Синтаксис C++

1. Основные принципы синтаксиса

Программа на C++ — это (как и в других языках) последовательность команд. Большинство команд должны заканчиваться точкой с запятой.

Структура программы формируется фигурными скобками, т.е. блоки функций, блоки if'ов, циклов и т.п. указываются с помощью фигурных скобок. В отличие от питона, отступы в программе на C++ не имеют никакого значения для компилятора. С точки зрения компилятора можно ставить отступы как хотите, и вообще разбивать программу на строки как хотите и т.д. (Есть некоторые исключения, типа директив компилятора, см. выше, и однострочных комментариев, см. ниже.) Тем не менее, конечно, рекомендуется ставить отступы аналогично тому, как они ставятся в питоне (ну и на самом деле в любом другом языке программирования) — чтобы программу было удобнее читать.

Комментарии в C++ бывают двух типов: однострочные — они начинаются с двух слешей подряд (`//`) и длятся до конца строки, и многострочные — начинаются с `/*` и идут до `*/`. Например:

```
#include <iostream>

using namespace std;

int main() {
    int a, b;   // это комментарий
    cin >> a >> b;   /* и
    это
    тоже
    комментарий */ int s = a + b;
    cout << s << endl;
    return 0;
}
```

Язык C++ чувствителен к регистру заглавные и маленькие буквы различаются. В простейших программах принято использовать только маленькие буквы. Большие буквы обычно используются в типах (именах классов) и в названиях глобальных констант и макросов, в наших программах вам такое редко будет нужно.

Переменные определяются в основном внутри функций, но также можно определить и *глобальные* переменные — их надо определять вне всех функций:

```
#include <iostream>
```

```
using namespace std;

int a, b;

int main() {
    cin >> a >> b;  // тут теперь используются глобальные a и b
    int s = a + b;
    cout << s << endl;
    return 0;
}
```

Глобальные переменные будут видны во всех функциях, определенных ниже (по коду программы) самих переменных. Вообще, глобальные переменные не рекомендуется использовать, но в простых программах вы можете их использовать, если они действительно нужны в разных функциях (например, если вы пишете поиск в глубину, то можно граф сделать глобальной переменной).

2. Целочисленные типы данных и переполнения

В отличие от питона, в котором тип для целых чисел один и он может хранить сколько угодно большие числа (переходя на длинную арифметику при необходимости), в C++ есть очень много разных типов для целых чисел, и у каждого свои границы допустимого интервала значений. При этом типы жестко не определены; допустимый интервал у одного типа может быть разный в разных компиляторах или даже при разных опциях одного компилятора.

- **int** — основной, наиболее широкоупотребимый тип. Хранит числа от -2^{31} до $2^{31}-1$, либо (в зависимости от компилятора и опций) от -2^{63} до $2^{63}-1$, занимает соответственно 4 или 8 байт.
- **unsigned int** (так и пишется, с пробелом!), или сокращенно **unsigned** — *беззнаковый* (т.е. не хранит знак числа, а вместо него хранит дополнительный бит значения числа) аналог int, хранит числа от 0 до $2^{32}-1$ или до $2^{64}-1$, занимает соответственно 4 или 8 байт (столько же, сколько и int).
- **long long int**, или сокращенно **long long** — хранит числа от -2^{63} до $2^{63}-1$, занимает 8 байт.
- **unsigned long long int**, или сокращенно **unsigned long long** — беззнаковый аналог long long'a, хранит числа от 0 до $2^{64}-1$, занимает 8 байт.

Важной особенностью целочисленных типов в C++ являются **переполнения**. Если вы попытаетесь сохранить в переменную

значение за пределами допустимого диапазона ее типа, то вместо этого сохранится какое-то другое значение, принадлежащее допустимому диапазону. При этом в C++ не возникнет никакой ошибки, просто молча получится неправильный ответ.

Поэтому всегда, когда работаете с целочисленными типами данных, помните про опасность переполнения. Всегда оценивайте, какое максимальное значение может получиться в той или иной переменной, и проверяйте, влезет ли оно в тип. Если не влезает в 4-байтный int, то лучше сделайте переменную `long long` (вообще говоря, никто не мешает вообще все переменные делать `long long`, но тогда вы рискуете, что какие-то большие массивы не пройдут по ограничению памяти, плюс `long long` тоже может переполниться). Если вы видите, что ответ не влезает даже в `long long`, то тут уже надо думать. Возможно, в конкретном компиляторе есть 16-байтовый целочисленный тип (типа `int128_t` или `__int128`), но это далеко не всегда так, ну и он тоже может переполниться. Или вам надо использовать длинную арифметику. Или придумать другой алгоритм, в котором не будут возникать такие большие числа.

