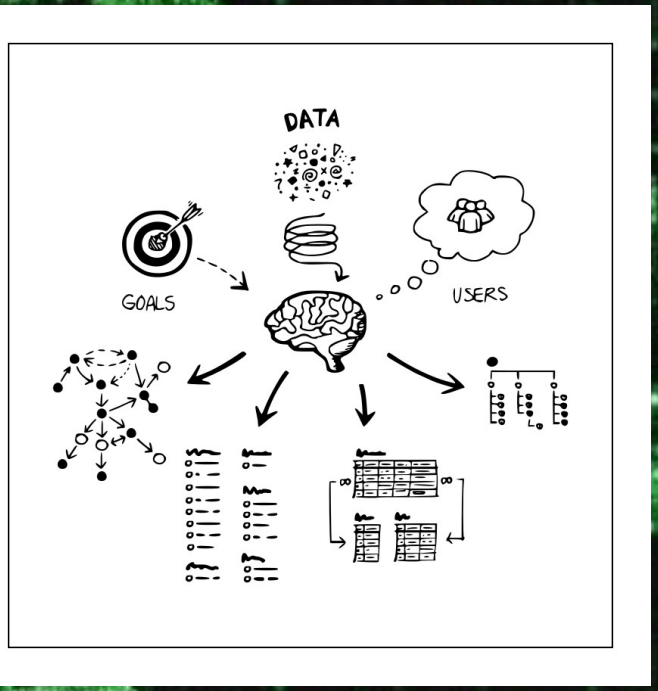
Τι συμβαίνει με τους Pointers ;

Δομημένος Προγραμματισμός

Τσαγκατάκης Ιωάννης

ΕΛΛΗΝΙΚΟ ΜΕΣΟΓΕΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ

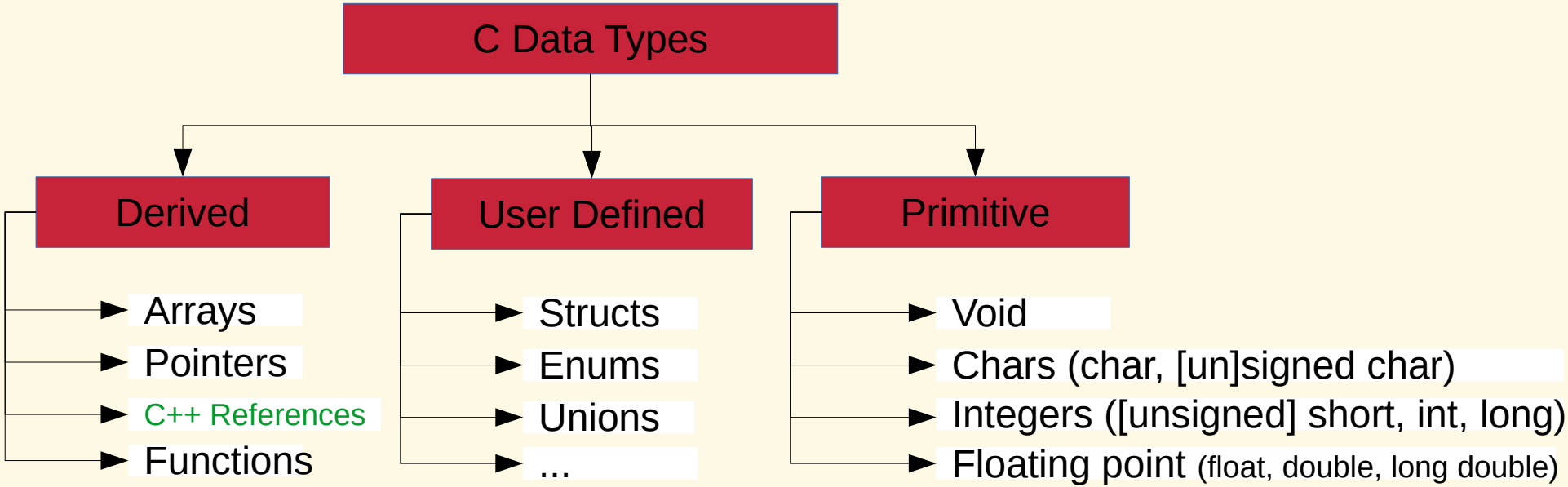
Τι συμβαίνει με τους Pointers ;



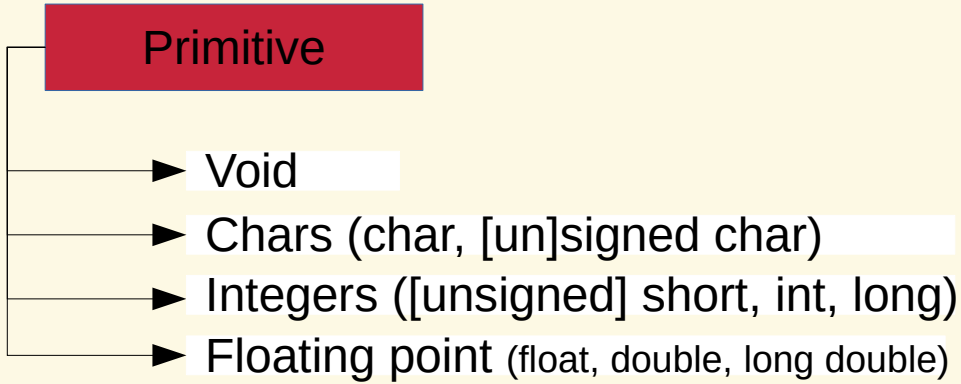
Οι τυποι

της γλωσσάς C

C Data Types


C Data Types
Derived
User Defined
Primitive
Arrays
Pointers
C++ References
Functions
Structs
Enums
Unions
...
Void
Chars (char, [un]signed char)
Integers ([unsigned] short, int, long)
Floating point (float, double, long double)

Primitive data types



- Ξέρει για αυτούς ο επεξεργαστής
- Υπάρχει μια σχέση “μικρότερο ίσο” ανάμεσα στα διάφορα int και float.
- Για αριθμούς κινητής υποδιαστολής επιλέγουμε τον double και όχι τον float
- Ο τύπος char είναι στην πραγματικότητα ένας μικρός ακέραιος
  - Υπάρχει με και χωρίς πρόσημο
  - Καταλαμβάνει χώρο 8bit πάντα
  - Στην εποχής του UTF ένας χαρακτήρας μπορεί να έχει πολλά bytes

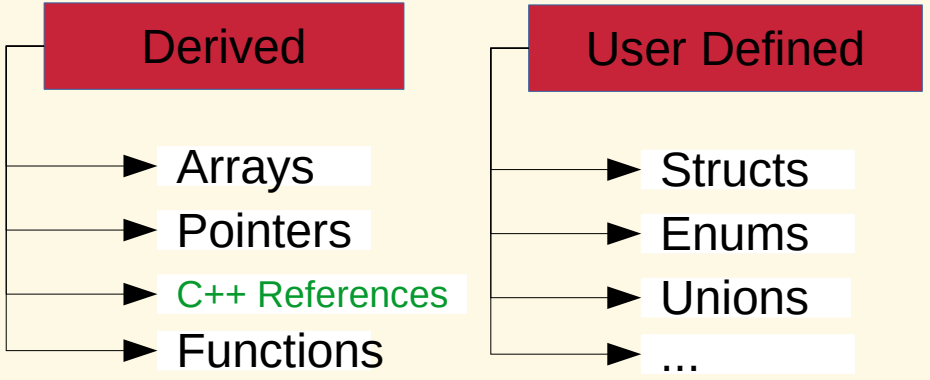
#include <stdint.h> (C99)

`int8_t,int16_t,int32_t,int64_t`	Ακέραιοι με ακριβώς 8,16,32,64 bit
`uint8_t,uint16_t,uint32_t,uint64_t`	Χωρίς πρόσημο
`[u]int_fastXX_t,[u]int_leastXX_t,`	Γρήγοροι, ελάχιστου μεγέθους
`[u]intmax_t`	Μεγαλύτερο δυνατό μέγεθος
`[U]INT_XX_MIN, [U]INT_XX_MAX`	Ελάχιστος και μέγιστος αριθμός (macro)

