

# ОБУЧЕНИЕ ПО ПРОГРАМИРАНЕ

Побитови операции. Endians – Big and Little. Литерали.  
Препроцесор.



# Побитови операции



# Побитови

## оператори

<< >> Шифтване (shift) на битовете на ляво или на дясно

$00001101 \gg 2 = 00000011$

$13 = 00001101$

$00001101 \ll 1 = 00011010$

$000011010 = 26$

~ Побитово отрицание (bitwise NOT)

$\sim 0001 = 1110$



# Побитови оператори

& Побитово И

a = 0011

&

b = 0101

R = 0001

| Побитово ИЛИ

a = 0011

|

b = 0101

r = 0111

^ XOR

a =

0011

^

b = 0101

r =

0110



# Свойства на побитовите оператори

1. За да променим  $n$  тия bit на променливата  $a$  на 1 трябва да използваме побитово ИЛИ

`int a = 15;` в двоичен вид това е числото 0000 1111

Ако искаме да сетнем нулата на пета позиция правим маска 0010 0000, като шифтнем  $1 \ll 5$ ; пъти. След това прилагаме побитово ИЛИ на променливата и маската:

0000 1111

|

0010 0000

= 0010 1111



# Свойства на побитовите оператори

2. За да направим на 0 съответния бит на n-та позиция, трябва да използваме побитово &

`int a = 35;` в двоичен вид това е числото 0010 0011

Ако искаме да променим единицата на пета позиция, правим маска

0010 0000, като шифтнем  $1 \ll 5$ ; пъти. След това приложим побитово

отрицание ~ на маската, за да получим 1101 111. След това, като приложим побитов оператор И, получаваме от 1 нула на 5-та позиция:

0010 0011

&

1101 1111

= 0000 0011





# Свойства на побитовите оператори

3. За да обърнем  $n$ -тия bit на променливата  $a$  на 1, трябва да използваме побитово XOR

`int a = 15;` в двоичен вид това е числото 0000 1111

Ако искаме да обърнем нулата на пета позиция, правим маска 0010 0000, като шифтнем  $1 \ll 5$ ; пъти. След това прилагаме побитово XOR на променливата и маската:

0000 1111

^

0010 0000

= 0010 1111



# Задачи за работа с битове:

1) Направете побиtovите операции върху целите числа 3 и 5 в main().  
Покажете резултата с функцията `printf("%d", res);`

AND operation	OR operation	XOR operation
00000011	00000011	00000011
& 00000101	00000101	^ 00000101
<hr/>	<hr/>	<hr/>
00000001	100000111	00000110
= 1	= 7	= 6

2) Какъв ще бъде резултатът от следната операция: `a = a ^ a;`  
?







# Задачи за работа с битове:

2) Създайте функция `bitAt`. Функцията получава параметър число и индекс. Тя връща стойността на бита на съответния индекс.



# Задачи за работа с битове:

3) Създайте функция `clearBit`. Тя получава като параметър число и индекс, и прави на нула бита на съответния индекс в числото.



# Задачи за работа с битове:

4) Създайте функция `setBit`. Тя получава следните параметри - число и индекс. Функцията трябва да "вдига" бита на съответния индекс в числото (под "вдига" ще разбираме да му присвои стойност 1).



# Задачи за работа с битове:

5) Създайте функция `reverseBit`. Тя получава следните параметри - число и индекс. Като резултат функцията обръща бита на съответната позиция в числото (съответно от 0 на 1 или от 1 на 0).





# Задачи за работа с битове:

6) Променете стойността на бита на дадено число на определена позиция.



# Задачи за самостоятелна

## работа:

7) За нулето в двоични битове на числата, намиращи се на четна позиция.





# Задачи за самостоятелна работа:

8\*) Намерете позицията на най-старшия бит, който е със стойност 1 в дадено число.



# Задачи за самостоятелна работа:

9) Направете на нула битовете в числа, намиращи се на позиции между 3 и 7.





# Задачи за самостоятелна работа:

10) Премахнете всички битове на число след  $n$ -тия бит включително.





