Структура на програма. Функции

Пламен Танов
Ненко Табаков
Мартин Вачовски
Технологично училище „Електронни системи“
Технически университет – София

версия 0.1

Въведение

- Функциите разделят големите задачи на по-малки (подход „разделяй и владей“) и дават възможност на програмиста да използва стар код, вместо всеки път да започва от нищото
- Когато ползваш дадена функция е необходимо само да знаеш какво прави тя, без да е необходимо да знаеш как точно го прави
- Една програма на **C** може да е съставена от повече от един сорс файла
- Съществуват правила за видимост на различните имена (променливи, типове и т.н.)

Функции

```
return_type function_name(arguments) {
   /* function body */
}
```

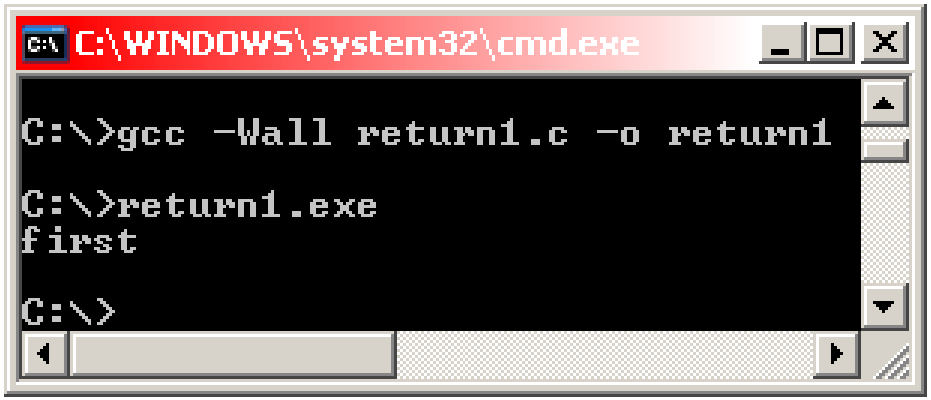
- Аргументите (тип и име), ако ги има, се изреждат разделени със запетайки
- Връщаната стойност може да бъде и void, т.е. функцията не връща стойност
- Стойност се връща чрез оператор **`return;`** При неговото срещане <u>се излиза незабавно</u> от функцията. Ако не се срещне **`return;`** върнатата стойност е случайна

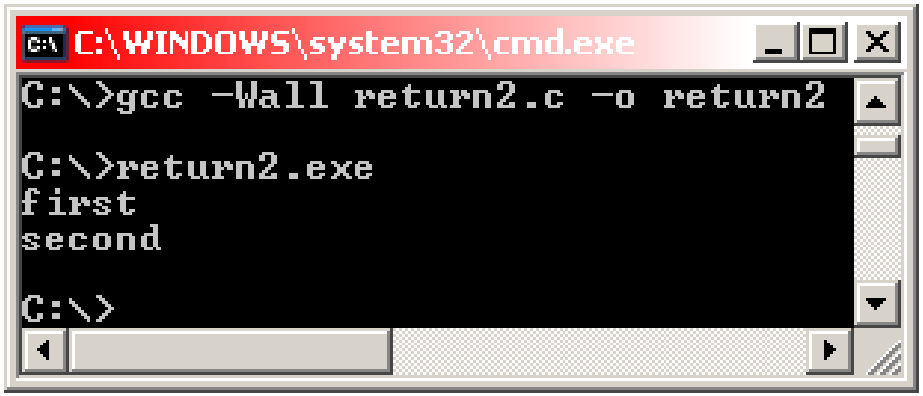
```
int abs(int a) {
   return (a>=0)?a:-a;
}
```

Пример

```
#include <stdio.h>
int main() {
  printf("first\n");
  return 0;
  printf("second\n");
}
```

```
#include <stdio.h>
int main() {
  printf("first\n");
  printf("second\n");
  return 0;
}
```





Пример$_1$

Програма, която извежда само редовете, в които се среща дадена дума („ould“).
Логическа структура:

```
while (има още един ред)
   if (думата се съдържа в реда)
      изведи го
```

```
getline(), strindex(), printf();
```

Пример$_2$

```
#include <stdio.h>
#define MAXLINE 1000 /* maximum input line length */

int getline(char line[], int max);
int strindex(char source[], char searchfor[]);
char pattern[] = "ould"; /* pattern to search for */

/* find all lines matching pattern */
int main() {
   char line[MAXLINE];
   int found = 0;
   while (getline(line, MAXLINE) > 0)
      if (strindex(line, pattern) >= 0) {
         printf("%s", line);
         found++;
      }
   return found;
}
```

Пример$_3$

```
/* getline: чете нов ред в s, връща дължината му */
int getline(char s[], int lim) {
   int c, i;
   i = 0;
   while (--lim > 0 && (c=getchar()) != EOF && c != '\n')
      s[i++] = c;

   if (c == '\n')
      s[i++] = c;

   s[i] = '\0'; //слага край на низа
   return i;
}
```

Пример$_4$

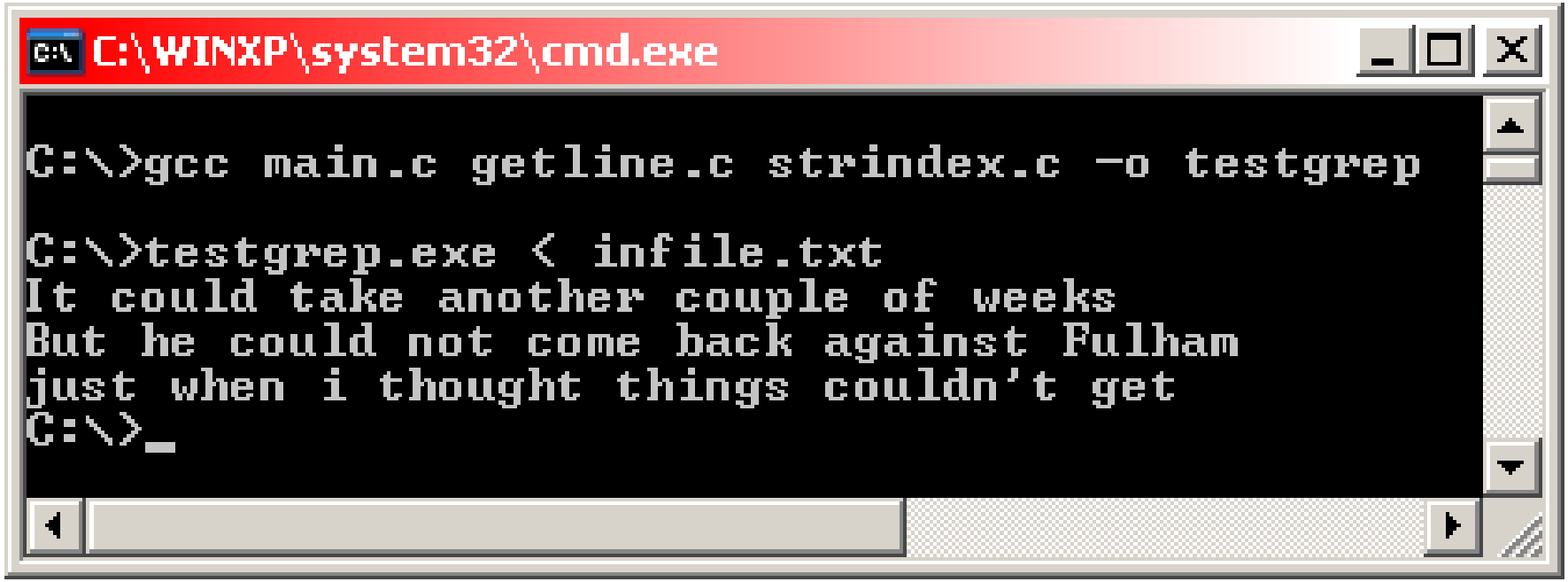
```
/* strindex: връща къде в s се среща t, -1 ако не се */
int strindex(char s[], char t[]) {
   int i, j, k;

   for (i = 0; s[i] != '\0'; i++) {

      //сравнява всеки един символ докато съвпадат
      for (j=i, k=0; t[k]!='\0' && s[j]==t[k]; j++, k++)
         ;

      if (k > 0 && t[k] == '\0')
         /* ако е достигнат края на t,
            явно всички символи съвпадат */
         return i;
   }

   //няма съвпадение
   return -1;
}
```

