1 Занятие №1.

1.1 Первая программа

Рассмотрим такую задачу: *написать программу, которая вводит пары чисел и вычисляет их НОД (наибольший общий делитель)*. Для вычисления будем использовать соотношение

$$\gcd(a,b) = \gcd(b, a \bmod b).$$

1.1.1 Текст программы

Полный текст программы приведён ниже.

```
1  #include <stdio.h>
2  /* a first program */
3  int main(void)
4  {
5          int a, b;
6          int c;
7          while (scanf("%d%d", &b, &c) == 2) {
8                  if (b <= 0 || c <= 0) {
9                          printf("Invalid␣parameters\n");
10                 } else {
11                         // invariant: GCD(a,b)==GCD(b,a%b)
12                         do {
13                                 a = b;
14                                 b = c;
15                                 c = a % b;
16                         } while (c != 0);
17                         printf("%d\n", b);
18                 }
19         }
20         return 0;
21 }
```

Эта программа в цикле считывает два целых числа, вычисляет НОД этих чисел и печатает результат. Источник, из которого считываются числа называется *стандартным потоком ввода*. Обычно он связан с вводом с клавиатуры, но в дальнейшем мы рассмотрим, как можно перенаправлять на стандартный поток ввода произвольные файлы. Результат работы печатается на *стандартный поток вывода*. Обычно это экран или окно на нём. Опять-таки, в дальнейшем мы рассмотрим, как можно перенаправлять вывод программы. Клавиатура и дисплей, отображающий информацию, вместе называются *терминалом*.

Теперь мы эту программу по шагам, демонстрируя базовые возможности языка **Си**. Как видно из текста программы, она записывается в свободном формате, то есть все пробелы и переводы строк при компиляции программы игнорируются. Поэтому для придания тексту программы большей наглядности используются отступы. Существует много стилей расстановки отступов в тексте программы (по вопросу «наилучшего» стиля много лет идут «религиозные» войны). В дальнейшем мы будем придерживаться стиля отступов предложенного авторами языка в своей Книге.

1.1.2 Комментарии в тексте программы

Вставить комментарий в текст программы или закомментировать существующий фрагмент можно следующими способами.

```
11                          // invariant: GCD(a,b)==GCD(b,a%b)
```

Такой комментарий начинается с двух символов `//` и продолжается до конца строки[1]. Существует возможность продлить строчный комментарий на следующую строку, поставив в самом конце строки символ `\`. Между ним и концом строки не должно быть пробельных символов.

```
2 /* a first program */
```

Это — блочный комментарий, который начинается с двух символов `/*` и заканчивается первыми встретившимися символами `*/`. Такой комментарий может располагаться на нескольких строках текста программы. Иногда для лучшего выделения комментария в тексте программы каждую строку начинают с дополнительного символа `*`, например так.

```
/* a first program - calculate GCD of two numbers
 * Public domain
 */
```

Обратите внимание, что такие комментарии не могут быть вложенными. Появление первой же пары символов `*/` означает завершение всего комментария. Поэтому нельзя комментировать функции целиком, если они уже содержат блочные комментарии. Так, следующий фрагмент:

```
1 /*
2 int simple_func(int n)
3 {
4   /* add 1 to argument */
5   return n + 1;
6 }
7 */
```

вызовет синтаксическую ошибку при компиляции программы. Блочный комментарий, начинающийся на строке 1 фактически заканчивается на строке 4. Для того, чтобы отключить компиляцию большого фрагмента кода используются директивы препроцессора, которые мы рассмотрим позднее. Поэтому *никогда не комментируйте блочным комментарием фрагменты кода*!

Внутри комментария одного вида специальные символы `//` и `/*`, начинающие комментарий не действуют. Поэтому

```
1 /* + 1 + 2 // + 3 + 4 */ + 5
+ 6
```

эквивалентно

```
1  + 5
+ 6
```

Каждый комментарий считается эквивалентным одному символу пробела. Поэтому `a/**/b` эквивалентно `a␣b`, а не `ab`.

[1] Строго говоря, такой комментарий был официально введён только в стандарте C99, но к этому моменту его поддерживали все существующие компиляторы.