Για περισσότερα, καθώς και για προσδιοριστικά για την printf :

https://en.cppreference.com/w/c/types/integer

Παράγωγοι και ορισμένοι τύποι

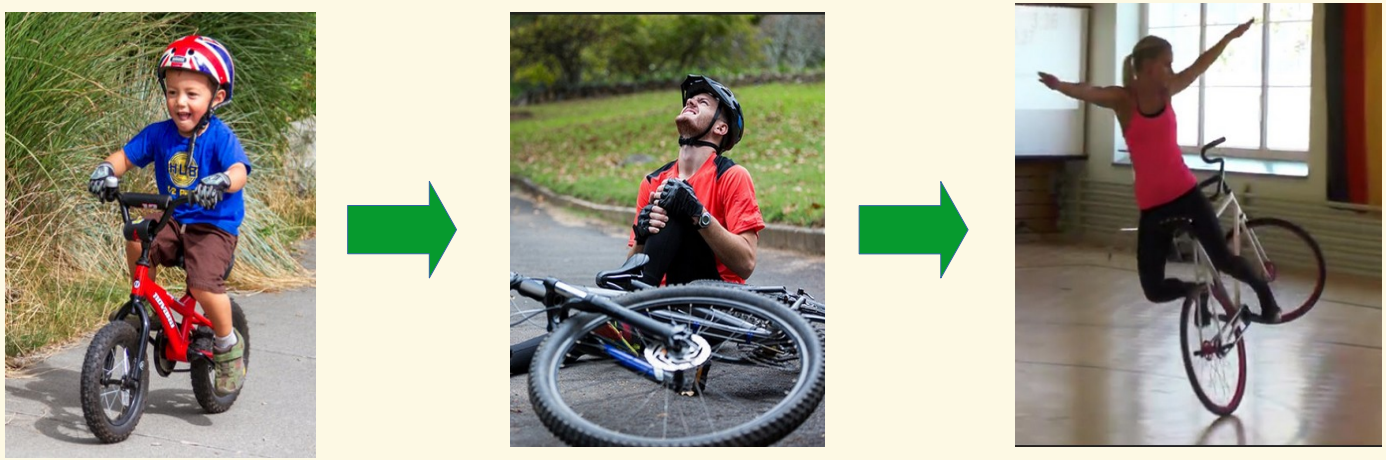


- Συνδυάζοντας τους υπάρχοντες τύπους (και με την βοήθεια των δεικτών) μπορούμε να φτιάξουμε τους δικούς μας τύπους που να ταιριάζουν στο πρόβλημα μας.
- Μπορούμε να συνδυάσουμε τα στοιχεία μεταξύ τους πχ array of structs που να έχει ενα union σαν μέλος
- Θα δούμε σιγά σιγά τι είναι αυτά ξεκινώντας από τους δείκτες.

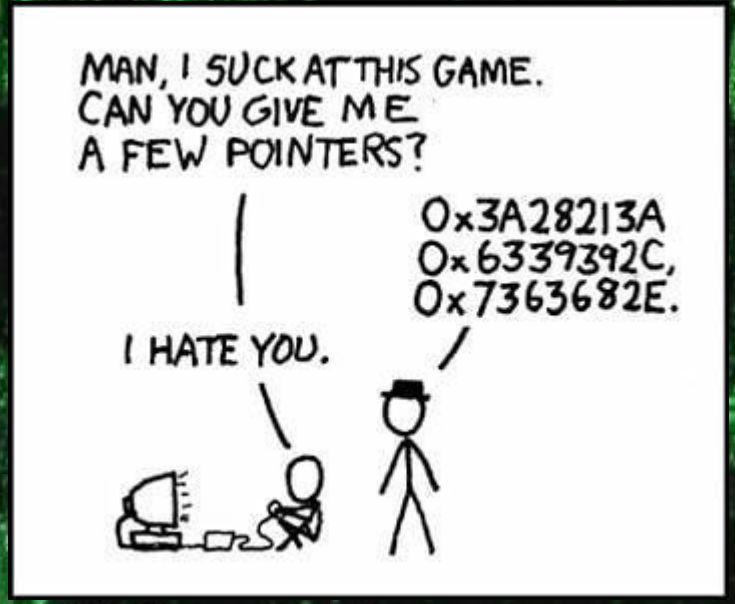
Πριν ξεκινήσουμε ...

Οι δείκτες είναι ένα δύσκολο (και δυνατό) χαρακτηριστικό της γλώσσας C. Αλλά αν καταλάβεις τις βασικές έννοιες θα δεις πως είναι εύκολοι.

Μοιάζουν με το ποδήλατο ή το πατίνι. Την μια μέρα νομίζεις πως είναι αδύνατο να το μάθεις και την επόμενη κάνεις ακροβατικά. Και το ποδήλατο δεν το ξεχνάς ποτέ.



Τι συμβαίνει με τους Pointers ;

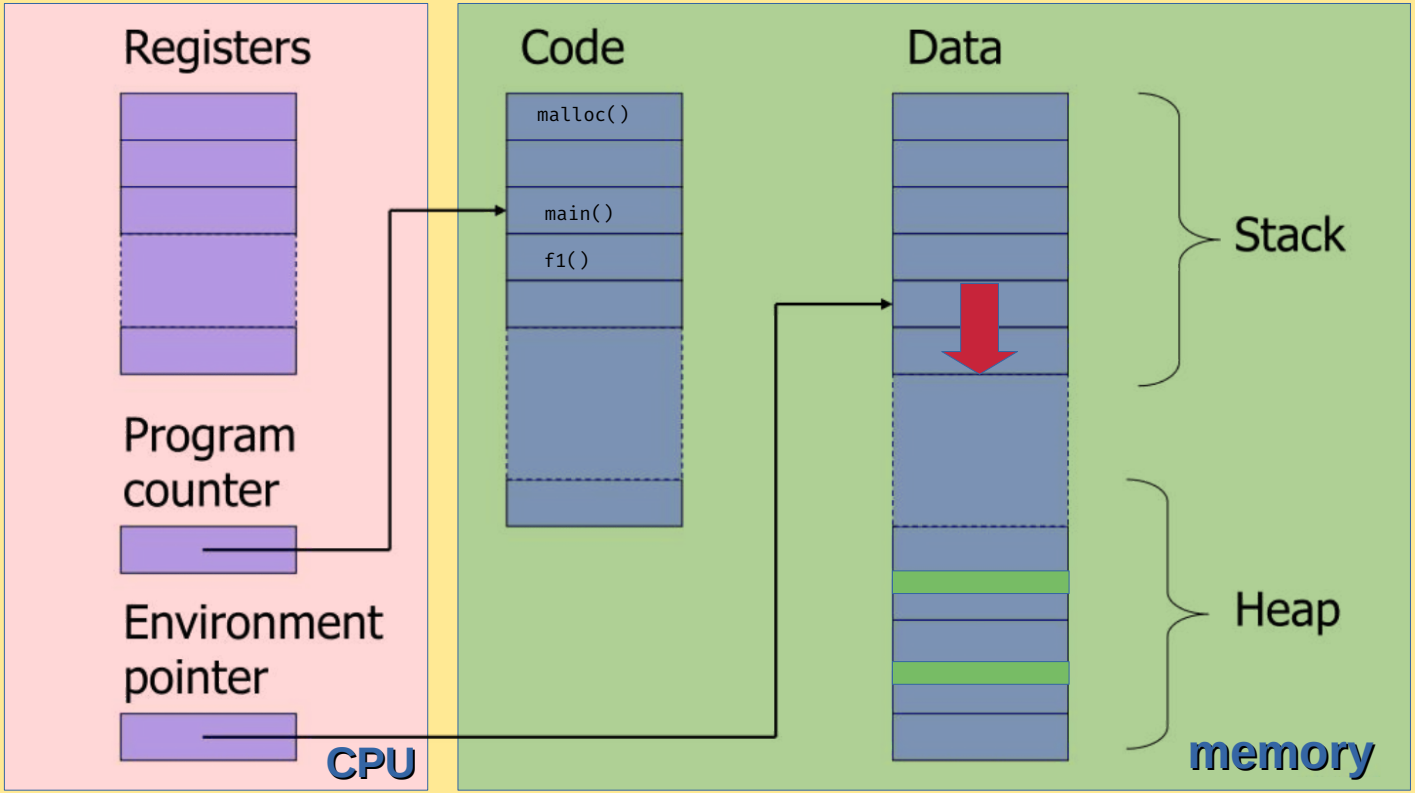


Μια προχωρημένη εισαγωγή και επανάληψη

Οι διευθύνσεις της μνήμης

- Η μνήμη είναι χωρισμένη σε διαδοχικά κελιά μεγέθους `sizeof(char)`,συνώνυμο του 1byte ή 8bit.
- Κάθε κελί μνήμης έχει ένα αύξοντα αριθμό, που συνήθως το γράφουμε στο δεκαεξαδικό σύστημα αρίθμησης, που τον λέμε **διεύθυνση μνήμης**.
- Θεωρούμε ότι δεν υπάρχει κελί μνήμης με αριθμό μηδέν.
- Κάθε μεταβλητή (αλλά και κάθε συνάρτηση), υπάρχει κάπου στην μνήμη και δεσμεύει κάποιο χώρο. Για παράδειγμα ο **int** θέλει `sizeof(int)` συνήθως 4 bytes.
- Η διεύθυνση του 1ου κελιού που δεσμεύεται για κάποια μεταβλητή,την ονομάζουμε **διεύθυνση μνήμης της μεταβλητής**, και την παίρνουμε με τον τελεστή **&**