Програма, състояща се от повече от един файл

- По една програма може да работи повече от един човек без това да пречи на останалите
- Лесно различими, логически отделени части (по-лесна поддръжка на кода)
- По-лесно използване на стар код в нови програми
- По-бърза компилация (компилират се само тези файлове, които са променяни)
- По-добър стил

Пример

- Файлове: main.c, getline.c и strindex.c



Видимост на променливи и функции

- Една променлива може да се използва (да се вижда) от мястото на нейната декларация/дефиниция до края на блока (или файла), в който тя е декларирана/дефинирана
- За да ползваме променлива, която е дефинирана в друг файл ползваме **`extern`** декларация
- Една променлива може да скрие друга променлива със същото име, но дефинирана в блок (съставен оператор)

Пример

```
#include <stdio.h>
int somefunc();
int main() {
   int k = 5;

   //k се „вижда“, i и z не се „виждат“:
   printf(“k = %d\n“, k);
   //ГРЕШКА: printf(“i = %d, z = %d\n“, i, z);
   {
      int i = 0;
      //k и i се „виждат“, z не се „вижда“:
      printf(“k = %d, i = %d\n“, k, i);
      //ГРЕШКА: printf(“z = %d\n“, z);
   }
}

int z = 0; //z се „вижда“ от тук надолу ...
int somefunc() {
   printf(“z = %d\n“, z);
}
```

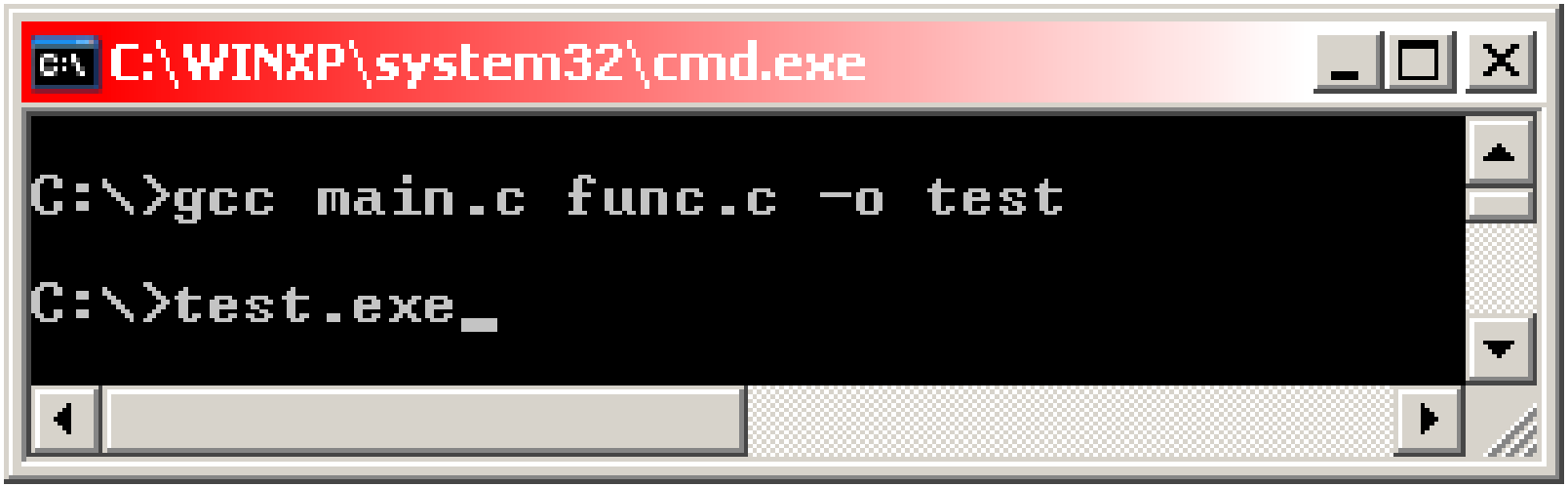
Пример

Файл 1 (main.c):

```
extern int i; //декларация

int main() {
 ++i;// i = 6 ...
}
```

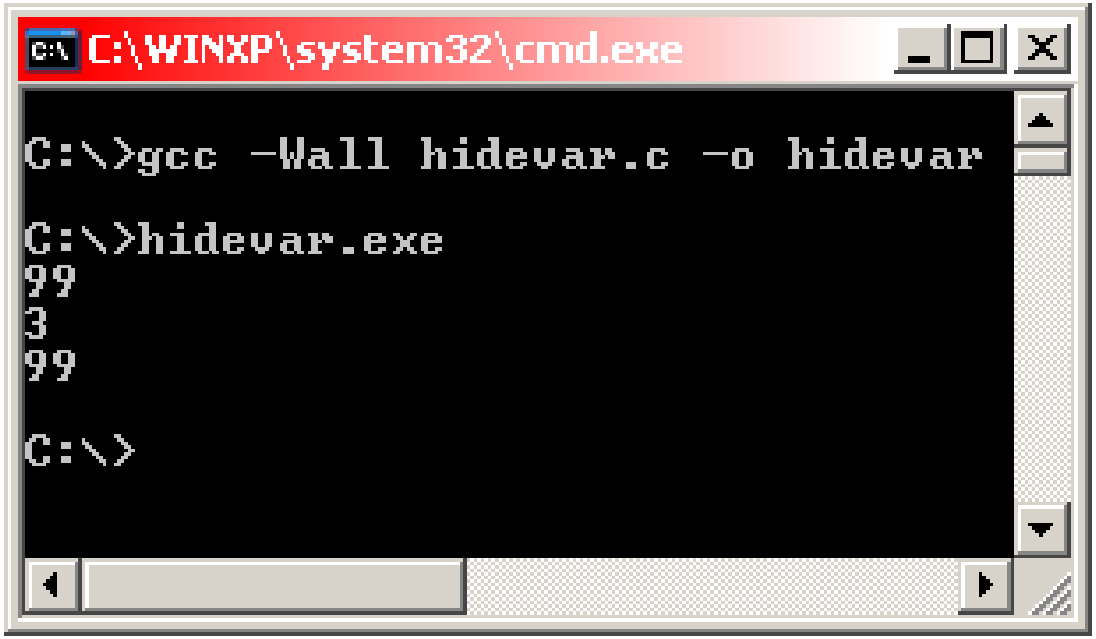
Файл 2 (func.c):

```
/* дефиниция, тук се заделя
място за променливата */

int i = 5;
```