1.1.3 Директива препроцессора

```
1  #include <stdio.h>
```

Это — так называемая директива препроцессора. Перед основной фазой трансляции в текст программы будет добавлено содержимое файла `stdio.h`, которые находится в одном из стандартных каталогов операционной системы. Собственно язык **Си** не определяет никаких функций, которые поддерживают операции ввода/вывода, работу со строками и другие необходимые возможности. Все они вынесены в стандартную библиотеку **Си**. В **UNIX**-системах стандартную библиотеку языка **Си** часто называют **libc**. Файл `stdio.h` — это один из файлов стандартной библиотеки, в котором определяются типы данных, константы, переменные и функции, необходимые для операций ввода/вывода. Наличие такой директивы препроцессора указывает, что в программе будут использоваться функции ввода/вывода. Поскольку почти всякая программа что-либо вводит или выводит, включение файла `stdio.h` будет присутствовать почти в каждой программе.

1.1.4 Функция `main`

```
3  int main(void)
4  {

21 }
```

Это — определение функции с именем `main`. В языке **Си** отсутствует деление на процедуры и функции, все выполняемые объекты называются функциями, даже если они не возвращают значения-результата.

Программа на языке **Си** является совокупностью функций и глобальных определений, то есть понятие о каком-то главном блоке, с которого начинается работа, отсутствует. В программе должна быть определена функция с именем `main`, именно эта функция будет вызвана первой, после того, как завершится инициализация объектов стандартной библиотеки.

Стандарт языка **Си** диктует, чтобы функция `main` была объявлена как возвращающая целое значение. Если это не так, Вы, скорее всего получите предупреждение от компилятора[2]. Дело в том, что функция `main` вызывается операционной средой, и поэтому должна удовлетворять требованиям операционной среды, то есть её тип параметров и тип возвращаемого значения жёстко фиксированы. Один из вариантов — это функция, которая не принимает параметров и возвращает значение целого типа, как в нашем примере. Итак, *всегда объявляйте функцию `main` как возвращающую значение целого типа*!

Фигурные скобки после заголовка функции — это составной оператор, аналог **`begin`** и **`end`** языка **Паскаль**. Тело функции в языке **Си** заключается в составной оператор, даже если оно состоит из одного оператора. Иногда составной оператор мы будем просто называть *блоком*.

В начале любого составного оператора, а не только того, которые образует тело функции, могут стоять определения переменных. Эти переменные, как правило, существуют ограниченное время, пока составной оператор не завершил работу.

В **C90** все определения локальных переменных должны размещаться до операторов, а в стандарте **C99**, как и в языке **Си++**, определения локальных переменных и операторы могут быть перемешаны. Вы можете объявить локальную переменную в середине блока, а также в заголовках цикла `for`.

[2] В **Си++** правила жёстче — это будет ошибка компиляции.

1.1.5 Определения переменных

```
5         int a, b;
6         int c;
```

Это — определение локальных переменных a, b, c, которые имеют целый тип.

Простейшее определение переменных имеет такой вид.

```
определение = тип список_переменных ";"
список_переменных = переменная { "," переменная }
переменная = имя [ "=" значение ]
```

При определении переменная может быть сразу проинициализирована некоторым значением, например.

```
int a = 4;
```

Если определение локальной переменной не содержит инициализации, то начальное значение такой переменной не определено. Так, в нашем случае начальное значение переменных a, b и c не определено, то есть в них содержится «мусор», оставшийся на стеке по ходу выполнения программы.

Регистр букв (заглавные—строчные) в идентификаторах и ключевых словах значим, то есть Main и main — это два различных идентификатора. Тем не менее, в программе не рекомендуется использовать имена, различающиеся только регистром букв.

1.1.6 Простые типы данных

В языке Си определены следующие основные простые типы.

_Bool	целый тип, достаточный для хранения значений 0 и 1 (C99);
char	целый тип, достаточный для хранения символов;
int	целый тип;
float	вещественное число с одинарной точностью;
double	вещественное число с двойной точностью;
_Complex	модификатор комплексного типа (C99).

Чтобы определить тип более точно используются квалификаторы **short**, **long**, **signed**, **unsigned**, в результате можно использовать следующие простые типы данных, которые перечислены в таблице 1.

Ключевое слово, записанное в квадратных скобках, означает, что оно может быть опущено. Так, тип **[signed] long [int]** может записываться и как **signed long**, и как **long int**, и как **signed long int**, и просто как **long**. Кроме того, ключевые слова могут свободно переставляться друг относительно друга, то есть тот же самый тип может задаваться и как **long int signed**.

