Язык программирования C++. Введение.

М. А. Ложников

5 сентября 2019 г.
Ревизия: 3

Это предварительная невыверенная версия. Читайте на свой страх и риск.

1 Введение

В настоящее время язык программирования C++ кардинально отличается от языка программирования C. Правильнее считать, что это два принципиально разных языка. Тем не менее, язык C++ включает в себя многие парадигмы языка C.

Не смотря на то, что часть синтаксиса языка C++ похожа на синтаксис C, язык программирования C++ не является в строгом смысле надстройкой над языком C, то есть не любая программа, написанная на чистом C будет компилироваться компилятором C++. Во многом это связано с более широким списком ключевых слов, а также с более строгой типизацией. Тем не менее, в большинстве случаев, программы, написанные на C, компилируются компилятором C++ (возможно, с небольшими правками).

На данном этапе мы будем постепенно знакомиться с возможностями языка C++.

2 Пространства имён

Предположим, что в программе используются две различные библиотеки, каждая из которых предоставляет функцию `f()`. В таком случае эти библиотеки будет затруднительно использовать в одном файле исходного кода из-за конфликта имён. В C++ эту проблему предлагается решать при помощи пространств имён.

```
1  namespace foo {
2    void f() {
3      // реализация функции f() из пространства имён foo
4    }
5  }
6
7  namespace bar {
8    void f() {
9      // реализация функции f() из пространства имён bar
10   }
11 }
12
13 /* Без пространств имён был бы конфликт имён из-за повторного определения
14    функции f(). */
15
```

```
16 void g() {
17   foo::f(); // вызов функции f() из пространства имён foo
18   bar::f(); // вызов функции f() из пространства имён bar
19 }
```

Таким образом, код (определения и реализации функций, константы) каждой библиотеки должен находиться в своём собственном пространстве имён.

2.1 Пример использования пространств имён

Listing 1: Файл foo.hpp

```
1 namespace foo {
2   // Определение функции f()
3   void f();
4 }
```

Listing 2: Файл foo.cpp

```
1 namespace foo {
2   // Реализация функции f()
3   void f() {
4   }
5 }
```

Listing 3: Файл bar.hpp

```
1 namespace bar {
2   // Определение функции f()
3   void f();
4 }
```

Listing 4: Файл bar.cpp

```
1 namespace bar {
2   // Реализация функции f()
3   void f() {
4   }
5 }
```

Listing 5: Пример использования пространств имён

```
1 #include <foo.hpp>
2 #include <bar.hpp>
```

```
3
4 void g() {
5   foo::f(); // вызов функции f() из пространства имён foo
6   bar::f(); // вызов функции f() из пространства имён bar
7 }
```

2.2 Пространство имён std

В заголовочных файлах языка C++ все стандартные функции вынесены в пространство имён `std`. Рассмотрим два примера.

```
1 #include <stdio.h>
2
3 /* В C++ доступны все стандартные функции языка C. Этот код компилируется
4    компилятором C++ и работает. */
5 int main() {
6   printf("Hello world!\n");
7   return 0;
8 }
```

Предыдущий пример компилируется и работает, однако, в C++ лучше не использовать заголовочные файлы языка C напрямую. Вместо `stdio.h` следует использовать `cstdio`, вместо `stdlib.h` — `cstdlib`, вместо `math.h` — `cmath` и т. д. Дело в том, что в новых файлах все функции вынесены в пространство имён `std`. Кроме того, в заголовочных файлах языка C++ некоторые функции могут работать с аргументами различных типов данных. Например, функция `std::fabs()` из `<cmath>` может работать с любыми целочисленными и вещественными аргументами, в то время как функция `fabs()` из `<math.h>` принимает только аргументы типа `double` (подстановка целочисленного аргумента приведёт к конвертации типов).

```
1 #include <cstdio>
2
3 /* В заголовочном файле "cstdio" функции вынесены в пространство имён std. */
4 int main() {
5   std::printf("Hello world!\n");
6   return 0;
7 }
```

2.3 Директива using namespace

В тех случаях, когда лень писать `std::` можно использовать директиву `using namespace`.

```
1 #include <cstdio>
2
3 using namespace std;
4
5 /* Теперь всё, что объявлено в пространстве имён std доступно в корневом
6    пространстве имён. */
```

```
int main() {
  printf("Hello world!\n");
  return 0;
}
```

Никогда не следует использовать директиву `using namespace` в заголовочных файлах, поскольку она подставится при включении соответствующего заголовочного файла.