Πως βλέπει η **C** τον υπολογιστή



Storage Class

- **register**
  - Αν μια μεταβλητή δηλωθεί σαν **register** δεν υπάρχει στην μνήμη, αλλά μόνο στους καταχωρητές του επεξεργαστή, άρα δεν έχει διεύθυνση μνήμης.
  - Αυτό είναι μια αίτηση που ο επεξεργαστής μπορεί να αγνοήσει και ήταν χρήσιμο πριν δεκαετίες. Σήμερα ο compiler είναι έξυπνος και την αγνοεί έτσι και αλλίως.
- **volatile**
  - Μια μεταβλητή που είναι σε ειδική θέση μνήμης και αλλάζει ανεξάρτητα από το πρόγραμμα, για παράδειγμα συνδεμένη με κάποιο αισθητήριο. Θα διαβαστεί από την μνήμη κάθε φορά που θα χρειαστεί η τιμή της.
  - Κοιτάξτε αν ο μεταγλωττιστής παρέχει κάτι καλύτερο (πχ atomic operators) γιατί υπάρχουν μικροπροβλήματα σε επεξεργαστές 64bit. Η μοντέρνα C επίσης παρέχει καλύτερες εναλλακτικές όταν έχουμε πολλά νήματα να τρέχουν ταυτόχρονα σε πολλούς επεξεργαστές και μοιράζονται την ίδια μνήμη.
- **auto**
  - Λέει ότι μια μεταβλητή υπάρχει στην στοίβα. Επειδή όλες οι μεταβλητές που έχουν οριστεί μέσα σε συναρτήσεις είναι εξ ορισμού δεν την χρησιμοποιούμε. Έχει μια εντελώς διαφορετική χρήση στην **C++**.

```c
#include <stdio.h>

int main(void) {
  const volatile int local = 10;
  int *ptr = (int*) &local;

  printf("Initial value of local : %d \n", local);

  *ptr = 100;

  printf("Modified value of local: %d \n", local);

}
```

- Ο κώδικας προσπαθεί να εξομοιώσει μια μεταβλητή **volatile** με την βοήθεια **pointers** και **aliasing**.
- Θα βγάλει νόημα αργότερα, προς στιγμήν δείτε μόνο πως κάναμε την δήλωση.
- Για να το δούμε θα πρέπει να παλέψουμε με τον μεταγλωττιστή, γιατί έχουν γίνει πολύ έξυπνοι.

Storage Class

- **extern**
  - Η μεταβλητή δεσμεύει μνήμη σε άλλη μονάδα μεταγλώττισης σε ποιο πολύπλοκα προγράμματα, όπου ο κώδικας είναι σε πολλά αρχεία.
- **static**
  - Κάνει μια μεταβλητή καθολική (**global**) όταν την δούμε μέσα σε σώμα συνάρτησης. Θα υπάρχει στην μνήμη μια φορά και δεν θα είναι μέσα στο stack frame.
  - Διαφορετικά μαρκάρει ένα σύμβολο (μεταβλητή ή συνάρτηση) σαν τοπικό στην μονάδα μεταγλώττισης.
- **restrict**
  - Λέει ότι ένας δείχτης προσδιορίζει μια περιοχή μνήμης που δεν έχει **aliasing**. Χρήσιμο σε αριθμητικές υπορουτίνες για να μπορέσουν να αξιοποιήσουν της δυνατότητες παραλληλίας που έχουν οι μοντέρνοι επεξεργαστές.
  - Υπάρχει μόνο στις δηλώσεις συναρτήσεων και άρα δεν δεν storage class.
  - Είναι η μόνη δεσμευμένη λέξη της **C** που δεν υπάρχει στην **C++**.

```
#include <cstdio>

void fcount() {
   static int callCount = 0;
   ++callCount;
   printf("The function \"fcount\" was
called %d times.\n", callCount);
}

int main() {
   fcount();
   fcount();
   fcount();
}
```

Scope and lifetime

	Scope	Lifetime
Global	Όλη η μονάδα μεταγλώτησης	Όσο τρέχει το πρόγραμμα
Static	Μέσα στην συνάρτηση που έχει οριστεί	Όσο τρέχει το πρόγραμμα
Auto (local)	Μέσα στην συνάρτηση που έχει οριστεί	Μέχρι η συνάρτηση να επιστρέψει μια τιμή.
Dynamic (malloc)	Όποια συνάρτηση γνωρίζει ένα δείκτη που δείχνει στην περιοχή της μνήμης.	Μέχρι να καλεστεί η συνάρτηση **free()**

Ο πίνακας συμβόλων

- Κατά την μεταγλώττιση ο compiler δημιουργεί και συμπληρώνει ένα πίνακα (**symbol table)** που περιέχει τις θέσεις μνήμης, τον τύπο και το μέγεθός (αυτό που το βρίσκουμε με τον τελεστή `sizeof()`), για κάθε μεταβλητή.
  - Ο compiler δηλαδή αντιστοιχεί τα ονόματα σε διευθύνσεις μνήμης
  - Συνήθως μας είναι εντελώς αδιάφορη η τιμή της διεύθυνσης μνήμης.
  - Για μεταβλητές που είναι στην στοίβα κρατάει σχετικές διευθύνσεις.
- Εκτός από τις μεταβλητές διευθύνσεις μνήμης έχουν και οι συναρτήσεις του κώδικα, αυτές που φτιάχνουμε, αλλά και αυτές της βιβλιοθήκης.

Τι είναι ένας δείκτης

- Ένας δείκτης είναι μια μεταβλητή που περιέχει την διεύθυνση μνήμης μιας άλλης μεταβλητής.
  - Μπορούμε να διαβάσουμε ή να γράψουμε την τιμή της μεταβλητής που δείχνει, μέσω του pointer.

    Αυτό το λέμε **deference** ή έμμεση αναφορά.
  - Σε μια μεταβλητή μπορεί να δείχνουν πολλοί δείκτες
    Αυτό το λέμε **aliasing**.

Χρήση των δεικτών

- Για να μπορεί μια συνάρτηση να αλλάξει τα ορίσματα της.
- Για να μπορεί μια συνάρτηση να επιστρέφει πολλαπλές τιμές.
- Για να μειώσουμε τον όγκο των αντιγραφών για την κλήση μιας συνάρτησης.
- Για να “φτιάξουμε” συμβολοσειρές ή πίνακες.
- Για να έχουμε πίνακες με δυναμικό μέγεθος.
- Για να φτιάξουμε **δομές δεδομένων** όπως ‘δυναμικά συνδεμένες λίστες’, ‘δέντρα’, ...

Περισσότερες χρήσεις των δεικτών

- Πέρασμα συναρτήσεων σαν ορίσματα σε συναρτήσεις.
- Χρήση του μηχανισμού DMA – πχ συνάρτηση realloc()
- Μείωση της πολυπλοκότητας του κώδικά που γράφουμε !
- Αύξηση την ταχύτητας εκτέλεσης
- Μείωση της απαιτούμενης μνήμης για να τρέξει ενα πρόγραμμα.
- Για να έχει το πρόγραμμα μας “ενδιαφέροντα” bugs,
  κενά ασφαλείας και να καταρρέει με ένα θεαματικό τρόπο !

