Little и Big endian (мала и велика крајност)

- Узмимо да је ширина меморије на неком процесору 8 бита, тј. да сваких 8 бита има своју адресу.

  Питање: Како ћемо у меморију таквог процесора сместити 32-битну вредност?

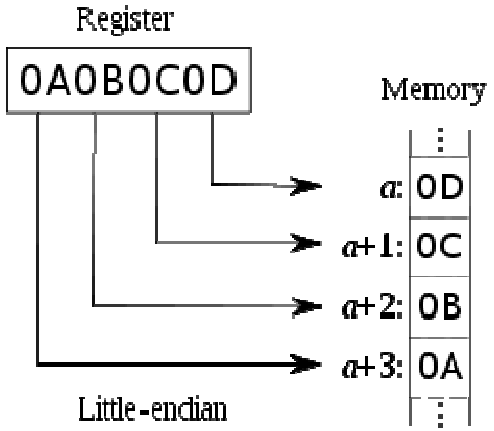
  Одговор: Поделићемо 32-битну вредност на 4 8-битне, нпр. вредност 0x0A0B0C0D се може поделити на следећа четири дела:

  0x0A, 0x0B, 0x0C и 0x0D, поређаних по значајности, од највеће ка најмањој.

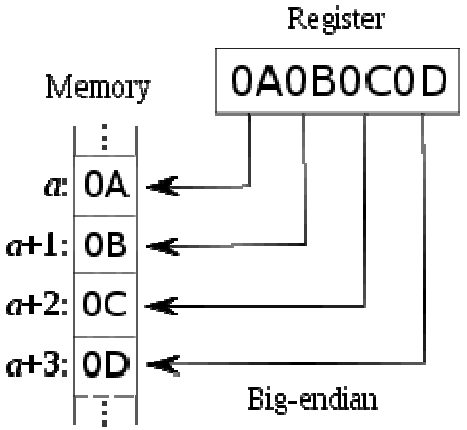
- Два су начина да се та четири дела сместе у меморију:
  - Делови мањег значаја иду на мању адресу - Литл ендијан
  - Делови већег значаја иду на мању адресу - Биг ендијан
- Са ког краја се вредности смештају у меморију највише зависи од самог процесора. Рецимо:
  - MIPS – може се изабрати LE и BE
  - Intel - LE
  - ARM - LE

Little и Big endian

- Део **најмањег** значаја се уписује први = **little** endian



- Део **највећег** значаја се уписује први = **big** endian



Little и Big endian Илустрација разлике

1. Смести 0x0A0B0C0D у меморију као 32-битну вредност (рецимо int типа)
2. Прочитај из тих меморијских локација као 2 узастопне 16-битне вредности
3. Прочитај из тих меморијских локација као 4 узастопне 8-битне вредности
4. Прочитане вредности:

16bit access:

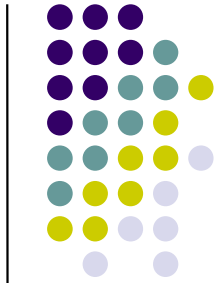
- **LE**: 0x0C0D и 0x0A0B
- **BE**: 0x0A0B и 0x0C0D

8bit access:

- **LE**: 0x0D, 0x0C, 0x0B и 0x0A
- **BE**: 0x0A, 0x0B, 0x0C и 0x0D

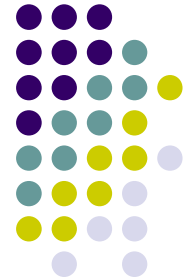
Размена података између рачунара

- Постоји договор око формата којим се шаљу подаци преко мреже. **Мрежни стандард каже: big-endian.**
- За пребацивање у мрежни формат звати функцију **`hton`** (host-to-network). На big-endian рачунарима не ради ништа, али да би програм био портабилнији важно је користити **`hton`** увек пре слања података преко мреже.
- Аналогно постоји и функција **`ntoh`** (network-to-host) која служи за читање података са мреже.
- Функције за конверзију:
  - **`htons()`** – "Host to Network Short" 16бит
  - **`htonl()`** – "Host to Network Long" 32бит
  - **`ntohs()`** – "Network to Host Short" 16бит
  - **`ntohl()`** – "Network to Host Long" 32бит



