Лекция 7. Символьный ввод-вывод и проверки достоверности. Функции.

Ревун Артем Леонидович
ст. преп. кафедры вычислительных систем

Курс «Программирование», осенний семестр, 2024
Сибирский государственный университет телекоммуникаций и информатики (г. Новосибирск)

Содержание раздела

- Рассмотрим различия между буферизированным и не буферизированным вводом
- Понятие условия конца файла и возможность его моделирования с клавиатуры
- Использование перенаправления потов вывода и подключения программы к файлам
- Создание более “дружественных” пользовательских интерфейсов

Эхо-программы (Echo-program)

- Эхо-программы встречаются в различных направлениях разработки. Задача демонстрировать обратную связь в зависимости от действия пользователя, которые были обработаны программой.
- Иногда эхо-программы просто демонстрируют работоспособность приложений. (*Подразумевается что их базовая настройка выполнена верно*).
- В случае разработки программ человеко-машинного взаимодействия, часто приходится писать эхо-программы позволяющие программисту понять действия программы в ответ на действия пользователя. (UX-разработка)

Эхо-программы (Echo-program)

```
#include <stdio.h>
int main(void)
{
        char ch;
        while ( (ch = getchar()) != '#')
        putchar(ch);
    return 0;
}
```

```
Планируете ли вы сдать экзамен по программированию? [Enter]
Планируете ли вы сдать экзамен по программированию?
Нет # у меня автомат.[Enter]
Нет
```

Буферизированный ввод (продолжение)

```
$ ./a.out
Планируете[Enter]
```

Буфер										
	П									
	л	П								
	А	л	П							
	н	а	л	П						
	и	н	а	л	П					
	…									
	\n	е	т	е	у	р	и	н	а	л	П

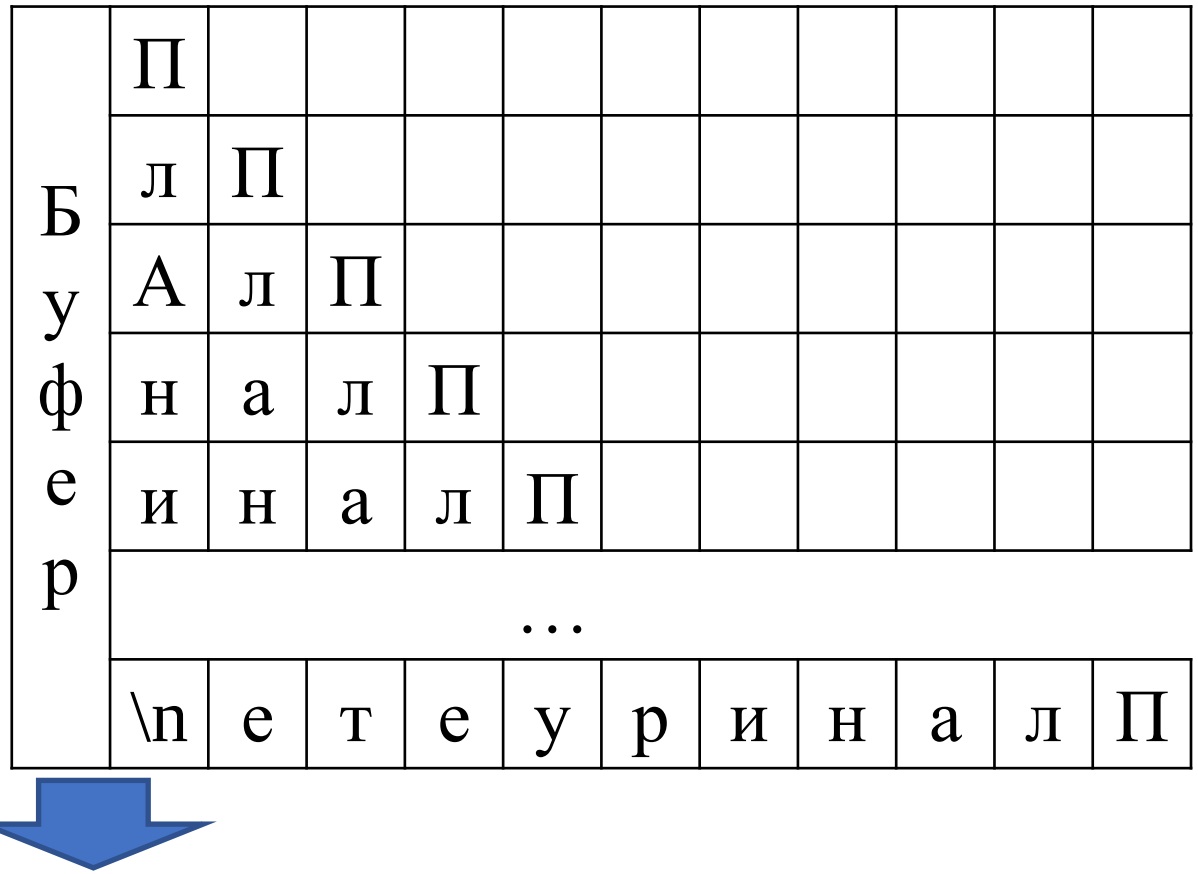
Преимущества:

- Передача нескольких значений ввода в одном *буфере* быстрее чем отправка значений по одному.
- До нажатия Enter можно править значения передаваемые в *буфер*.

Содержимое буфера передаётся программе после нажатия Enter

Буферизированный ввод

```
$ ./a.out
Планируете[Enter]
```



Проиллюстрирован *буферизированный ввод*, когда вводимые символы накапливаются и хранятся во временной памяти, которую называют *буфером*.

Содержимое буфера передаётся программе после нажатия Enter

Небуферизированный ввод

```
$ ./a.out
ППллааннииррууееттее
```

Проиллюстрирован *небуферизированный ввод*, когда вводимые символы немедленно отправляются на обработку программы.

Преимущества: нажатие клавиши мгновенно обрабатывается программой, что может быть использовании при разработке интерактивных приложений или компьютерных игр.

Виды буферизированного ввода

Существуют два вида буферизации ввода-вывода - *полностью буферизированный* и *построчно буферизированный*.

При *полностью буферизированном* вводе-выводе буфер отправляется в программу после достижения состояния переполнения. Размер буфера зависит от системы и меняется в зависимости от запуска разрабатываемого приложения*.

При *построчно буферизированном* вводе-выводе буфер сбрасывается всякий раз, когда появляется символ новой строки ('\n')

Ограничения ANSI

ANSI Си построчно буферизированный ввод считается стандартом.

Реализация не буферизированного и буферизированного ввода зависит от как *hardware* так и *software* составляющей.

В Unix есть ряд функций, которые входят в состав библиотеки Unix, но не является частью стандарта Си, которые позволяют изменить стандартное поведение функции getchar(). Однако, имеется множество “подводных камней”.

Файлы с точки зрения Си

Файл – именованная область долговременной памяти, где хранится информация.

В Си много библиотечных функций предназначенных для открытия, чтения, записи и закрытия файлов.

В Си работа с файлами доступна с использованием *стандартного пакета ввода-вывода*, является унифицированным интерфейсом поддерживающим специфику различных систем.

Однако есть и возможность работать с файлами на *низкоуровневом вводе-выводе*, при этом придётся знать специфику системы и учитывать все ограничения и особенности.

Отличия систем при работе с файлами

- Некоторые системы хранят содержимое файла в одном месте, а информацию о нем — в другом.
- Одни системы встраивают описание файла в сам файл, а другие нет.
- При работе с текстами многие системы для обозначения конца строки применяют одиночный символ новой строки ('\n’).
- Другие могут использовать для этого комбинацию символов возврата каретки (CR) и перевода строки (LF) (‘\r\n’).
- Некоторые системы измеряют размер файлов до ближайшего байта, а другие — в блоках байтов.ы

Работа с файлами в Си (Поток ≈ Файл)

Концептуально работая с файлами в Си, это работа с *потоком*. Работа с *потоком* это абстракция, т.к. в Си все особенности работы с файловой системы нивелируются за счёт *унификации* операций. С точки зрения Си нет разницы:

- *ввод* информации в файл = *ввод* информации с клавиатуры
- *считывание* информации из файла = *вывод* информации в консоль

В следствии этого, при *клавиатурном* вводе можно использовать те же приемы, что и при работе с *файлами* и наоборот. Программе читающей файл, необходим способ обнаружения *конца файла*, поэтому все функции в Си оснащены этим механизмом, в том числе и функции *клавиатурного* ввода-вывода.

Свойство обнаружения конца файла

1. Предусматривается помещение в конец файла специального символа ctrl+d (чаще в производных Unix) или ctrl+z (чаще в производных MS-DOS). Но в современных системах и вовсе такой символ может не ставится!!!
2. Операционная система оценивает размер файла и знает его размер и сама оценивает процесс чтения файла, если достигнут конец файла, система сама принимает решение что конец файла достигнут.