Пример

```
#include <stdio.h>

int main () {
    int nc=99;
    printf ("%d\n", nc);

    {
        int nc = 3;
        printf("%d\n", nc);
    }

    printf("%d\n", nc);

    return 0;
}
```



Променливи

- Глобални (външни) – дефинирани извън функция
- Вътрешни – дефинирани в блок (функция) или аргумент на функция

```
int global=6;//глобална

int main () {
   int b=7;//вътрешна
...
   return 0;
}
```

Глобални променливи

- Една **C** програма може да съдържа „външни“ (глобални) обекти – променливи и функции.
- Дефинират се извън блок и могат да се ползват от много функции
- По подразбиране са достъпни и от функции компилирани отделно (в друг файл)
- Съществуват от стартирането на програмата до нейния край

Пример$_1$

Калкулатор, който използва обратна полска нотация, т.е. всеки оператор е след операндите:
`(1 – 2) * (4 + 5)` в обратна полска нотация:
`1 2 – 4 5 + *`

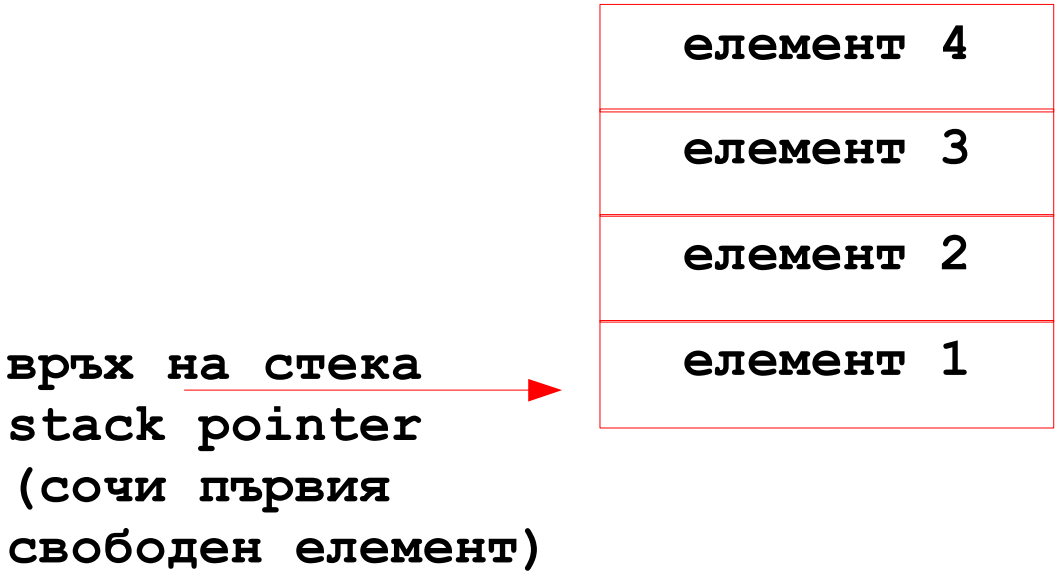
Това дава възможност да не се ползват скоби за определяне на приоритета при изчислението на израза

Стек$_1$

Калкулатор, който използва обратна полска нотация, т.е. всеки оператор е след операндите:

`(1 – 2) * (4 + 5)` в обратна полска нотация:

`1 2 - 4 5 + *`

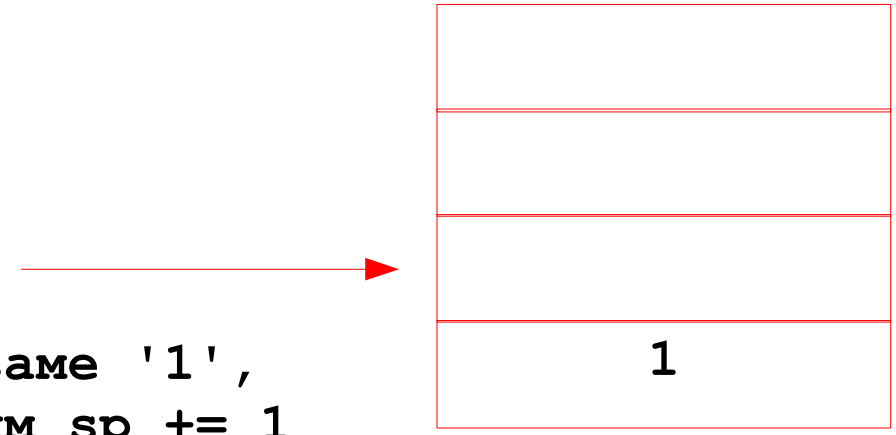


Стек$_2$

Калкулатор, който използва обратна полска нотация, т.е. всеки оператор е след операндите:

`(1 – 2) * (4 + 5)` в обратна полска нотация:

`1 2 - 4 5 + *`



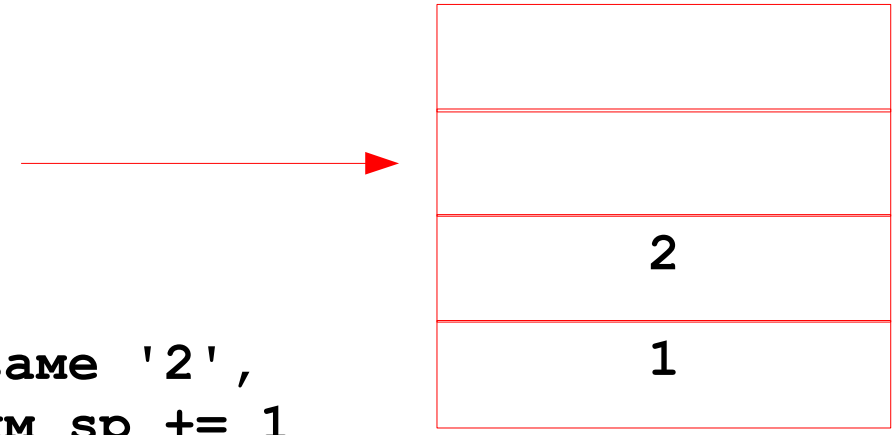
Вмъкваме '1',
местим sp += 1

Стек$_3$

Калкулатор, който използва обратна полска нотация, т.е. всеки оператор е след операндите:

`(1 – 2) * (4 + 5)` в обратна полска нотация:

`1 2 - 4 5 + *`



Вмъкваме '2',
местим sp += 1

Стек$_4$

Калкулатор, който използва обратна полска нотация, т.е. всеки оператор е след операндите:

`(1 – 2) * (4 + 5)` в обратна полска нотация:

`1 2 – 4 5 + *`

За да извършим операцията '-' трябва да извадим два елемента: местим sp -= 2

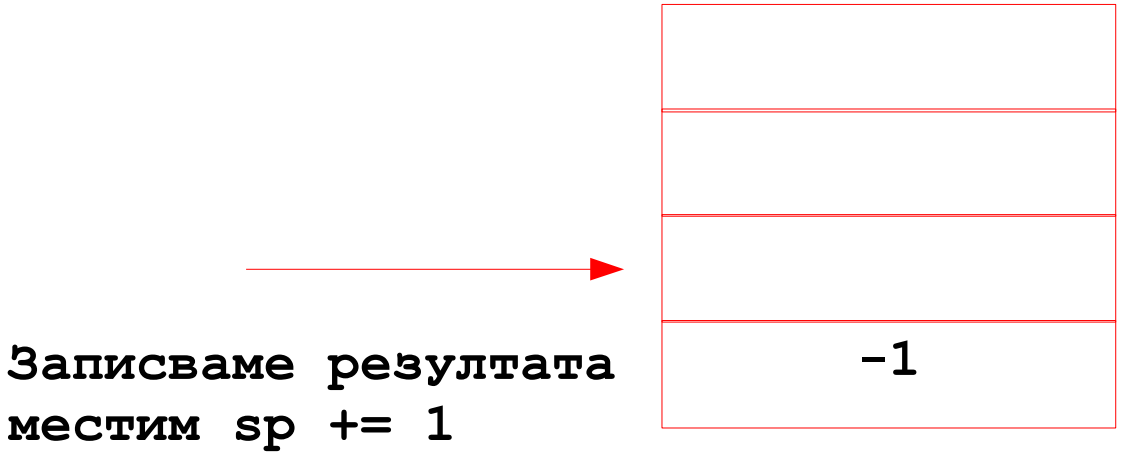
Елемент1: 1
Елемент2: 2
Елемент1 – Елемент2
1 – 2 = -1

Стек$_5$

Калкулатор, който използва обратна полска нотация, т.е. всеки оператор е след операндите:

`(1 – 2) * (4 + 5)` в обратна полска нотация:

`1 2 – 4 5 + *`

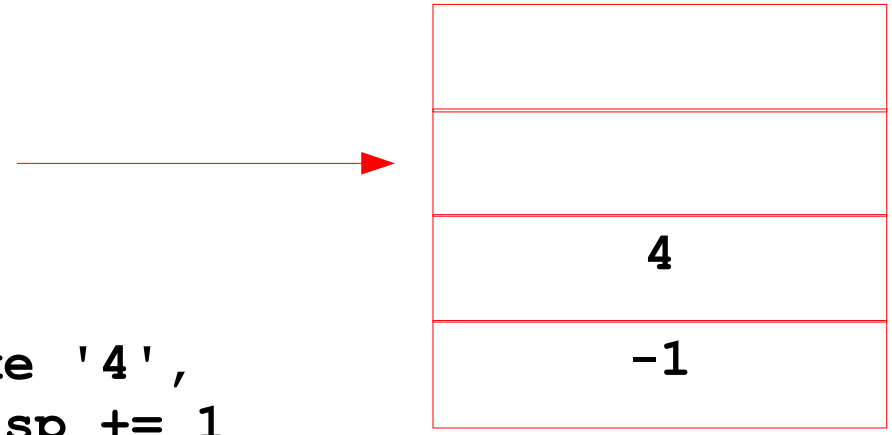


Стек$_6$

Калкулатор, който използва обратна полска нотация, т.е. всеки оператор е след операндите:

`(1 – 2) * (4 + 5)` в обратна полска нотация:

`1 2 – 4 5 + *`



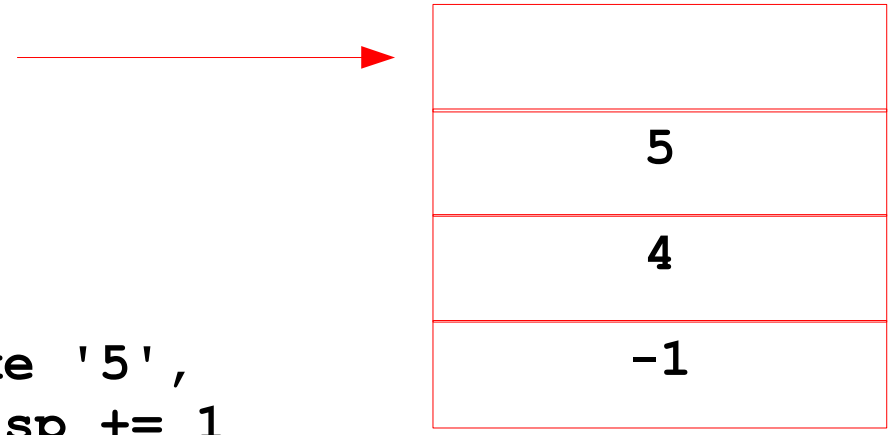
Вмъкваме '4',
местим sp += 1

Стек$_7$

Калкулатор, който използва обратна полска нотация, т.е. всеки оператор е след операндите:

`(1 – 2) * (4 + 5)` в обратна полска нотация:

`1 2 – 4 5 + *`



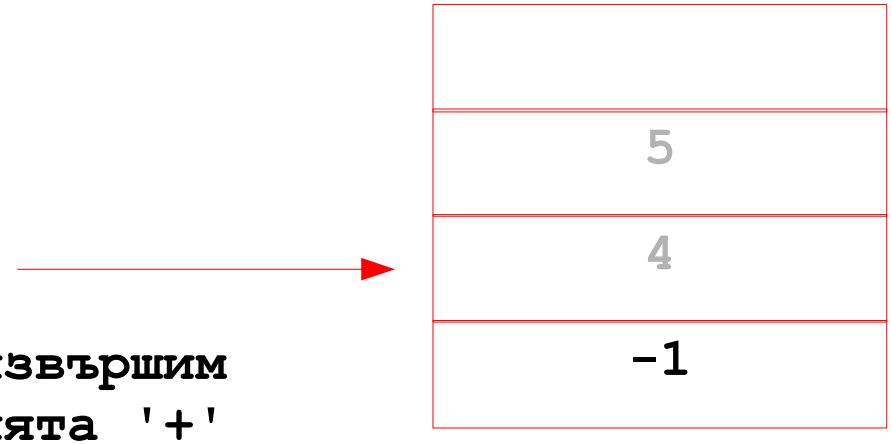
Вмъкваме '5',
местим sp += 1

Стек

Калкулатор, който използва обратна полска нотация, т.е. всеки оператор е след операндите:

(1 – 2) * (4 + 5) в обратна полска нотация:

1 2 - 4 5 + *



За да извършим операцията '+' трябва да извадим два елемента: местим sp -= 2

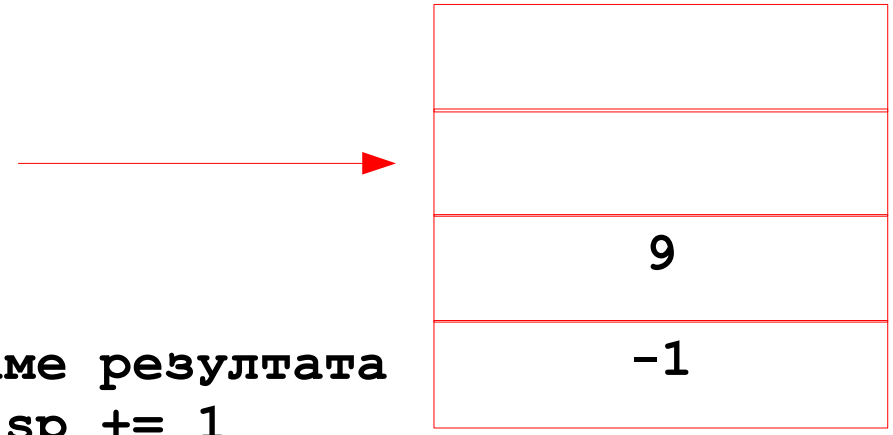
Елемент1: 4
Елемент2: 5
Елемент1 + Елемент2
4 + 5 = 9