3. Арифметические операции

Сложение, вычитание и умножение делаются также, как и в других языках, через +, - и *, тут ничего особенного. Специального оператора для возведения в степень нет, пишите цикл :) (ну или быстрое возведение в степень, или pow, в зависимости от ситуации).

А вот с делением есть особенности. Неполное частное берется оператором /, остаток берется оператором %, но при этом нет прямого способа разделить два целых числа так, чтобы получилось вещественное (т.е. в C++ / — это питоновский //, а аналога питоновскому / нет). Чтобы получить вещественное деление, вам надо явно сделать так, чтобы хотя бы одно из чисел было вещественное.

Например:

```
int x = 10, y = 3;
cout << x / y;   // выведет 3
cout << 1.0 * x / y;  // сделали числитель вещественным, выведет 3.33333
```

Частный, но очень важный случай — запись `1/2` дает ноль. Чтобы получить 0.5, надо написать, например, `1.0/2` (ну или напрямую `0.5`, конечно).

4. Присваивания, auto и ++

Присваивание делается одиночным равенством:

```
s = a + b;
```

(Это подразумевает, что у вас уже есть переменная s, куда вы просто хотите записать новое значение.). Также есть сокращенные операторы присваивания как в питоне: +=, -=, *=, /=, %=. Мы также видели, что присваивания можно использовать сразу при объявлении переменной:

```
int a = 10;
```

В таком случае также вместо конкретного типа можно использовать специальное слово auto, которое обозначает «используй тот тип, который в правой части выражения» (это появилось только в C++11):

```
int a, b;
...
auto c = a + b;  // тип выражения a+b — int, поэтому переменная
c получается тоже int
```

Запись auto a = 10 не очень понятна (какого типа 10 — int? unsigned? long long?..), поэтому ее не надо использовать. А вот если справа сложное выражение, то вполне можно использовать auto.

Есть также специальные конструкции ++ и --, которые обозначают увеличить или уменьшить переменную на 1:

На самом деле, тут есть два варианта записи этих операторов: `a++` и `++a`, и аналогично с `--`. Оба увеличивают `a` на единицу, но отличаются возвращаемым значением, т.е. значением самого выражения (которое используется, если вы написали типа `b = a++` или например вызываете функцию: `foo(a++)`). При записи `a++` возвращаемое значение будет равно старому значению `a` (типа сначала запомни значение `a`, потом увеличь его на 1), при `++a` — новому (типа сначала увеличь, потом используй значение `a`), и аналогично с `--`:

```
int a = 10;
a++;  // увеличить a на 1, получается a == 11
a--;  // уменьшить на 1, получается обратно 10
```

Но вообще использовать результат операторов `++` и `--` — это плохая практика, не делайте так. Пишите `a++` отдельной командой, и тогда проблем не будет.

Квадратный корень вычисляется через `sqrt`, для него надо подключить заголовочный файл `cmath` (`#include <cmath>`). Модуль вычисляется через `abs`.

5. Условный оператор (if) и логические операции

Записывается так:

```
if (условие) {
    код
} else {
    код
}
```

Часть else, конечно, может быть опущена:

```
if (условие) {
    код
}
```

Важно тут следующее. Во-первых, условие обязательно заключается в круглые скобки. Во-вторых, сам код заключается в фигурные скобки; именно они определяют, какой код находится внутри if'а. Исключение — если в `if` только одна команда, то можно фигурные скобки не писать. Но это не рекомендуется делать, за исключением ситуаций, когда команда очень простая.

В условии, как и в питоне, можно использовать сравнения (`>`, `>=`, `<`, `<=`, `==`, `!=`), обратите внимание, что сравнение делается двойным равенством (собственно, как и в питоне, и в отличие от паскаля).

Важный момент тут — что C++ не выдает ошибку, если вы напишете одиночное равенство, а не двойное:

```
if (a = b) {...}
```

но это уже вовсе не сравнение, это присваивание! и поэтому работает совсем не так, как вы можете думать. Это очень частая ошибка, особенно у тех, кто переходит с паскаля. Питон в такой ситуации выдает ошибку, а вот C++ — нет.