# Задачи за самостоятелна работа:

11\*) Разбийте число като сума от степени на двойката.



# Задачи за самостоятелна работа:

12) Вдигнете всички битове на 32 битово число,  
на позиции делищи се на 3.



# Задачи за самостоятелна работа:

13\*) Намерете броя на позициите на битовете, в които две числа се различават.

14\*) Обърнете битовете на число, които се намират на нечетна позиция.





# Big Endian, Little Endian



# Big Endian, Little Endian

- ▶ ***Big-endian** и **Little-endian** са два начина за съхраняване на много-байтови типове данни .*
- ▶ В машините **Little-Endians** **последният байт** от двоичното представяне на типа данни **се съхранява първи**.
- ▶ В машините **Big-Endians** **първият байт** от двоичното представяне на типа данни **се съхранява първи**.



# Побайтово представяне на int, float и указател.

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

```
/* function to show bytes in memory, from location start */
```

```
void show_mem_rep(char *start, int n)
{
    int i;
    for (i = 0; i < n; i++)
        printf(" %.2x", start[i]);
    printf("\n");
}
```

```
int main()
{
    int i = 0x01234567;
    show_mem_rep((char *)&i, sizeof(i));
    getchar();
    return 0;
}
```



# Big Endian, Little Endian

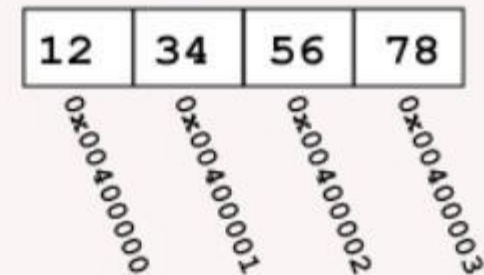
Имаме едно число, което е 4 байта(32 бита). Има производители, които поставят старшия байт първи (Big Endian) като Motorola, а Intel и AMD са решили да поставят младшия байт първи (Little Endian)

0x AA BB CC DD

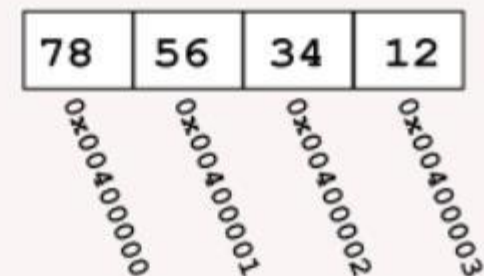
BE AA BB CC DD

LE DD BB CC AA

## Big Endian



## Little Endian



# C program to check processor endians

```
#include <stdio.h>
#include
<inttypes.h> int
main(void)
{
    uint32_t
    data; uint8_t
    *cptr;
    data = 1; //Assign data
    cptr = (uint8_t *)&data; //Type
    cast if (*cptr == 1)
    {
        printf("little-endian");
    }
    else
        if (*cptr == 0)
        {
            printf("big-endian");
        }
    return 0;
}
```



# Да проверим на нашия компютър как се представят битовете:

**Упражнение 1** Big and Little Endian (подредба на байтовете: старши байт първи или младши байт първи)

```
#include <stdio.h>

void print_bytes(const char* pcVal, unsigned uSize) {
    unsigned i;

    for (i = 0; i < uSize; i++)
        printf("%#02x ", (int) (pcVal[i] & 0xFF));

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







# Big Endian, Small Endian


```
#include <stdio.h>

void LEtoBE(unsigned int *t) {
    *t = (*t >> 24) | (*t << 24) |
        ((*t << 8) & 0x00ff0000u) |
        ((*t >> 8) & 0x0000ff00u);
    return;
}