Для этого состояния было назначено имя EOF ("end of file") – конец файла. EOF = -1 и описан в в файле stdio.h

Пример программы с EOF

```
#include <stdio.h>
int main(void)
{
        char ch;
        while ( (ch = getchar()) != EOF)
        putchar(ch);
    return 0;
}
Планируете ли вы сдать экзамен по программированию? [Enter]
Планируете ли вы сдать экзамен по программированию?
Нет # у меня автомат.[Enter]
Нет # у меня автомат.
^D (Linux Ctrl+D) или ^Z (Windows Ctrl+Z)
```

Работа с буферизированным выводом

```
    int guess = 1;
    printf("Выберите целое число в интервале от 1 до 100. Я
попробую угадать его");
    printf("\nНажмите клавишу 'y' если моя догадка верна
и");
    printf("\nклавишу 'n' в противном случае.\n");
    printf("Вашим числом является %d?\n", guess);
    while (getchar() != 'д')
        printf("Ладно, тогда это %d?\n", ++guess);
    printf("Я знал что у меня получится!\n");
    return 0;
```

Результат работы программы

```
PS.\'main.exe'
Выберите целое число в интервале от 1 до 100. Я попробую
угадать его. Нажмите клавишу 'y' если моя догадка верна и
клавишу 'n' в противном случае. Вашим числом является 1?
n
Ладно, тогда это 2?
Ладно, тогда это 3?
n
Ладно, тогда это 4?
Ладно, тогда это 5?
n
Ладно, тогда это 6?
Ладно, тогда это 7?
```

Отбрасывание остатка входного потока

```
    while (getchar() != 'д')
    {
        printf("Ладно, тогда это %d?\n", ++guess);
        while (getchar() != '\n')
            continue;
    }
Выберите целое число в интервале от 1 до 100. Я попробую
угадать его. Нажмите клавишу 'y' если моя догадка верна и
клавишу 'n' в противном случае. Вашим числом является 1?
net
Ладно, тогда это 2?
```

Работа с буферизированным выводом

```
    int guess = 1;
    printf("Выберите целое число в интервале от 1 до 100.
Я попробую угадать его");
    printf("\nНажмите клавишу 'y' если моя догадка верна
и");
    printf("\nклавишу 'n' в противном случае.\n");
    printf("Вашим числом является %d?\n", guess);
    while (getchar() != 'д'){
        printf("Ладно, тогда это %d?\n", ++guess);
       while (getchar() != '\n') continue;
    }
    printf("Я знал что у меня получится!\n");
    return 0;
```

А ещё ошибки были?

Ошибка проверки корректности значений

```
.\'main.exe'
Выберите целое число в интервале от 1 до 100. Я попробую
угадать его
Нажмите клавишу 'y' если моя догадка верна и клавишу 'n' в
противном случае. Вашим числом является 1?
no
Ладно, тогда это 2?
net
Ладно, тогда это 3?
naverno
Ладно, тогда это 4?
```

Решение ошибки корректности значений

```
while (response = getchar() != 'y')
{
    if(response == 'n')
        printf("Ладно, тогда это %d?\n", + + guess);
    else
        printf("Принимаются только варианты y или n.\n");
    while (getchar() != '\n')
        continue;
}
```

А ещё ошибки остались?

Ошибка проверки корректности значений

```
.\'main.exe'
Выберите целое число в интервале от 1 до 100. Я попробую
угадать его
Нажмите клавишу 'y' если моя догадка верна и клавишу 'n' в
противном случае. Вашим числом является 1?
yarmarka
Я знал что у меня получится!
```

Лекция 7. Символьный ввод-вывод и проверки достоверности. Функции.

Ревун Артем Леонидович
ст. преп. кафедры вычислительных систем

Курс «Программирование», осенний семестр, 2024
Сибирский государственный университет телекоммуникаций и информатики (г. Новосибирск)

Содержание раздела

- Ключевые слова: return
- Операции: * (унарная), & (унарная)
- Функции и их определение
- Использование аргументов и возвращаемых значений
- Использование переменных-указателей в качестве аргументов функций
- Типы функций
- Прототипы ANSI C
- Рекурсия