3 Ввод/вывод

Не смотря на то, что в C++ работают средства ввода/вывода языка C, в C++ есть свои собственные средства ввода/вывода.

3.1 Потоковый вывод

```
/* Заголовочный файл <iostream> используется для подключения средств ввода/вывода. */
#include <iostream>

int main() {
  /* Для записи в стандартный поток вывода используется специальная переменная cout,
     которая является объектом класса std::ostream (о классах на следующих занятиях).
     Манипулятор endl служит для перевода строки, вместо него можно использовать "\n".
     Манипулятор endl отличается от "\n" тем, что сбрасывает содержимое внутренней
     памяти в поток вывода. */
  std::cout << "Hello world" << std::endl;

  int a = 2;
  double b = 3.14;

  /* Пример вывода переменных. */
  std::cout << "a = " << a
            << " b = " << b << std::endl;
  return 0;
}
```

3.2 Потоковый ввод

```
#include <iostream>
/* В заголовочном файле <string> описан класс std::string для работы со строками. */
#include <string>

int main() {
  int a;
  double b;
  // Переменная s является объектом класса std::string, то есть имеет тип std::string.
  std::string s;

  /* Для чтения из стандартного потока ввода используется специальная переменная
```

```
12      std::cin, которая является объектом класса std::istream. Обратите внимание,
13      что стрелки направлены в другую сторону. */
14    std::cin >> a;
15
16    /* Любой поток можно неявным образом преобразовывать к типу bool. В языке C++
17      переменная типа bool может принимать одно из двух значений: true (true = 1,
18      истина) или false (false = 0, ложь). Если поток приводится к типу bool
19      со значением false, значит считывание не удалось.
20    */
21    if (!std::cin) {
22      std::cout << "Can't read a!" << std::endl;
23    }
24
25    /* Оператор >>, применённый к потоку, возвращает сам поток, поэтому конструкцию можно
26      упростить. */
27    if (!(std::cin >> b)) {
28      std::cout << "Can't read b!" << std::endl;
29    }
30
31    /* Также можно читать строки. */
32    if (!(std::cin >> s)) {
33      std::cout << "Can't read s!" << std::endl;
34    }
35    return 0;
36  }
```

3.3 Поток cerr для вывода ошибок

Для вывода ошибок как правило используется специальный поток вывода `std::cerr`.

3.3.1 Пример 1

```
1  #include <iostream>
2
3  int main() {
4    int n;
5
6    if (!(std::cin >> n)) {
7      std::cerr << "Can't read n!" << std::endl;
8      return 1;
9    }
10
11   if (n < 0) {
12     std::cerr << "n < 0!" << std::endl;
13     return 2;
14   }
15
16   if (n == 0) {
17     // Выводим результат в стандартный поток вывода.
18     std::cout << "Result: 0" << std::endl;
```

```
    return 0;
  }

  /* В языке C++ разрешено перемешивать объявления переменных и код. В некоторых
     случаях переменные следует объявлять по мере необходимости, а не в начале блока. */
  double sum = 0;

  /* Внутри цикла for можно объявлять переменные-счётчики. Рекомендуется именно так и
     делать, чтобы не плодить лишние сущности. */
  for (int i = 0; i < n; i++) {
    double number;

    if (!(std::cin >> number)) {
      std::cerr << "Can't read the next number!" << std::endl;
      return 1;
    }

    sum += number;
  }

  sum /= n;

  std::cout << "Result: " << sum << std::endl;
  return 0;
}
```

