C++ Beginners Guide

Aleksandr Vakulenko

21 марта 2024 г.

Содержание

1 Hello, world!

Так принято называть программу, впервые запускаемую при изучении нового языка программирования. Ее задачей является демонстрация нескольких важнейших особенностей языка программирования - общая структура любой программы и механизм вывода сообщений пользователю.

Можно сказать, что видимым(или полезным) для пользователя результатом работы любой программы, выполняемой на любом виде вычислительной системы, является последовательность так называемых побочных эффектов. Ведь обычно от того, что в процессоре переместились порции электрического заряда радости мало, подобное совсем не оправдывает расхода изрядной суммы на новый блестящий системный блок. То ли дело появление красивой подвижной картинки на мониторе, которая как раз и является побочным эффектом.

Конечно хотелось бы сразу перейти к чему-то столь же эффектному, но начать придется с малого. С вывода текста в консоль. Вывод сообщений это самое первое, что следует изучить взявшись за новый язык программирования.

Ниже приведен исходный текст программы, которая выводит к консоль сообщеине “Hello, World!”. Строк в этой программе совсем мало, но при этом можно разобрать около десятка ключевых мест.

Листинг 1: Hello world!

```
1 #include <iostream>
2
3 int main()
4 {
5     std::cout << "Hello, World!\n";
6     return 0;
7 }
```

Строка 1: символ # означает, что дальше идет директива препроцессора, в данном случае это директива include, которая подключает в исходный код содержимое файла iostream. Данный файл содержит часть заголовков стандартной библиотеки, а именно описание типов и методов для работы с потоками ввода/вывода. При этом имя взято в угловые скобки <>, это означает, что данный файл находится “у компилятора”, а не в папках пользовательского проекта. Пока об этом можно не задумываться.

На строке 3 **объявляется** и **определяется** функция с именем main. Функцией называется фрагмент кода, который может быть вызван для исполнения из других мест. Иногда функции имеют входные данные и/или еще и выходные данные. Принято называть процедурой функцию, которая ничего не возвращает. В данном курсе мы так делать не будем, разделение на функции и процедуры совершенно условно и не имеет большой ценности. К тому же в языках C и C++ функции всегда что-то возвращают, просто иногда это что-то является абсолютным ничто

(void). Изучим подробнее в разделе о функциях. А пока скажем только необходимый минимум. функция main особая - она обязана существовать в программе и при том только одна. Важность очевидно следует из ее предназначения - операционная система вызывает именно ее в тот момент(на самом деле не сразу, а после некоторых предварительных действий) когда мы запускаем исполняемый файл (*.exe). Возвращает она при этом целое число (int), но вот кому и зачем? Обратно операционной системе, после завершения своей работы, а переданное значение может быть как-то интерпретировано ОС или пользователем. Типичное значение в случает отсутствия ошибок это число 0.

Приведем сразу общее синтаксическое правило: тип возвращаемого значения указывается перед именем функции. После имени идут круглые скобки, в которых через запятую указываются типы входных аргументов. Само тело функции, т.е. тот самый код, являющейся сутью конкретной функции, приводится в фигурных скобках. Функций, которые возвращают не void, обязаны завершаться оператором return, которому передано какое-то значение (указанного типа).

Таким образом строка 6 означает следующее: выходим из функции обратно (в точку вызова), возвращая целое число типа int со значением 0. В данном случае эта строка не несет пользы лично нам, но необходима по правилам. Хотя и пользу когда-нибудь мы из этого получим.

В этот момент стоит упомянуть, что почти все строчки в исходном коде C/C++ заканчиваются точкой с запятой (;). В нашей программе это только две строки 5 и 6. Вполне уместен вопрос “А почему на остальных нет, это же тоже строки?”. Ответ на этот казалось бы простой вопрос не так уж и прост. Вероятно даже настолько сложен, что не появится в этом курсе. Если говорить коротко, то точка с запятой нам нужна только там, где компилятор может не понять, что строка на самом деле кончилась. На строке 1 компилятору очевиден конец строки. Строки 3-7 компилятор видит единым целым, примерно так:

```
1 int main() { ... }
```

И тут компилятору тоже очевидно, где это **определение** закончено. А вот если бы это было не **определение**, а **объявление**:

```
1 int main();
```

То точка с запятой была бы необходима. В отличии, например, от Python в C++ компилятор не видит переносов строк, отступов (tab) и пробелов, вообще всего, что называется “Whitespace character”. Позднее мы встретимся с ситуацией, когда отсутствие точки с запятой приводит к ошибке в логике работы программы, а не к ошибке компиляции. И ответим на старую шутку “Почему если компилятор может сообщить мне, что на какой-то строке пропущена точка с запятой, он не может сам ее там поставить?”.