Η συνάρτηση scanf()

- Χρησιμοποιήσαμε ήδη δείκτες χωρίς να το ξέρουμε με την συνάρτηση scanf(), όπου προσθέταμε στο όνομα της μεταβλητής τον τελεστή “διεύθυνση μεταβλητής” γνωστό και σαν **&**.

```
int a;
scanf("This is the value %d", &a);
printf("Input value read : a = %d", a);
```

- Περνώντας την διεύθυνση μνήμης της μεταβλητής στην **scanf()** αυτή μπορεί να αλλάξει την τιμή της μεταβλητής.
- Το **&a** λοιπόν σημαίνει την διεύθυνση στην μνήμη που υπάρχει η μεταβλητή.

Δηλώσεις (**declarations**)

dataType *var_name;

- **dataType**: Ο τύπος που δείχνει ο δείκτης.
  Κάθε έγκυρος τύπος, γνωστός όταν γίνετε η δήλωση.
- **var_name**: Το όνομα του δείκτη.
  Κάθε έγκυρο όνομα μεταβλητής της **C**.

Παραδείγματα δηλώσεων

```
int *ip;          // Pointer to an int

double *id;       // Pointer to a floating point number

float *ifl;       // Pointer to a small floating point number

char *ich;        // Pointer to a character
```

Προσοχή στις δηλώσεις

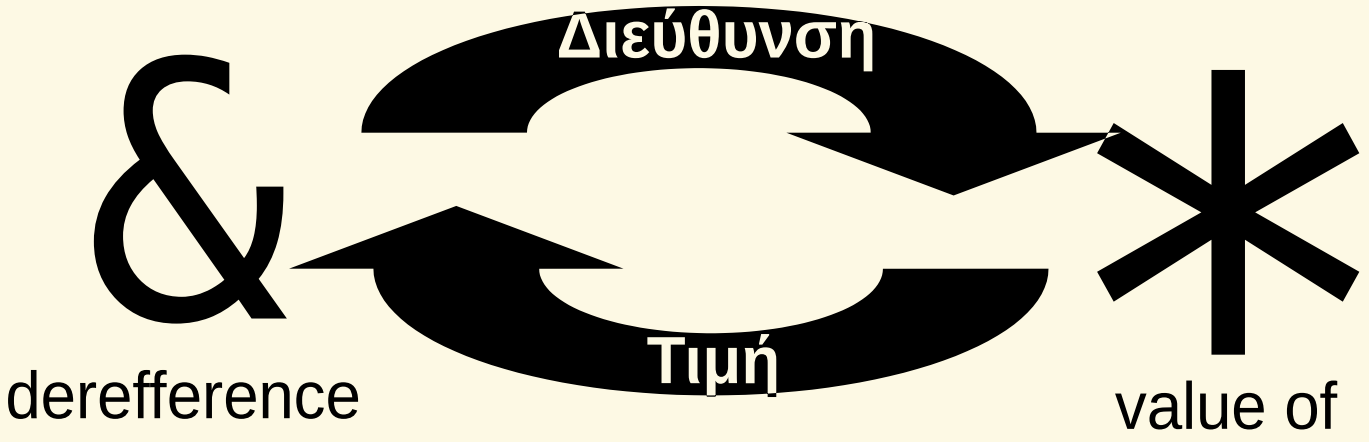
```
int *ptr1, ptr2, i, j;
```

- Αυτό δεν δηλώνει 4 pointer, αλλά ένα pointer και 3 int !
  - Ο σωστός τρόπος: `int *ptr1, *ptr2, *i, *j;`
  - Ένας τρόπος να το δούμε είναι ότι το '`*ptr2`' είναι ένα int.
  - Είναι το ίδιο αν έχουμε κενά μεταξύ του '`*`' και του '`ptr2`'
  - Μπορούμε να τα συνδυάζουμε μέσα στην ίδια δήλωση:

    ```
    int *ptr1, *ptr2, i, j;
    ```

    που θα ορίσει δύο pointers και δύο int.

Τα σύμβολα & και *



Τυπώνοντας τιμές διευθύνσεων

Ο παρακάτω πίνακας δείχνει πως μπορούμε να τυπώσουμε τιμές διευθύνσεων μνήμης με την βοήθεια της συνάρτησης **`printf()`**

Προσδιοριστικό	Ερμηνεία
`%u`	Σαν ακέραιο αριθμό χωρίς πρόσημο
`%x`	Σαν δεκαεξαδικό αριθμό
`%lx`	Σαν δεκαεξαδικό αριθμό long
`%o`	Σαν οκταδικό αριθμό
`%p`	Σαν δείκτη (implementation specific) τυπικά σαν το %x

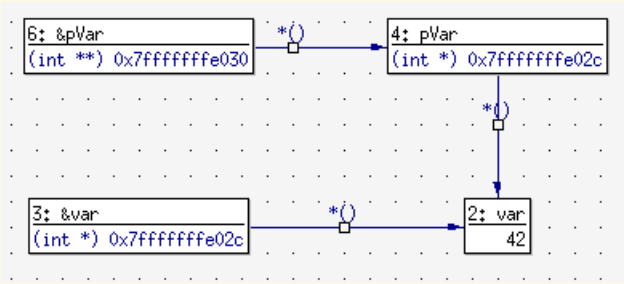
```
printf("Address of pi: %x Value: %d\n",&pi, pi);
```

Παράδειγμα

```c
#include <stdio.h>

int main() {
   int var = 42;
   int *ip = &var;

   puts("Pointers variables and addresses\n");
   printf("The value of 'var' is : %d\n", var);
   printf("The address of 'var' is : %p\n", &var);                    // Note &var

   printf("The value of 'ip' is : %lx\n",  (unsigned long int) ip);   // Note the type casting (machine dependend)
   printf("The address of 'ip' is : %p\n", &ip);                      // Note &ip
   printf("The value that 'ip' points -> %d\n", *ip);                 // Note *ip
}
```

Μεταβλητές και δείκτες

```c
int main() {
  int var = 42;
  int *pVar = &var;
  assert(*pVar == 42);
}
```



Symbol Table

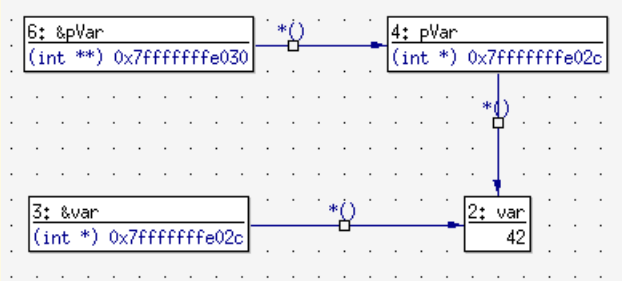
Όνομα	Τύπος	Μέγεθος	Διεύθυνση Μνήμης	Τιμή
var	int	sizeof(int) 4		
pVar	int *	sizeof(int*) 8		

- Η τιμή μιας μεταβλητής: var, pVar
- Η διεύθυνση μνήμης : **&**var, **&**pVar
- Η τιμή, που δείχνει ένας δείκτης: ***pVar**
- **Όλοι οι δείκτες πιάνουν το ίδιο χώρο στη μνήμη**
  sizeof(int *) == sizeof(void *)
  (στις μοντέρνες αρχιτεκτονικές υπολογιστών)
  - Αυτό μαζί με τους δείκτες σε συναρτήσεις είναι η βάση για να φτιάξουμε “αντικείμενα” χωρίς την C++

Η εικόνα φτιάχτηκε με την βοήθεια του ddd (GNU data display debugger)

Μεταβλητές και δείκτες:2

```c
int main() {
  int var = 42;
  int *pVar = &var;
  assert(*pVar == 42);
}
```



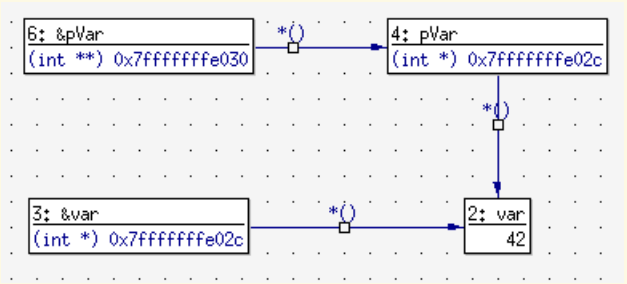
Symbol Table

Όνομα	Τύπος	Μέγεθος	Διεύθυνση Μνήμης	Τιμή
var	int	sizeof(int) 4	**0x7FF..02C**	
pVar	int *	sizeof(int*) 8	0X7FF..030	

