ΗΡΥ 201
ΨΗΦΙΑΚΟΙ ΥΠΟΛΟΓΙΣΤΕΣ

Εισαγωγή

Ν. Αλαχιώτης

Αναφορές

- Ιστοσελίδα : **courses.ced.tuc.gr** : Ελέγχετε συχνά την σελίδα του μαθήματος για ανακοινώσεις, και πληροφορίες σχετικά με το μάθημα και το εργαστήριο.
  (ΣΗΜΑΝΤΙΚΟ: ΕΓΓΡΑΦΗ ΣΤΟ ΜΑΘΗΜΑ!!!)

- "Οργάνωση και Σχεδίαση Υπολογιστών" , David A. Patterson, John L. Hennessy

- SPIM: www.cs.wisc.edu/~larus/spim.html

- Σημειώσεις καθηγητή (courses)

Γιατί ΗΡΥ 201?

- Απομυθοποίηση του «μαύρου κουτιού» Η/Υ
- Να αποκτήσετε αρκετές γνώσεις για να καταλαβαίνετε πώς λειτουργεί ένας υπολογιστής
- Υπόβαθρο για άλλα μαθήματα:
  - Μεταφραστές
  - Οργάνωση Υπολογιστών
  - Ενσωματωμένα συστήματα
  - κ.α.

ΠΡΟΑΠΑΙΤΟΥΜΕΝΗ ΓΝΩΣΗ

  - Γλώσσα προγραμματισμού υψηλού επιπέδου (C, ...)
  - (Πολύ) απλές γνώσεις φυσικής, μαθηματικών, κλπ

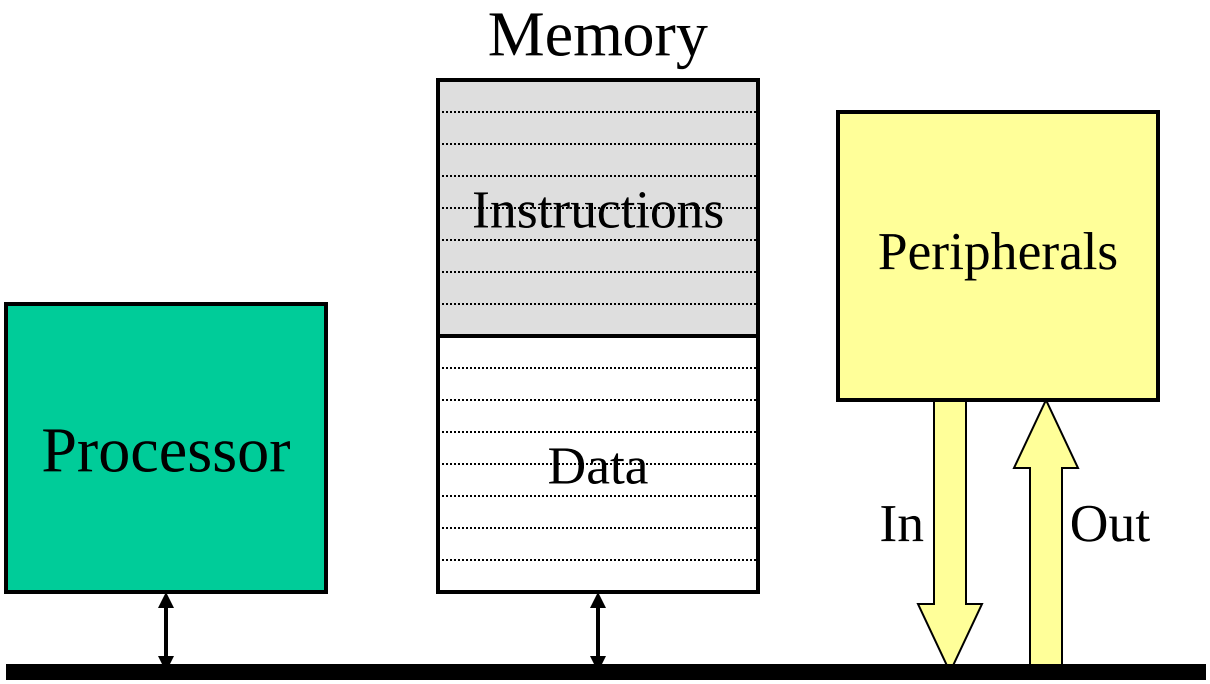
Υλη Μαθήματος

- Ο υπολογιστής σε αφηρημένο (abstract) επίπεδο
- Αναπαράσταση/**αποθήκευση** δεδομένων σε υπολογιστές
  - Ακέραιοι, σταθερή και κινητή υποδιαστολή, πράξεις (+, -, ...)
  - Συμβολοσειρές, Στοίβα, κ.α.
- Κλήση διαδικασιών και συναρτήσεων
- Γλώσσα C «χαμηλού» επιπέδου
- Γλώσσα μηχανής και συμβολομεταφραστή (assembly)
- Διαδικασία παραγωγής εκτελέσιμου προγράμματος
- Απλό μοντέλο διασύνδεσης επεξεργαστή με περιφεριακές συσκευές
- Διακοπές και εξαιρέσεις

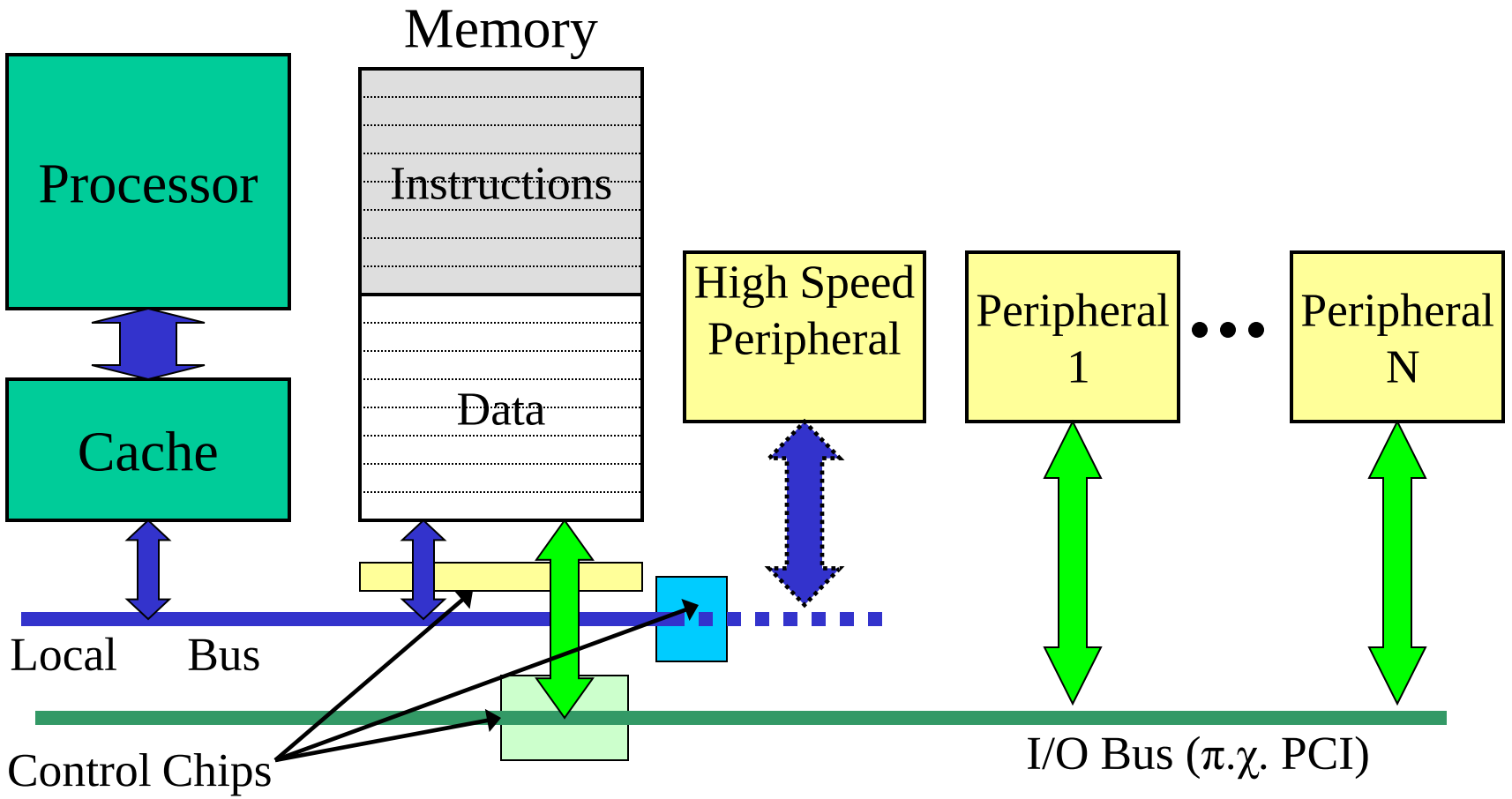
Συνήθης Δομή Η/Υ

- Κύρια πλακέτα (μητρική) με τα βασικά εξαρτήματα: Επεξεργαστή, Μνήμη, απλά περιφερειακά, Δίαυλος Επέκτασης (PCI, κ.α.)
- Κάρτες Περιφερειακών για:
  - Επεκτασιμότητα
  - Κόστος ανάλογο απαιτήσεων (τιμή, λειτουργικότητα, ...)
- Ελεγκτές άλλων Διαύλων Επέκτασης
  - SATA, IDE, SCSI για δίσκους