3.3.2 Пример 2

```
#include <iostream>

int main() {
  int n = 0;
  double sum = 0;
  double number;
  /* Эта конструкция будет считывать действительные числа из стандартного потока ввода
     до тех пор, пока они считываются. */
  while (std::cin >> number) {
    sum += number;
    n++;
  }

  if (n > 0)
    sum /= n;

  std::cout << "Result: " << sum << std::endl;
  return 0;
}
```

4 Ключевое слово const

В языке C++ можно явно запретить менять значения переменных. Это делается для того, чтобы избежать ошибок, связанных со случайным изменением переменной. Попытка изменить значение такой переменной приводит к ошибке компиляции.

```
const int a = 5;

// a = 10; // Ошибка на этапе компиляции.

/* ptr является указателем на элементы типа const int. Это означает, что элементы
   массива нельзя менять, однако само значение указателя может быть изменено. */
const int* ptr;

// Сам указатель менять можно
ptr = (const int*) malloc(sizeof(int));

// ptr[0] = 10; // Ошибка на этапе компиляции.

/* Ошибка компиляции, функция free() принимает указатель на void (без const), значит,
   теоретически она может менять содержимое массива, а мы передаём указатель на
   константу. */
  free(ptr);

int* const constPtr = (int* const) malloc(sizeof(int));

constPtr[0] = 10; // Корректно

// constPtr = NULL; // Ошибка на этапе компиляции

/* Всё в порядке */
free(constPtr);

const int* const constPtrToConst = (const int* const) malloc(sizeof(int));

// constPtrToConst[0] = 10; // Ошибка на этапе компиляции
// constPtrToConst = NULL; // Ошибка на этапе компиляции

/* Ошибка компиляции, функция free() принимает указатель на void (без const), значит,
   теоретически она может менять содержимое массива, а мы передаём указатель на
   константу. */
  free(constPtrToConst);
```

В случае с указателями лучше читать написанное справа налево.

```
  /* Синтаксис: (const) ТИП* (const) ИМЯ;
     ИМЯ есть (константный) указатель на (константный) ТИП (соответствие в обратном
     порядке). */
```

4.1 Преобразования типов

Указатели на неконстантный тип могут быть неявно преобразованы к указателям на константный тип. Обратное преобразование приводит к ошибке компиляции (ошибка нейтрализуется при помощи `const_cast`, но об этом позже).

5 Lvalue ссылки

В C++ для каждой переменной можно создать ссылку, которая будет как бы вторым именем для той же самой переменной. Ссылки могут быть любого типа. Разберём на примере типа `int`.

```
int a = 5;
/* Переменная refToA является ссылкой на переменную a. Ссылка обязательно должна
   быть инициализирована некоторой переменной того же типа (или другой такой же
   ссылкой) сразу при объявлении. */
int& refToA = a;

/* Синтаксис: ТИП& ИМЯ ССЫЛКИ = ИМЯ ПЕРЕМЕННОЙ; */

/* Если поменять значение ссылки, то значение переменной, на которую она ссылается
   также поменяется. */
refToA = 10;
std::cout << "a = " << a << std::endl; // Выведет 10.

/* Ссылки бывают константными. Значения, на которые они указывают нельзя поменять
   напрямую (запрещено компилятором). */
const int& constRefToA = a;

/* Неконстантная ссылка не может указывать на константу. */
const int b = 5;
const int& constRefToB = b; // Корректно.
// int& refToB = b; // Ошибка компиляции.

/* Ссылки можно инициализировать другими ссылками, однако неконстантную ссылку нельзя
   инициализировать константной ссылкой, это приведёт к ошибке компиляции (ошибку
   можно нейтрализовать при помощи const_cast). */
int& anotherRefToA = refToA;
// int& refToConstRef = constRefToA; // Ошибка компиляции.
```