- Η τιμή μιας μεταβλητής: var, pVar
- Η διεύθυνση μνήμης : **&**var, **&**pVar
- Η τιμή, που δείχνει ένας δείκτης: ***p**Var

Μεταβλητές και δείκτες:3

```c
int main() {
  int var = 42;
  int *pVar = &var;
  assert(*pVar == 42);
}
```



Symbol Table

Όνομα	Τύπος	Μέγεθος	Διεύθυνση Μνήμης	Τιμή
var	int	sizeof(int) 4	**0x7FF..02C**	42
pVar	int *	sizeof(int*) 8	0X7FF..030	**0x7FF..02C**

- Η τιμή μιας μεταβλητής: var, pVar
- Η διεύθυνση μνήμης : **&**var, **&**pVar
- Η τιμή, που δείχνει ένας δείκτης: ***p**Var

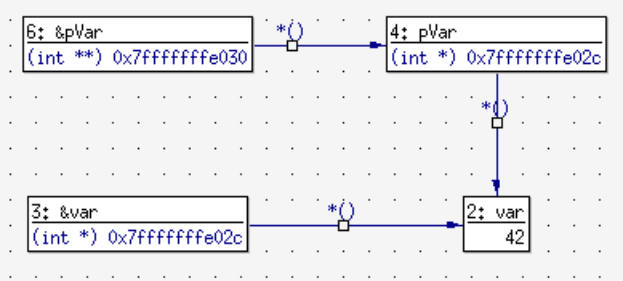
```
9: *var
<error: Cannot access memory at address 0x2a>
```

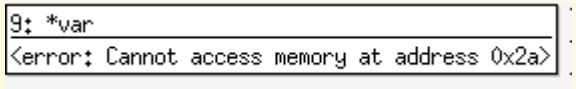
- Όλοι οι δείκτες πιάνουν το ίδιο χώρο στη μνήμη

sizeof(int *) == sizeof(void *)

Μεταβλητές και δείκτες:4

```c
int main() {
  int var = 42;
  int *pVar = &var;
  assert(*pVar == 42);
}
```



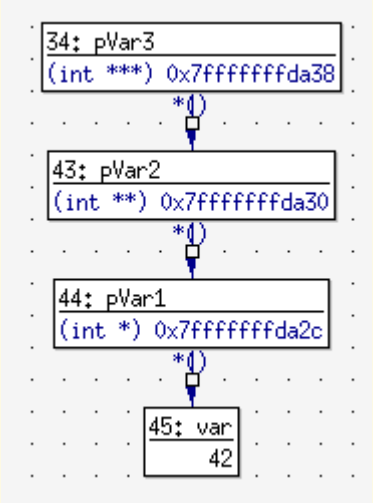


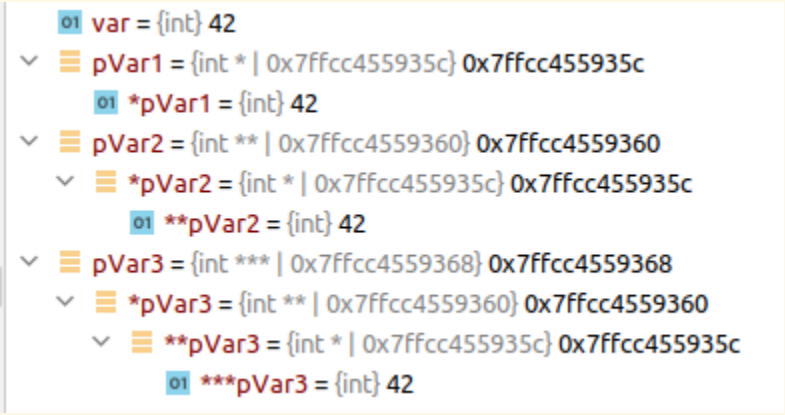
	Τιμή	Διεύθυνση μνήμης &	Αναφορά τιμής *
var	42	0×7FF..02C	💣💣
pVar	0×7FF..02C	0X7FF..030	42

- Η τιμή μιας μεταβλητής: var, pVar
- Η διεύθυνση μνήμης : **&**var, **&**pVar
- Η τιμή, που δείχνει ένας δείκτης: ***p**Var
  - ***var** Η τιμή που έχει η “απαγορευμένη” διεύθυνση μνήμης 0x2a η 42 στο δεκαδικό σύστημα → κατάρρευση του προγράμματος

Ένας δείκτης σε ένα δείκτη, ...

```
int var = 42;
int *pVar1;
int **pVar2;
int ***pVar3;

pVar1 = &var;
pVar2 = &pVar1;
pVar3 = &pVar2;
```





Σημείωση: Οι τιμές των διευθύνσεων μνήμης σε δυο διαφορετικούς debuggers είναι διαφορετικές, αλλά αυτό δεν έχει καμία σημασία. Μας ενδιαφέρει η μεταξύ τους σχέση. Πως θα τυπώναμε την τιμή 42 ξεκινώντας από την pVar3 με την `printf()`;

Τι είναι ένας **void** pointer

- Είναι ένας δείκτης που δείχνει σε κάποια “απλή θέση μνήμης” ή αλλιώς ένας pointer γενικού σκοπού.
- Μπορεί να γίνει **assignment** σε οποιοδήποτε άλλο τύπο pointer
- Μπορεί να δεχτεί οποιοδήποτε άλλο τύπο pointer
  - Ακόμα και χωρίς να δηλωθεί ένα ρητό **casting**.
- Ο compiler όμως δεν έχει ιδέα σε τι τύπο δείχνει, και δεν μπορεί να κάνει αριθμητική δεικτών ούτε να κάνει **dereference** (σε τι;).
  - Για τον ίδιο λόγο δεν έχουν κανένα νόημα οι τελεστές **++**, **--**
- Ένα δυνατό και επικίνδυνο εργαλείο

Μερικά κακά παιδιά

- **Wild Pointers**
  - Είναι οι δείκτες που δεν έχουν αρχικοποιηθεί με κάποια αρχική τιμή.
- **Dangling Pointers**
  - Είναι οι δείκτες που κάποια στιγμή έδειχναν σε μια έγκυρη διεύθυνση μνήμης αλλά αυτή δεν υπάρχει πλέον.
    - Δεν υπάρχει πλέον το stack frame
    - Έχει δοθεί πίσω η μνήμη (συναρτήσεις malloc/free)
- Στο πρόγραμμα μας δεν θέλουμε τέτοια κακά παιδιά

Οι 3 χρήσεις του ‘*’

- Για δήλωση pointers
- Για derefference
- Για πολλαπλασιασμό

```
int a;

int *p;

int z = *p;

a = 3 * *p;
```

Το μέγεθος ενός δείκτη

- Μπορούμε να βρούμε το μέγεθος ενός δείκτη με τον γνωστό τελεστή **sizeof()**.
  - Τυπικά θα είναι όσα τα bits της αρχιτεκτονικής της CPU. Δηλαδή 8bit σε μια μηχανή 64bit.
  - Το μέγεθος των δεικτών θα είναι πάντα το ίδιο.
    - Αυτό είναι πολύ σημαντικό και είναι η βάση για να κάνεις **αντικειμενοστραφή προγραμματισμό** σε **C**. Είναι η βάση για να υλοποιηθούν πράγματα όπως ο **πολυμορφισμός** και οι **ιδεατές (virtual) συναρτήσεις** που θα δείτε σε άλλα εξάμηνα.

Ερώτηση 1

```c
void fun(int x) {
    x = 30;
}

int main() {
    int y = 20;
    fun(y);
    printf("%d", y);
    return 0;
}
```

Τι θα τυπώσει το πρόγραμμα;

1) 30
2) 20
3) Compiler Error
4) Runtime Error

Σωστή απάντηση

.

https://www.geeksforgeeks.org/c-pointers-question-1

Ερώτηση 2

```c
void fun(int *p) {
    *p = 30;
}

int main() {
    int y = 20;
    fun(&y);
    printf("%d", y);
    return 0;
}
```

Τι θα τυπώσει το πρόγραμμα;

1) 30
2) 20
3) Compiler Error
4) Runtime Error

Σωστή απάντηση

.