Функция:определение и концепция

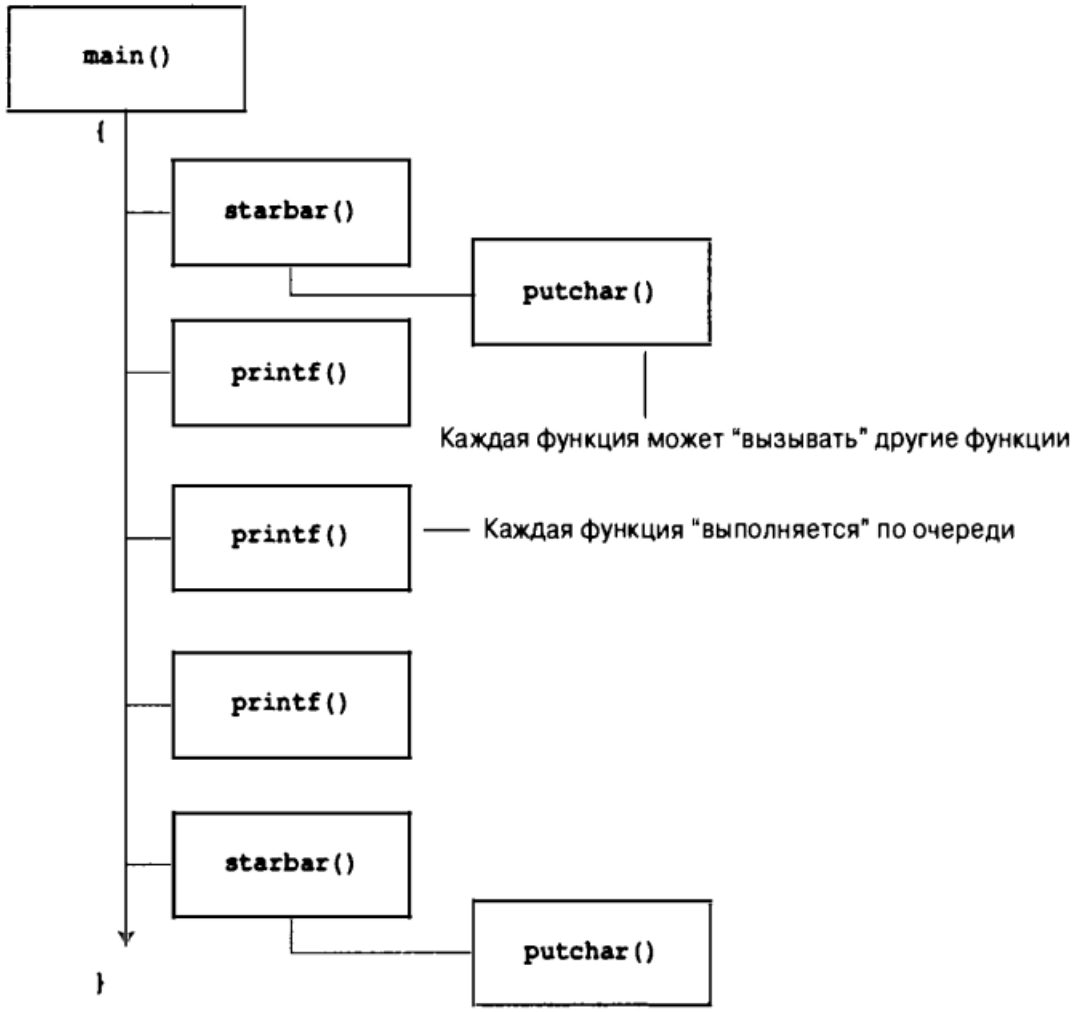
Функция – это самодостаточная единица в коде программы, созданная для выполнения отдельной задачи.
Основная цель функции – это *декомпозиция*.
Декомпозиция – процесс разделения целого (системы/задачи/программы), на отдельные части. Каждая часть выполняет свою (микро) задачу и в совокупности они образуют систему.
Преимущества использования функций:

- Упрощение разработки
- Упрощение тестирования
- Гибкость при внесении изменений в отдельные функции
- Возможность повторно использовать одну и ту же функцию в различных других программах/системах/задачах

Пример функции

```
#include <stdio.h>
#define WIDTH 40
void starbar(void) ; /* прототип функции */
int main (void){
    starbar();printf();printf();printf();starbar();  /*
использование функциий starbar  */
    return 0;
}
void starbar(void){ /* определение функции */
    int count;
    for (count = 1; count <= WIDTH; count++)
        putchar('*');
    putchar('\n');
}
```

Поток выполнения операций



Понятие черного ящика

При использовании сторонних функций программисты часто предпочитают не задумывается о том, как работает та или иная функция, такой подход, называется *чёрным ящиком*.