Является ли тип **char** знаковым или беззнаковым, зависит от конкретной реализации компилятора языка Си. Поэтому везде, где требуется выполнять арифметические операции, зависящие от знаковости, необходимо явное указание знаковости типа **char**.

Размер типа **char** является единицей измерения размера всех других типов. Постулируется, что переменная типа **char** занимает один байт. Таким образом может получиться, что байт будет содержать, например, 9 битов. Все прочие типы имеют размер, кратный целому числу байтов.

Тип **int** имеет размер машинного слова на данной архитектуре. Если машинное слово — 16 бит, то и тип **int** будет иметь такой же размер.

char	целый тип, достаточный для хранения кода символа.
signed char	знаковый целый тип, достаточный для хранения символа.
unsigned char	беззнаковый целый тип, достаточный для хранения символа.
[signed] short [int]	короткое целое со знаком.
unsigned short [int]	короткое целое без знака.
[signed] int	целое.
unsigned [int]	целое без знака.
[signed] long [int]	длинное целое.
unsigned long [int]	длинное целое без знака.
[signed] long long [int]	более длинное целое (введён стандартом C99).
unsigned long long [int]	более длинное целое без знака (введён стандартом C99).
float	число с плавающей точкой одинарной точности.
double	число с плавающей точкой двойной точности.
long double	число с плавающей точкой расширенной точности.
_Complex float	Комплексный тип с вещественной и мнимой частью типа **float** (C99).
_Complex double	Комплексный тип с вещественной и мнимой частью типа **double** (C99).
_Complex long double	Комплексный тип с вещественной и мнимой частью типа **long double** (C99).

Таблица 1: Простые типы языка Си

Все знаковые типы имеют точно такой же размер, как и беззнаковые типы. Способ хранения знаковых чисел стандартом не определён, но все существующие архитектуры хранят знаковые числа в формате дополнения до 2.

Стандарт определяет следующие неравенства для размеров целых типов.

$$\texttt{sizeof(char)} < \texttt{sizeof(short)} \leqslant \texttt{sizeof(int)} \leqslant \texttt{sizeof(long)} \leqslant \texttt{sizeof(long long)}$$

$$\texttt{sizeof(short)} < \texttt{sizeof(long)}$$

$$\texttt{sizeof(int)} < \texttt{sizeof(long long)}$$

Все современные архитектуры имеют байт, состоящий из 8 бит. На многих современных 32-битных архитектурах целые типы имеют следующие размеры:

short 2 байта (16 бит).
int 4 байта (32 бита).
long 4 байта (32 бита).
long long 8 байтов (64 бита).

Как правило, 64-битные архитектуры имеют следующие размеры целых типов:

short 2 байта (16 бит).
int 4 байта (32 бита).
long 8 байтов (64 бита).
long long 8 байтов (64 бита).

Для типов с плавающей точкой точность (количество битов мантиссы) и диапазон представления (количество битов порядка) растёт от **float** до **long double**. Для архитектуры **ix86** обычно размер типа **float** равен 4 байта, типа **double** — 8 байтов, типа **long double** — 12 байтов, из которых используется 10.

1.1.7 Функция ввода данных

```
7                    scanf("%d%d", &b, &c)
```

Это — вызов функции `scanf` для считывания данных со стандартного потока ввода (обычно стандартный поток ввода — это клавиатура). Функция `scanf` первая функция стандартной библиотеки языка **Си**, которую мы рассмотрим.

Первый аргумент функции это строка формата (строки мы будем рассматривать позже, а пока заметьте, что они записываются в кавычках), которая определяет, значения каких типов должны быть считаны. Остальные аргументы задают переменные, в которые должны быть считаны значения.

Строка формата состоит из спецификаторов ввода полей. Каждый спецификатор ввода начинается со знака `%` («процент»). Простейшие спецификаторы ввода перечислены ниже.