Ερώτηση 3

```c
#include <stdio.h>

int main()
{
    int *ptr;
    int x;

    ptr = &x;
    *ptr = 0;

    printf(" x = %d\n", x);
    printf(" *ptr = %d\n", *ptr);

    *ptr += 5;
    printf(" x  = %d\n", x);
    printf(" *ptr = %d\n", *ptr);

    (*ptr)++;
    printf(" x = %d\n", x);
    printf(" *ptr = %d\n", *ptr);

    return 0;
}
```

Τι θα τυπώσει το πρόγραμμα;

Α. x = 0, *ptr = 0, x = 5, *ptr = 5, x = 6, *ptr = 6

Β. x = ??, *ptr = 0, x = ??, *ptr = 5, x = ??, *ptr = 6

Γ. x = 0, *ptr = 0, x = 5, *ptr = 5, x = ??, *ptr = ??

Δ. x = 0, *ptr = 0, x = 0, *ptr = 0, x = 0, *ptr = 0

Σωστή απάντηση .

Ερώτηση 5

```
int arr[] = {10, 20, 30, 40, 50, 60};
int *ptr1 = arr;
int *ptr2 = arr + 5;


size_t diffel = ptr2 - ptr1;
printf("Elements are: %d.",  diffel);


size_t diffmem = (char *) ptr2 - (char *) ptr1;
printf("Bytes are: %d",   diffmem);
```

Τι θα τυπώσει το πρόγραμμα;

Α. `5, 20`
Β. `20, 20`
Γ. `5 ,5`
Δ. `Compile error`
Ε. `Runtime error`

Σωστή απάντηση

Ερώτηση 6

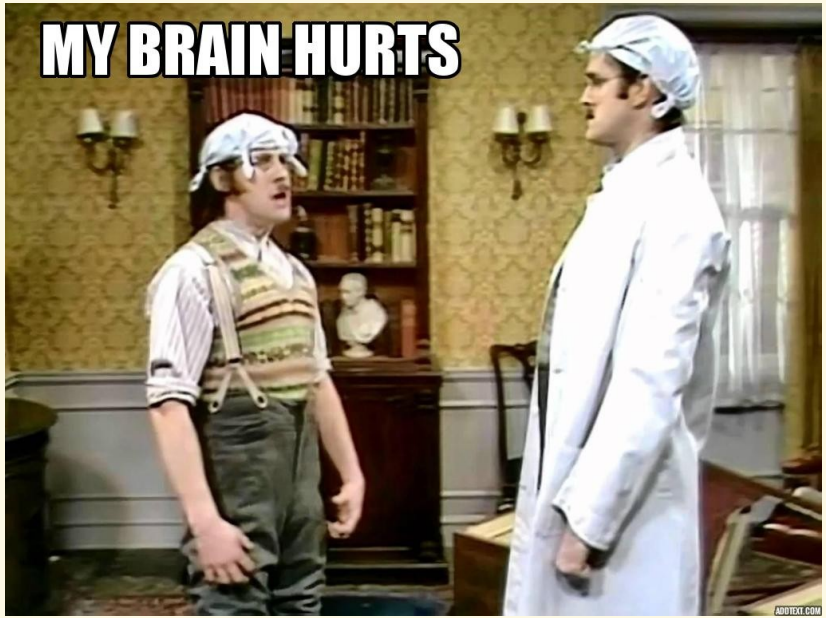
```
int arr[] = {10, 20, 30, 40, 50, 60};
int *ptr1 = arr;
int *ptr2 = arr + 5;


size_t diffel = ptr2 - ptr1;
printf("Elements are: %u.",  diffel);


size_t diffmem = (void *) ptr2 - (void *) ptr1;
printf("Bytes are: %u",   diffmem);
```

Τι θα τυπώσει το πρόγραμμα;

Α. `5, 20`
Β. `20, 20`
Γ. `5 ,5`
Δ. `Compile error`
Ε. `Runtime error`

Σωστή απάντηση

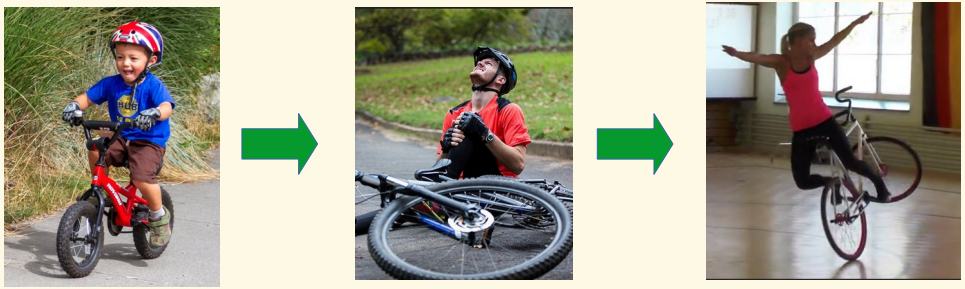
Προπαντός Ψυχραιμία !



Οι δείκτες είναι ένα δύσκολο (και δυνατό) χαρακτηριστικό της γλώσσας C. Αλλά αν καταλάβεις τις βασικές έννοιες θα δεις πως είναι εύκολοι.

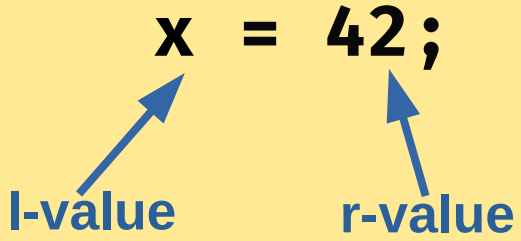
Μοιάζουν με το ποδήλατο ή το πατίνι. Την μια μέρα νομίζεις πως είναι αδύνατο να το μάθεις και την επόμενη κάνεις ακροβατικά.

Και το ποδήλατο δεν το ξεχνάς ποτέ.



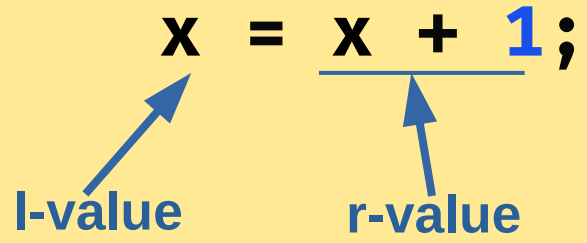
Μια άλλη προσέγγιση στους δείκτες

- Μια ποιο επιστημονική και δικηγορίστικη προσέγγιση πάνω στους δείκτες και τις τιμές.
- Είναι ο εσωτερικός τρόπος της **C** για να ελέγξει αν μια **ανάθεση τιμής (asignment)** είναι έγκυρος.
- Το **l-value** είναι αυτό που υπάρχει αριστερά (left) μιας ανάθεσης, ενώ το **r-value** αυτό που υπάρχει δεξιά (right).
  - Στην πράξη αναθέτουμε μια **τιμή** σε μια **θέση μνήμης**.
- Μπορεί να μοιάζει προφανής τώρα, αλλά υπάρχουν και κάποιες πολύπλοκες αναθέσεις.



Οι τιμές **r-value** και **l-value**

- Αν μια έκφραση δεν έχει **l-value** τότε δεν μπορεί να βρεθεί στην αριστερή (left) πλευρά του **assignment**. Με άλλα λόγια αριστερά του **=** θα πρέπει να υπάρχει μια διεύθυνση μνήμης.
  - Για παράδειγμα το '1 = x +1' δεν είναι έγκυρο γιατί το 1, δεν είναι **l-value**. Είναι έγκυρο το 'array[1] = x+1';
- Οι σταθερές έχουν μόνο **r-value** δηλαδή δεν έχουν κάποια διεύθυνση μνήμης (τουλάχιστον στην C++).
- Οι συναρτήσεις έχουν μόνο **l-value** και είναι const.
- Οι μεταβλητές έχουν και **l-value** και **r-value**.
- Οι μεταβλητές τύπου δείκτη (pointer) έχουν σαν **r-value** την τιμή **l-value** μιας άλλης μεταβλητής. Δηλαδή η τιμή τους είναι μια διεύθυνση μνήμης.