Стек$_9$

Калкулатор, който използва обратна полска нотация, т.е. всеки оператор е след операндите:

`(1 – 2) * (4 + 5)` в обратна полска нотация:

`1 2 – 4 5 + *`



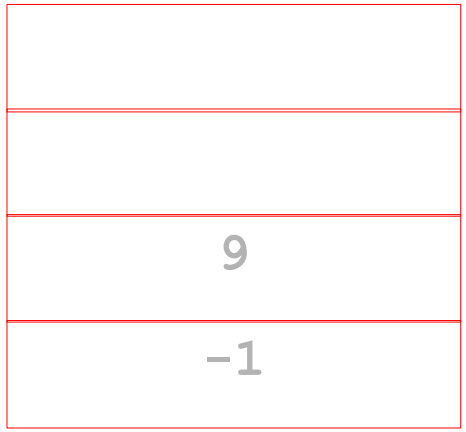
Записваме резултата местим sp += 1

Стек$_{10}$

Калкулатор, който използва обратна полска нотация, т.е. всеки оператор е след операндите:

`(1 – 2) * (4 + 5)` в обратна полска нотация:

`1 2 - 4 5 + *`



За да извършим операцията '*' трябва да извадим два елемента: местим sp -= 2

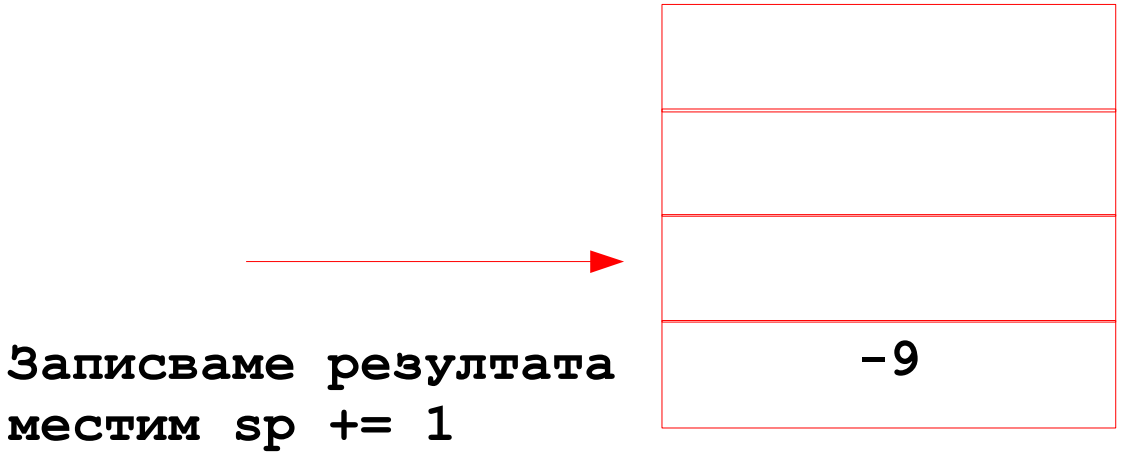
Елемент1: -1
Елемент2: 9
Елемент1 * Елемент2
-1 * 9 = -9

Стек

Калкулатор, който използва обратна полска нотация, т.е. всеки оператор е след операндите:

`(1 – 2) * (4 + 5)` в обратна полска нотация:

`1 2 - 4 5 + *`



Пример$_2$

Калкулатор, който използва обратна полска нотация, т.е. всеки оператор е след операндите:

(1 – 2) * (4 + 5) в обратна полска нотация:

1 2 – 4 5 + *

```
while (има нов оператор или операнд)
   if (число)
      добави го в стека (push)
   else if (операция) {
      прочети операндите от стека (pop)
      извърши операцията
      добави резултата в стека (push)
   } else if (нов ред)
      извеждаме и изтриваме върха на стека (pop)
   else
      грешка
```

Пример$_3$

```
#include <stdio.h>
#include <stdlib.h> /* for atof() */
#define MAXOP 100   /* max size of operand or operator */
#define NUMBER '0'  /* signal that a number was found */

int getop(char []);
void push(double);
double pop(void);

/* reverse Polish calculator */
int main() {
   int type;
   double op2;
   char s[MAXOP];
   while ((type = getop(s)) != EOF) {
      switch (type) {
         case NUMBER:
            push(atof(s));
            break;
```

Пример$_4$

```
        case '+':
           push(pop() + pop());
           break;
        case '*':
           push(pop() * pop());
           break;
        case '-':
           op2 = pop();
           push(pop() - op2);
/* push(pop()-pop()) е ГРЕШНО, защото редът на извикване е от
значение! */
           break;
        case '/':
           op2 = pop();
           if (op2 != 0.0)
              push(pop() / op2);
           else
              printf("error: zero divisor\n");
        break;
```

Пример$_5$

```
        case '\n':
           printf("\t%.8f\n", pop());
           break;
        default:
           printf("error: unknown command %s\n", s);
           break;
      } //край на switch
   } //край на while
   return 0;
} //край на main()
```

Пример 6

```
#define MAXVAL 100   /* максималната дълбочина на стека */
int sp = 0;          /* следващата празна позиция в стека */
double val[MAXVAL];  /* данните на стека */
/* push: push f onto value stack */
void push(double f) {
   if (sp < MAXVAL)
      val[sp++] = f;
   else
      printf("error: stack full, can't push %g\n", f);
}
/* pop: pop and return top value from stack */
double pop(void) {
   if (sp > 0)
      return val[--sp];
   else {
      printf("error: stack empty\n");
      return 0.0;
   }
}
```

Пример$_7$

```
#include <ctype.h>
int getch(void);
void ungetch(int);
/* getop: get next character or numeric operand */
int getop(char s[]) {
   int i, c;
   while ((s[0] = c = getch()) == ' ' || c == '\t');
   s[1] = '\0'; //добавя null terminator на низа
   if (!isdigit(c) && c != '.')
      return c; /* not a number */
   i = 0;
   if (isdigit(c)) /* collect integer part */
      while (isdigit(s[++i] = c = getch()));
   if (c == '.') /* collect fraction part */
      while (isdigit(s[++i] = c = getch()));
   s[i] = '\0'; //добавя null terminator на низа
   if (c != EOF)
      ungetch(c);
   return NUMBER;
}
```

Пример$_8$

```
#define BUFSIZE 100
char buf[BUFSIZE];   /* buffer for ungetch */
int bufp = 0;        /* next free position in buf */

/* get a (possibly pushed-back) character */
int getch(void) {
   return (bufp > 0) ? buf[--bufp] : getchar();
}

/* push character back on input */
void ungetch(int c) {
   if (bufp >= BUFSIZE)
      printf("ungetch: too many characters\n");
   else
      buf[bufp++] = c;
}
```