Нас не интересует что внутри *чёрного ящика*, если мы не занимаемся разработкой этой функции.

Для использования функции достаточно знать ее входные аргументы и возвращаемый ею результат (если он есть) и/или влияние функции на программу в целом.

Функция, процедура и сигнатура

Функция отличается от ***процедуры***, только тем, что всегда возвращает значение отличное от void (пустоты).

Возвращаемое значение ***функции*** указывается перед названием функции когда происходит ее определение в коде.

Если ***функция*** не возвращает значение, её называют ***процедурой***.

int main(void) – предполагает, что в случае выполнения ***функции*** main, возвращается значение типа int. Само по себе int здесь и является сигнатурой ***функции***.

При компилировании важно, чтобы сигнатуры выходных и выходных значений функции соответствовали корректности при дальнейшей обработки этого выходного значения.

Понятие сигнатуры (signature)

```c
#include <stdio.h>
int max(int a, int b);
int main (void){
    printf("%d",max(10, max(5,15)));
    return 0;
}
int max(int a, int b){
    return (a>b)?a:b;
}
//max(5,15) -> int(15) -> max(10,15) -> int(15)
//Своего рода сигнатура это байтовое и битовое представление данных.
```

Синтаксис определения функций

```
int max(int a, int b); //допустимо
int max2v(int,int);    //допустимо
int max3v();не допустимо, не определена реализация
...
int max(int a, int b){//допустимо
    return (a>b)?a:b;
}
int max2v(int x,int y){//допустимо, но объявить имена переменных
    return (x>y)?x:y;
}
int max3v(x,y){//не допустимо, но раньше работало
    int x,y;
    return (x>y)?x:y;
}
```

Оператор return

Оператор **return** – останавливает выполнение любых операций, переводя выполнение программы к точке из которой была вызвана функция, а также возвращает лишь одно описанное при определении функции значение указанного типа данных.

```
int imin(int n, int m ) {
if (n < m)
    return n;
else
    return m;
printf("Оператор, который никогда не  выполнится.\n");
}
```

Прототипы функций

Концептуально *прототип* функции описывает возвращаемое значение, количество аргументов и типы аргументов.

Использование *прототипов* позволяет повысить читаемость кода, а также перейти к созданию программ из нескольких файлов.

Аргументы могут быть не объявлены в *прототипе* (void)
Допустимы также прототипы с неизвестным количеством аргументов **int printf(const char *, …);**