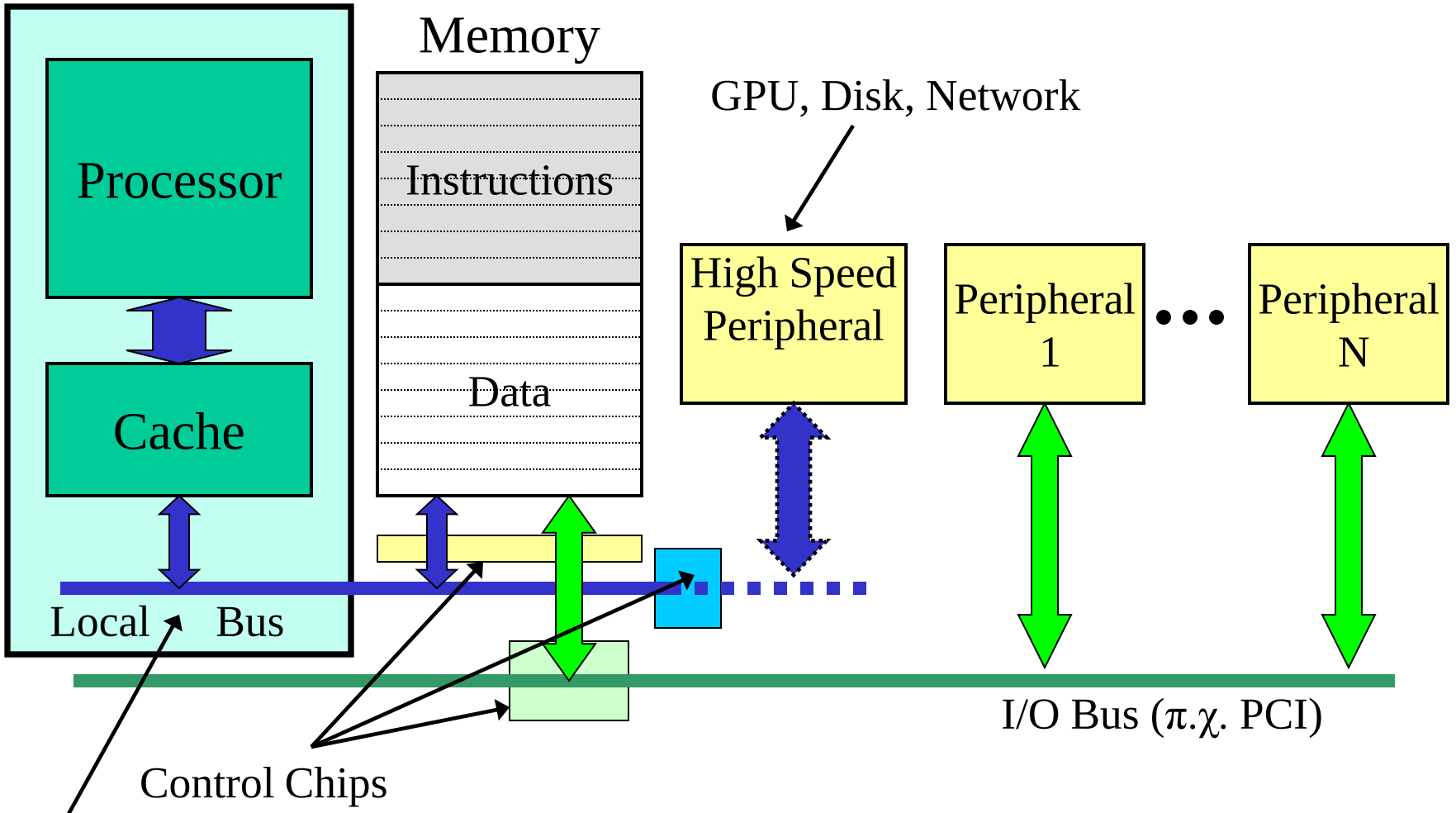
Απλό Μοντέλο Υπολογιστή



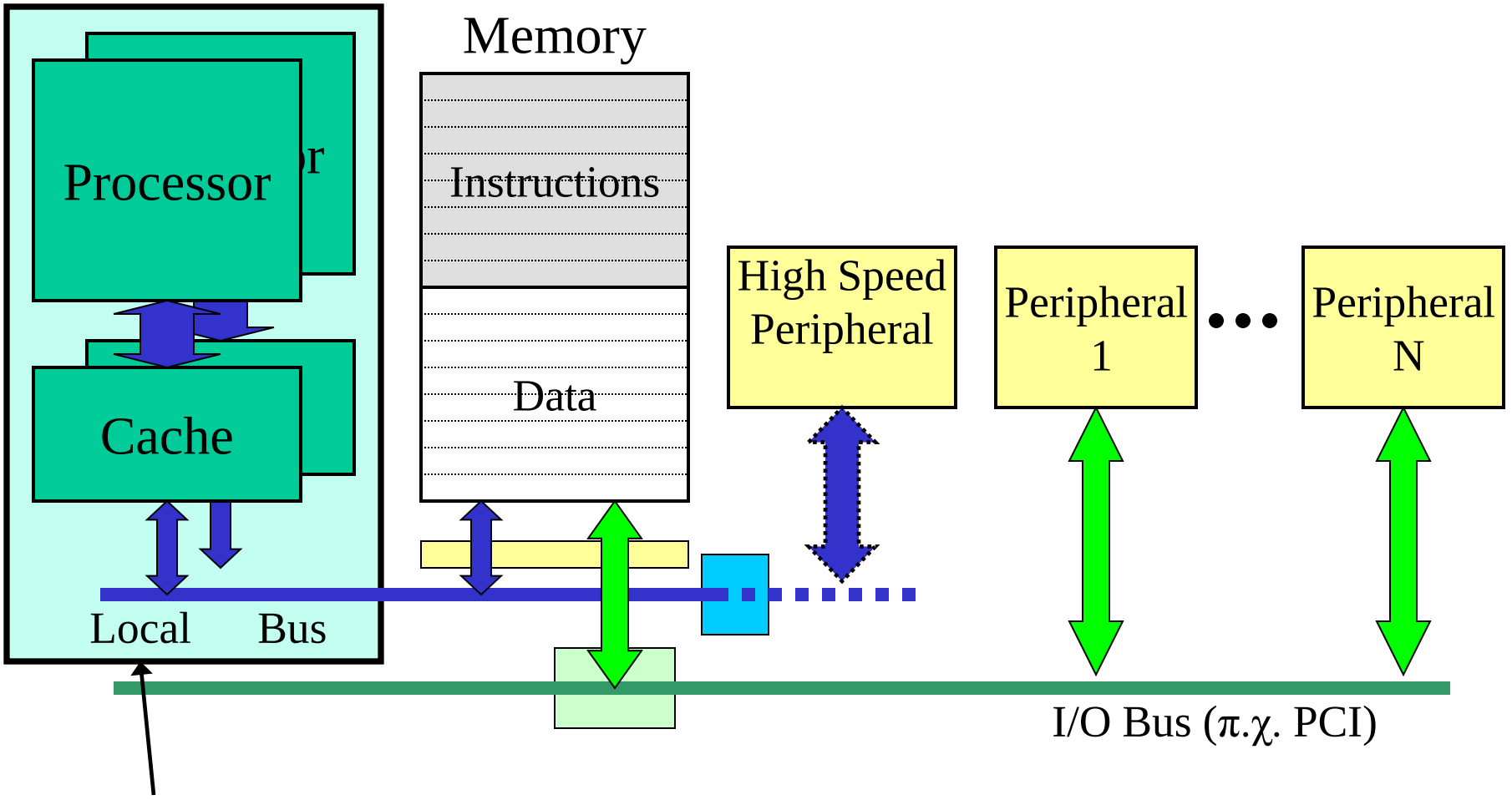
(Πιο) Ρεαλιστικό Μοντέλο Υπολογιστή



(Πιο+) Ρεαλιστικό Μοντέλο Υπολογιστή



(Πιο++) Ρεαλιστικό Μοντέλο Υπολογιστή



Αφαίρετική σκέψη (Abstraction)

- Communications of the ACM, April 2007

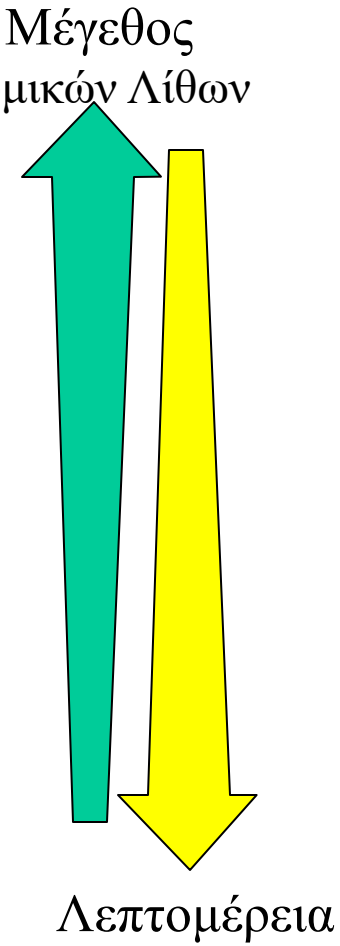
IS ABSTRACTION THE KEY TO COMPUTING?

