Циклы

Циклы это конструкции языка для повторения одинаковых действий много раз подряд.

Конструкция `while (<expr>) { <loop> }` позволяет выполнить statement'ы из фрагмента кода `<loop>` 0 или более раз пока условие `<expr>` истинно. Фрагмент кода `<loop>` называется *телом* цикла. Например, следующий код выведет числа 0, 1, 2, 3, 4:

```
int x = 0;


while (x < 5) {        // это условие выполнения (инвариант)


  printf("%d\n", x);   // это тело цикла

  x = x + 1;           // это тело цикла


}
```

Каждый раз перед выполнением тела цикла программа проверяет, выполняется ли условие `x < 5`. Если условие верно, то мы выполняем тело цикла, опять проверяем условие и идём дальше по кругу. Если условие становится ложным, то тело цикла больше не будет перезапускаться.

Если условие никогда не было истинным, цикл будет пропущен и не выполнится ни одного раза.

Условие также называют *инвариант*. В математике инвариантом часто называют утверждение, которое сохраняется истинным на протяжении последовательности действий. Например, для арифметической прогрессии 1, 3, 5, 7... инвариантом можно назвать утверждение "текущий член прогрессии больше предыдущего". Для цикла, однако, в этом термине есть нюанс: условие, конечно, было верно перед началом итерации тела цикла, но как только тело начало выполняться, уже не имеет значения, истинно оно или ложно. Во время выполнения цикла условие может становиться то ложным, то истинным. Например, этот цикл будет выполняться бесконечно:

```
int x = 0;


while (x == 0) {

  x = 10;
```

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

  x = 0;

}
```

А в этом примере цикл будет выполнен один раз, но условие будет ложно уже в начале выполнения его тела:

```
int x = 0;


int check() {

  if (x == 0) {

    x = 1;

    return 1;

  }

  else { return 0; }

}


...
```

```
while (check ()) {

  // любой дальнейший вызов check() вернёт 0


  printf("Loop\n");

}
```

Решите задачу в соответствии с указаниями в комментариях. Вам доступна функция `int read_int()`.

Sample Input:

```
1
```

Sample Output:

```
1 2 3 4 5 6 7 8 9 10
```

1task – program

Выход из цикла

Можно прервать исполнение цикла "на середине" с помощью таких statement'ов:

- `break` приведёт к немедленному выходу из цикла. Программа продолжит исполняться со statement'а, которое следует после тела цикла. Например, этот код выведет строчку `0 1 2 3 4 Out.`
- `int i = 0;`
-
- `while (1) {`
- `   printf("%d ", i);`
- `   i = i + 1;`
- `   if (i > 4) break;`
- `}`
-

  `printf("Out");`

  Обратите внимание: без `break` цикл бы выполнялся бесконечное количество раз, ведь значение 1 всегда истинно.

- `continue` приведёт к тому, что текущая итерация цикла завершится, и мы опять окажемся на стадии проверки условия. Иными словами, мы сразу попадём на следующую итерацию цикла (или выйдем из него, если условие ложно). Код ниже выведет `1 2 7 8 9 10` -- если 2 < i < 7, то мы пропускаем вывод.
- `int i = 0;`
-
- `while (i < 10) {`

```
    i = i + 1;
    if (i > 2 && i < 7 ) {
      continue;
    }
    printf("%d ", i);
}
```

Если `i` попадает в интервал (2,7) мы выполняем `continue` и перепрыгиваем вывод `printf`, оказываясь на следующей итерации цикла.

Конечно, выйти из цикла можно и с помощью `return`, который вообще завершит выполнение текущей функции. Из функций, которые ничего не возвращают (помечены `void`), можно выйти с помощью `return` без аргументов.