int main(void) {
    unsigned int cell = 0x12345678u;
    printf("%x\n", cell);
    LEtoBE(&cell);
    printf("%x\n", cell);
    return 0;
}
```



# Препроцесор

Препроцесорът (от англ. preprocessor) на езика С прави предварителна обработка на файловете без да променя синтаксиса на езика. Тази обработка може да се илюстрира като замяна на един текст с друг. За удобство на програмиста са въведени команди към препроцесора, които се наричат **директиви** и имат префикс '#'.  
  


# Препроцесор

- изпълнява предварителна текстообработка на изходния файл и не е нищо по различно от малко по интелигентен текстов редактор
- препроцесорът не променя синтаксиса
- текстообработката се управлява с команди
- директиви - команди на препроцесора (започват с #)
- препроцесорът не е част от компилатора
- препроцесорът се изпълнява преди компилатора
- обикновено грешките при препроцесора са при указване на заглавен файл или при макроси
- препроцесорът не разбира от C-и типове, а просто обработва стрингове



# Препроцесор

Двете най - често използвани характеристики са:

`# include` служи за включване съдържанието на файл на мястото, където е използвана директивата `# include` по време на препроцесорна обработка преди компилация и `#define` за заместване на символ с произволна последователност от символи. С нея се дефинира макрос за текстозамяна (търси текст XXX и го заменя с текст YYY). Пример:

```
#define CONST_PI (3.14159) /* macro for search/replace */
```

Горната директива се указва на препроцесора при всяко срещане на стринга `CONST_PI` **да бъде заменен** с (3.14159).



# Включване на файлове

Всеки един от тези директиви:

```
#include "filename"
```

и

```
#include <filename>
```

се замества със съдържанието на файла преди компилацията, а в .h файловете се намират декларациите на функциите. Ако името на файла е поставено в кавички, предпроцесорът търси файла в текущата директория, в която се намира изходната програма.

Ако не го открие там или ако името е оградено с < и >, търсенето следва дефинирани от реализацията правила, за да намери файла. Всеки включен файл може да съдържа в себе си `#include` редове.

Често пъти в началото на сорс файла има няколко `#include` реда, `#define` стейтмънти и `extern` декларации или достъп до прототипните декларации на библиотечните функции като `<stdio.h>`



# Включване на файлове с препроцесор

`#include` е предпочитаният начин за свързване на декларациите при големи програми. Той гарантира, че всички изходни файлове ще притежават едни и същи дефиниции и декларации на променливите и по този начин се предотвратява възможността за появата на грешка.

Когато съдържанието на един включен файл бъде променено, всички файлове, които зависят от него, трябва да бъдат компилирани отново.





# Замяна посредством макроси

Дефиницията има следната форма:

```
#define име текст
```

Когато препроцесорът намери *име*, то в кода той го заменя с *текст* а - това представлява. Името в `#define` има същия формат както имената на променливите. Текстът за замяна е произволен.

Обикновено текстът за замяна продължава до края на `#define` реда, но при по дълги дефиниции той може да се разпростре на няколко реда, като се поставя `\` накрая на всеки ред, след който ще има продължение.

Областта на действие на името, дефинирано с `#define`, започва от мястото на дефиниция и стига до края на изходния файл.

Замяната не се осъществява ако името се среща с низ кавички:

```
#define YES 1
```

няма да се замени в `printf("YES");`



# Замяна посредством макроси

Към всяко дефинирано име може да се приложи всякакъв текст за замяна. Например:

```
#define forever for( ; ; ) /*безкраен цикъл*/
```

дефинира дума `forever`, която когато се срещне в кода, ще се замени от безкрайния условен цикъл `for`.

Възможно е също да дефинирате макрос с аргументи, тогава текстът за замяна може да бъде различен при различните извиквания на макроса. Например може да дефинираме макрос, наречен `max`:

```
#define max(A, B) (A) > (B) ? (A): (B)
```

Въпреки че наподобява извикване на функция, при срещането на `int x = max(4, 5);` в кода предпроцесорът ще замени аргументите направо в израза:

`int x = 4 > 5 ? 4 : 5 ;` и израза ще върне `x = 5`, защото 4 не е по голямо от 5



# Замяна посредством макроси

Това може да доведе до проблем, ако параметрите не са поставени в скоби.

Изразът:

```
int p=1,q=2,r=3,s=4;
```

```
int x = max(p+q, r+s);
```

ще бъде заменен с :