Why is it that some software engineers and computer scientists are able to produce clear, elegant designs and programs, while others cannot? Is it possible to improve these skills through education and training? Critical to these questions is the notion of abstraction.

By JEFF KRAMER

Επίπεδα Αφαίρεσης (Abstraction)

- Πότε αρκεί το «μαύρο κουτί» και πότε πρέπει να σκεφτόμαστε τη (σύνθετη) πραγματικότητα;
  - 1 μαυρο κουτί
  - Λίγα κουτάκια (επεξεργαστής, μνήμη, περιφεριακά)
  - Επεξεργαστής, μνήμη, κάρτες επέκτασης, δίαυλοι, ...
  - Λογικά κυκλώματα, ολοκληρωμένα εξαρτήματα, τάσεις, ρεύματα
  - Τρανζίστορς, δίοδοι, πυκνωτές αντιστάσεις, ...
  - Φυσική στερεάς κατάστασης, ηλεκτρόνια, κβαντικά φαινόμενα...



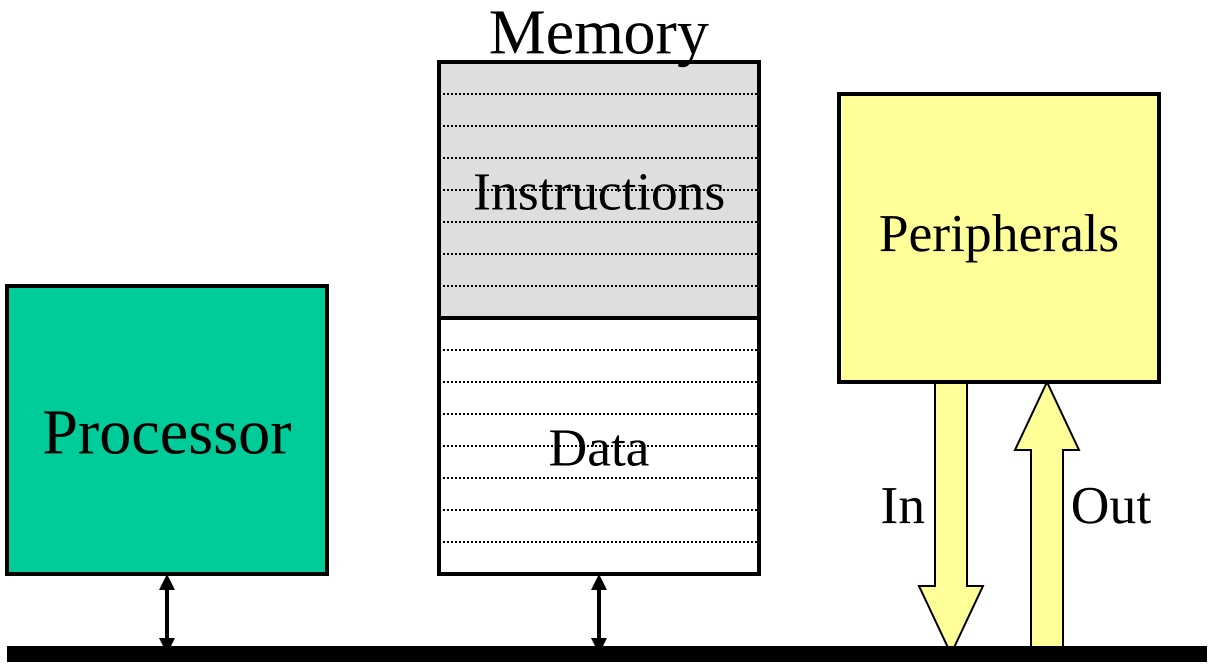
Επίπεδα Αφαίρεσης (Abstraction) 2

- Θέλουμε αρκετή πληροφορία ώστε να μην υπεραπλουστεύουμε το μοντέλο
- Πρέπει να «κρύψουμε» όση περισσότερη πληροφορία μπορούμε για απλότητα

➢ Επιλογή κατάλληλου επίπεδου αφαίρεσης
  - ➢ Απλό, και περιγραφικό!
  - ➢ Περιέχει «ικανές» πληροφορίες για να εκτιμούμε τις «επιδόσεις» και να αποφασίζουμε σωστά

Μοντέλο Υπολογιστή HPY201

- Ενα απλό μοντέλο Η/Υ είναι αρκετό για τους περισσοτερους σκοπούς του HPY 201!



Επίπεδα Αφαίρεσης σε Η/Υ

- Εφαρμογές
- Βιβλιοθήκες
- Λειτουργικό/Μεταφραστές
- Συμβολομεταφραστές
- Κώδικας Μηχανής

Διεπαφή S/W, H/W

- Επεξεργαστής, Μνήμη, Περιφερειακά
- Καταχωρητές, αθροιστές, πύλες
- Τρανζίστορ
- Ηλεκτρόνια, πεδία
- Κβαντικά φαινόμενα

Επεξεργαστο-κεντρική θεώρηση

- Στο μάθημα βλέπουμε τον κόσμο «μέσα» από τον επεξεργαστή:
  - Είσοδος: μεταφορά πληροφορίας **από** τον έξω κόσμο
  - Εξοδος: μεταφορά πληροφορίας **πρός** τον έξω κόσμο
- Μνήμη: πίνακας 2^n θέσεων, Mem[0.. 2^n –1]
- Υπόλοιπος κόσμος: ένα σύννεφο εισόδου/εξόδου