Итак, коротенько разобрав основы синтаксиса простейшей программы перей-

дем наконец к содержательной части. Строка 5, которая и создает наш желаемый побочный эффект, содержит целых шесть различных элементов. Первый это пространство имен std. Вообще пространства имен бывают разные, их можно создавать самому, они могут быть частью сторонней библиотеки. В данном случае речь идет о пространстве имен, в котором живут все имена элементов стандартной библиотеки. Сделано это (и вообще пространства имен) для того, чтобы программисты между собой (и компилятор с ними же) не путали дублированные имена. Согласитесь, что после двух часов работы с векторами у вас появится имя типа vector, не имеющее ничего общего со стандартным контейнером std::vector. Примерно по этой же причине не рекомендуется использовать “using namespace std” , которое выпускает на волю все существующие там имена, а мы засекаем время до появления ошибки коллизии имен.

Осознав необходимость применения пространств имен давайте теперь научимся доставать из него нужные имена. Для этого применяется “Оператор расширения области видимости”. После имени пространства имен следуют два двоеточия, после чего нужное нам имя из этого пространства. Честно говоря, лучше один раз увидеть, чем читать про это.

Таким образом можно начать разбирать написанное на строке 5 - имя cout из пространства имен std стандартных функций. Уже неплохо, осталось понять, что это за такой cout и почему в его сторону направлены стрелочки от строки, которая потом появляется в выводе программы.

Сама по себе строка, представленная в нашем коде, является так называемым строковым литералом, т.е. неизменяемый текст, известный к моменту компиляции. Сам текст взят в двойные кавычки, а в конце идет ’$\backslash n$’, который означает перенос строки (ascii символ номер 10).

А вот две стрелочки влево от строки к cout это как ни странно функция, точнее оператор. Те кто изучал до этого чистый C могут признать побитовый сдвиг влево, что весьма странно, никаких битов тут явно нет, да и стоять от этого оператора справа должно целое число, а никак не строка.

В C++ ряд операторов можно наделить своим смыслов. Кто еще к этому моменту не перешел по ссылке на статью об операторах в C и C++, то перейдите и обратите внимание на колонку “Перегружаемый”. В данном случае оператор << означает вывод в поток, он так определен для объектов типа stream. Объект cout как раз является таким потоком (вывода).

Вообще забегая вперед скажем, что потоком называется такое место, куда (откуда) уходят (приходят) данные, и мы не знаем когда они прекратятся. Можно представить себе транспортерную ленту, которая уходит за горизонт. Вы оператором << кладете на нее строку, и смотрите как она уплывает вдаль. Вполне вероятно, что там ее и вовсе никто ее не ждет. Но ни нас, ни потока это не касается в момент вывода <<, cout обещает попробовать передать вашу строку операционной системе, а она попытается показать ее в консоли, скорее всего у нее это получится.

На этом и завершаем изрядно затянувшееся повествование о семи строчках программы “Hello world”.

2 Переменные и память.

3 Ветвление.

4 Циклы.

5 Функции.

При написании относительно большой программы практически сразу же встанет вопрос о дублировании однотипного кода. Рассмотрим типичную обучающую задачу по программированию. которая формулируется так: Напишите программу, которая вычисляет векторное произведение $\vec{C} = \vec{A} \times \vec{B}$. В коде ниже для этой цели объявлена простейшая структура Vec3 из трех вещественных чисел, а в основной части программы инициализированы два вектора $\vec{A} = (1, 2, 3), \vec{B} = (1, 1, 0)$ и третий $\vec{C}$ поэлементно заполняется координатами в соответствии с правилами векторного умножения и станет равен $(-3, 3, -1)$. Далее идет блок вывода вектора на экран, красиво оформленный расстановкой скобок вокруг координат и запятых с пробелами между ними.