Декларираме и дефинираме

- Декларацията описва характеристиките на променливата (например типът й).
- При дефиниция се заделя и място за променливата и евентуално тя се инициализира
- Една променлива може да бъде декларирана на няколко места (например: в няколко файла), но се дефинира (и заделя място в паметта) само веднъж

```
//декларираме:
extern int sp;
extern double val[];
```

```
//дефинираме:
int sp;
double val[MAXVAL];
```

Заглавни (**`header`**) файлове

- Заглавните файлове в **`C`** са файлове, с разширение **`.h`**, които обикновено съдържат декларации (прототипи) на функции
- Когато една програма се състои от повече от един файл е добре декларациите (прототипите) на функциите да се запишат в отделен **`.h`** файл. При нужда, след това е по-лесно да се смени прототипа на дадена функция

Пример

main.c:

```
#include <stdio.h>
#include <stdlib.h>
#include "calc.h"
#define MAXOP 100
int main() {
   ...
}
```

getop.c:

```
#include <stdio.h>
#include <ctype.h>
#include "calc.h"
int getop(char s[]){
   ...
}
```

stack.c:

```
#include <stdio.h>
#include "calc.h"
#define MAXVAL 100
int sp = 0;
double val[MAXVAL];
void push(double f){
... }
double pop(void) {
... }
```

calc.h:

```
#define NUMBER '0'
void push(double);
double pop(void);
int getop(char []);
int getch(void);
void ungetch(int);
```

getch.c:

```
#include <stdio.h>
#define BUFSIZE 100
char buf[BUFSIZE];
int bufp = 0;
int getch(void) {
...}
void ungetch(int c) {
...}
```

Глобални **static** променливи

- Модификаторът **static**, приложен към дадена <u>глобална променлива</u> или <u>функция</u> оказва, че тя може да бъде ползвана („виждана“) само от функциите в нейния **.c** файл
- По този начин тези променливи се скриват от външния свят (другите файлове) и могат да бъдат променяни от него единствено косвено

```
В stack.c: sp и val се променят само от функциите от същия
файл - push(double f) и pop(void), които се извикват от друг
файл (main.c). Т.е. sp и val трябва да са static:

static int sp = 0;        //следващата празна позиция в стека
static double val[MAXVAL];        //данните на стека
void push(double f) { ... }
double pop(void) { ... }
```

Вътрешни `static` променливи

- Модификаторът **`static`**, приложен към дадена <u>вътрешна променлива</u> оказва, че тя няма да бъде унищожена след излизането от блока, а ще запази стойността си и при следващото преминаване от там.
- По този начин се дава възможност една функция да пази дадена информация докато програмата се изпълнява
- Поведението й е като на глобална променлива, но видима само от функцията, в която е дефинирана

Пример

```
#include <stdio.h>

int sum(int i);
int main () {
   int i = 0;
   for (;i<10;i++)
      sum(i);

   printf("%d\n", sum(0));
//извежда се 45!
   return 0;
}

int sum(int add) {
   static int s = 0;
   return s+=add;
}
```

≠

```
#include <stdio.h>

int sum(int i);
int main () {
   int i = 0;
   for (;i<10;i++)
      sum(i);

   printf("%d\n", sum(0));
//извежда се 0!
   return 0;
}

int sum(int add) {
   int s = 0;//всеки път=0!
   return s+=add;
}
```

`register` променливи$_1$

- Модификаторът **`register`**, приложен към дадена <u>променлива</u> дава препоръка на компилатора, че тя ще бъде ползвана много активно
- Идеята е тези променливи да се разположат в регистрите на процесора, което води до по-бързи и по-малки програми
- Компилаторът е свободен да игнорира тази препоръка – променливата се създава като нормална

```
register int x;
register char c;
```

`register` променливи$_2$

- Може да бъде прилаган само към **automatic** променливи (а не към глобални или **static**) и към формални параметри на функции
- Типът на променливите, които могат да са **register** зависи от системната архитектура
- На **register** променлива не може да се взима адреса (унарна операция **&**) без значение дали тя наистина е сложена в регистър или не

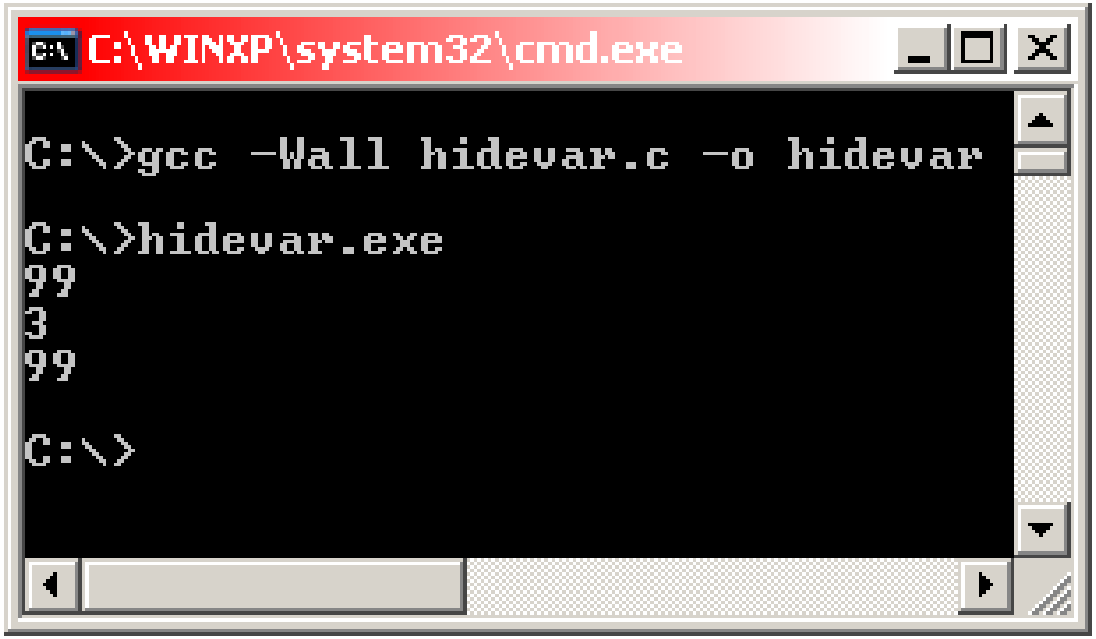
```
int f(register unsigned m, register long n) {
   register int i;
   ...
}
```

Блокова структура$_1$

- **`C`** не е блоково структуриран език в смисъла на Паскал – функция не може да се дефинира в друга функция
- Може да се дефинира променлива в началото на всеки блок и тя съществува до съответната затваряща фигурна скоба (края на блока)
- Всеки път като се мине през началото на дадения блок се създава нова променлива (с изключение на **`static`** променливите)

Пример

```
#include <stdio.h>

int main () {
    int nc=99;
    printf ("%d\n", nc);

    {
        int nc = 3;
        printf("%d\n", nc);
    }

    printf("%d\n", nc);

    return 0;
}
```



Блокова структура$_2$

- Дефинирането на променлива в блок или аргументи на функция „скрива“ други променливи със същото име, дефинирани в по-външен блок
- Добрият стил изисква да не се използват дублиращи имена – възможността за откриване на грешка е много малка

```
int x;
int y;
int f(double x) { //може да са от различен тип
   double y;
//double x и y скриват int x и y
}
```

Инициализация на скаларни променливи

- Глобалните и **`static`** променливите автоматично се инициализират с 0
- Всички останали – имат неопределена стойност
- Променливи могат да се инициализират като след дефинирането им се добави **`= value`**
- За глобалните и **`static`** променливи **`value`** трябва да е константа, докато за останалите може да е всеки един израз, извикване на функция и т.н.