`%d`	Считать целое число (**int**). Перед чтением числа пропускаются все пробельные символы (пробелы, табуляции, переводы строк). Если первый непробельный символ не может начинать число, функция `scanf` завершается. Иначе число считывается либо пока не возникнет переполнения, либо пока не встретится символ, который не может быть частью числа. Если возникло переполнение, функция `scanf` завершается с неудачей. Если встретился нецифровой символ, чтение числа считается успешным, а этот символ не будет считан из потока.
`%ld`	Считать длинное целое число (**long int**). Используются те же самые правила, что и при чтении обычного целого числа.
`%Ld`	Считать длинное целое число (**long long int**). Используются те же самые правила, что и при чтении обычного целого числа.
`%f`	Считать вещественное число типа **float**. Применяются те же правила, что и при чтении целых чисел.
`%lf`	Считать вещественное число типа **double**. Применяются те же правила, что и при чтении целых чисел.
`%Lf`	Считать вещественное число типа **long double**. Применяются те же правила, что и при чтении целых чисел.
`%c`	Считать очередной символ из входного потока в переменную типа **char**.

Несколько спецификаций формата могут быть записаны в одной форматной строке. Например, `"%lf%d%f"` — считать вещественное значение в переменную типа `double`, затем целое значение в переменную типа `int`, затем вещественное значение в переменную типа `float`.

После спецификации формата перечисляются переменные, в которые будет записано значение. Для всех простых типов, которые были упомянуты выше, перед именем переменной *обязательно* должен стоять знак `&`. Это — унарная операция взятия адреса переменной, которую мы ещё рассмотрим в дальнейшем.

Количество спецификаций формата в форматной строке обязательно должно совпадать с количеством переменных, указанных после неё. Кроме того, тип, указанный в спецификации формата, обязательно должен совпадать с типом соответствующей переменной. Если вы где-то ошиблись, то многие компиляторы ничего не заметят и скомпилируют программу.

При выполнении такая программа будет либо давать неправильный результат, либо вообще завершаться аварийно.

Функция `scanf` возвращает количество успешно считанных полей, заданных в спецификации. В примере из программы, которую мы пишем, значение 2 означает, что успешно считаны два целых числа, 1 означает, что было считано только первое число, а второе — нет (закончился файл или до начала чтения числа встретился символ, которые не может быть частью целого числа). Если бы `scanf` вернул 0, это значит, что ни одна переменная не была считана из-за неверного задания входных данных в файле. Если данные закончились (то есть чтение дошло до конца файла) до того, как была считана хотя бы одна переменная, функция `scanf` вернёт значение, задаваемое константой `EOF`.

Игнорировать значение, возвращаемое функций `scanf` нельзя! То есть, язык, конечно, позволяет это делать, но программа, написанная таким образом будет работать некорректно, если пользователь ошибся при вводе.

1.1.8 Оператор цикла `while`

```
7         while (scanf("%d%d", &b, &c) == 2) {

19        }
```

Это — оператор цикла **`while`**. Цикл **`while`** выполняется (как и в **Паскале**) до тех пор, пока истинно выражение, записанное в скобках. В данном случае цикл будет выполняться, пока функция `scanf` возвращает значение 2, то есть пока считываются оба числа `b` и `c`. Круглые скобки здесь — часть оператора цикла **`while`**, а не выражения условия цикла. Круглые скобки не могут быть пропущены.

Телом цикла **`while`** может быть любой оператор и, в частности, составной оператор, как в разбираемой программе.

1.1.9 Операции сравнения и логические операции

В языке **Си** определены обычные операции сравнения чисел, которые записываются следующим образом:

`==` сравнение двух чисел на равенство.
`!=` сравнение на неравенство.
`>=` «больше или равно».
`>` «больше».
`<=` «меньше или равно».
`<` «меньше».

Обратите внимание, что сравнение на равенство записывается как два знака равенства `==`, а один знак равенства `=` — это операция присваивания!

Подробнее то, как вычисляются выражения, мы рассмотрим подробнее на следующих занятиях.

Для проверки нескольких условий используются логические операции-связки `||` и `&&`. Обратите внимание, что оба знака операции состоят из двух символов. Есть и операции `|` и `&`, это совсем другие операции, мы их рассмотрим позднее.

Операция `||` — логическое «или». Она даёт истинное значение, если хотя бы один из аргументов даёт истинное значение. При этом операция вычисляется по «короткой» схеме, то есть если первый аргумент операции дал «истину», второй даже не вычисляется.

