Statements, expressions

Программы на C пишутся в виде текста, который называется *исходный код*. Он может быть разбит на множество файлов, обычно у них расширение `.c`. Когда эти файлы пропускаются через специальную программу – *компилятор* – они становятся набором машинных инструкций, которые компьютер может выполнять напрямую.

Statement — это команда, инструкция, действие

Исходный код на C описывает последовательность выполнения действий. Поэтому в каждой программе вам придётся привести шаги, которые приведут её к цели. Каждый шаг это команда: "Сделай то! Сделай это!" Отсюда и название такого стиля программирования: *императивный*.

Описание каждого действия в исходном коде программы называется *statement (утверждение, высказывание)*. Они перечисляются через точку с запятой:

```
...

какой-то statement ;

еще один statement ;

...
```

Точка с запятой означает: "Сначала выполни то, что слева от меня, затем – то, что справа!". Существует много видов statement'ов: `if`, `return`, `while` и другие. Постепенно мы будем с ними знакомиться.

Когда мы говорим о выполнении программы по шагам, шагом будет как раз выполнение одного statement'а, после чего можно остановиться и посмотреть на состояние программы.

Expression — это выражение, подсчёт чисел или других данных

Второй важный тип кусочков программ называется *expression* (выражение). Они описывают вычисления, в которых есть конечный результат, то есть некоторые данные.

Вот примеры выражений:

- `42` посчитается как 42
- `4 + 9` посчитается как 13
- `2 * (3 + 1)` посчитается как 8
- `2 * 3 + 1` посчитается как 7
- `11 / 5` посчитается как 2, потому что деление двух целых чисел считается целочисленным.
- `11 % 5` посчитается как 1, это остаток от целочисленного деления 11 на 5

Арифметические действия имеют привычный нам приоритет: сначала умножение и деление, потом сложение и вычитание.

Выражения могут быть частью statement'ов, что мы сейчас увидим.

Первые примеры statement'ов

Сразу начнём писать кусочки программ, которые могут сделать что-то осязаемое. Научимся выводить текст на экран!

Выводим текст в одну строку

Чтобы вывести фиксированную строчку, например, `Hello!`, мы пишем такой statement (обратите внимание на двойные кавычки):

```
printf("Hello!") ;
```

Результат:

```
Hello!
```

Можно разбить строку на части и выводить каждую отдельным вызовом `printf`. Например, этот код тоже выводит Hello! в одну строчку и без пробелов:

```
printf("Hel") ;
```

```
printf("lo!") ;
```

Результат:

```
Hello!
```

Как видите, между последовательными `printf` не выводятся ни пробелы, ни переводы строк.

Выводим несколько строк текста

Чтобы вывести несколько строчек текста, нужно напечатать специальный код `\n` между ними:

```
printf("\n");
```

Не перепутайте направление косой черты (слэша)!

Например:

```
// вывести две строчки: первая это "Hello", вторая это "world"
```

```
printf("Hello");
```

```
printf("\n");
```

```
printf("world!");
```

Результат:

```
Hello
world!
```

Код переноса строки может быть частью строки:

```
// то же самое:
printf("Hello\nworld!"); // тут код \n это часть строчки
```

Результат:

```
Hello
world!
```

Считаем выражения и выводим числа

Чтобы вывести на экран результат подсчёта выражения, который является целым числом, необходимо использовать более сложную конструкцию:

```
printf("%d", <выражение>);
```

Здесь и далее `<угловые скобки>` мы используем для описания мест, куда можно что-то вписать; в угловых скобках мы пишем пояснение. Вместо `<выражение>` можно подставить выражение любой сложности; результатом его вычисления должно быть обязательно целое число. Например:

```
printf("%d", 42);            // вывести 42
printf("%d", 20*2 + 2);      // вывести 42
printf("%d", 7* (8-1) - 7); // вывести 42
```

А следующий код уже приведёт к ошибке, и программа аварийно завершит работу:

```
printf(20*2 + 2);
```

... потому что мы забыли написать `"%d"`.

Пока мы привыкнем писать конструкции языка для достижения нужных результатов, но не всегда будем их детально понимать.