Логические операции записываются так: and — `&&`, or — `||`, not — `!`. Пример:

```
if ((year % 400 == 0) || (year % 4 == 0 && !(year % 100 == 0)))
```

(конечно, можно было и просто написать `year % 100 != 0`). Конструкции `elif` в C++ нет. Но она и не нужна — вы прекрасно можете просто писать `else if`:

```
if (...) {
    ...
} else if (...) {
    ...
} else if (...) {
    ...
} else {
    ...
}
```

6. Циклы

Цикл `while` пишется так, как вы, наверное, уже ожидаете:

```
while (условие) {
    код
}
```

Как и в `if`, тут обязательно брать условие в скобки, и тело цикла заключается в фигурные скобки, исключение — если тело цикла состоит из одной команды, скобки можно не ставить (но все равно рекомендуется). Работает цикл `while` так же, как и в других языках.

А вот цикл `for` в C++ пишется и работает довольно необычно. В простейшем случае он пишется так:

```
for (int i = 0; i < n; i++) {
    код
}
```

В общем виде в заголовке `for` есть три части, разделенные точкой с запятой. Первая часть (`int i = 0` в примере выше) — что надо сделать перед циклом (в данном случае — объявить переменную `i` и записать туда ноль). Вторая часть (`i < n`) — условие продолжения цикла: это условие будет проверяться перед самой первой итерацией цикла и после каждой итерации, и

как только условие станет ложным, выполнение цикла закончится (аналогично условию while). И третья часть (`i++`) — что надо делать после каждой итерации до проверки условия.

То есть запись выше обозначает: заведи переменную `i`, запиши туда ноль, дальше проверь, правда ли, что `i<n` и если да, то выполняй тело цикла, потом делай `i++`, опять проверяй `i<n`, если все еще выполняется, то опять выполняй код и делай `i++`, и т.д., до тех пор, пока в очередной момент не окажется `i>=n`.

Примеры:

```
for (int i = n - 1; i >= 0; i--)  // цикл в обратном порядке
for (int i = 0; i < n; i+= 2)  // цикл с шагом 2
for (int i = 0; !found && i < n; i++)  // цикл закончится когда found станет true, или i >= n
for (int i = 1; i < n; i *= 2)  // цикл по степеням двойки
```

То есть на самом деле `for` в C++ — очень мощный вид цикла, такой, что даже обычный `while` является частный случаем `for` (потому что в for можно просто опустить ненужные части заголовка: `for (; условие;)` полностью эквивалентно `while (условие)`). Но настоятельно рекомендуется использовать `for` только в тех ситуациях, когда у вас есть явная «переменная цикла», которая как-то последовательно меняется, и тогда в заголовке `for` вы упоминаете только ее. Если вам надо что-то сложнее, пишите `while`.

Обратите также внимание, что переменную цикла принято объявлять прямо в заголовке цикла. В частности, такая переменная не будет видна снаружи цикла — ну и правильно, если вы пишете цикл `for`, нечего использовать переменную цикла после цикла. И заодно это позволяет например написать два цикла `for` подряд с одной и той же переменной, причем эти переменные не обязаны иметь одинаковый тип:

```
for (int i = 0; i < n; i++) {
    код, тут i -- int
}
// тут переменной i нет вообще
for (unsigned int i = 1; i < m; i *= 2) {
    код, тут i -- unsigned
}
```

Есть еще одна форма цикла `for`, которая появилась в C++11 — это так называемый range-based for. Это уже чистый аналог питоновского `for ... in`,

который позволяет итерироваться не по range, а по более-менее любому объекту (массиву, строке и т.п.). На C++ это пишется так:

```
for (int i : v) {
    код
}
```

здесь предполагается, что v — это массив int'ов, и тогда i последовательно принимает все значения элементов этого массива.

В частности, тут часто удобно использовать auto:

```
for (auto i : v) {
    ...
}
```

у переменной i получится такой же тип, как у элементов массива.

Команды break и continue есть и работают в точности так же, как в питоне и паскале; в частности, можно писать while (true) и далее в коде использовать break.

Кроме того, есть еще цикл do-while с проверкой условия после итерации, я его не буду описывать (хотя там ничего сложного), он бывает довольно редко нужен (точнее даже практически никогда, не случайно в питоне нет его эквивалента).