Операция `&&` — логическое «и». Она даёт значение «ложь», если хотя бы один из аргументов даёт значение «ложь». Как и предыдущая, эта операция вычисляется по «короткой» схеме. Если первый аргумент дал значение «ложь», второй аргумент даже не вычисляется.

Обратите внимание, что в языке **Си** отсутствует логический тип как таковой. Везде, где требуется логическое значение «ложь» или «истина», может использоваться любое целое выражение (и вообще, любое скалярное выражение). Значение 0 понимается, как «ложь», а любое значение, не равное 0, как «истина». Тем не менее, операции отношения и логические операции вырабатывают в качестве значения «истины» вполне определённое значение — 1.

В новом стандарте **C99** тип **`_Bool`** всё же введён. Но все правила, описанные выше, продолжают работать.

1.1.10 Условный оператор `if`

```
 8                    if (b <= 0 || c <= 0) {

10                    } else {

18                    }
```

Это — условный оператор. Обратите внимание, что *круглые скобки (и) являются частью условного оператора, и, соответственно, не могут быть опущены*. Специальное ключевое слово **`then`** отсутствует. И после условия, и после ключевого слова **`else`** могут находиться по одному оператору программы. Для наглядности мы используем составной оператор в обоих случаях, хотя в первой части оператора составной оператор необязателен. Как обычно, часть **`else`** может быть опущена.

Рассматривая нашу задачу вычисления НОД двух чисел, нужно заметить, что этот условный оператор выполняет очень важную функцию: проверяет входные данные на допустимость. Алгоритм вычисления НОД, использованный в нашей программе, работает корректно только при положительных значениях аргументов. С другой стороны, программа не может никак влиять на входные данные, которые пользователь будет задавать программе. Поэтому *при чтении данных от пользователя проверка корректности входных данных обязательна*! Это — одно из важнейших требований, которым должна удовлетворять надёжная программа.

Часть проверок на допустимость входных данных выполняет функция `scanf`. Она проверяет, что действительно на стандартном потоке ввода заданы два числа, каждое из которых представимо в типе **`int`**. Только в этом случае функция `scanf` вернёт результат 2.

1.1.11 Оператор `do while`

```
12                    do {

16                    } while (c != 0);
```

Это — оператор цикла с постусловием. Отличия от цикла **`repeat until`** языка **Паскаль** перечислены ниже:

1. Ключевые слова **`do`** и **`while`** не образуют блока. Поэтому, если в цикле необходимо записать несколько операторов, нужно использовать составной оператор.

2. Круглые скобки после ключевого слова **`while`** являются частью оператора цикла, а не выражения, и поэтому обязательны.

3. Цикл выполняется, пока условие, записанное после **`while`** истинно.

1.1.12 О расстановке «;»

В языке Си «точки с запятой» являются составной частью каждого оператора, кроме составного, поэтому должны обязательно ставится после них. После составного оператора точка с запятой никогда не должна ставиться!

1.1.13 Арифметические выражения и присваивания

```
13                              a = b;
14                              b = c;
15                              c = a % b;
```

В языке определены обычные арифметические операции:

- `+` сложение.
- `-` вычитание.
- `*` умножение.
- `/` деление. Если оба операнда операции — целые выражения, деление будет выполняться как деление нацело, результатом будет тоже целое значение. Если хотя бы один из операндов отрицателен, то результат операции целочисленного деления зависит от реализации. Если хотя бы один операнд — вещественное число, деление будет выполняться над вещественными числами и даст вещественный результат.
- `%` взятие остатка от деления. Применимо только к целым выражениям. Если хотя бы один из операндов отрицателен, то результат операции взятия остатка зависит от реализации.

При выполнении операций сложения и вычитания с целыми числами арифметическое переполнение или переносы не диагностируется. Программа продолжит работать как ни бывало. Но при выполнении операций умножения и деления, а также сложения и вычитания с вещественными числами, ошибки диагностируются и вызывают аварийное завершение программы.

Операция присваивания = даёт результат, равный присвоенному значению, поэтому допустимо использование нескольких присваиваний подряд, например:

```
a = b = c = 0;
```

1.1.14 Функция вывода данных

```
9                       printf("Invalid␣parameters\n");

17                      printf("%d\n", b);
```

Функция `printf` печатает значения на стандартный поток вывода (обычно это экран). Первый параметр функции — это строка формата печати. Строка формата может содержать обычные литеры, которые будут просто печататься, а может содержать спецификации формата печати, похожие на рассмотренные нами при разборе функции `scanf`. Простейшие из них перечислены ниже.