```
void f() {
  int i = 0;
  while (1) {
      i = i + 1;
      if (i > 10) {
          return; // выйдем из бесконечного цикла (и из функции) через 10 итераций
      }
  }
  printf("Never");        // до этой строчки мы никогда не дойдём.
}
```

Напишите функцию `is_square`, которая принимает число в качестве аргумента. Она должна возвращать 1 если число является квадратом, и 0 в противном случае.

Например, 99 является квадратом, т.к. 3^2 = 93^2=9, а 66 не является квадратом, т.к. не существует такого натурального числа nn, что n^2=6n^2=6.

[Про ошибку control reaches the end of non-void function.]

Sample Input:

```
16
```

Sample Output:

```
1
```

2task – program

Цикл for

Зачастую, когда мы думаем про цикл, мы хотим выделить некий набор значений и с каждым из них совершить какие-то действия. С помощью цикла `while` это можно сделать так:

```
void print_int(int a) { printf( "%d ", a ); }



int i = 0;

while ( i < 10 ) {

    print_int( i );

    i = i + 1;

}
```

Обратите внимание на три кусочка кода, которые постепенно изменяют `i` от 0 до 10:

1. Инициализация (объявляет переменную `i`):

   ```
   int i = 0;
   ```

2. Инвариант (условие продолжения):

   ```
   i < 10
   ```

3. Шаг (на каждой итерации цикла обновляет `i`):

   ```
   i = i + 1;
   ```

Эти три кусочка кода служат единой цели: "изменять `i` от 0 до 10 с шагом 1".
Цикл `for` позволяет сгруппировать логику изменения `i` в одном месте:

```
for( int i = 0; i < 10; i = i + 1 ) {

    print_int( i );
```

```
}
```

Совершенно необязательно, чтобы инициализация, инвариант и шаг были как-то связаны с изменением переменной. Например, функция `f` делит свой аргумент на 2 и выводит его, пока не дойдёт до нуля:

```
void f(int z) {

    for (int i = z; i > 0 ; printf("%d\n", i) ) {

        i = i / 2;

    }

}
```

В общем случае цикл `for` выглядит так:

```
for( <statement инициализации>; <expression инвариант>; <statement шаг> ) {

   <тело цикла>

}
```

- инициализация выполняется перед всем остальным;
- инвариант проверяется перед каждой итерацией цикла;
- шаг выполняется после каждой итерации цикла (и `continue` не пропускает его)

Для чисел от 1 до 100 включительно выведите их делители большие 1 в порядке возрастания. Вывод начинается так:

```
1:

2: 2

3: 3

4: 2 4

5: 5

6: 2 3 6

...
```

После каждого делителя должен быть пробел, в том числе в конце строки.

3task – program

В предыдущем задании вас мог удивить шаблон для решения. В нём фигурировали следующие функции:

```
int divides(int a, int b) { return a % b == 0; }

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

Зачем делать такие маленькие функции, код которых умещается в одну строчку?

1. Код легче читать. Функции дают имена последовательностям действий, которые имеют смысл в контексте нашей программы.
   К примеру, бегло взглянув на выражение `a % b == 0` мы понимаем, что в нём считается остаток от деления. Однако остаток от деления может считаться для совершенно разных целей. Напротив, увидев `divides(a,b)`, мы понимаем, что нас интересует именно делимость.
2. В маленьких функциях сложнее сделать ошибку, а если она вкрадывается, то её легко локализовать и исправить.
3. Маленькие функции легко протестировать. Если вы сомневаетесь в её правильности, достаточно запустить её на нескольких наборах аргументов и посмотреть результат. Это одна из причин, почему функции должны принимать все необходимые им данные через аргументы.
4. Маленькие функции легко и быстро писать, что экономит ваши силы и интеллектуальные ресурсы.
5. Современные компиляторы умеют оптимизировать программы так, что большое количество маленьких функций не ведёт буквально ни к какому ухудшению производительности: там, где нужно, они встраиваются прямо в место вызова.

Пишите маленькие функции, которые делают мало действий – они позволят вам разрабатывать очень большие и сложные программы, .

Простое число -- это число, большее единицы, которое делится только на 1 и на само себя.

Напишите функцию `is_prime`, которая проверяет число на простоту и возвращает 1 если число простое и 0 в противном случае.

Функция должна корректно работать для любых знаковых чисел.

Sample Input:

```
13
```

Sample Output:

```
1
```

4task – program