Пример

```
int x = 1;
char squota = '\'';
long day = 1000L * 60L * 60L * 24L; // milliseconds/day
```

```
int binsearch(int x, int v[], int n) {
   int low = 0;
   int high = n - 1;
   int mid;
   ...
}
```

≡

```
int low, high, mid;
low = 0;
high = n - 1;
```

Инициализация на масиви$_1$

- Масиви могат да се инициализират чрез стойности разделени с “**,**“ и заградени в **{ }**
- Когато е пропуснат броя на елементите компилаторът го пресмята като брои броя на елементите при инициализация

```
int days[] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
//компилаторът сам пресмята, че това е масив с 12 елемента
```

Инициализация на масиви$_2$

- Ако има по-малко инициализирани елементи – следващите се запълват с 0. Без значение дали променливата е **`automatic`**, **`static`** или глобална
- Не се допуска да има повече инициализиращи елементи от големината на масива

```
int week[7] = {6, 7, 8, 7, 5};
//масив със 7 елемента със стойности: 6, 7, 8, 7, 5, 0, 0

//ГРЕШКА: масив с 3 елемента има 4 инициализиращи стойности
int err[3] = {1, 2, 3, 4};
```

Инициализация на масиви от символни низове

- Може символите да се затворят в “” без 0 на края
- Може да се използва и стандартния начин за масив

```
//масив с 5 елемента (има 0 на края!):
char pattern[] = "ould";
```

≡

```
char pattern[] = { 'o', 'u', 'l', 'd', '\0' };
```

Рекурсия$_1$

- Рекурсия имаме, когато функция извиква себе си директно или индиректно
- Задължително трябва да има някакво условие, при което рекурсията да спира
- При всяко едно извикване функцията разполага с нов набор от **`automatic`** променливи
- Рекурсивният код е по-компактен и по-лесен за разбиране от итеративния (не рекурсивния)
- Често се ползва за рекурсивни структури като дървета, графи и т.н.

Рекурсия$_2$

- Пряка

```
int some_func(int i) {
    //...
    if(i!=1)
        some_func(i-1);
    //...
}
```

Косвена (не пряка)

```
int some_func1(int i) {
    //...
        some_func2(i-1);
    //...
}
```

```
int some_func2(int i) {
    //...
        some_func1(i+1);
    //...
}
```

Пример

```
#include <stdio.h>

/* printd: извежда n като десетично число */
void printd(int n) {
   if (n < 0) {
      putchar('-');
      n = -n;
   }
   if (n / 10)           //условие за край на рекурсията
      printd(n / 10);    //рекурсивно извикване
   putchar(n % 10 + '0');
}
```

Пример

```
/* qsort: сортира v[left]...v[right] в нарастващ ред */
void qsort(int v[], int left, int right) {
   int i, last;
   void swap(int v[], int i, int j);
   if (left >= right) /* do nothing if array contains */
      return;         /* fewer than two elements */
   swap(v, left, (left + right)/2); //move partition elem
   last = left;                     //to v[0]
   for (i = left + 1; i <= right; i++) /* partition */
      if (v[i] < v[left])
         swap(v, ++last, i);
   swap(v, left, last); /* restore partition elem */
   qsort(v, left, last-1);
   qsort(v, last+1, right);
}
void swap(int v[], int i, int j) {
   int temp = v[i];
   v[i] = v[j];
   v[j] = temp;
}
```

Предпроцесор. Вмъкване на файл

```
#include "filename"
#include <filename>
```

- Всеки такъв ред се заменя със съдържанието на **`filename`**
- **`"filename"`** се търси в директорията на сорс кода и ако не е намерен там или името е заградено в **`<>`** (а не в **`""`**) - в директорията със стандартните библиотеки
- В такива файлове обикновено се записват прототипи на функции, често ползвани **`#define`** и други. Така се гарантира, че цялата програма ще ползва еднакви декларации, прототипи и други

Предпроцесор. Замяна на текст 1

```
#define name заменящ текст
```

- За „**name**“ важат същите правила като за име на променлива
- „**заменящ текст**“ е текстът до края на реда. Ако на края на реда има **** то той продължава и на следващия ред

```
#include <stdio.h>
#define H "Hello, World"

int main () {
   printf ("%s\n", H);
   return 0;
}
```

≡

```
#include <stdio.h>
#define H "Hello, \
World"
int main () {
   printf ("%s\n", H);
   return 0;
}
```

Предпроцесор. Замяна на текст


```
#define name заменящ текст
```

- Преди да се подаде на компилатора текста на програмата се преработва съгласно правилата до:

```
//... съдържанието на <stdio.h>
int main () {
   printf ("%s\n", "Hello, World");
   return 0;
}
```

```
#include <stdio.h>
#define H "Hello, World"

int main () {
   printf ("%s\n", H);
   return 0;
}
```

≡

```
#include <stdio.h>
#define H "Hello, \
World"
int main () {
   printf ("%s\n", H);
   return 0;
}
```

Предпроцесор. Замяна на текст$_2$

```
#define name заменящ текст
```

- Замяната се прави от съответния **#define** до края на файла
- Не се заменят части от дума или низове
- „**заменящ текст**“ може да е практически всичко

```
#include <stdio.h>
#define YES da
#define forever for (;;) /* безкраен цикъл */
int main() {
   printf(“YES“);     //НЯМА да се замести с da
   int YESMAN = 0;    //НЯМА да се замести с da (daMAN)

   forever printf(“aa“);   //извежда aa до безкрай
   //еквивалентно на: for(;;) printf(“aa“);
   return 0;
}
```

Предпроцесор. Замяна на текст 3

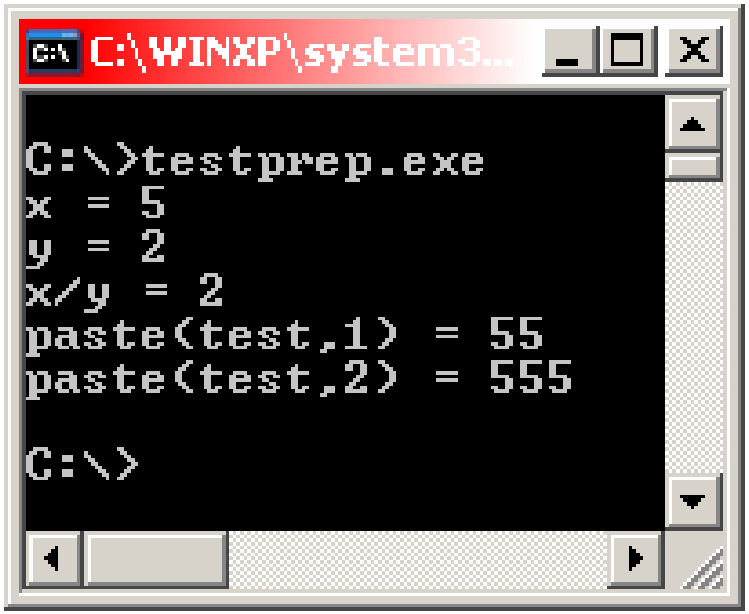
- Възможност за добавяне на аргументи – няма space между name и (!!!
- Възможност за премахване на име

```
//работи за всеки тип, който може да се сравнява:
//!!! няма празно място между max и (A, B) !!!
#define max(A, B) ((A) > (B) ? (A) : (B))
x = max(p+q, r+s); //x = ((p+q) > (r+s) ? (p+q) : (r+s));

max(i++, j++); //ГРЕШКА – по-голямото се увеличава два пъти!
               // ((i++) > (j++)? (i++) : (j++) )

#define square(x) x * x //ГРЕШКА: трябва да е ( (x) * (x) )
int t = square(z+1);    // = z+1 * z+1 = z+(1*z)+1
...
#undef square
int square(int x) {
   return x*x;
}
```