```
int x = ((p+q) > (r+s) ? (p+q) : (r+s));
```

Помислете какво би се случило, ако забравите да сложите скоби в макроса:

```
#define square(x) x * x
```

x = square(p+q); Определете точния ред на изчисленията? **(p+q \* p+q);**

Макросът може да работи с данни от всякакви типове, стига аргументите да бъдат съвместими. Няма нужда да работите с различни max за различните типове. В това се състои разликата между макросите и функциите.



# Предимства и недостатъци

Предимства:

- Могат да се употребяват различни типове.

Недостатъци:

- Макросите не могат да се дебъгват
- Макросите са като inline функциите - добавят се в изпълнимия файл.



# Замяна посредством макроси

Всички дефиниции с `#define` могат да се отменят с **`#undef`**

например:

```
#define square(x) (x) * (x) int x =  
square(2+3);
```

```
#undef square
```

```
//square(3)//ERROR
```

На следващия ред в кода вече не може да използваме макроса `square`. `#undef` се използва за да се покаже изрично, че дадена функция не е макрос, а истинска функция.



# Преобразуване в стрингов литерал #

Низове с кавички и формални параметри **не се заместват**. Обаче ако в текста за замяна името на даден параметър е предшествано от знака на препроцесора **'#'**, комбинацията ще бъде заместена до низ в кавички, като параметърът **ще бъде заместен от действителния аргумент, поставен в кавички:**

```
#define tostr(s) #s  
    printf("tostr: %s", tostr(xxxxxx));
```

Това ще принтира на екрана xxxxxx

```
printf("tostr: %s", tostr(x/y));
```

Знакът **'/'** се заменя с **'//'** , за да се изпише в стринга.



# Сливане на аргументи '##'

Ако даден параметър от текста за заместване стои непосредствено до символите **# #** , следван от втори аргумент, то двата аргумента се сливат в един или това се нарича още **конкатенация**.

```
#define concat(x,y) x##y  
    int concat(i,X);  
    iX = 1;
```





# Условно включване на код

Възможно е да контролирате действието на препроцесора с помощта на условни оператори, които се проверяват по време на предпроцесорната обработка. Това ви дава възможност да включвате код избирателно, в зависимост от стойността на условията, изчислени по време на компилацията. Ред, съдържащ `#if`, оценява израза след него като константа. Ако изразът има ненулева стойност, се включват редовете след `#if`, докато препроцесорът не срещне ред `#endif` или `#elif`. (препроцесорна конструкция `#elif` е аналогична на `else if()`)

```
int main(void){  
    #if 1*1  
        printf("Test 1\n");  
    #endif  
    #if 1*0  
        printf("Test 2\n");  
    #endif
```





# Условно включване на код

Изразът `define` име в `#if` има стойност 1, ако името е дефинирано, в противен случай стойността му е 0

Искаме съдържанието на един `.h` файл (кръстен например `some_header.h`) да бъде включено само веднъж при използването му в `.c` файл, трябва да го декларираме така:

```
#ifndef SOME_HEADER_H
#define SOME_HEADER_H
#include <stdio.h>
void f(); //some function prototype #endif
```





# УСЛОВНО ВКЛЮЧВАНЕ НА КОД

Когато го включваме в един компилационен файл с разширение .c (например file.c), ние го добавяме така :

```
#include "some_header.h" int
```

```
main(void){
```

```
    #ifndef SOME_HEADER_H
```

```
        printf("Will never be printed");
```

```
    #endif
```

```
        return 0;
```

```
}
```

Вътре в main() функцията питаме дали е дефинирано SOME\_HEADER\_H и понеже то е дефинирано в #include "some\_header.h" то printf никога не се изпълнява.



# Директиви

<code>#define</code>	дефинира макрос (за текстозамяна)
<code>#include</code>	включва заглавен файл
<code>#undef</code>	изтрива дефиниран макрос
<code>#ifdef</code>	връща истина, ако макросът е дефиниран
<code>#ifndef</code>	връща истина, ако макросът не е дефиниран
<code>#if</code>	проверява условие (което се оценява по време на компилация) дали е истина
<code>#else</code>	алтернатива на <code>#if</code>
<code>#elif</code>	<code>#else</code> и <code>#if</code> в една директива
<code>#endif</code>	край на условна конструкция
<code>#error</code>	извежда грешка на <code>stderr</code>
<code>#pragma</code>	предава команди на компилатора