C Language and Memory

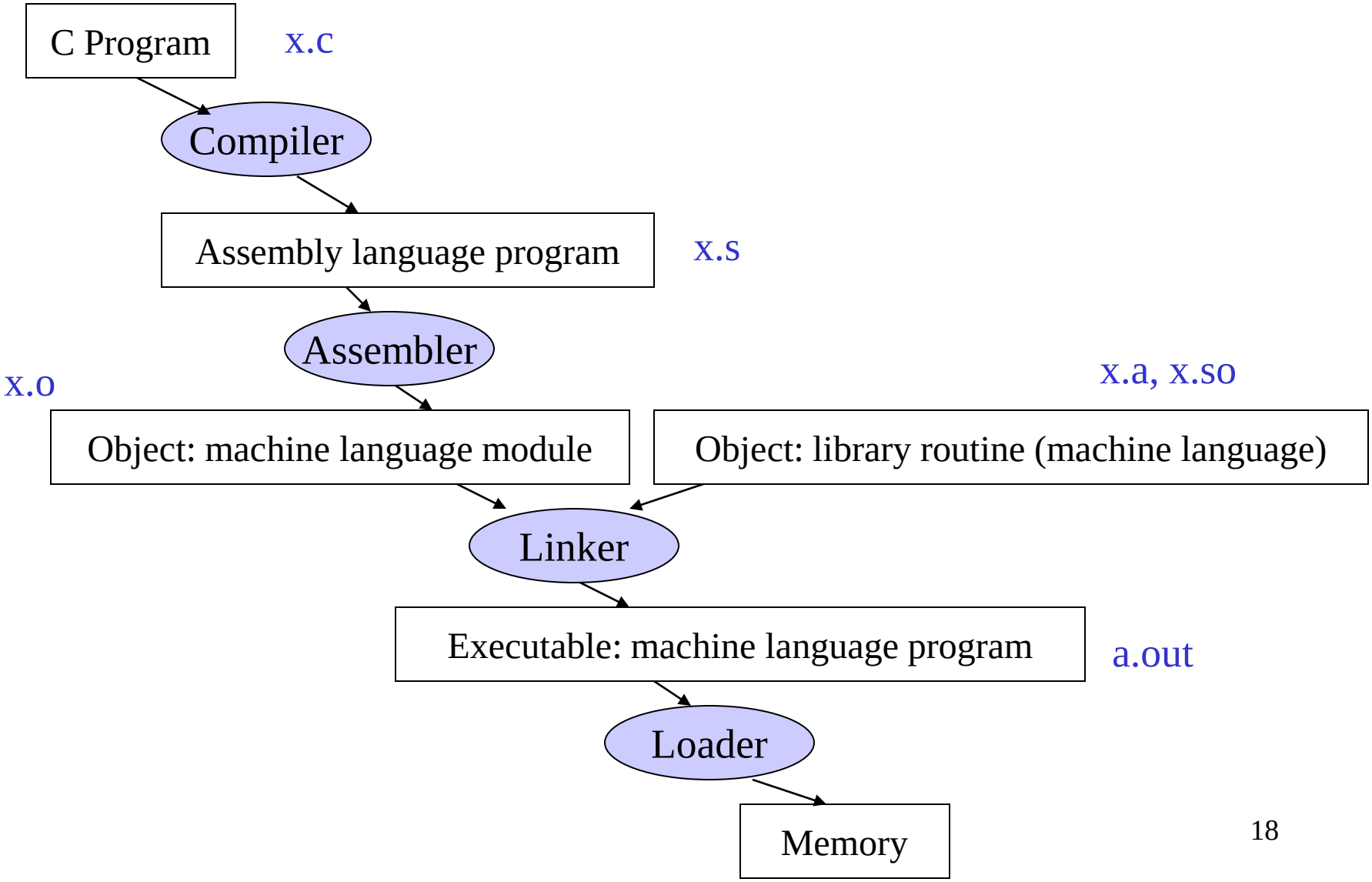
Example

Convert to assembly:

```
void strcpy (char x[], char y[])
{
    int i;
    i=0;
    while  ((x[i] = y[i]) != `\0' )
       i += 1;
}
```

```
strcpy:
 addi     $sp, $sp, -4
 sw       $s0, 0($sp)
 add      $s0, $zero, $zero
L1: add   $t1, $s0, $a1
 lb       $t2, 0($t1)
 add      $t3, $s0, $a0
 sb       $t2, 0($t3)
 beq      $t2, $zero, L2
 addi     $s0, $s0, 1
 j        L1
L2: lw    $s0, 0($sp)
 addi     $sp, $sp, 4
 jr       $ra
```

Starting a Program



C operator priorities

Precedence	Operator	Description	Associativity
1	++ --	Suffix/postfix increment and decrement	Left-to-right
	()	Function call	
	[]	Array subscripting	
	.	Structure and union member access	
	->	Structure and union member access through pointer	
	(*type*){*list*}	Compound literal(C99)	
2	++ --	Prefix increment and decrement	Right-to-left
	+ -	Unary plus and minus	
	! ~	Logical NOT and bitwise NOT	
	(*type*)	Type cast	
	*	Indirection (dereference)	
	&	Address-of	
	sizeof	Size-of	
	_Alignof	Alignment requirement(C11)	

C operator priorities

Precedence	Operator	Description	Associativity
3	* / %	Multiplication, division, and remainder	Left-to-right
4	+ -	Addition and subtraction	
5	<< >>	Bitwise left shift and right shift	
6	< <=	For relational operators < and ≤ respectively	
	> >=	For relational operators > and ≥ respectively	
7	== !=	For relational = and ≠ respectively	
8	&	Bitwise AND	
9	^	Bitwise XOR (exclusive or)	
10	|	Bitwise OR (inclusive or)	
11	&&	Logical AND	
12	||	Logical OR	

C operator priorities

Precedence	Operator	Description	Associativity
13	?:	Ternary conditional	Right-to-Left
14	=	Simple assignment	
	+= -=	Assignment by sum and difference	
	*= /= %=	Assignment by product, quotient, and remainder	
	<<= >>=	Assignment by bitwise left shift and right shift	
	&= ^= |=	Assignment by bitwise AND, XOR, and OR	
15	,	Comma	Left-to-right

Motivation

- Real programs use: **integers, chars, floats, doubles**.
- Real programs use abstractions like **arrays, lists, trees, stacks, and queues**.
- The data associated with these abstractions must be **ordered** within memory.
- Assembly language does not provide convenient ways to access data in memory like high-level languages do.
- It is necessary to calculate explicitly the locations of data within the structure.

C peculiarities

- Conditional expressions:

```
e = a < d ? a++ : a = d;
e = ( ((a < d) ? (a++) : a) = d ) ??? Non standard!
```

- Short-circuit evaluation:

```
if ( a(param1) || b(param2) ) { /* then *? } else {
/( else */ }
```

a() will be called. b()? ONLY if a(param) evaluates to false

Example:

if ((a!=0) && printf("%d\n", a)) ;

if (a != 0) printf("%d\n", a);

- pointer arithemtic

*(p + 3) ???

MEMORY ALLOCATION (I)

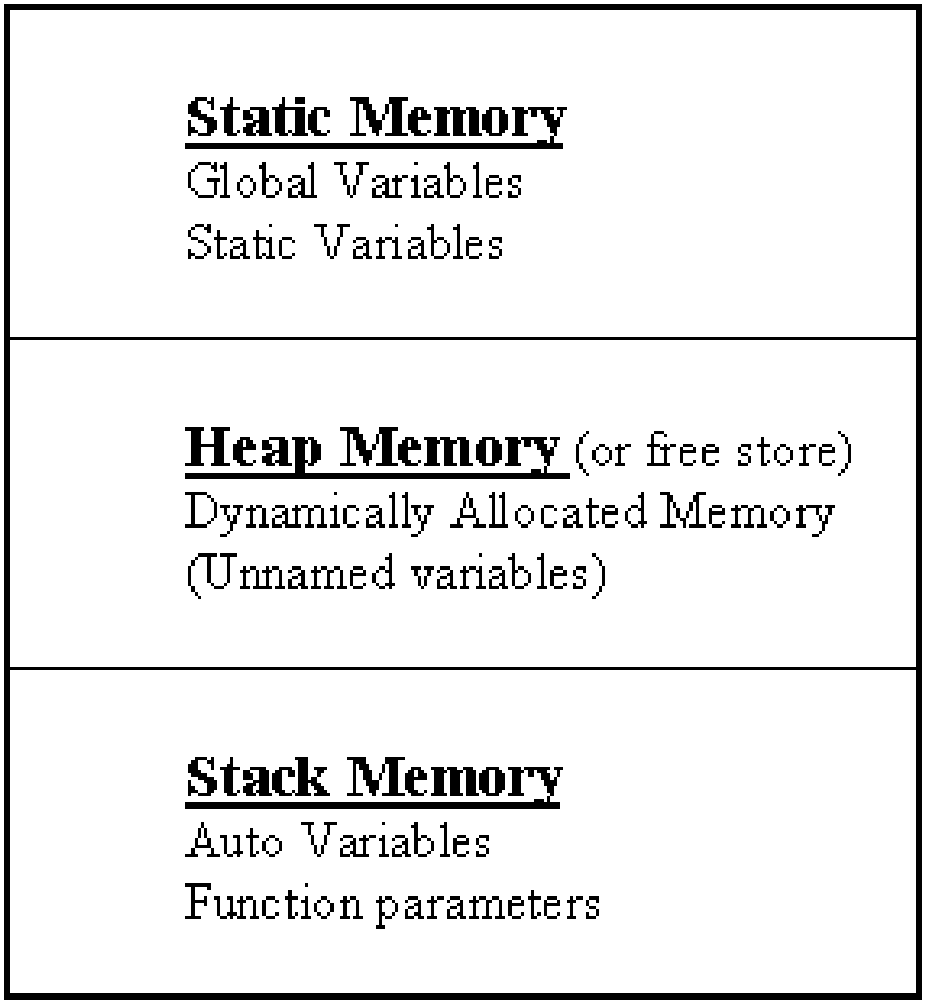
- Memory is allocated for variables in two ways:
    - Static (or at compile-time)
    - Dynamic (or at run-time)

Memory Allocation (II)

In C and C++, three types of memory are used by programs:

- **Static memory -** where global and static variables live
- **Heap memory -** dynamically allocated at execution time

  - "managed" memory accessed using pointers
- **Stack memory -** used by automatic variables



Memory allocation of variables (II)

- **STATIC DATA** : One region of memory is reserved for **static data**, variables that are never created or destroyed as the program runs, such as constants variables.
- Static Memory is allocated at **compile time.**

Dynamic Memory Allocation

- **HEAP** : When a new object is created, C/Java allocates space from a pool of memory called the **heap**.
- *In C*, functions such as **malloc()** are used to dynamically allocate memory from the **Heap**.
- *In C++*, this is accomplished using the **new** and **delete** operators
- **new** is used to allocate memory during execution time
  - returns a pointer to the address where the object is to be stored
  - always returns a pointer to the type that follows the **new**