Напишите один statement, который выводит на экран результат сложения числа 17283 и произведения четырёх чисел: 5, 6, 7 и 8. Не забудьте, что каждый statement завершается точкой с запятой!

Про ошибку `implicit declaration of function `printf`.` Если вы видите такое сообщение об ошибке:

```
main.c:2:1: warning: implicit declaration of function ‘printf’
```

то проверьте, что у вас написано внутри скобок!

1task.c – program

Функции

Функция это часть исходного кода программы, описывающая действия, направленные на достижение единой цели. У функции есть *имя*, отражающее эту цель, и *тело*, состоящее из самих вычислений.

Пример создания новой функции

Мы создаём новую функцию, указывая её *определение*. Например, так выглядит определение функции с именем `proc`, которая выводит несколько строчек на экран:

```
void proc() {
   printf("Hello!");                  // Тело функции состоит
   printf("\n");                      // из этих трёх statement'ов
   printf("From ITMO university");    // Каждый завершается точкой с запятой
}
```

Ключевое слово `void` означает, что функция служит всего лишь контейнером для нескольких statement'ов; мы её определяем чтобы дать последовательности действий имя – в данном случае, это имя `proc`. Тело всегда пишется `{ в фигурных скобках }`.

Определение любой функции соответствует шаблону:

```
void <имя функции>() {
   <тело функции>
}
```

Здесь и далее встречается `<текст в угловых скобках>`. Он обозначает такие места в программе, куда можно что-то вписать, и объясняет, что именно. Например, вместо `<имя функции>` можно подставить `f`, и это будет значить, что мы определяем функцию с именем `f`. Сами угловые скобки, конечно, нужно стереть.

Сравните шаблон описания функции и её пример:

`void <имя функции>() {`
 	
`  <тело функции>`
 	
`}`
	`void proc() {`
	`  printf("Hello");`
	`  printf(" world!");`
	
	`}`

Нельзя писать statement'ы вне функций!

Так нельзя:

```
printf("Something");
```

```
void proc() {
  ...
}
```

Функция это инструкция, а не сами действия

Определив функцию мы не запустим действия внутри неё автоматически. Определение функции это инструкция по выполнению определённых в ней действий. Ясно, что инструкция и процесс выполнения действий по этой инструкции это разные вещи. Скоро мы научимся *запускать* функции на выполнение.

Напишите определение функции с именем `print_newline`, которая переведёт вывод на новую строку, то есть выведет код `"\n"`.

Не нужно писать полную программу. Если требуется написать функцию, то нужно написать только её (и можно добавлять вспомогательные функции); также не подключайте заголовочные файлы, даже если вы уже знаете, что это такое.

В именах функций заглавные и строчные буквы отличаются. `print_newline` и `print_newLine` и `Print_NewLine` это разные функции.

2task.c – program

Вызов функций

Дав имя кусочку кода мы можем обратиться к нему используя это имя – *вызвать функцию,* написав её имя и скобки. Вызов функции с именем `proc` (см. выше) будет выглядеть так:

```
proc();
```

В момент вызова программа обращается к телу функции и выполняет его. После этого программа продолжает свою работу с места вызова функции `proc` .

Рассмотрим такой пример. Это уже маленькая программа на C, которую можно запустить.

```
1  void p() {
2      printf("Hello!");
3      printf("\n");
4  }
5
6  void main() {
7      p();       // тело функции main
8      p();       // тело функции main
9      p();       // тело функции main
10 }
```

Запустить пример в отдельном окне.

Исходный код программы – это чертёж, схема, инструкция по её выполнению. Когда мы запускаем программу, она начинает исполняться с функции `main`. Такая функция должна быть в любой программе.

Содержимое функции `main` выполняется последовательно. На строчке 7 произойдёт первый вызов к `p`; и выполнение main приостановится пока мы не выполним функцию `p`.
Внутри `p` будут выполнены строчки 2 и 3, программа выведет `Hello!`

Теперь тело вызванной функции `p` закончилось, мы возвращаемся в место вызова и выполняем следующий statement, уже на строчке 8. Аналогично будут произведены вызовы на строчках 8 и 9, после чего функция `main` закончится и программа завершит работу.