7. Массивы

Массив — это последовательность объектов того же типа, которые занимают смежную область памяти. Традиционные массивы в стиле C являются источником многих ошибок, но по-прежнему являются общими, особенно в старых базах кода. В современных C++ мы настоятельно рекомендуем использовать std:: Vector или std:: Array вместо массивов в стиле C, описанных в этом разделе. Оба этих типа стандартных библиотек хранят свои элементы в виде непрерывного блока памяти. Однако они обеспечивают гораздо большую безопасность типов и итераторы поддержки, которые гарантированно указывают на допустимое расположение в последовательности. Дополнительные сведения см. в разделе контейнеры.

Одномерные массивы в C++

Одномерный массив — массив, с одним параметром, характеризующим количество элементов одномерного массива. Фактически одномерный массив — это массив, у которого может быть только одна строка, и n-е количество

столбцов. Столбцы в одномерном массиве — это элементы массива. На рисунке 1 показана структура целочисленного одномерного массива **a**. Размер этого массива — 16 ячеек.

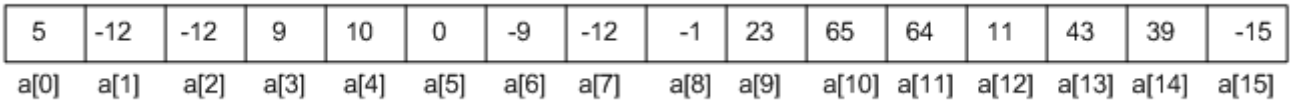

Рисунок 1 — Массивы в C++

Заметьте, что максимальный индекс одномерного массива **a** равен 15, но размер массива 16 ячеек, потому что нумерация ячеек массива всегда начинается с 0. Индекс ячейки – это целое неотрицательное число, по которому можно обращаться к каждой ячейке массива и выполнять какие-либо действия над ней (ячейкой).

```
//синтаксис  объявления одномерного массива в C++:
/*тип данных*/  /*имя одномерного массива*/[/*размерность
одномерного массива*/];
//пример объявления одномерного массива, изображенного на
рисунке 1:
int a[16];
```

где, int — целочисленный тип данных;
a — имяодномерного массива;
16 — размер одномерного массива, 16 ячеек.

Всегда сразу после имени массива идут квадратные скобочки, в которых задаётся размер одномерного массива, этим массив и отличается от всех остальных переменных.

8. Логический тип данных

Логический тип данных называется bool и может принимать два значения: true и false (с маленькой буквы). Как и в других языках, в переменную типа bool можно записывать напрямую результаты сравнений и других условий; и переменную типа bool можно использовать напрямую в if'ах, while'ах и т.п.

В отличие от других языков, bool — тоже *целочисленный тип*. Если вы пишете арифметическое выражение, то false превращается в 0, а true — в 1. Аналогично, логические операции на самом деле принимают не

только `true`/`false`, но и произвольные числа: `0` считается `false`, а все остальные значения — `true`:

```
bool x = 1 + 2;  // 1 + 2 == 3, превратится в true.
int y = x;  // x == true, превратится в 1.
int z = x + 10;  // x == true, превратится в 1, 1 + 10 == 11.
if (z) {  // работает так же, как if (z != 0).
}
cout << true << '\n';  // выведет 1.
cout << false << '\n';  // выведет 0.
cin >> x;  // ожидает на вход либо 0, либо 1, другие числа или строки нельзя.
```

Но в целом не стоит так писать, в некоторых случаях это может приводить к незаметным ошибкам. Пишите проверки полностью (`z != 0`), как в `if`'ах, так и при сохранениях `int` в `bool` и в подобных случаях, ну и не используйте арифметические операции с `bool`.

9. Функции

Функция в общем виде определяется так:

```
int foo(int x, double y, string s) {
    ...
}
```

Это определена функция `foo`, которая принимает три параметра: `x` типа `int`, `y` типа `double` и `s` типа `string`, и возвращает тип `int`. Если аргументов нет, то надо обязательно написать пустые скобки: `int foo() {...}`. Внутри функции для завершения функции и возврата значения используется команда `return <значение>`.

Любая ветка исполнения функции обязана завершаться командой `return <значение>`, ее отсутствие — это undefined behavior (см. ниже), т.е. в случае ее отсутствия программа может вести себя вообще как угодно. (Исключение — функции, возвращающие `void`, см. ниже.)

Особый случай — функции, не возвращающие ничего («процедуры», если пользоваться терминами паскаля). Для таких функций надо указать специальный тип возвращаемого значения `void`:

```
void foo() {
    ...
}
```