5.1 Передача аргументов в функцию по ссылке

В языке C++ при передаче в функцию параметры копируются. Для того, чтобы поменять аргумент изнутри функции, можно передать его по ссылке.

```
#include <iostream>

void foo(int a) {
  a = 10; // Снаружи значение переданного аргумента не поменяется.
}
```

```
6
7  void bar(int& b) {
8    b = 20; // А теперь поменяется.
9  }
10
11 int main() {
12   int c = 0;
13
14   foo(c);
15
16   std::cout << "c = " << c << std::endl; // Выведет 0.
17
18   bar(c);
19
20   std::cout << "c = " << c << std::endl; // Выведет 20.
21   return 0;
22 }
```

В тех случаях, когда в функцию передаётся аргумент сложного типа, то есть он долго копируется, например, строка типа `std::string`, однако этот аргумент не должен меняться снаружи, его следует передавать по константной ссылке.

```
1  #include <iostream>
2  #include <string>
3
4  void PrintLine(const std::string& str) {
5    std::cout << str << std::endl;
6  }
7
8  /* Если размер типа данных передаваемого аргумента превышает 16 байт, то его имеет смысл
9     передавать по ссылке. Для типа int это конечно же не нужно, однако как то так это
10    может выглядеть. */
11 int Sum(const int& left, const int& right) {
12   return left + right;
13 }
```

6 Выделение/освобождение памяти

Не смотря на то, что в C++ можно выделять/освобождать память при помощи функций `std::malloc()`/`std::free()` из `<cstdlib>`, этого делать не рекомендуется. Вместо этого следует использовать операторы `new[]`/`delete[]`.

```
1  /* Выделение памяти для массива элементов типа int. Память выделяется под 10 элементов.
2     Обратите внимание, что здесь указывается количество элементов, а не байт. */
3  int* arr = new int[10];
4
5  /* Синтаксис: ТИП* ПЕРЕМЕННАЯ = new ТИП[кол-во элементов]; */
6
```

```
for (int i = 0; i < 10; i++)
  arr[i] = i * i;

/* Оператор delete[] освобождает память, выделенную оператором new[]. */
delete[] arr;

/* Синтаксис: delete[] ПЕРЕМЕННАЯ; */
```

6.1 Пример: сумма элементов массива

```
#include <iostream>

bool ReadArray(int* arr, int n) {
  for (int i = 0; i < n; i++) {
    if (std::cin >> arr[i])
      return false;
  }

  return true;
}

/* Передача массива в функцию по указателю позволяет менять его содержимое внутри
   функции. В случае если содержимое массива меняться не должно, сама возможность может
   потенциально привести к ошибке. Поэтому считается хорошим тоном саму эту возможность
   запретить. Это делается при помощи ключевого слова const. В данном случае arr
   является указателем на элементы типа const int, то есть элементы массива не могут
   меняться. */
int Sum(const int* arr, int n) {
  int result = 0;

  for (int i = 0; i < n; i++)
    result += arr[i];

  return result;
}

int main() {
  int n;

  if (!(std::cin >> n)) {
    std::cerr << "Can't read n!" << std::endl;
    return 1;
  }

  if (n < 0) {
    std::cerr << "n < 0!" << std::endl;
    return 2;
  }

```

```
40  if (n == 0) {
41    std::cout << "Result: 0" << std::endl;
42    return 0;
43  }
44
45  /* Оператор new[] генерирует исключение (см. следующие занятия) в случае ошибки
46     выделения памяти. Проверку возвращаемого значения делать не нужно. */
47  int* arr = new int[n];
48
49  /* Тип данных int* можно неявно преобразовать к const int*. Всё в порядке. */
50  if (!ReadArray(arr, n)) {
51    std::cerr << "Can't read array!" << std::endl;
52    return 4;
53  }
54
55  std::cout << "Result: " << Sum(arr, n) << std::endl;
56
57  delete[] arr;
58
59  return 0;
60 }
```