Функции, которые вызываются внутри `main`, тоже выполнятся. И функции, которые вызывают функции, вызванные в `main`, тоже выполнятся (и так далее). А функции, до которых нельзя добраться по цепочке вызовов из `main`, никогда не будут выполнены.

Представим, что мы уже определили несколько функций:

```
void greet() {
    printf("Hello, ");
}

void b() {
    printf("Boris");
}

void v() {
    printf("Vladimir");
}

void print_newline() {
    printf("\n");
}
```

Вызывая эти функции, напечатайте на экран следующее:

```
Hello, Boris
Hello, Vladimir
Hello, Boris
```

Пользоваться функцией `printf` напрямую в этом задании запрещено.

В текстовое поле нужно ввести только последовательность вызовов функций (как будто мы внутри тела функции `main`).

Функции с одним аргументом

Рассмотрим функцию, выводящую на экран результат вычисления выражения 40+2.

```
void print_int() {

   printf("%d", 40 + 2);

}
```

Эта функция всегда выводит число 42. Нельзя переиспользовать её чтобы вывести на экран другое число.

Функции, выводящие другие числа, выглядят похоже:

```
void print_int1() {

   printf("%d", 99 );

}



void print_int2() {

   printf("%d", 8762 * 73 );

}
```

Эти функции различаются только выражением в скобках после `printf`. Вместо них можно сделать одну универсальную функцию `print_int`, которая способна выводить результат вычисления любого выражения. Для этого возьмём функцию, написанную для конкретных значений данных, а также покажем её вызов из другой функции:

```
void print_int() {




   printf("%d", 40 + 2 );

}



void main() {
```

```
    print_int();


}
```

Мы можем заменить любые выражения (expressions) в функции `print_int` на именованные *аргументы*. Для этого необходимо одновременно внести в программу три изменения:

```
void print_int( int x ) {
//               ^^^^^ добавили в скобки тип аргумента и его имя


    printf("%d", x );
}   //          ^^^  заменили 40 + 2 на x


void main() {


    print_int( 40 + 2 );
//            ^^^ указали значение для аргумента `x` в месте вызова
}
```

Функция `print_int` принимает один аргумент с именем `x` типа `int`. Аргументы могут быть разных типов, но пока что мы знаем только один тип целых чисел `int`.

Когда мы вызываем функцию с каким-то значением аргумента, например, 42, происходит следующее:

1. Все вхождения аргумента функции в её теле заменяются на 42.

```
2. /*
3. Тело функции
4. void print_int( int x ) {
5.    printf("%d", x );
6. }
```

```
7.
8. становится:
9. */
10.
   printf("%d", 42 );
```

11. Функция выполняется с учётом этой подстановки.

Теперь в зависимости от того, как мы вызовем функцию `print_int`, на экран будут напечатаны разные числа:

```
print_int( 40 + 2 );   // выведет 42

print_int( 9 * 2 );   // выведет 18

print_int( 0 );   // выведет 0
```

Если значение аргумента является сложным выражением, а не просто числом, то сначала оно будет подсчитано. Это значит, что `print_int( 40 + 2 )` эквивалентно не такому действию:

```
printf( "%d", 40 + 2 )
```

... а такому:

```
printf( "%d", 42 )
```

Иными словами, если аргумент – сложное выражение, то он подсчитается один раз, а затем это значение будет подставлено в функцию.

Вам дано определение функции:

```
void greet( int n ) {
   printf("Hello ");
   printf("%d", n);
   printf("\n");
}
```

Вызывая функцию `greet` с разными аргументами, напечатайте на экран следующее:

```
Hello 10

Hello 20

Hello 42
```

Пользоваться функцией `printf` напрямую в этом задании запрещено.

4task.c – program

Функции с несколькими аргументами

Функции с несколькими аргументами выглядят похоже: вместо одного аргумента в скобках мы указываем несколько через запятую. Следующая функция печатает два числа, каждое на новой строке:

```
void print_int2(int arg1, int arg2) {
   printf("%d", arg1);
   printf("\n");
   printf("%d", arg2);
}
```

Эта функция принимает два аргумента: arg1 и arg2. Эти аргументы – целые числа. Вот как можно вызвать эту функцию с аргументами 10 и 33:

```
print_int2( 10, 33 );
```

Результатом этого вызова будет такой текст:

```
10

33
```

Кстати, `printf` это тоже функция, которая определена в стандартной библиотеке языка C. Конструкция `printf(аргументы)` – это тоже вызов функции.