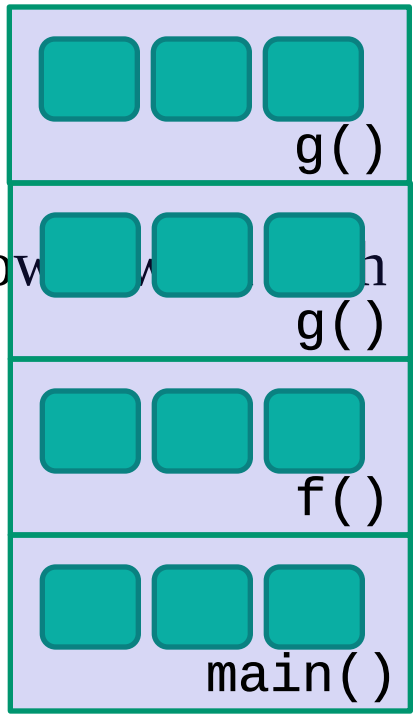
Overview of memory management

- Stack-allocated memory
  - When a function is called, memory is allocated for all of its parameters and local variables.
  - Each active function call has memory on the stack (with the current function call on top)
  - When a function call terminates,

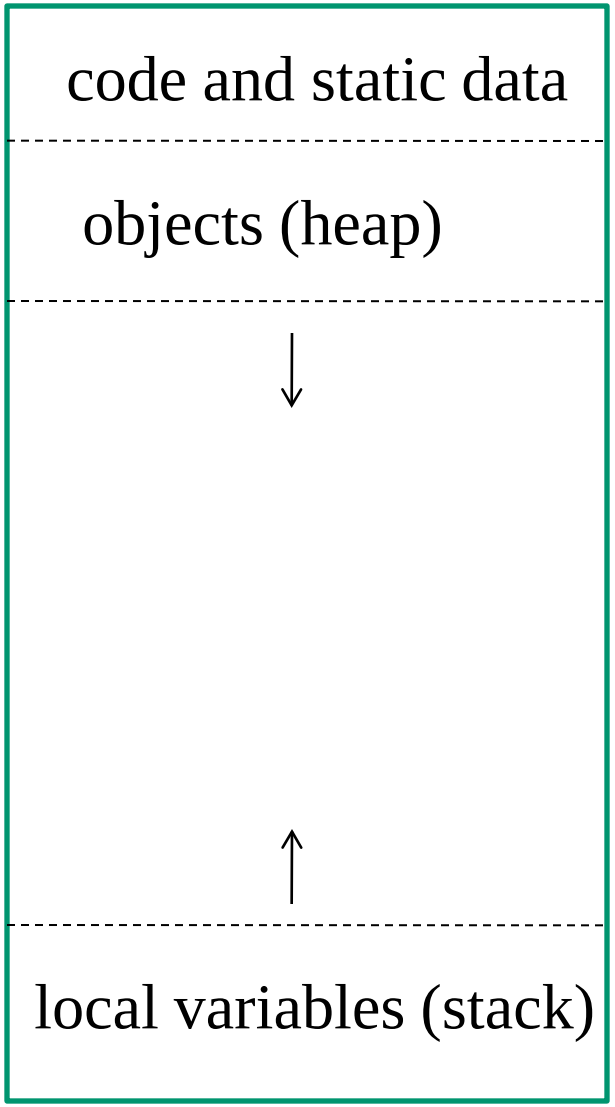
    the memory is deallocated ("freed up")
- In classical architectures, the stack and heap grow [illegible] other for flexibility.
- Ex: `main()` calls `f()`,

  `f()` calls `g()`

  `g()` recursively calls `g()`



Memory allocation of variables (I)



LOW Address

HIGH Address

Memory

- A *memory cell* is a unit of memory with a unique address.
- The entire memory is a collection of memory cells.
- Each memory cell holds eight bits, or 1 byte.
- A word designates the amount of memory used to store an integer, e.g., 1 word = 4 bytes

Byte Addressing

- A method of addressing where each byte has a unique address
- The ordering of the bytes within the word can either be *little-endian* or *big-endian*.
- Little-endian numbers the bytes from the least significant to the most signifcant.
- Big-endian numbers the bytes from the most significant to the least signifcant.
- suppose we have a 32 bit quantity, written as $90AB12CD_{16}$

Big Endian

Address	Value
1000	90
1001	AB
1002	12
1003	CD

Little Endian

Address	Value
1000	CD
1001	12
1002	AB
1003	90

Storing an integer

- Assume a 32-bit word. The number of bits used to store an integer is 32 or 4 bytes. It can also be thought of as an array of 4 bytes.
- By convention, the smallest of the four byte addresses is used to indicate the address of the word.
- The smallest of the byte addresses in a word must be a multiple of four -- **words are aligned**

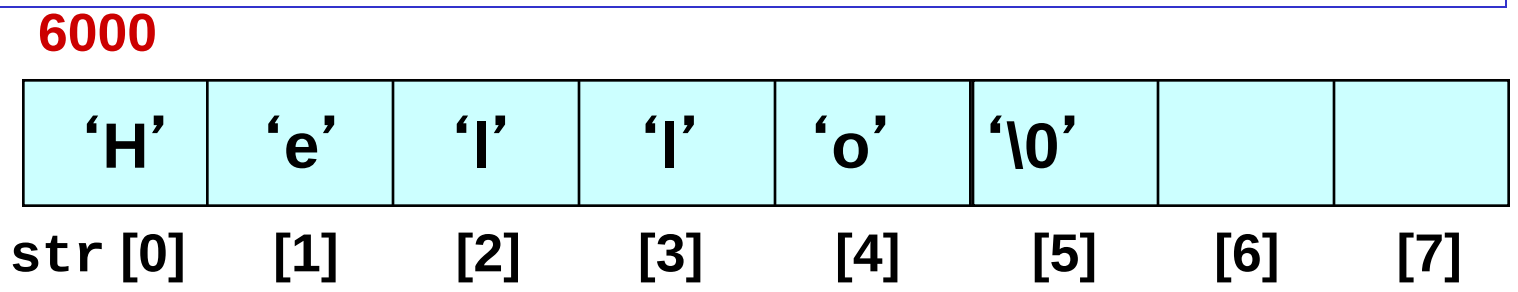
Array

- The array is the most important and most general data structure.
- All other data structures can be implemented using an array.
- The computer memory itself is organized as a large, one-dimensional array.
- All uses of the memory are simply allocations of part of this gigantic array.

Recall that . . .

char str [8];

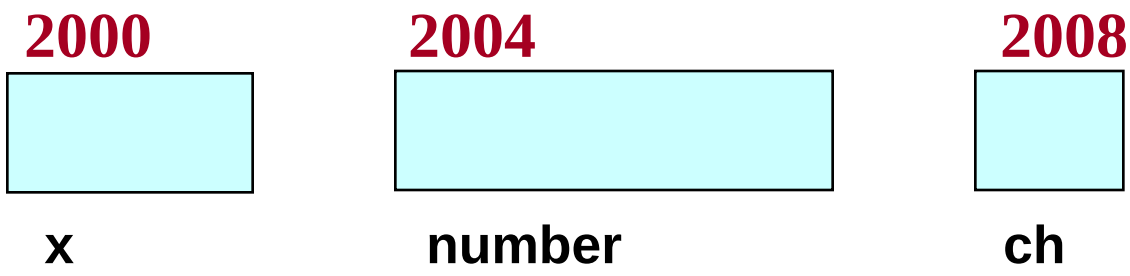
- **`str`** is the base address of the array.
- We say **`str`** is a pointer because its value is an address.
- It is a pointer constant because the value of **`str`** itself cannot be changed by assignment. It "points" to the memory location of a `char`.