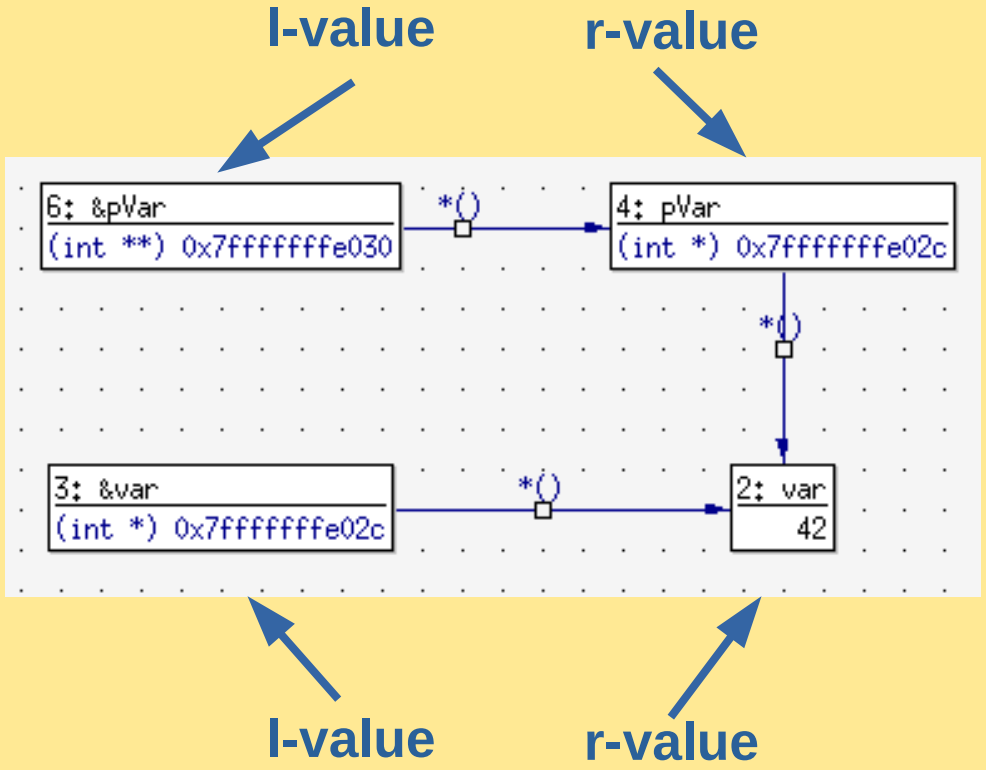


Πάρε την **r-value** του **x** και πρόσθεσε ένα. Βάλε την τιμή στην **l-value** του **x**.

- Ο τελεστής **&x** επιστρέφει την **l-value** του x.
- Ο τελεστής ***p** επιστέφει την **r-value** του p.

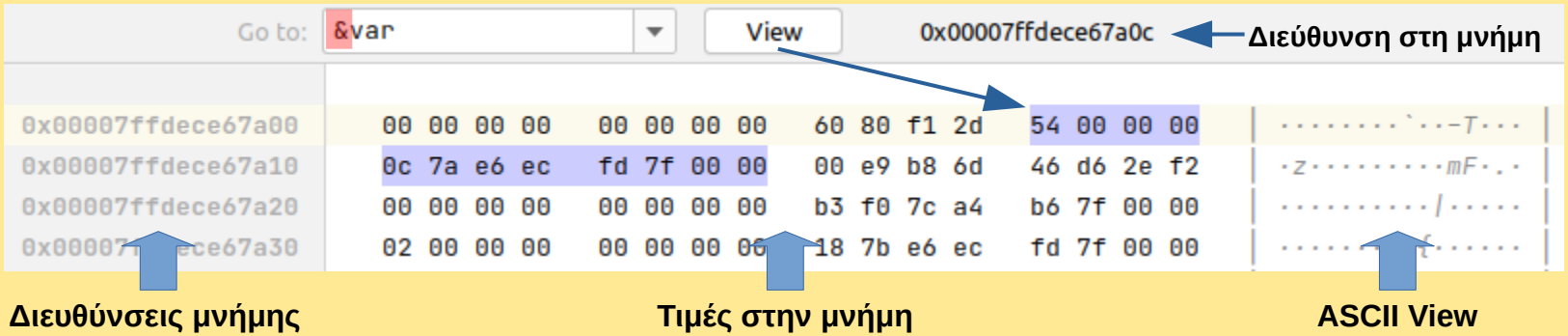
Δείκτες και rl-τιμές

```
int var = 42;
// lValue(var) some address, rValue(var) == 42

int *pVar = &var;
// lValue(pVar) some other address
// rValue(pvar) == lValue(var)

*pVar = 2 * var;
// rValue(pVar) <- rValue(var) * rValue(2)
// lValue(var) <- rValue(var) * rValue(2)
```



Μια ματιά στην μνήμη

```
int var = 42;

int *pVar = &var;

*pVar = 2 * var;
```



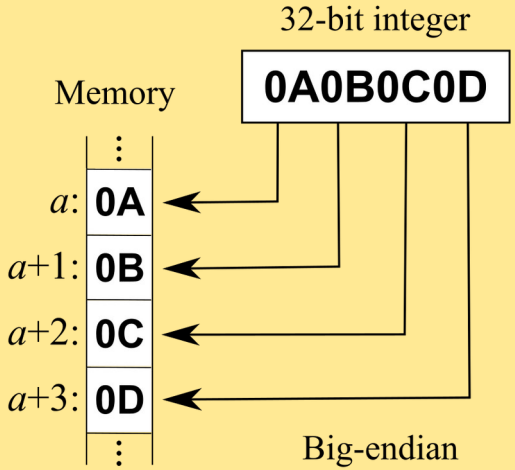
Η μεταβλητή **var** είναι στην διεύθυνση μνήμης **0x7ffdece67a0c** και περιέχει την τιμή **0x00000054**.

Η μεταβλητή **pVar** (στην διεύθυνση μνήμης **0x7ffdece67a10**) περιέχει την διεύθυνση μνήμης της **var**.

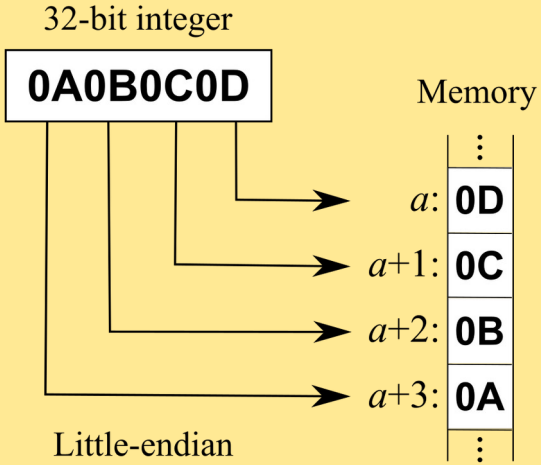
Η τιμή του **0x00000054** στο δεκαεξαδικό είναι 5*16+4 = 84 στο δεκαδικό.

Προσέξτε πως η τιμές είναι γραμμένες ανάποδα στην μνήμη.
Στην μνήμη **54 00 00 00** και όχι **00 00 00 54,** μια ιδιομορφία του επεξεργαστή **Pentium**.

Από ποια πλευρά σπάμε το αυγό;



- Η σειρά αποθήκευσης στην μνήμη μπορεί να είναι διαφορετική από την αναπαράσταση στην CPU.
- Δεν μας ενδιαφέρει, εκτός αν στέλνουμε κάτι σαν bytes στο δίκτυο, τα αποθηκεύουμε σε ένα αρχείο ή κοιτάζουμε την μνήμη σε ένα debugger.
- Τα πρωτόκολλα δικτύου υπολογιστών κάνουν τους κατάλληλους μετασχηματισμούς δεδομένων ώστε δύο υπολογιστές με διαφορετικό endianness να επικοινωνούν.
- Κάποιοι επεξεργαστές μπορεί να υποστηρίξουν και τα δυο.
- Οι Pentium είναι Litle Endian