Листинг 2: Векторное произведение (плохая версия)

```
1  #include <iostream>
2
3  struct Vec3{
4      float x,y,z;
5  };
6
7  int main()
8  {
9      // init vectors
10     Vec3 A = {1.1, 2.2, 3.3};
11     Vec3 B = {1, 1, 0};
12
13     // cross product
14     Vec3 C;
15     C.x = A.y*B.z - A.z*B.y;
16     C.y = A.z*B.x - A.x*B.z;
17     C.z = A.x*B.y - A.y*B.x;
18
19     // print result
20     std::cout << "(" << C.x
21               << ", " << C.y
22               << ", " << C.z << ")\n";
23
24     return 0;
25 }
```

Может показаться, что такая программа хороша и менять ее незачем. Обычно начинающий программист приводит следующий аргумент: "Она работает, результат правильный". Действительно, когда в задании указано, что необходимо

написать программу, которая считает векторное произведение, такой аргумент будет совершенно справедлив. Любые дальнейшие изменения кода не оправданы, задача решена, ответ правильный. Скорее всего в подобном коде даже не будет создаваться структура, а будет лишь 9 переменных float x1,x2,x3,y1,y2, ...,z3 с комментарием: "У нас же всего три вектора, зачем что-то еще?"; И вот в этом кроется главная проблема. Вспомнить формулу для векторного произведения - это не главная задача программиста. Язык C++ это не инженерный калькулятор, а возможность реализовать высокоуровневые абстракции, при этом не отрываться от низкоуровневых возможностей и не терять (почти) в производительности.

Давайте попробуем разобраться к каким проблемам приведет нас такой код и как сделать его лучше. Для начала разберем вывод вектора на экран. Это три строки, которые форматируют вывод в такой - (x, y, z). Стоит отметить, что для компилятора эти три строки видятся как одна строка (одно полное выражение). Поэтому мы можем в любом месте программы безнаказанно переносить строку для сокращения кода в ширину. Стандартным пределом по ширине считается 80 символов, это не обязательное требование, но строго рекомендуемое. Тем не менее для наших глаз это три строки, поначалу может быть не очень очевидные. Три координаты вектора, разделенные строковыми литералами, выводятся подряд. Если мы захотим вывести на экран еще один вектор, то придется скопировать код и везде заменить $\vec{C}$ на какой-то другой. На пятый раз это станет настолько скучно и неудобно, что начнутся ошибки по невнимательности.

Обратите внимание на выравнивание кода по оператору <<, которое так же не является синтаксически значимым, но строго рекомендуется для улучшения читаемости.

Часть с векторным произведением еще хуже, эта проблема там проявляется трижды - в двух операндах произведения - $\vec{A}, \vec{B}$ и в результате $\vec{C}$. В случае произведения других векторов нужно будет сделать до пятнадцати замен похожих друг на друга букв. Тут шанс ошибки где-то 90% в первом же копировании.

Так же стоит отметить, что суть фрагментов кода сейчас поясняют только комментарии, которые обычно и вовсе не ставятся. А без них через неделю-две можно будет только догадываться, что там происходит и зачем оно надо.

Давайте сделаем первый логический шаг в сторону улучшения:

Листинг 3: Попытка исправить печать значений

```
1     // print result
2     Vec3 vec_to_print = C;
3     std::cout << "(" << vec_to_print.x
4               << ", " << vec_to_print.y
5               << ", " << vec_to_print.z << ")\n";
```

Таким образом мы решили проблему подстановки вектора в различные места, заменив подстановкой в единственное. При этом проблема повторного использования кода в другом месте ухудшилась, теперь копировать надо четыре строки,

для банального вывода данных на экран это многовато. Напрашивается решение проблемы - нужно как-то так сделать, чтобы весь однотипный код был куда-то вынесен, получил бы имя, по которому на него можно сослаться и как-то понимал, что мы хотим подставить туда переменную C. Такое решение есть и это функция.

Листинг 4: Функция печати вектора на экран

```
1   void print_vec3(Vec3 vec_to_print)
2   {
3       std::cout << "(" << vec_to_print.x
4                 << ", " << vec_to_print.y
5                 << ", " << vec_to_print.z << ")\n";
6   }
7
8   ...
9
10  main()
11  {
12      ...
13      print_vec3(A);
14      print_vec3(B);
15      print_vec3(C);
16      ...
17  }
```

Для начала замечание: для сокращения объема многоточием в листинге кода будут обозначаться несущественные в данный момент фрагменты кода. А теперь давайте разберемся в происходящем.

На строке 1 начинается **объявление** и **определение** функции. Первым идет тип данных, который функция возвращает. В данном случае это "ничто"(void) так как наша функция и не должна что-либо возвращать. Иногда функции, которые ничего не возвращают, называются процедурами, но в языках C/C++ синтаксических различий нет, функция всегда что-то возвращает, просто иногда это "ничто".