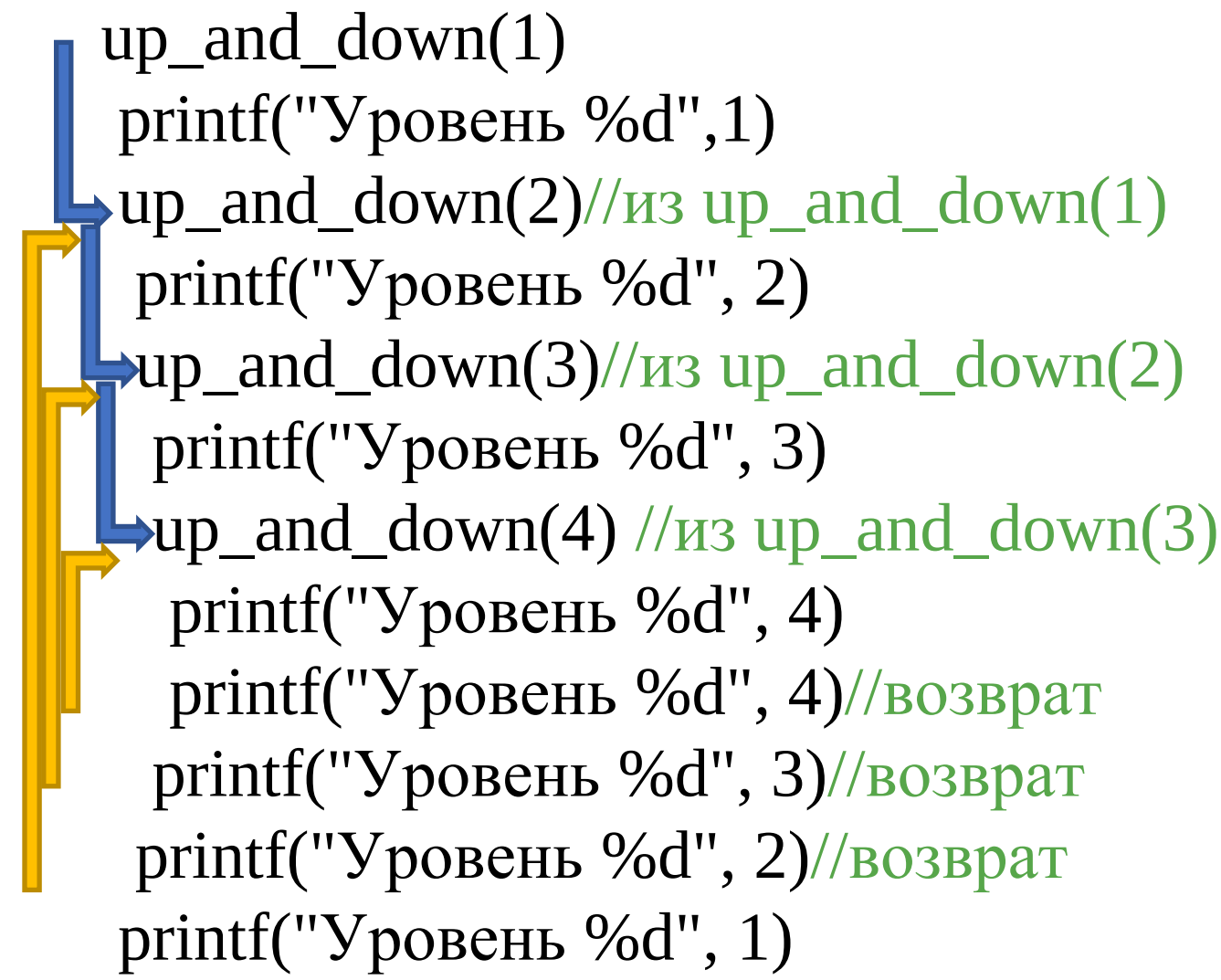
Рекурсия

В языке Си функция может вызывать сама себя, этот процесс и называется рекурсией.

Сложность рекурсии в том, что не всегда явно описано условие окончание рекурсии, что приводит к ее не завершению. Однако иногда использование рекурсии бывает крайне **удобным**.

Рекурсия пример и последовательность

```c
#include <stdio.h>
void up_and_down(int);
int main(void){
up_and_down(1);
return 0;
}
void up_and_down(int n){
printf("Уровень %d\n", n);
if (n < 4)
    up_and_down(n+1);
printf("Уровень %d\n",n);
```



Рекурсия пример и сравнение с циклом

```
#define ull unsigned long long  //удобное сокращение
ull fact (ull n) ;//прототипы на цикле
ull rfact(ull n);//прототип рекурсивной
ull fact(ull n) {// функция, основанная на цикле
    ull  ans;
    for (ans =1; n > 1; n--)
      ans *= n;
    return ans;
}
ull rfact(ull n){// рекурсивная версия
    ull ans;
    if (n > 0) ans= n * rfact(n-1);
    else ans = 1;
    return ans;
}
```

```
Эта программа вычисляет факториалы.
Введите значение в диапазоне 0-20 (q для завершения):
20
цикл: факториал 20 = 2432902008176640000        0.000000 заняло
рекурсия: факториал 20 = 2432902008176640000    0.010000 заняло
```

Рекурсия преимущества и недостатки

Преимущества:

- Простота при решении ряда задач при программировании.

Недостатки:

- Использование приводит к потери ресурсов.
- Сложно документировать и поддерживать.
- Легко достигает экспоненциальной сложности при двойной рекурсии.

Рекурсия: область видимости

```
unsigned long Fibonacci(unsigned n);
...
Fibonacci(5);
unsigned long Fibonacci(unsigned n) {
if (n > 2)
    return Fibonacci(n-1) + Fibonacci(n-2);
else
    return 1;
}
```

https://recursion.vercel.app/

Локальные и глобальные переменные

```
unsigned long r = Fibonacci(40);
Fibonacci(40)->Fibonacci(39)+ Fibonacci(38) 16byte
Fibonacci(39)->Fibonacci(38)+ Fibonacci(37) 16byte
Fibonacci(38)->Fibonacci(37)+ Fibonacci(36) 16byte
Fibonacci(37)->Fibonacci(36)+ Fibonacci(35) 16byte
…
Fibonacci(3)->Fibonacci(2)+ Fibonacci(1) 16byte
+ на каждую вызванную функцию по 4byte на
переменную unsigned n.
```