Замечание 6.1 *Обратите внимание, что в примере выше нет проверки того, что память выделилась. Это связано с тем, что оператор* `new[]` *использует механизм исключений для обработки ошибок, о котором пойдёт речь на следующих занятиях. В данном примере программа вылетит в момент вызова оператора* `new[]` *в случае ошибки выделения памяти. Поэтому нет смысла проверять возвращаемое значение оператора* `new[]`*.*

7 Список задач

Задача 1 *(Источник:* `https://leetcode.com/problems/is-subsequence/`*) Даны две последовательности целых чисел заданной длины: $\{A_i,\ i = 1, \ldots, N\}$ и $\{B_j,\ j = 1, \ldots, M\}$. Проверить, является ли последовательность B подпоследовательностью последовательности A.*

Время работы программы: $O(N + M)$.

Задача 2 *(Источник:* `https://leetcode.com/problems/two-city-scheduling/`*). Предположим, что в некоторой конторе работает $2N$ человек, каждого из которых необходимо отправить в командировку в один из двух городов: A или B. Программе на вход поступает число N и два вещественных массива длины $2N$:* `costFlyToA` *и* `costFlyToB` *таким образом, что элементы* `costFlyToA[i]` *и* `costFlyToB[i]` *соответствуют стоимости отправки человека с номером i в города A и B соответственно. Программа должна вывести минимально возможную стоимость отправки сотрудников таким образом, что каждый сотрудник должен поехать ровно в один из городов A или B, и в каждом городе должно побывать ровно N человек.*

Время работы программы: $O(N \log N)$.

Задача 3 *(Источник:* `https://leetcode.com/problems/best-time-to-buy-and-sell-stock-ii/`*) Дана последовательность $\{a_i,\ i = 1, \ldots, N\}$. В элементе последовательности a_i записана стоимость некоторого товара в i-ый день. Разрешается продавать и покупать товар в любой день, однако нельзя купить товар, если он уже приобретён, для покупки его нужно сначала продать. Требуется посчитать наибольшую возможную прибыль.*

Время работы программы: $O(N)$.

Задача 4 *(Источник: `https://leetcode.com/problems/majority-element/`) Дан непустой массив длины N. Найти в массиве элемент, встречающийся более $N/2$ раз. Считайте, что такой элемент существует.*

Время работы программы: $O(N)$.

Задача 5 *(Источник: `https://leetcode.com/problems/k-closest-points-to-origin/`) Дано N точек на плоскости. Выведите K ближайших к началу координат точек (в любом порядке). Расстояние между точками определяется евклидовой метрикой.*

Время работы программы: *в среднем* $O(N)$.

Задача 6 *(Источник: `https://leetcode.com/problems/sort-array-by-parity-ii/`) Дан массив длины $2N$, половина элементов которого чётные, а вторая половина — нечётные. Отсортируйте массив таким образом, чтобы на месте чётных индексов стояли чётные элементы, а на месте нечётных — нечётные. Ответом является любой массив, удовлетворяющий требуемому условию.*

Время работы программы: $O(N)$.

Задача 7 *(Источник: `https://leetcode.com/problems/sort-colors/`) Дан массив длины N, элементами массива являются числа 0, 1 или 2. Отсортируйте массив таким образом, чтобы в начале располагались все нули, затем все единицы и, наконец, в конце массива лежали все двойки.*

Время работы программы: $O(N)$, *задача решается за один проход по массиву.*

Список литературы

[1] *Бьерн Страуструп* Язык программирования C++. М:Бином. 2011.

[2] `https://en.cppreference.com`

[3] `https://en.cppreference.com/w/cpp/language/namespace`

[4] `https://en.cppreference.com/w/cpp/language/namespace_alias`

[5] `https://en.cppreference.com/w/cpp/io/cin`

[6] `https://en.cppreference.com/w/cpp/io/cout`

[7] `https://en.cppreference.com/w/cpp/language/constant_expression`

[8] `https://en.cppreference.com/w/cpp/language/cv`

[9] `https://en.cppreference.com/w/cpp/language/const_cast`