Декларације и дефиниције променљивих (и помало функција)

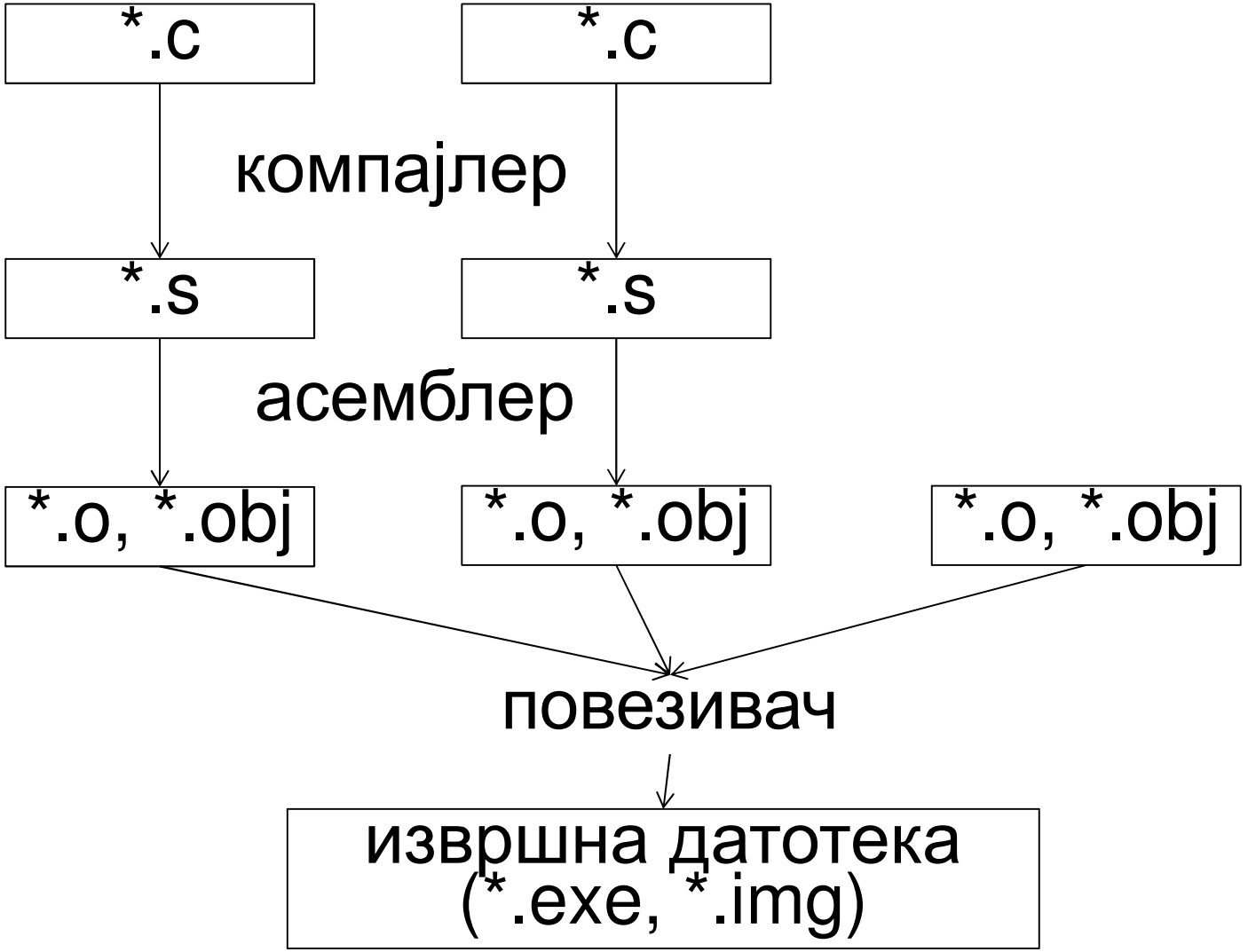
Особине променљивих



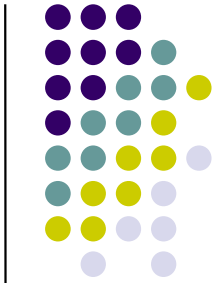
- Назив и тип...

- Назив и тип променљиве нису њене једине особине!
- Све особине променљиве се саопштавају приликом њеног декларисања.
- Неке особине су предвиђене стандардом, али могу постојати неке особине специфичне за конкретну платформу и конкретан компајлер.

Билдовање програма - слика



Формирање назива променљиве



- Први знак мора бити слово или _.
- Након првог знака и бројеви могу бити коришћени.
- Последица претходна два: размаци нису дозвољени.
- Променљива се не може звати исто као резервисана реч.
- Велика и мала слова нису исто, па тако **Mile** није иста променљива као **mile**.
- Само прва 63 знака гарантовано јединствено дефинишу променљиву.
  - Само 31 у случају екстерне променљиве
- Коришћење _ на почетку се обесхрабрује јер је по договору резервисано за неке библиотечке и системске идентификаторе.

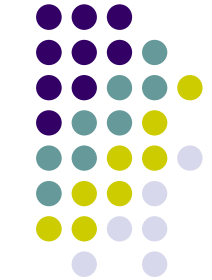
Савети:

- Имена трабају бити смислена и информативна.
  - **studentAge** или **student_age** даје више информација о променљивој и њеној сврси него само **age**, или, не дај Боже, само **a**.
- Када се име састоји од више речи, оне могу бити раздвојене уметањем _, или се могу визуелно раздвојити тако што ће наредна реч почети великим словом.
- Употребити један принцип именовања и држати га се. У случају рада у тиму на неком пројекту придржавати се правила именовања која у том пројекту важе.

Дефиниција и декларација променљиве