Локальные и глобальные переменные

```c
#include <stdio.h>
int a = 5, b = 10; //глобальная переменная a,b
int imax(int, int); //прототип imax
int main(){
  printf("%d %d\n", a,b);
  int a=0, b=0; //локальная переменная a,b функции main
  scanf("%d %d", &a, &b);printf("%d %d\n", a,b);
  printf("%d", imax(a,b));printf("%d %d\n", a,b);return 0;
}
int imax(int a, int b){
     //локальная переменная a,b функции imax
    printf("%d %d\n", a,b);
    return (a>b)?a:b;}
```

```
5 10
100 200
100 200
100 200
200
100 200
```

Использование заголовочных файлов (1)

Одним из вариантов декомпозиции кода, является создание много файловой программы.
Для реализации программ и подпрограмм используются файлы с форматом расширения .c
Для прототипов и констант, обычно используются файлы с форматом расширения .h

```c
//info.h
#define NAME "GIGATHINK, INC."
#define ADDRESS "101 Megabuck Plaza"
#define PLACE "Megapolis, CA 94904"
#define WIDTH 40
#define SPACE ' '
void show_n_char (char ch, int nurn);
```

```c
//info.c
#include <stdio.h>
void show_n_char (char ch, int nurn)
{
    int count;
    for (count = 1; count <= nurn; count++)
    putchar(ch);
}
```

Использование заголовочных файлов (2)

```c
#include <stdio.h>
#include <string.h>
#include "info.h"
int main(void)
{
    int spaces;
    show_n_char('*', WIDTH); putchar('\n');
    show_n_char(SPACE, 12); printf ("%s\n", NAME);
    spaces = (WIDTH - strlen(ADDRESS) ) / 2;
    show_n_char (SPACE, spaces); printf ("%s\n", ADDRESS);
    show_n_char(SPACE, (WIDTH - strlen(PLACE) ) /2);
    printf ("%s\n", PLACE);
    show_n_char('*', WIDTH); putchar('\n');
    return 0;
}
```

Использование заголовочных файлов (3)

Заголовочные файлы достаточно подключить с использованием команды препроцессора #include "название_заголов_файла.c“

Для компиляции в много файловой программы требуется при компиляции указать, все файлы в формате .c
gcc main.c info.c – o main

Для удобства компиляции можно (~~нужно~~) использовать утилиту make или bash-скрипы.

Операция извлечения адреса (& унарный)

Оператор унарный & (амперсант) предоставляет *адрес* по которому хранится значение переменной. По сути своей *адрес* – это расположение данных в оперативной памяти компьютера. В большинстве систем *адреса* в памяти представлены в виде шестнадцатеричных чисел (*hexadecimal numbers*).

```
int number = 20;
printf("Значение - %d, Адрес - %p", number, &number);
```

./main

```
Значение - 20, Адрес - 00000018B0DFFD8C
```

Понимание операции получения адреса, является одной из наиважнейших концепций языка Си.

Доработка с извлечением адреса (&)

```
global 5 - 00007FF602D03000, 10 - 00007FF602D03004
5 10
local 5 - 00000044B67FFBFC, 10 - 00000044B67FFBF8
imax 5 - 00000044B67FFBC0, 10 - 00000044B67FFBC8
after imax local 5 - 00000044B67FFBFC, 10 - 00000044B67FFBF8
```

```c
#include <stdio.h>
int a = 5, b = 10;
int imax(int a, int b);
int main(){
    printf("global %d - %p, %d - %p \n", a,&a,b,&b);
    int a=0, b=0;
    scanf("%d %d", &a, &b);
    printf("local %d - %p, %d - %p \n", a,&a,b,&b);
    imax(a,b);
    printf("after imax local %d - %p, %d - %p \n", a,&a,b,&b);
    return 0;
}
int imax(int a, int b){
    printf("imax %d - %p, %d - %p \n", a,&a,b,&b);
    return (a>b)?a:b;
}
```

Задача изменения переменных (местами)

Кажется, что задача обмена переменных значением проста.

В языке программирования Python a,b = b,a; ~~Да, но как работает?~~

a = 5, b = 10 => a,b = b,a => new(a=10, b=5)

В Си нет возможности изменить значения переменных без использования буфера однотипного изменяемым значениям.

```
int a=5,b=10; //значения которые нужно поменять
int temp=0; //буфер однотипен по данным
temp = a;
a = b;
b =temp;
```

Понадобится при реализации алгоритмов замены, сортировки.