```
#include <stdio.h> /* include stdio function prototypes */  
#define CONST_PI (3.14159) /* macro for search/replace */ #define
```

```
MAX(a,b) (((a)>(b)) ? (a) : (b))
```

```
#define log_trace(s) \  
    printf(_FILE    " : " s) /* multiline macro */
```

```
    #define log_error(...) fprintf(stderr,_VA_ARGS_) /*  
променлив брой аргументи */
```

```
#ifdef DEBUG  
    dump_mem_to_file("/tmp/mem.txt");/*dump memory to file*/  
#endif
```



# Предефинирани макроси

__DAT__	текуща дата на препроцесорна обработка (MMM DD YYYY)
__E__	
__TIME__	текущо време на препроцесорна обработка (HH:MM:SS)
__FILE__	текущ файл на обработка
__LINE__	текуща линия, като десетична константа
__STDC__	ако компилаторът поддържа ANSI стандарт е 1

(\*) има още предефинирани макроси, които са специфични за конкретния компилатор



```
/* header guard, the same as #pragma once */  
#ifndef _FILE_H_  
#define _FILE_H_  
    /* header definitions */ #endif /*  
_FILE_H_ */  
  
#if PRODUCT_VERSION < 3  
    #error "Incompatible product version" #endif  
  
/* send options to the C-compiler */  
#pragma warning (disable : 1234)
```



# Задачи:

Задача 1. Дефинирайте константи `PI`, `E`, и други с помощта на макроси. Използвайте ги в кода.

Задача 2. Напишете макрос `swap(t, x, y)`, който променя местата на двата аргумента от тип `t`.

Задача 3. Реализирайте условна компилация в зависимост от макрос `DEBUG`



Задача 4. Използвайте предефинирани макроси

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

```
main() {
```

```
    printf("File      :%s\n",      FILE ); /* текущ файл */
```

```
    printf("Date      :%s\n",      DATE ); /* дата */
```

```
    printf("Time      :%s\n",      TIME ); /* време */
```

```
    printf("Line      :%d\n",      LINE ); /* ред */
```

```
    printf("ANSI      :%d\n",      _STDC ); /* ANSI */
```

```
    return 0;
```

```
}
```





**Задача 5. Напишете макрос с един параметър, който печата съобщение, само при дефиниран макрос `DEBUG`. Ако `DEBUG` не е дефиниран, не печата нищо.**

**Задача 6. Напишете макрос с променлив брой параметри, който извиква `printf` със префикс `"TRACE: "`**



**Задача 9. Напишете макрос, който свързва два идентификатора в един общ (конкатениране на идентификатори)**

**Задача 10. Напишете макрос, който прави идентификатор на низ (стринг)**



Задача 11. Напишете функциите като макроси:

1. Функция  $AVG(x, y)$ , която смята средното аритметично на две подадени като параметър числа.
2. Функция  $AVG$ , която смята средното аритметично на числата от определен диапазон. В  $[a, b]$  средното на всички цели числа.
3. **Напишете функцията повдигане на степен, която повдига  $x$  на степен  $y$**

**`#define STEPEN(x,y) exp(y*log(x))`**

защото  $\exp(\log(a))=a$   
и  $k*\log(x)=x^k$



Задача 11. Напишете функциите като макроси:

4. Напишете функцията `TOUPPER`, която прави малката буква `a` в голяма `A`

5.



Задача 12. Напишете макро `GENERIC_MAX(type)`, което трябва да се замени със следната функция:

```
int int_max(int x, int y){ return x > y  
? x : y;  
}
```

или

```
float float_max(float x, float y){  
return x > y ? x : y;  
}
```

или

```
char char_max(char x, char y){  
return x > y ? x : y;  
}
```



# Задачи

**Задача 11\*. Напишете програма, която да премахва всички коментари в една C програма.**



# Целочислени литерали

123	/* десетичен литерал */
0321	/* осмичен */
0xab	/* hex int */
38	/* тип: int, 4Bytes */
38u	/* тип: unsigned int */
38l	/* тип: long, 8Bytes */
38ul	/* тип: unsigned long, 8Bytes */
38LL	/* тип: long long int */



# Литерали с плаваща запета

30.1F /\* float, 4Bytes \*/

30.1 /\* double, 8Bytes \*/

- 30.1

301.0e-1 /\* double \*/

0.301e3

30.1L /\* long double float, 16Bytes \*/





## Задача 2. Числови литерали

```
#include <stdio.h>
int main() {
    printf("Size of literal default integer %d\n", sizeof(3));
    printf("Size of literal U: %ld\n", sizeof(3U));
    printf("Size of literal l: %ld\n", sizeof(3l));
    printf("Size of literal L: %ld\n", sizeof(3L));
    printf("Size of literal LL: %ld\n", sizeof(3LL));
    printf("Size of literal default floating point: %ld\n",
sizeof(3.1));
    printf("Size of literal F: %ld\n", sizeof(3.1F));
    printf("Size of literal D: %ld\n", sizeof(3.1D));
    printf("Size of literal L: %ld\n", sizeof(3.1L));
}
```



# Символни литерали

- символи, оградени с единични кавички
- backslash и код на символ
- 'a' /\* symbol 'a' \*/
- '\037' /\* octal symbol code \*/
- '\x1A' /\* hex symbol code \*/
- '\n' /\* нов ред \*/
- '\t' /\* табулация \*/
- '\\' /\* обратна наклонена черта \*/
- '\u02FF' /\* разширени символи, повече от байт \*/



## Задача. Специални символни литерали (escape sequences)

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

```
int main() { int
```

```
    i;
```

```
    char arrEscChar[] = { '\t', '\n', '\r', '\v', '\\', '\'', '\"', '\a', '\b', '\f' };
```

```
    '\?';
```

```
    for (i = 0; i < sizeof(arrEscChar); i++) {
```

```
        printf("Escape Character %d '%c'\n", (int)(arrEscChar[i]), arrEscChar[i]);
```

```
    }
```

```
    return 0;
```

```
}
```



**Задача 6. Запишете в четири последователни байтове в паметта със стойност 0xAA(10101010) и разпечатайте съдържанието на горните байтове, ако типът е: float, signed int, unsigned int**

**Задача 7. Запишете в осем последователни байта в паметта със стойност 0xBB(1011 1011) и разпечатайте съдържанието на горните байтове, ако типът е: double, signed long long, unsigned long long**



# Константи

## Дефиниция

```
const Ctype cmyConstIdentifier = init_literal;
```

## Декларация

```
extern const Ctype cmyConstIdentifier;
```

## Използване на препроцесора за създаване на константа

```
#define CONST_PI (3.14159)
```



# Особености на константите

- 1) Константите са подобни на променливите, но не променят стойността си.
- 2) При декларация на константа, задължително се инициализира.
- 3) Декларацията на константа е подобна като на променлива.
- 4) Стойността на константата е ясна по време на компилиране.
- 5) Според Код конвенцията към името се добавя префикс с (от `const`).



# string литерали

Обграждат се от двойни кавички.

Имат скрит терминиращ символ '\0'

Последователни string-ове се свързват от

компилятора. "This is"

"a long string"

". "



# string литерали

Технически погледнато string литерал (поредица) представлява поредица в паметта от еднакви по размер променливи т.е. масив от символи. Вътрешното представяне на всеки низ завършва с нулев символ '\0' накрая, така че физическото пространство, необходимо за съхранение на низа, е с единица по голямо от символите между кавичките. Това представяне показва, че няма ограничение за дължината на низовете, но програмите трябва да обхождат целия низ, за да определят дължината му.





**ЗА ДОМАШНА РАБОТА  
СА  
ВСИЧКИ ЗАДАЧИ В ЧЕРВЕНО**