- **Декларација** описује променљиву, дајући довољно информација да компајлер може знати како са променљивом треба да поступа.
- **Дефиниција** представља заузимање физичког простора за променљиву, а може укључивати и њену иницијализацију.
- Декларација и дефиниција могу бити одвојене или спојене.
  - Код променљивих су у пракично свим случајевима, осим у једном, дефиниција и декларација спојене.
  - Код функција је разлика између дефиниције и декларације јаснија.

Примери декларације (и дефиниције)



- Једна декларација (са дефиницијом) у једном реду

```
int age;
float amountOfMoney;
char initial;
```

- Више декларација (са дефиницијама) у једном реду

```
int age, houseNumber, quantity;
float distance, rateOfDiscount;
char firstInitial, secondInitial;
int* p1, p2, p3; // sta je ovde problem?
```

- Код функција
  - Декларација:

```
int foo(int x);
```

  - Декларација са дефиницијом:

```
int foo(int x)
{
  return x + 1;
}
```

Досег - видљивост променљивих

- Досег променљиве одређује се на основу места декларисања.
- Променљиве могу имати три досега:
  - **Датотечки досег** - тзв. глобалне променљиве.
    - Ван било ког блока или листе параметара
  - **Блоковски досег** - тзв. локалне променљиве.

```
{
   int x;
   ...
}
```

```
void foo(int x)
{
   ...
}
```

  - **Прототипски досег** - специјалан случај концептуалне природе, али суштински небитан.

```
void foo(int x);
```

Досег - видљивост променљивих

- Две различите променљиве не могу имати исто име.
- Али:

```
int milorad;
void foo()
{
    int milorad;
    ...
}
```

- Избегавати!