Предпроцесор. Замяна на текст

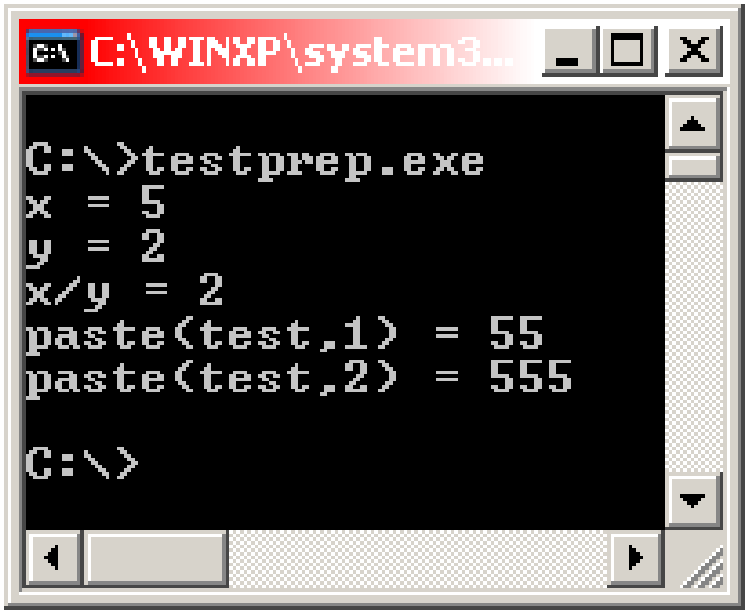
- # към низ
- ## конкатенация

```
#include <stdio.h>
#define dprint(expr) printf(#expr " = %d\n", expr)
#define paste(front, back) front ## back

int main() {
    int x = 5, y=2;
    int test1 = 55, test2 = 555;
    dprint(x);
    dprint(y);
    dprint(x/y);
//printf("x/y" " = %d\n", x/y);
    x=paste(test,1);//x = test1 = 55
    dprint(paste(test,1));
    dprint(paste(test,2));
    return 0;
}
```



Пример

```
/*
преди компилация горната програма се преобразува от
предпроцесора до следното:
*/
//... съдържанието на <stdio.h> ...

int main() {
   int x = 5, y=2;
   int test1 = 55, test2 = 555;
   printf("x" " = %d\n", x);
   printf("y" " = %d\n", y);
   printf("x/y" " = %d\n", x/y);
   x = test1;

   printf("paste(test,1)" " = %d\n", test1);
   printf("paste(test,2)" " = %d\n", test2);
   //"abc" "123" ≡ "abc123" (лекция 2, 7 слайд)

   return 0;
}
```



Пример

С код

```
#include <stdio.h>
#define dprint(expr) printf(#expr " = %d\n", expr)
#define paste(front, back) front ## back

int x = 5, y=2;
int test1 = 55, test2 = 555;
dprint(x);
dprint(y);
dprint(x/y);
x=paste(test,1);//x = test1 = 55
dprint(paste(test,1));
dprint(paste(test,2));
```

С код, минал през предпроцесора

```
int x = 5, y=2;
int test1 = 55, test2 = 555;
printf("x" " = %d\n", x);
printf("y" " = %d\n", y);
printf("x/y" " = %d\n", x/y);
x = test1;
printf("paste(test,1)" " = %d\n", test1);
printf("paste(test,2)" " = %d\n", test2);
```

Предпроцесор. Директиви за условия

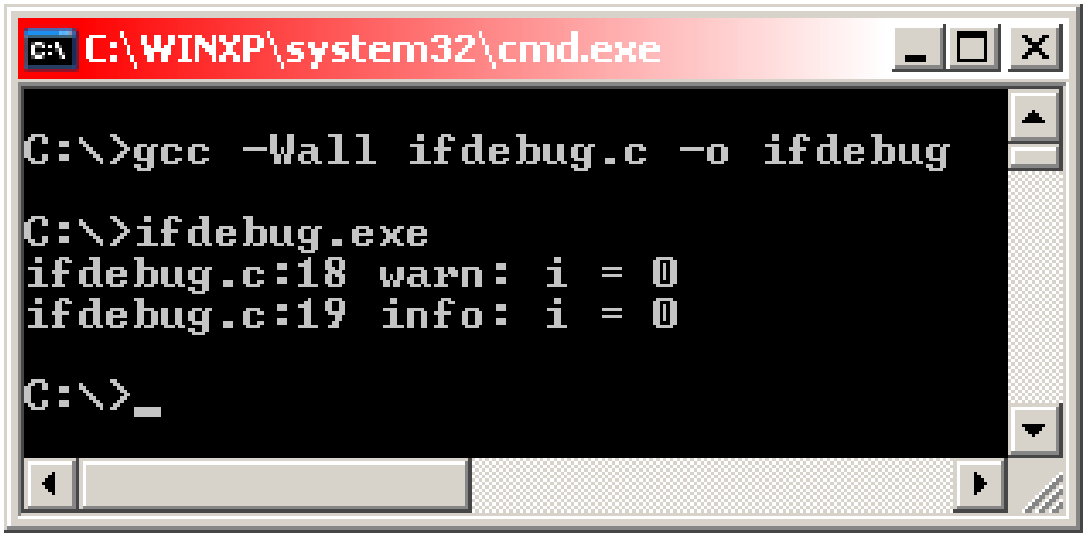
- **#if, #ifdef, #ifndef**
- **#endif, #elif, #else**

```
#ifndef HDR
#define HDR
// hdr.h
#endif
```

≡

```
#if !defined(HDR)
#define HDR
// hdr.h
#endif
```

```
#if defined(CREDIT)
   credit();
#elif defined(DEBIT)
   debit();
#else
   printError();
#endif
```

Пример

```
#include <stdio.h>
#define DEBUG_LEVEL 5
#if DEBUG_LEVEL>=3
   #define info(expr) printf("%s:%d info: " #expr " = %d\n",\
      __FILE__, __LINE__, expr)
#else
   #define info(expr)
#endif
#if DEBUG_LEVEL>=1
   #define warn(expr) printf("%s:%d warn: " #expr " = %d\n",\
      __FILE__, __LINE__, expr)
#else
   #define warn(expr)
#endif

int main() {
   int i = 0;
   warn(i);
   info(i);
   return 0;
}
```