Далее идет имя функции print_vec3. Так как компилятор должен понимать какую конкретно функцию мы хотим вызывать, то для простоты пока скажем, что имена у функций, находящихся в одной области видимости должны быть уникальные. Более сложные случаи будут рассмотрены в следубщих разделах.

В скобках после имени функции перечисляются типы входных аргументов (если они есть). В случае **определения** функции - эти аргументы должны тут же получить имена. Мы сохраняем то же имя временной переменной, что и в прошлом коде vec_to_print.

В фигурных скобках далее идет тело функции, т.е. тот самый код, который будет исполняться при ее дальнейшем вызове. В данном случае (и только) принято переносить открывающую фигурную скобку на новую строку. Тело функции

располагается с отступом на ширину 4 (или 8) знаков.

Таким образом у нас получилось, что фрагмент кода получил имя и один переменный аргумент, передаваемый из вне (полный эквивалент строки 2 листинга 3). Осталось этой функций воспользоваться. На строках 13, 14, 15 эта функция вызывается подряд для всех трех векторов. Выглядит намного приятнее, чем 12 строчек однотипного кода. При этом и имя функции и имя аргумента в теле функции имеют короткие понятные имена, однозначно дающие понять для чего они нужны.

Выше уже было сказано про эквивалентность ручного копирования в новую переменную и передачи аргумента в функцию при вызове. Давайте разберемся с этим подробнее. В прошлый раз на строке 2 листинга 3 создавалась новая переменная vec_to_print, которая с этого момента существовала до конца функции main(), что кажется странным так как ее суть полностью исчерпывается тремя последующими строками.

Замечание: дословное копирование кода листинга 3 несколько раз подряд - это ошибка, переменная vec_to_print может быть объявлена только один раз. Попытка это исправить только еще больше запутает код.

Теперь же переменная vec_to_print является **локальной переменной** функции print_vec3. Её **время жизни** ограничено строками 1-5 (листинг 4), за пределами этой функции ее не существует, а имя свободно для использования. В момент вызова функции на строках 13-15 (листинг 4) значения переменных A, B, C поочередно и независимо копировались в локальную переменную vec_to_print. И каждый раз при выходе из функции компьютер "забывал"её значение и некоторым образом очищал память. Пока скажем об этом процессе только то, что это место, где живет vec_to_print при каждом вызове, называется стеком и отложим подробный разбор до следующих разделов.

В завершение приведем улучшенный код нашей программы, векторное произведение также заменено вызовом соответствующей функции (листинг 5). В данном примере демонстрируется еще одна возможность. Для того, чтобы скрыть служебные функции подальше можно предварительно только **объявить**, что такие функции существуют (строки 7-8), а **определить** (строки 23-37) их ниже или вообще в другом файле, также в **объявлении** функций могут отсутствовать фактические имена аргументов. К моменту вызова (строки 14, 16-17) компилятор знает, что такие функции существуют, при этом наличие их тела (**определения**) его пока не волнует. В случае отсутствия тела произойдет ошибка компиляции, но на моменте **компоновки**, а не **трансляции** исходного кода в объектный. Также обратите внимание, что функция cross_product имеет возвращаемый тип Vec3. На строке 32 создается локальная переменная tmp, а при возврате на строке 36 будет произведено копирование ее значения обратно в место вызова. Имя tmp происходит от temporary (временный) и часто подчеркивает, что переменная имеет столь короткое время жизни, что придумывать ей особое имя незачем, а её назначение понятно из контекста.

Листинг 5: Векторное произведение (хорошая версия)

```
1  #include <iostream>
2
3  struct Vec3{
4      float x,y,z;
5  };
6
7  void print_vec3(Vec3);
8  Vec3 cross_product(Vec3, Vec3);
9
10 int main()
11 {
12     Vec3 A = {1, 2, 3};
13     Vec3 B = {1, 1, 0};
14     Vec3 C = cross_product(A, B);
15
16     print_vec3(A);
17     print_vec3(B);
18     print_vec3(C);
19
20     return 0;
21 }
22
23 void print_vec3(Vec3 vec_to_print)
24 {
25     std::cout << "(" << vec_to_print.x
26               << ", " << vec_to_print.y
27               << ", " << vec_to_print.z << ")\n";
28 }
29
30 Vec3 cross_product(Vec3 argA, Vec3 argB)
31 {
32     Vec3 tmp;
33     tmp.x = argA.y*argB.z - argA.z*argB.y;
34     tmp.y = argA.z*argB.x - argA.x*argB.z;
35     tmp.z = argA.x*argB.y - argA.y*argB.x;
36     return tmp;
37 }
```

6 Ссылки.