Видљивост променљивих ван датотеке - затвореност

- Односи се на променљиве датотечког досега.
- **Никада се не играти овога са променљивама блоковског досега!!!**
- 1. Променљива је дефинисана у датотеци и користи се само у њој - **затворена (приватна)**

```
static int x; // definicija i deklaracija – promenljiva nije vidljiva spolja
```

- 2. Променљива је дефинисана у датотеци али се користи и у некој другој датотеци - **јавна**

```
int x; // definicija i deklaracija – promenljiva je vidljiva spolja
```

- 3. Променљива није дефинисана у датотеци, а користи се у њој (дефинисана је у некој другој датотеци) - **спољна**

```
extern int x; // samo deklaracija – x mora biti definisano negde drugde
```

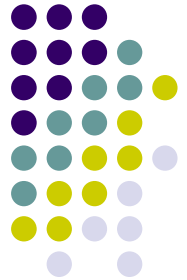
- Синтаксно је омогућено и ово у истој датотеци:

```
extern int x;
int x;
```

- А може и ово, и још штошта, али то не треба радити - само горње је корисно:

```
static int x;
int x;
```

Спољна - јавна



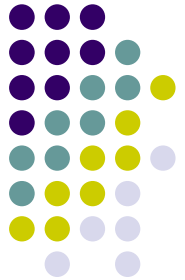
a.c

```
int a_x;
```

b.c

```
void foo()
{
...
x = a_x;
...
}
```

Спољна - јавна



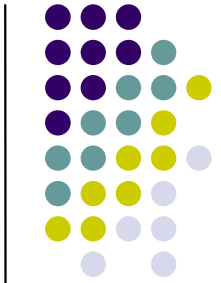
a.c

```
int a_x;
```

b.c

```
extern int a_x;

void foo()
{
...
x = a_x;
...
}
```

Спољна - јавна



a.h

```
#ifndef _A_H_
#define _A_H_

extern int a_x;

#endif
```

a.c

```
#include "a.h"


int a_x;
```

b.c

```
#include "a.h"


void foo()
{
...
x = a_x;
...
}
```

Видљивост функција ван датотеке - затвореност

- Односи се на функције датотечког досега.
- **Не правити функције блоковског досега.**
- 1. Функција је дефинисана у датотеци и користи се само у њој - **затворена (приватна)**

```
static void foo(int x)
```

- 2. Функција је дефинисана у датотеци али се користи и у некој другој датотеци - **јавна**

```
void foo(int x) // u datoteci mora biti data i njena definicija
```

- 3. Функција није дефинисана у датотеци, а користи се у њој (дефинисана је у некој другој датотеци) - **спољна**

```
void foo(int x); // data je samo deklaracija, extern nije potrebno, a ni preporucljivo!
```

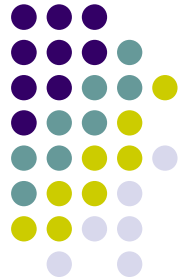
- Синтаксно је омогућено и ово у истој датотеци:

```
void foo(int x);
void foo(int x) {...}
```

- А може и ово, и још штошта, али то не треба радити - само горње је корисно:

```
static void foo(int x);
void foo(int x) {...}
```

Спољна - јавна



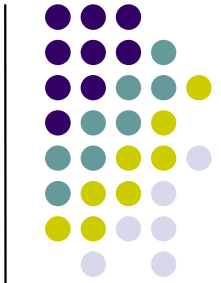
a.c

```
int a_bar()
{
...
}
```

b.c

```
int a_bar();

void foo()
{
...
x = a_bar();
...
}
```

Спољна - јавна



a.h

```
#ifndef _A_H_
#define _A_H_

int a_bar();

#endif
```

a.c

```
#include "a.h"


int a_bar()
{
...
}
```

b.c

```
#include "a.h"


void foo()
{
...
x = a_bar();
...
}
```

Трајност променљиве (објекта)

- Одређује када променљива настаје и када нестаје.
- Три врсте трајности:
  - **Аутоматска**

    Настајање и нестајање променљиве аутоматски контролише компајлер, тј. променљива настаје само у блоку где треба и нестаје чим више није потребна.