`%d` печать целого (**`int`**) значения со знаком.
`%u` печать беззнакового целого (**`unsigned int`**).
`%ld` печать длинного целого (**`long`**) со знаком.
`%lu` печать беззнакового длинного целого (**`unsigned long`**).
`%Ld` печать длинного целого (**`long long`**) со знаком.
`%Lu` печать беззнакового длинного целого (**`unsigned long long`**).
`%c` печать литеры.
`%f` печать вещественного числа (**`double`**).
`%Lf` печать вещественного числа (**`long double`**).

В форматной строке могут использоваться специальные последовательности символов: `\n` — переход на следующую строку, `\t` — символ табуляции (для равного отступа столбцов). Это примеры так называемых специальных последовательностей. Хотя в тексте программы они записываются двумя (или более) байтами, в работающей программе им соответствуют один байт — соответствующий управляющий символ в кодировке, с которой работает система. Чтобы напечатать знак процента, он повторяется дважды.

Знак взятия адреса `&` для всех пока рассмотренных нами типов не ставится! Аргументы печати должны быть точно того типа, который указан в спецификации формата (исключения — см. ниже). Они должны идти в том же порядке, в котором перечислены в строке формата. В противном случае ваша программа будет в лучшем случае печатать что-то странное, а в худшем — аварийно завершаться.

Для печати значений типа **`short`** или **`signed char`** нужно использовать спецификатор печати чисел типа **`int`**. Для печати значений типа **`unsigned short`** или **`unsigned char`** нужно использовать спецификатор печати чисел типа **`unsigned int`**. Для печати значений типа **`float`** используется спецификатор печати значений типа **`double`**.

1.1.15 Возврат значения из функции

```
20          return 0;
```

Поскольку функция `main` объявлена как возвращающая целое значение, необходим оператор, который определит возвращаемое значение. Оператор **`return`** вызывает завершение работы функции и возврат значения, указанного в операторе.

Относительно самого возвращаемого значения пока заметим, что функция `main` в обычных случаях должна возвращать значение 0.

1.2 Вторая программа

Рассмотрим следующую задачу: *удалить из входного потока все пробельные литеры*.

```
1 #include <stdio.h>
2 int main()
3 {
4         char c;
5         while (scanf("%c", &c) == 1) {
6                 if (c != '␣') printf("%c", c);
7         }
8         return 0;
9 }
```

1.2.1 Символьные константы

Замечание: слово «символ» — очень сильно перегружено разными смыслами, поэтому иногда употребляют слово «литера».

Литерные константы записываются в апострофах (одинарных кавычках). Например, `'a'` — литера `a`. Кроме того, для специальных символов применяется следующая запись: `'\''` — литера «апостроф», `'\\'` — литера «обратная косая черта» (backslash), `'\n'` — литера перехода на новую строку, `'\t'` — литера табуляции.

Обратите внимание, что литерные константы имеют тип **`int`**, а не **`char`**.

1.2.2 Признак конца файла

В задачах, которые мы с Вами будем рассматривать, программа, как правило, должна в цикле считывать входные данные, обрабатывать их и выводить результат. Как правило, чтение ведётся не до получения специального значения, а до наступления *конца файла*. После того, как при чтении из файла был достигнут конец файла, данных в нём не осталось, и дальнейшие операции чтения к данному файлу неприменимы и их использовать нельзя.

Когда программа считывает данные с клавиатуры (терминала), пользователь должен сам указать, в какой момент ввод данных программы он считает завершённым. Для этого нажимается специальная комбинация клавиш (Ctrl-Z в Windows, Ctrl-D в UNIX-системах). Эта комбинация клавиш обычно не появляется в файле в виде символа, но приводит к тому, что устанавливается флаг конца файла и все функции ввода данных (в нашем случае пока только `scanf`) будут возвращать специальное значение -1 (`EOF`). Если функция `scanf` возвращает значение `EOF`, значение всех считывамых в данном вызове `scanf` переменных не определено. Таким образом, *признак конца файла — это специальное значение, возвращаемое функциями чтения из файла, а не особое значение в самом файле.*