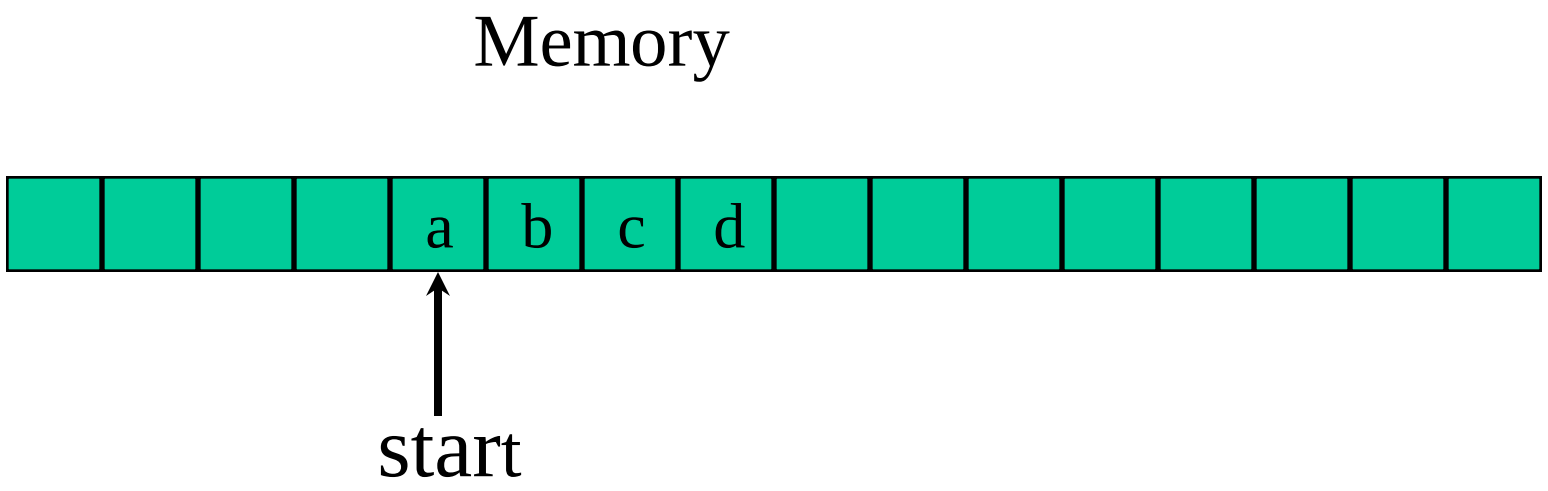
Addresses in Memory

- When a variable is declared, enough memory to hold a value of that type is allocated for it at an unused memory location. This is the address of the variable

```
int     x;
float   number;
char    ch;
```



1D Array Representation In Java, C, and C++



- 1-dimensional array x = [a, b, c, d]
- map into contiguous memory locations

- location(x[i]) = start + i

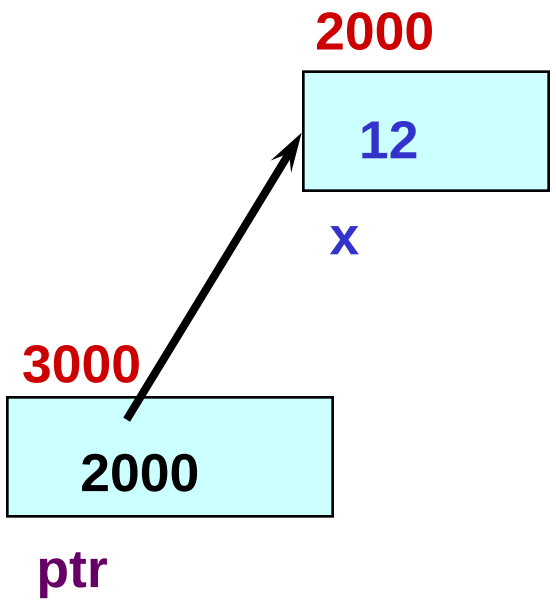
What is a pointer variable?

- A pointer variable is a variable whose value is the address of a location in memory.
- To declare a pointer variable, you must specify the type of value that the pointer will point to, for example,

```
int*   ptr; // ptr will hold the address of an int

char*  q;   // q will hold the address of a char
```

Using a Pointer Variable

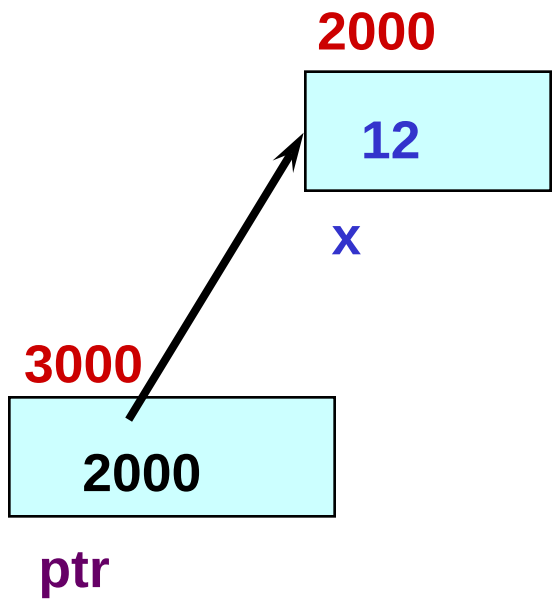
```
int  x;
x = 12;

int*  ptr;
ptr = &x;
```



NOTE: **Because ptr holds the address of x, we say that ptr "points to" x**

*: dereference operator

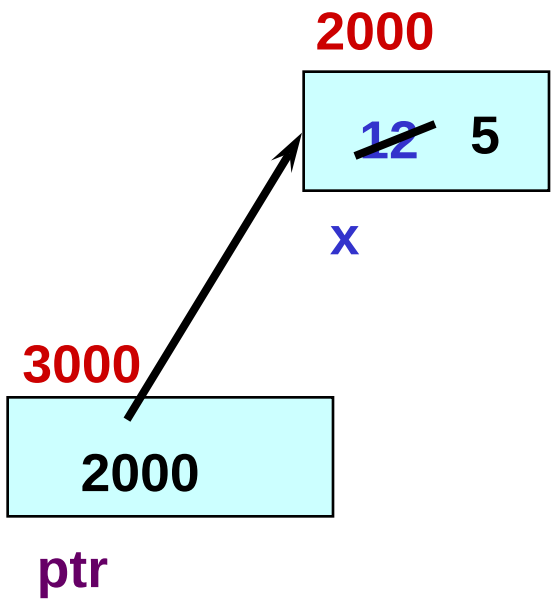
```
int  x;
x = 12;


int*  ptr;
ptr = &x;

cout  <<  *ptr;
```



NOTE: **The value pointed to by ptr is denoted by *ptr**

Using the Dereference Operator

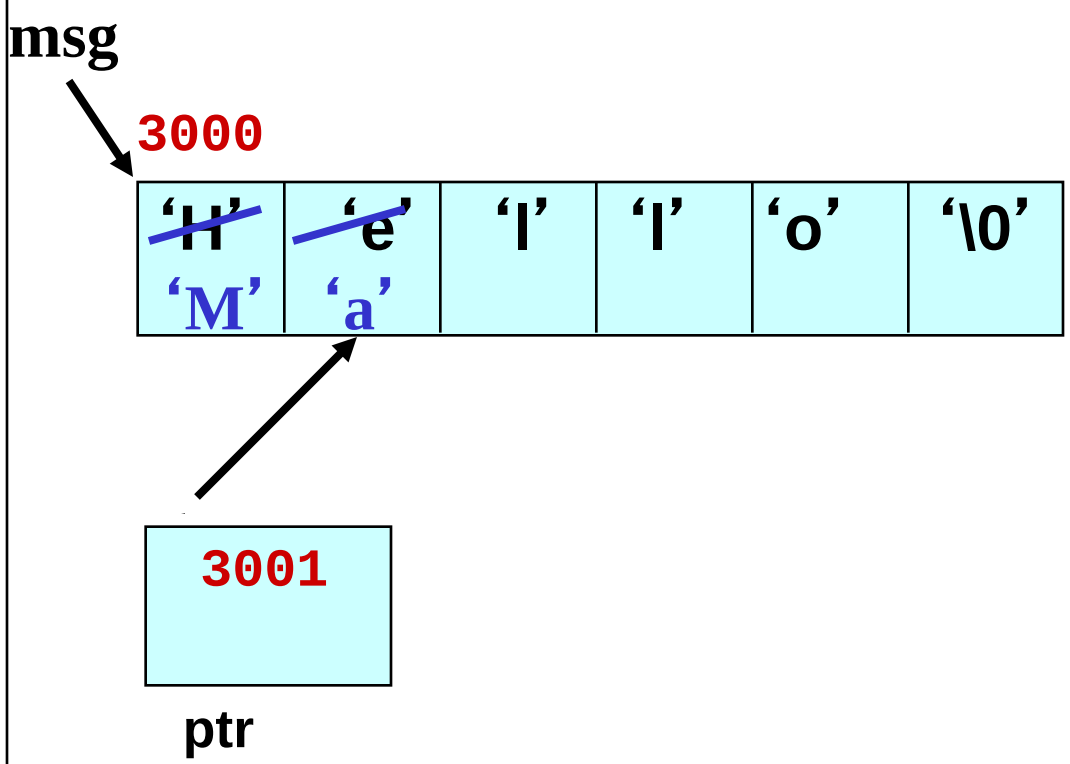
```
int  x;
x = 12;

int*  ptr;
ptr = &x;

*ptr = 5;
```




```
// changes the value at the
   address ptr points to 5
```

Using a Pointer to Access the Elements of a String

```
char   msg[ ] =“Hello”;
char*  ptr;
ptr  =  msg;
*ptr  =  ‘M’  ;
ptr++;
*ptr =  ‘a’ ;
```