    Резервисана реч **auto**. Подразумевана трајност за променљиве блоковског досега. Променљиве датотечког досега **не могу** имати аутоматску трајност. Када настану неиницијализоване су.
  - **Статичка**

    Променљива настаје једном, на почетку програма и траје све до краја програма.

    Резервисана реч **static (али не онај статик за затвореност него неки други ☺)**. Подразумевана (и једина) трајност променљивих датотечког досега. Променљиве блоковског досега **могу** бити статичке трајности. Када настану иницијализоване су на 0, осим ако другачије није експлицитно наведено.
  - **Алоцирана**

    Настајање и нестајање променљиве одређује програмер коришћењем функција из stdlib.h. Није променљива у правом смислу јер нема име, већ се референцира преко показивача. Иницијализованост зависи од функције која је употребљена за стварање.

Табела – за променљиве

	Затворена	Јавна	Спољна
Аутоматска Глобална	Нема	Нема	Нема
Статичка Глобална	`static int x;`	`int x;`	`extern int x;`
Аутоматска Локална	`{ auto int x; }`	Нема	Нема
Статичка Локална	`{ static int x; }`	Нема	Као да нема

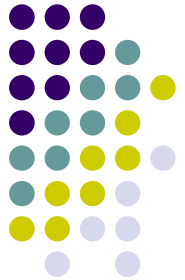
Код који је закошен се може изоставити.

Уочити два смисла `static` речи: 1) `static` наспрам `auto`, и 2) `static` као одредница да је променљива затворена (да није јавна или спољна)

Табела – за функције

	Затворена	Јавна	Спољна
~~Аутоматска~~ Глобална	`static void foo()`	`void foo() {}` Са дефиницијом.	`void foo();` Само декларација.
~~Статичка~~ Глобална			
~~Аутоматска~~ Локална	Нема (не користити)	Нема (не користити)	Нема (не користити)
~~Статичка~~ Локална			

Пример



```
/* uninitialized global variable */
int x_global_uninit;

/* initialized global variable */
int x_global_init = 1;

/* uninitialized global variable (static)*/
static int y_global_uninit;

/* initialized global variable (static) */
static int y_global_init = 2;

/* global variable that exists somewhere
 * else in the program */
extern int z_global;
```

```
int foo(int x_local)
{
  /* uninitialized local variable */
  int y_local_auto_uninit;

  /* initialized local variable */
  int y_local_auto_init = 3;

  /* uninitialized local variable */
  static int y_local_static_init;

  /* initialized local variable */
  static int y_local_static_init = 4;
}
```

Колико треба користити спољне променљиве

- У принципу, избегавање коришћења спољних променљивих представља добру програмерску праксу зато што:
  - оне нарушавају модуларност програма
  - отежавају дебаговање
- Али некада је то неопходно, па је савет да се спољне променљиве користе обазриво.

Колико треба користити јавне променљиве

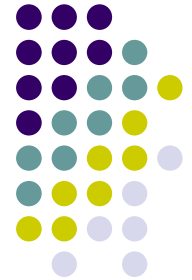
- Јавне променљиве су неопходне ако постоје одговарајуће спољне променљиве у другим датотекама.
- Али зашто би користили јавне променљиве ако не постоји одговарајућа декларација спољне променљиве у некој другој датотеци?

  Одговор: Из незнања или аљкавости.
- Ако се глобална променљива неће користити негде другде ван датотеке онда је добра пракса декларисати је као затворену (приватну).
- Ако се то не уради могу настати и проблеми: Шта ако у две датотеке имамо јавну променљиву истог имена?

