin >> n;
7      int min, max;
```

```
8       int t;          /* Переменная, в которую мы считаем первый элемент
            последовательности */
9       std::cin >> t;
10          min = max = t; /* Предполагаем, что первый элемент - и минимум, и максимум
                */
11      /* Мы уже считали первый элемент, поэтому отсчитываем с единицы */
12          for(int i = 1; i < n; ++i)
13          {
14                  int a;
15                  std::cin >> a;
16          /* Сравниваем каждый новый элемент с текущим минимумом/максимумом */
17          if (a > max) max = a;
18          if (a < min) min = a;
19          }
20          std::cout << "min: " << min << std::endl;
21          std::cout << "max: " << max << std::endl;
22          return 0;
23  }
```

Если же у нас есть какие-то условия, которым должен соответствовать минимум или максимум, то можно воспользоваться так называемым **значением-меткой** (sentinel value) - таким значением, которое **не** соответствует условию, и показывает, что данный минимум/максимум не существует (например, нет чисел, удовлетворяющих какому-то условию, по которому ищется экстремальное значение). Так, к примеру, если бы нам надо было найти наибольшее чётное число, мы бы могли написать так:

```
1   #include <iostream>
2
3   int main()
4   {
5           int n;          /* Длина последовательности */
6           std::cin >> n;
7       /* Значение-метка: -1 является нечётным числом, если максимум после работы
            цикла остался равен этому значению, значит, чётных чисел не было */
8           int max = -1;
9           for(int i = 0; i < n; ++i)
10          {
11                  int a;
12                  std::cin >> a;
13          /* Условие: если число a является чётным и:
14             - либо a больше текущего максимума
15             - либо максимум равен значению-метке */
16          if ((a % 2 == 0) && ((a > max) || (max == -1))) max = a;
17          }
18      /* Проверяем, нашли ли мы хотя бы одно чётное число */
19      if (max != -1)
20      {
21                  std::cout << "max_even: " << max << std::endl;
22          }
23          else
24          {
25                  std::cout << "no_even_numbers" << std::endl;
26          }
27          return 0;
28  }
```

Наконец, если мы по каким-то причинам не можем использовать значения-метки, но у нас есть **диапазон** входных значений (то есть сказано, что все элементы, например, по модулю не превосходят 10000), то мы можем установить начальные значения минимума и максимума за пределами этого диапазона. **Важно**:

максимум должен быть **меньше меньшего**, минимум - **больше большего** (поскольку значение минимума может только уменьшаться, а максимума - только увеличиваться).

Пример нахождения нечётного минимума, если все числа не превосходят 5000:

```
1  #include <iostream>
2
3  int main()
4  {
5          int n;          /* Длина последовательности */
6          std::cin >> n;
7      /* Минимум — больше большего: наибольшее значение 5000, поэтому здесь поставим
           5001 */
8          int min = 5001;
9          for(int i = 0; i < n; ++i)
10         {
11                 int a;
12                 std::cin >> a;
13         /* Заметим, что min меняется тогда и только тогда, когда очередное число
               меньше, чем min, то есть min может только уменьшаться */
14         if ((a % 2 != 0) && (a < min)) min = a;
15         }
16     /* Строго говоря, 5001 можно трактовать как значение-метку — или просто
           проверить, не выходит ли наш минимум за допустимый диапазон */
17     if (min <= 5000)
18     {
19                 std::cout << "min odd: " << min << std::endl;
20         }
21         else
22         {
23                 std::cout << "no odd numbers" << std::endl;
24         }
25         return 0;
26 }
```

2.3 Порядковый номер минимума/максимума

Идея абсолютно такая же, как и в случае с поиском минимума/максимума... правда, значения, равные минимуму или максимуму могут встречаться несколько раз. Поэтому в задании должно быть уточнение: надо ли найти положение первого или последнего минимума/максимума. В первом случае мы проверяем, меньше ли/больше ли текущий элемент, чем наш минимум/максимум, и если условие выполняется - записываем порядковый номер считанного элемента. Во втором - строгую проверку (меньше/больше) надо заменить на **нестрогую** (не больше/не меньше).

Продемонстрируем это на примере: пусть надо определить первое и последнее вхождения максимума в последовательность:

```
1  #include <iostream>
2
3  int main()
4  {
5          int n;          /* Длина последовательности */
6          std::cin >> n;
7          int max;
8          int imaxFirst;      /* Порядковый номер первого максимума */
9          int imaxLast;       /* Порядковый номер последнего максимума */
10         int t;
11         std::cin >> t;
12         max = t;
```

```
13          /* Будем считать, что у первого элемента порядковый номер 1 */
14          imaxFirst = imaxLast = 1;
15          /* Переменная цикла всегда на 1 меньше порядкового номера элемента
                последовательности */
16          for(int i = 1; i < n; ++i)
17          {
18                  int a;
19                  std::cin >> a;
20                  /* Первый максимум: строгое неравенство */
21                  if (a > max)
22                  {
23                          max = a;
24                          imaxFirst = i + 1;
25                  }
26                  /* Последний максимум: нестрогое неравенство */
27                  if (a >= max)
28                  {
29                          max = a;
30                          imaxLast = i + 1;
31                  }
32          }
33      std::cout << "max: " << max << ", first seen at: " << imaxFirst << ", last seen
            at: " << imaxLast << std::endl;
34          return 0;
35  }
```

2.4 Поиск элемента в последовательности

Пусть нам требуется определить, есть ли в последовательности элемент с заданным значением, и если есть - то на каком месте. Задача очень похожа на порядковый номер минимума/максимума, с той лишь разницей, что проверяем мы элемент на строгое равенство. Опять же, в зависимости от задачи нам может потребоваться найти порядковый номер первого/последнего вхождения заданного элемента. Для этого можно использовать значение-метку.

```
1   #include <iostream>
2
3   int main()
4   {
5           int n;          /* Длина последовательности */
6           std::cin >> n;
7           int toFind;   /* Значение, которое надо найти */
8           std::cin >> toFind;
9       /* Первое и последнее вхождения заданного элемента; значение -1 является
            значением-меткой, означающим, что заданное число в последовательности
            отсутствовало */
10          int firstIdx = -1;
11          int lastIdx = -1;
12          for(int i = 0; i < n; ++i)
13          {
14                  int a;
15                  std::cin >> a;
16                  if (a == toFind)
17                  {
18                          /* Поиск первого вхождения: если здесь стоит что-то,
                                отличное от значения-метки, не перезаписываем порядковый
                                 номер */
19                          if (firstIdx == -1) firstIdx = i + 1;
```

```
20                          /* Поиск последнего вхождения: в любом случае
                               перезаписываем порядковый номер */
21                          lastIdx = i + 1;
22                  }
23          }
24      /* Проверяем, нашли ли мы хотя бы одно совпавшее число. Достаточно проверить
           только порядковый номер первого вхождения */
25      if (firstIdx != -1)
26      {
27                  std::cout << "first occurrence: " << firstIdx << std::endl;
28                  std::cout << "last occurrence: " << lastIdx << std::endl;
29          }
30          else
31          {
32                  std::cout << "number not in sequence" << std::endl;
33          }
34          return 0;
35  }
```

3 Управление ходом цикла

В ряде случаев (как, например, в пункте 2.4 при нахождении первого вхождения) алгоритм строится так, что выполнять все итерации цикла бессмысленно; возникают и другие ситуации, в которых текущую итерацию цикла следует прервать и перейти к следующей. Для выполнения этих действий в языке **C++** существуют два ключевых слова: **break** и **continue**.

Первое прерывает выполнение текущего цикла (если циклы вложены - то прерывается только самый внутренний цикл). В пункте 2.4 при нахождении первого вхождения разумно сделать именно это: в таком случае код для нахождения первого вхождения выглядел бы так:

```
1           for(int i = 0; i < n; ++i)
2           {
3                   int a;
4                   std::cin >> a;
5                   if (a == toFind)
6                   {
7                           /* Поиск первого вхождения: если здесь стоит что-то,
                               отличное от значения-метки, не перезаписываем порядковый
                                номер */
8                           if (firstIdx == -1)
9                           {
10                                  firstIdx = i + 1;
11                                  break;
12                          }
13          }
```

Второе переносит выполнение цикла **в конец тела цикла**, условно - после последнего выражения внутри цикла. В случае с циклом перечисления это означает, что будет выполнен *шаг цикла* и проверено *условие*.

Следует отметить, что эти ключевые слова следует использовать весьма ограниченно: они затрудняют чтение кода и в некоторых случаях (это в большей степени касается написания кода, выполняемого, например, на видеопроцессорах) не улучшают или даже ухудшают его производительность.