2D Arrays

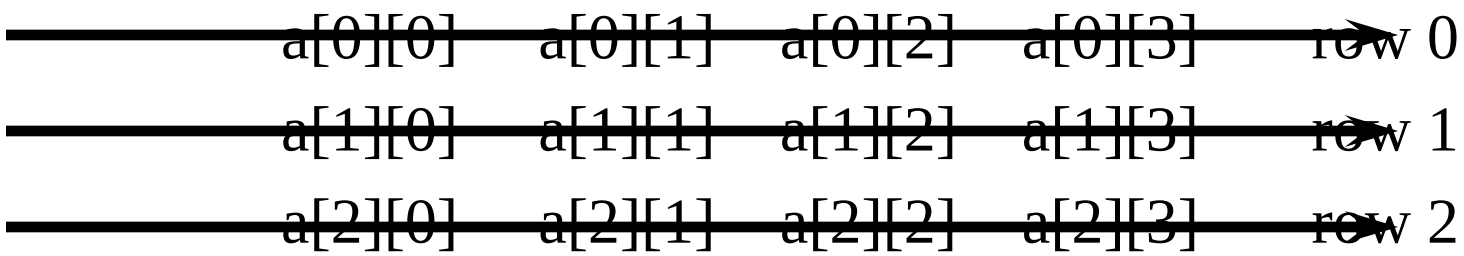
The elements of a 2-dimensional array a declared as:

int a[3][4];

may be shown as a table

a[0][0]	a[0][1]	a[0][2]	a[0][3]
a[1][0]	a[1][1]	a[1][2]	a[1][3]
a[2][0]	a[2][1]	a[2][2]	a[2][3]

Rows Of A 2D Array



Row-major (C, C++, Java?)

Column-major (Fortran, ???)

Row-Major Mapping

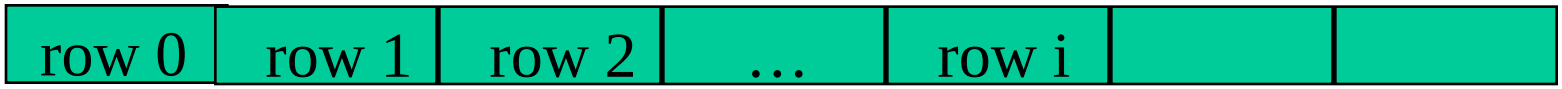
- Example 3 x 4 array:

a b c d

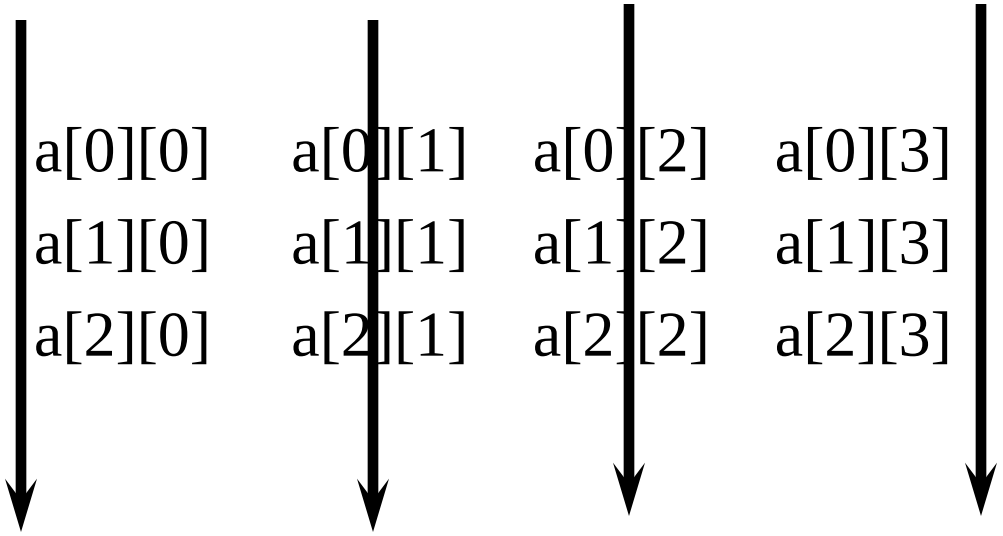
e f g h

i j k l

- Convert into 1D array y by collecting elements by rows.
- Within a row elements are collected from left to right.
- Rows are collected from top to bottom.
- We get y[] = {a, b, c, d, e, f, g, h, i, j, k, l}



Columns Of A 2D Array



Column-Major Mapping

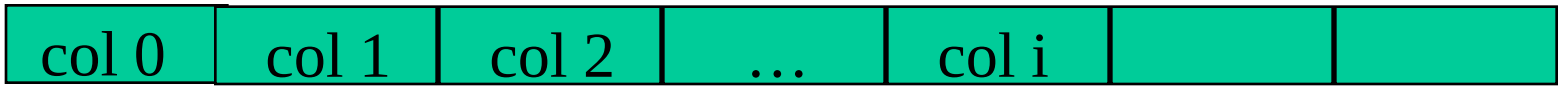
Example 3 x 4 array:

a b c d

e f g h

i j k l

- Convert into 1D array y by collecting elements by columns.
- Within a column elements are collected from top to bottom.
- Columns are collected from left to right.
- We get y = {a, e, i, b, f, j, c, g, k, d, h, l}



Ψηφιακοί Υπολογιστές

Δυαδική Αριθμητική

Συστήματα Αρίθμησης

- Φυλακισμένοι: I, II, III, IIII, ~~IIII~~ , ...
  - η θέση των I δεν έχει σημασία για το μέγεθος του αριθμού
- Λατινικό: I, II, III, IV, V, VI, … IX, X, XI, …
  - η σχετική θέση συμβόλων στον αριθμό αλλάζει τη σημασία τους και μάλιστα σε συνάρτηση με τα άλλα ψηφία
- Αραβικό: XYZ με βάση το 10 σημαίνει:
  - $X * 10^2 + Y * 10^1 + Z * 10^0$
  - Γενικά: ο αριθμός με n δεκαδικά ψηφία:
    $d_{n-1}d_{n-2}\ldots d_1d_0$ υπολογίζεται από το άθροισμα: $\sum_{i=0}^{n-1} d_i * 10^i$
- Με βάση (Base, radix) το 10 τιμές ψηφίων 0...9 (σύνολο 10 ψηφία)
  περισσότερο σημαντικό ψηφίο (MS digit) => μεγαλύτερος εκθέτης
  λιγότερο σημαντικό ψηφίο (LS digit) => μικρότερος εκθέτης

Άλλες βάσεις αριθμών

- Δυαδικό: βάση το 2, ψηφία 0, 1
  - Παράδειγμα: $010101_2 = 21_{10}$
- Οκταδικό: βάση το 0, ψηφία 0, 1,...,7 (σύνολο 8 ψηφία)
  - Παράδειγμα: $0127_8 = 1*8^2 + 2*8^1 + 7*8^0 = 64+16+7 = 87_{10}$
- Δεκαεξαδικό: βάση το 16, ψηφία 0, 1,...9, A, B, C, D, E, F (σύνολο 16 ψηφία)
  - Παράδειγμα: $1B3_{16} = 1*16^2 + B*16^1 + 3*16^0 =$ $1*256+11*16+3*1 = 435_{10}$
  - Αντιστοιχία δεκαεξαδικών ψηφίων:
    A = 10, B = 11, C = 12, D = 13, E = 14, F = 15
  - Στην γλώσσα C:
    x = 0xf2; 0x0000, 0xFFFFFFFF, 0xDEADBEEF, 0xBAADF00D, 0xCAFEBABE (Wikipedia “hexspeak” για άλλα...)

Μετατροπή Δεκαδικό => Άλλη βάση

- Ο γενικός τρόπος για μετατροπή θετικών αριθμών από μιά βάση σε μια άλλη είναι μια ακολουθία διαιρέσεων. Π.χ. για τη μετατροπή του 171_{10} σε δεκαεξαδικό.
  - Διαιρούμε το 171 με το 16 (υπόλοιπο 11, πηλίκο 10).
  - Διαιρούμε το πηλίκο 10 με το 16 (υπόλοιπο 10, πηλίκο 0)
  - Η διαδικασία σταματάει όταν το πηλίκο φτάσει στο μηδέν.
  - Η αναπαράσταση του αριθμού στην νέα βάση είναι η ακολουθία υπόλοιπων των διαιρέσεων ξεκινώντας από το τέλος προς την αρχή: 10, 11 δηλαδή με δεκαεξαδικά ψηφία $171_{10} = AB_{16}$. Πραγματικά: 10*16 + 11 = 160 + 11 = 171.