Где декларисати/дефинисати и иницијализовати променљиве?

- Променљиве треба дефинисати искључиво у .c датотекама, никако у заглављима.
- То урадити при врху датотеке (обично одмах после укључивања заглавља) у случају глобалних променљивих.
- У случају локалних променљивих најпрепоручљивије место за декларацију је почетак функције, али могу се декларисати и касније (пожељно на почетку блока, иако од C99 то више није обавезно)
- Променљиве иницијализовати чим пре. Ако има смисла - чим су декларисане.

Квалификатори



- `volatile` - значи да одређена променљива може бити промењена, а да је програмски код није ни пипнуо. Ово се готово не сусреће нигде осим у програмирању наменских уграђених система.
- `const` - значи "само за читање" (read-only) од стране програма, односно програм вредност не може мењати.

Уочимо разлику:

1. **`const char* p`**
2. **`char const* p`**
3. **`char* const p`**

1. и 2. су исте ствари: декларација показивача на `const char`. То значи да се показивач може мењати али оно на шта показује не може.

3. декларише константни показивач на `char`. Дакле, сам показивач се не може мењати од стране програма (али оно на шта он показује - може).

Константне променљиве ☺

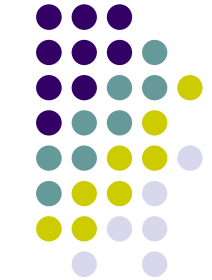
- Декларација константне променљиве иста као и декларација обичне променљиве - постојање резервисане речи `const` једина разлика.
- Реч `const` може бити са било које стране типа:

```
int const a = 1;
const int a = 2;
```

Ствар стила. Изабрати једну варијанту и држати се ње. Препорука је друга варијанта.

- Морају бити иницијализоване при декларацији јер не могу на други начин променити вредности из програма.
- #define је алтернативни начин дефинисања константи у програму, али постоје важне разлике.

#define насупрот const



```
#define SOME_CONST 1

const int some_const = 1;
```

Разлике?

- Константна променљива може бити сложеног типа
- Константна променљива има адресу и физичку представу у меморији, ако је потребна (компајлер је може уклонити ако није потребна)

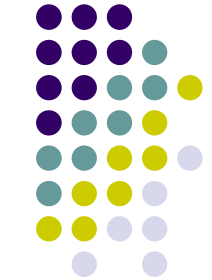
  ```
  int* x = &some_const;
  ```

  Затвореност у оба дата случаја?

  Избегавају се потенцијалне грешке, нпр.:

  ```
  #define SOME_CONST 5 + 2
  const int some_const = 5 + 2;
  x = 7 * SOME_CONST; y = 7 * some_const;
  ```

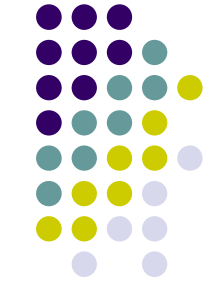
Пример за volatile



```
volatile int _flag;

void wait_for_flag()
{
  while(_flag == 0);
}
```

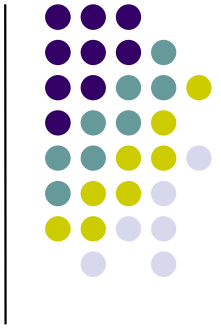
```
volatile int x;

int foo()
{
   x = 5;
   return x;
}

int foo1()
{
   //x = 5;
   return 5;
}
```

Пример за volatile



```
volatile uint8_t* flag = (uint8_t volatile*)0x12345678;
```

const volatile?



```
volatile uint8_t* flag = (uint8_t volatile*)0x12345678;

volatile uint8_t* const flag
    = (uint8_t volatile*)0x12345678;

volatile const uint8_t* const flag
    = (uint8_t const volatile*)0x12345678;

const uint8_t volatile * const flag
    = (volatile uint8_t const*)0x12345678;
```