Шаблон определения функции с *n* аргументами выглядит так:

```
void <имя функции>(<тип 1> <аргумент 1>, <тип 2> <аргумент 2>, ..., <тип n> <аргумент n>) {
   <тело функции>
}
```

Напишите функцию с именем `f`, которая принимает *через аргументы* два числа и печатает *на экран* их сумму с помощью `printf`. Посмотрите на пример выше – там функция `print_int2` принимает два числа через аргументы и просто печатает их; теперь нужно распечатать не числа по отдельности, а сложить их и распечатать результат.

Не нужно писать полную программу, если об этом явно не сказано в задании. Если требуется написать функцию, то нужно написать только её (и можно добавлять вспомогательные функции); также не подключайте заголовочные файлы, даже если вы уже знаете, что это такое.

5task.c – program

Возвращаемое значение функции

Как вы знаете, код внутри функции служит единой цели, и имя функции отражает эту цель. Зачастую цель функции в вычислении какого-то значения. Это подсчитанное значение мы можем вернуть из функции в другую функцию, которая её вызвала; оно называется *возвращаемым значением*.

Посмотрим на примере, как возвращать значения из функций. Функция `sum` вычисляет сумму двух чисел; вся программа считает сумму чисел 4 и 3 и выводит их на экран.

```
int sum(int x, int y) {
    return x + y;
}


void main() {
    printf("%d", sum(4, 3) );
}
```

Функция `sum` отличается от функций, которые мы рассматривали ранее:

- вместо ключевого слова `void` указан *тип возвращаемого значения*: `int` ;
- в функции присутствует специальное предложение `return <expr>`, где `<expr>` – произвольное выражение. При его выполнении функция сразу завершает свою работу и *возвращает* результат вычисления `<expr>`.
- вызов функции теперь можно использовать в качестве выражения. Его тип такой же, как тип возвращаемого значения. В данном случае это `int`. Вместо вызова функции подставляются данные, которые она вычислила.

Подведём итог. В общем случае мы объявляем функцию по следующему шаблону:

```
тип_возвращаемого_значения имя_функции (тип1 имя_параметра1, тип2
имя_параметра2...) {

    тело функции

}
```

В целях общности мы можем считать `void` специальным типом, который имеет только одно значение с которым ничего нельзя сделать: ни складывать, ни сохранять в памяти, ни проводить какие-то иные операции.

Код возврата программы

Функция `main`, с которой начинается выполнение программы, тоже возвращает число. Это так называемый *код возврата*. Программисты договорились, что если программа работала корректно, нужно вернуть код возврата 0; если же были какие-то ошибки, надо вернуть ненулевое значение и в документации на программу описать, каким именно ошибкам соответствует данный код.

В этом модуле для упрощения мы часто пишем `void main() {...}`, но правильнее писать:

```
int main() {

    ...

    return <код возврата>;

}
```

Напишите функцию `avg3` с тремя аргументами, которая вернёт среднее арифметическое своих аргументов. Подразумевается использование целочисленного деления, т.е. среднее чисел 3, 6 и 10 будет 6, а не 6.3333...

Не нужно писать полную программу, если об этом явно не сказано в задании. Если требуется написать функцию, то нужно написать только её (и можно добавлять вспомогательные функции); также не подключайте заголовочные файлы, даже если вы уже знаете, что это такое.

6task.c – program

Итоги

- Программы выполняются последовательно.
- Statement'ы это конструкции языка, соответствующие действиям, командам.
- Expression'ы это конструкции языка, соответствующие вычислениям.
- Мы научились печатать текст и целые числа с помощью функции printf.
- Кусочки кода изолируются внутри функций.
- Каждая функция имеет имя, отражающее её цель.
- Функции могут вызывать друг друга.
- Функции могут не иметь аргументов или иметь несколько аргументов, возможно разных типов.
- Функции могут возвращать значение с помощью конструкции return. Это значение будет подставляться в выражение вместо вызова функции.
- Как только функция дошла до return, она сразу завершается и передаёт значение тому, кто её вызвал.