Проблема return и решение проблемы

~~`return a,b;`~~

Оператор **return** может возвращать только одно значение. Этого мало? Язык Си считает, что этого достаточно, т.к. если возвращать "правильное" значение, задача решена.

Во многом эту проблему решают указатели. По сути своей **указатель** это переменная, которая не может хранить в себе значение, а может лишь хранить адрес который указывает на объект данных, поэтому и называется **указателем**.

Оператор разыменовывания (* унарный)

Получать значение адреса, мы уже научились (&).

Оператор разыменовывания – позволяет выяснить значение, которое хранится по указанному адресу, иначе операцию называют операцией *снятия косвенности*.

```
int a=5;
int *ptrA=&a;
int valA = *ptrA;
(*ptrA = &a) , (valA = *ptrA) == (valA = a)
```

Имя переменной	*a*	*ptrA*	*valA*
Адрес	00007FF602D03004	00000044B67FFBF8	00000044B67FFBC8
Значение	5	00007FF602D03004	5

Указатели (pointers)

Чтобы объявить указатель, после типа переменной указывается оператор * и имя для переменной. Учитывая все правила именования переменных в Си по стандарту ANSI.

```
int * q;
char *ch;
float *p, * v;
```

Созданы указатели на int, char и float.

Каждый указатель занимает столько же byte сколько и тип данных на который он указывает.

Есть операции которые можно выполнять только с указателями, а есть те, которые только с переменными.

Указатели (pointers): новые возможности (1)

Указатели позволяют получать от функции столько значений сколько нам нужно, без использования оператора **return**.

Если мы знаем адрес по которому хранится некоторый объект данных, мы можем передать этот адрес в функцию в качестве аргументов. Разыменовывание адреса, позволяет нам получить значение объекта данных.

Также это позволяет изменять объект данных даже если мы и не знаем имя объекта данных.

Указатели (pointers): новые возможности (2)

```c
#include <stdio.h>
void upper10(int * a, int * b);
int main(){int a=5, b=10;
    printf("before local %d - %p, %d - %p \n", a,&a,b,&b);
    upper10(&a, &b);
    printf("after local %d - %p, %d - %p \n", a,&a,b,&b);
    return 0;}
void upper10(int * a, int * b){
    printf("upper10 %d - %p, %d - %p \n", *a,a,*b,b);
    *a=(*a)+10;    *b=(*b)+10;}
```

```
before local 5 - 000000047233FFC6C, 10 - 000000047233FFC68
upper10 5 - 000000047233FFC6C, 10 - 000000047233FFC68
after local 15 - 000000047233FFC6C, 20 - 000000047233FFC68
```

Указатели (pointers): новые возможности (3)

```
before local 5 - 000000047233FFC6C, 10 - 000000047233FFC68
upper10 5 - 000000047233FFC6C, 10 - 000000047233FFC68
after local 15 - 000000047233FFC6C, 20 - 000000047233FFC68
```

Имя переменной	*a (main)*	*b (main)*
Адрес	000000047233FFC6C	000000047233FFC68
Значение	5	10

```
upper10(&a, &b);
```

Имя переменной	**a (upper10)*	**b (upper10)*
Адрес	000000F32CDFF700	000000F32CDFF708
Значение	000000047233FFC6C	000000047233FFC68

```
*a=(*a)+10;        *b=(*b)+10;
&a=5+10;           &b=10+10;
```

Задачи на практику

Читать:
Главу 8. Символьный ввод вывод и проверка достоверности.
Главу 9. Функции.
Изучить:

- *Перенаправление в Unix, Linux и командной строке Windows стр. 302*
- *Создание дружественного пользовательского интерфейса стр. 306*
- *Просмотр меню стр. 316*
- *Использование указателей для обмена данным и между функциями*

Задачи на практику

Смотреть вопросы для самоконтроля (322-323 стр и 364-365 стр)
ПР №7 Символьный ввод-вывод и проверки достоверности. Функции.
Выполнить 5 заданий по формуле:

```
id = номер_в_списке % 10 +1
count_z = 19;
z1 = id+(count_z%id)
z2 = ((id*3)%count_z)+1
z3 = ((id*3)+id)%count_z)+1
```

Задания выполнение не по формуле в зачёт не идут . Но это не мешает вам их выполнять.

Спасибо за внимание

Ревун Артем Леонидович
ст. преп. Кафедры вычислительных систем

Курс «Программирование», осенний семестр, 2024
Сибирский государственный университет телекоммуникаций и информатики (г. Новосибирск)