Μετατροπή Δεκαδικό => Δυαδικό

- Όμοια, η μετατροπή από δεκαδικό σε δυαδικό γίνεται με μια ακολουθία διαιρέσεων. Π.χ. Προσπαθούμε να μετατρέψουμε το 28_{10} σε δυαδικό .
  - Διαιρούμε το 28 με το 2 (υπόλοιπο 0, πηλίκο 14) (LS bit)
  - Διαιρούμε το 14 με το 2 (υπόλοιπο 0, πηλίκο 7)
  - Διαιρούμε το 7 με το 2 (υπόλοιπο 1, πηλίκο 3)
  - Διαιρούμε το 3 με το 2 (υπόλοιπο 1, πηλίκο 1)
  - Διαιρούμε το 1 με το 2 (υπόλοιπο 1, πηλίκο 0) (MS bit)
  - Η διαδικασία σταματάει όταν το πηλίκο φτάσει στο μηδέν.
  - Η αναπαράσταση του αριθμού 28_{10} σε δυαδικό είναι 11100_2
  - Πραγματικά: $1*2^4 + 1*2^3 + 1*2^2 + 0*2^1 + 0*2^0 = 28_{10}$.

Κλασματικοί Αριθμοί

- Πως μπορούμε να αναπαραστήσουμε δεκαδικούς αριθμούς όπως το 3,14 ?
- Πως μπορούμε να αναπαραστήσουμε πολύ μικρούς αριθμούς όπως το 0.00000000000000000000045691?
- Πως μπορούμε να αναπαραστήσουμε πολύ μεγάλους αριθμούς όπως το 9.349.398.989.787.762.244.859.087 ?

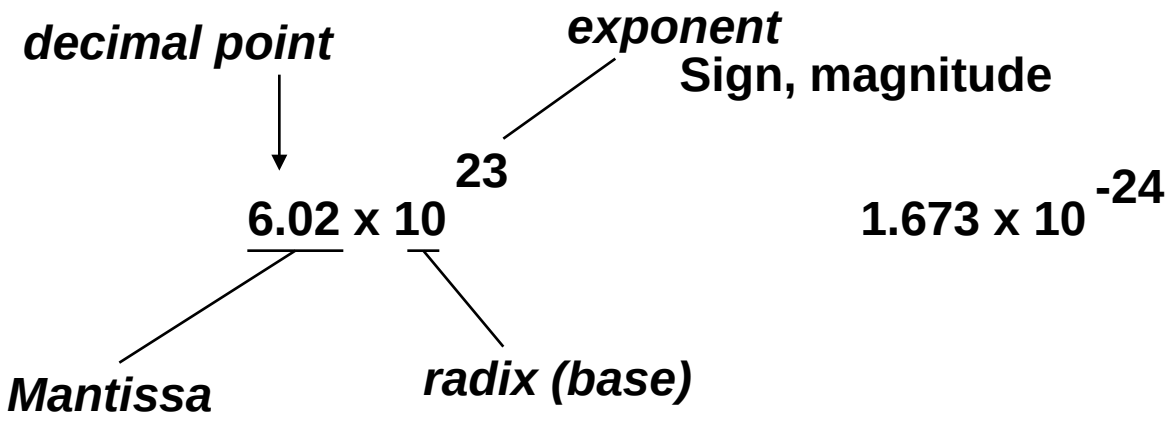
Αριθμοί Σταθερής Υποδιαστολής

- Όμοια με των ακεραίων, αλλά με επιπρόσθετο δεκαδικό κομμάτι π.χ. :
  - $1.0000_2 = 1.0_{10}$
  - $1.0010_2 = 1.125_{10}$
  - $1.0001_2 = 1.0625_{10}$
- Μετατροπή δεκαδικών σε δυαδικό αριθμό Σταθερής Υποδιαστολής:
  - Για το ακέραιο μέρος ο γνωστός τρόπος
  - Για το κλασματικό αντικαθιστούμε τους πολ/σμούς με διαιρέσεις
    - 1,625 = 1 + 0.625 x = 1
    - 0,625 * 2 = 1,25 = 1 + 0,25 y = 1 (MS)
    - 0,25 * 2 = 0,5 = 0 + 0,5 z = 0
    - 0.5 * 2 = 1,0 = 1 + 0.0 w = 1 (LS)

    Άρα $1.625_{10} = 1.101_2$

Αριθμοί Κινητής Υποδιαστολής

- Η αναπαράσταση κινητής υποδιαστολής είναι της μορφής XYZ * βάση^{ABC} όπως για παράδειγμα 2,37*10^{11}



Mantissa	exponent
m-bits	n-bits

Αριθμοί Κινητής Υποδιαστολής

- Η απόσταση μεταξύ διαδοχικών τιμών κινητής υποδιαστολής είναι 2^{-m} * exponent άρα πολύ μικρή για αρνητικές τιμές του εκθέτη και πολύ μεγάλη για (μεγάλες) θετικές τιμές του εκθέτη.
- Οι τιμές που μπορούν να αναπαρασταθούν αν το exponent είναι 8 bits και η μαντίσσα 23 bits

$$2^{-126}(1.0) \quad \text{εώς} \quad 2^{127}(2 - 2^{23})$$

$$1.8 \times 10^{-38} \quad \text{εώς} \quad 3.40 \times 10^{38}$$

Μετατροπή σε Αριθμούς Κινητής Υποδιαστολής

- Με κινητή υποδιαστολή, m = 24, n = 8, έχω $2^{-100} = 0,1 * 2^{-99}$. Το 0,1 είναι κανονικοποιημένη μαντίσσα, και το –99 μπορεί να αναπαρασταθεί με προσημασμένο αριθμό 8 bits, συνεπώς το 2^{-100} αναπαριστάται ακριβώς, ενώ με σταθερή υποδιαστολή χρειάζονται 0,0000...1 (100 ψηφία)
- **Π.χ. $17,6875_{10}$**
- Βήμα πρώτο: μετατροπή σε σταθερή υποδιαστολή
  - 17_{10} => 10001_2
  - 0,6875 * 2 = 1,375 => 1
  - 0,375 * 2 = 0,75 => 0
  - 0,75 * 2 = 1,5 => 1
  - 0,5 * 2 = 1,0 => 1
  - => $17,6875_{10} = 10001,1011_2$
- Βήμα δεύτερο: κανονικοποίηση (ολίσθηση της υποδιαστολής 5 θέσεις αριστερά με αντίστοιχη αύξηση του εκθέτη):
  - $17,6875 = 10001,1011 = 0,100011011 * 2^5$
  - Τελική αναπαράσταση: μαντίσσα = 0,100011011 εκθέτης = 5

Πρόσθεση Κινητής Υποδιαστολής

- **Πρόσθεση**: Για να προσθέσουμε δύο αριθμούς κινητής υποδιαστολής πρέπει να εξισώσουμε τους εκθέτες αυξάνοντας τον μικρότερο από τους δύο (ολισθαίνοντας κατάλληλα την αντίστοιχη μαντίσσα προς τα δεξιά). Κατόπιν, οι μαντίσσες προστίθενται, και το αποτέλεσμα κανονικοποιείται εάν χρειάζεται.
- Παράδειγμα $\mathbf{17,6875_{10} + 0,0625_{10}}$.
- $17,6875_{10} = 0,100011011 * 2^5$ και $0,0625_{10} = 0,1 * 2^{-3}$
  - Ολισθαίνω την μαντίσσα του |μικρότερου| αριθμού ($0,0625_{10}$) προς τα δεξιά αυξάνοντας τον εκθέτη:

  $0,1* 2^{-3} = 0,01 * 2^{-2} = 0,001 * 2^{-1} = ... = 0,000000001 * 2^5$

  - Πρόσθεση mantissas

  $0,100011011 + 0,000000001 = 0,100011100$

  - $0,100011100 * 2^5 = 10001,11 = 17\ ¾\ _{10} = 17,75_{10}$

Αφαίρεση Κινητής Υποδιαστολής

- **Αφαίρεση**: Για να αφαιρέσουμε δύο αριθμούς κινητής υποδιαστολής πρέπει να εξισώσουμε τους εκθέτες αυξάνοντας τον μικρότερο από τους δύο (ολισθαίνοντας κατάλληλα την αντίστοιχη μαντίσσα προς τα δεξιά). Κατόπιν, οι μαντίσσες αφαιρούνται, και το αποτέλεσμα κανονικοποιείται εάν χρειάζεται.
- Παράδειγμα: αφαίρεση $\mathbf{17{,}6875_{10} - 17_{10}}$
- $17{,}6875_{10} = 0{,}100011011 * 2^5$ ενώ το $17_{10} = 0{,}10001 * 2^5$
  - Εξίσωση εκθέτη: Δεν χρειάζεται αφού και οι δύο είναι ίσοι (5).
  - Αφαίρεση mantissas

    0,100011011 - 0,100010000 = 0,000001011
  - $0{,}000001011 * 2^5 = 0{,}1011 = 06875_{10}$
  - Κανονικοποίηση (εάν χρειάζεται):

    $0{,}000001011 * 2^5 = 0{,}00001011 * 2^4 = \ldots = 0,\ 1011 * 2^0$