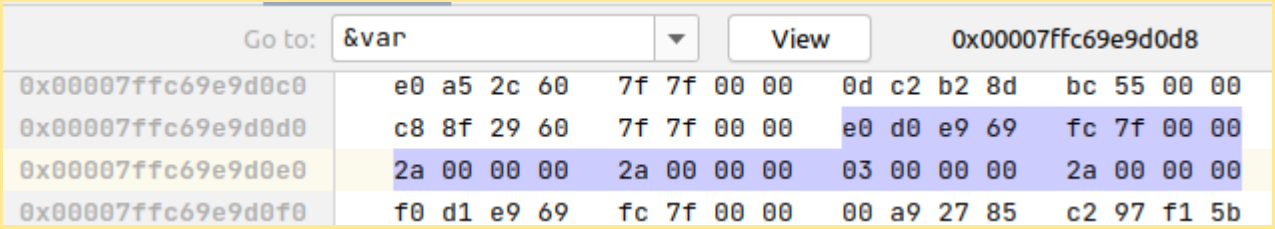


https://en.wikipedia.org/wiki/Endianness

Ερώτηση κατανόησης

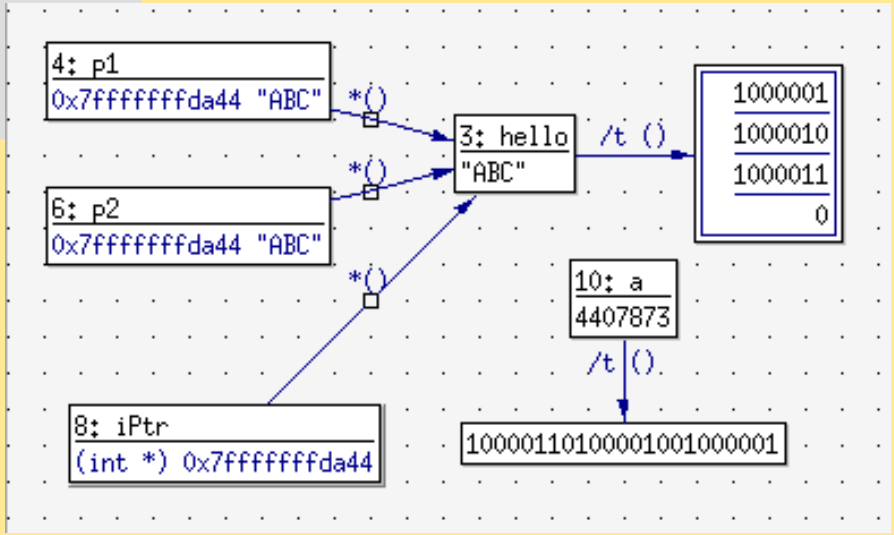
```
const int d = 2;
int var[] = {40, 41, 42, 43};

int *pVar = &var;

*(pVar + d) = 0;
```

- Είναι έγκυρος ο κώδικας;
- Εντοπίστε τα (rl)-value
- Τι κάνει ο κώδικας;
- Τι θα συμβεί αν d=4;



Τι είναι το **aliasing**

```
char hello[]="ABC";
char *p1 = hello;
char *p2 = hello;

int *iPtr = (int *) hello;

int a = *iPtr;
```

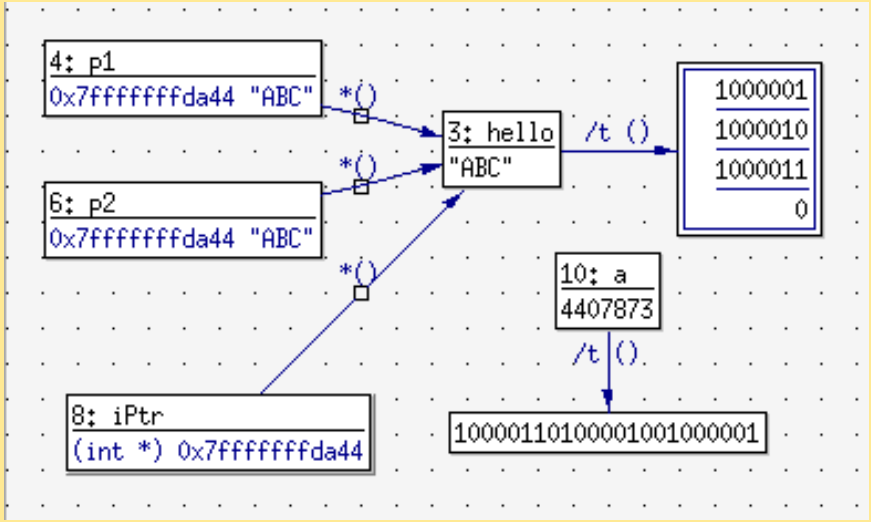


- Σε μια θέση μνήμης μπορεί να δείχνουν πολλοί δείκτες
- Οι δείκτες αυτοί μπορεί να έχουν διαφορετικούς τύπους !

Τι είναι το Aliasing (παράδειγμα)

```
char hello[]="ABC";
char *p1 = hello;
char *p2 = hello;

int *iPtr = (int *) hello;

int a = *iPtr;
```

Προσαρμογή (casting) ενός δείκτη σε ένα άλλο τύπο δείκτη.



Στο `{'A','B','C','\0'}` που περιέχει η μεταβλητή **hello** δείχνουν 3 δείκτες.

Οι δύο δείκτες **p1**, **p2** είναι τύπου χαρακτήρα και το βλέπουν σαν κείμενο.

Αλλά ο **iPtr** βλέπει την ίδια μνήμη σαν signed int !

Τι τιμές παίρνει ένας δείκτης;

- Ένας δείκτης μπορεί να **αρχικοποιηθεί** και να δείχνει σε μια **υπάρχουσα** μεταβλητή.
- Η να μην δείχνει πουθενά και να έχει την τιμή **NULL**.
- Αν ένας δείκτης δεν αρχικοποιηθεί θα δείχνει σε μια τυχαία θέση της μνήμης, που μπορεί να μην υπάρχει στον χώρο διευθύνσεων του προγράμματος.
- **Προσοχή**!
  - Πάντα αρχικοποιούμε τους δείκτες σε μια τιμή.
  - Πάντα είμαστε βέβαιοι πως ένας δείκτης δεν είναι **NULL** πριν τον χρησιμοποιήσουμε
  - Πάντα είμαστε βέβαιοι πως ένας δείκτης δείχνει σε θέση μνήμης που υπάρχει ακόμα.
    - Οι μεταβλητές στην στοίβα θα παύσουν να υπάρχουν όταν γίνει έξοδος από μια συνάρτηση.
    - Αλλά εξακολουθούν να υπάρχουν αν η συνάρτηση καλέσει μια άλλη συνάρτηση.
  - Αν η τιμή ενός δείκτη είναι λάθος, τότε τα πάντα μπορεί να συμβούν !

Τι είναι ένας NULL pointer

- Είναι ένας δείκτης που δεν δείχνει σε μια έγκυρη διεύθυνση μνήμης. Κάθε προσπάθεια πρόσβασης του θα δώσει σφάλμα υλικού.
- *Έχει την τιμή μηδέν*;
  Τεχνικά όχι, απλά η τιμή μηδέν είναι ισοδύναμη με αυτήν την τιμή. Στην πράξη σήμερα επειδή οι επεξεργαστές έχουν σχεδιαστεί για να τρέχουν προγράμματα στην C ή τιμή αυτή συμβαίνει να είναι η τιμή μηδέν.
- Η **C++** χρησιμοποιεί την δεσμευμένη λέξη **nullptr** και δεν είναι πλέον ένα macro του προεπεξεργαστή που δίνει την τιμή 0.
- Είναι καλύτερο να χρησιμοποιούμε την **nullptr** αντι για το **NULL**.

Χρήσεις του NULL/nullptr

- Για τερματισμό μιας αναδρομής
- Σε συναρτήσεις που επιστρέφουν τιμές σαν τιμή λάθους.
- Σαν μια τιμή φρουρού (sentinel)

Οι πολλές έννοιες του NULL

- Η έννοια του **NULL**
- Η σταθερά **NULL**
- Το **NULL** macro του **προεπεξεργαστή**

```
#define NULL ((void *) 0)
```

- Το **nullptr**
- Ο **ASCII** κώδικας NULL
- Μια κενή (null) συμβολοσειρά
- Το **null statement** (ένα σκέτο ';')
- Η τιμή μηδέν σε κάποιες περιπτώσεις

Έλεγχος για NULL/ nullptr

```c
#include <stdio.h>

int main() {
   int var = 42;
   int *ip = &var;

   if (ip) {
      puts("The pointer 'ip' is NULL!\n");
      puts("Try to dereference and the program will die!\n");
   }

   if (!ip) {
      puts("The pointer 'ip' is not NULL!\n");
      printf("I can get the value. It is %d.", *ip);
   }
}
```

Η ιστορία του NULL

Ο **Tony Hoare** ο δημιουργός του QuickSort κάτοχος βραβείου **Turing** (το Nobel της πληροφορικής) πρόσθεσε το **NULL** στην γλώσσα **ALGOL** το 1965 γιατί του φάνηκε χρήσιμο και εύκολο.

Ο φίλος του **Edsger Dijkstra**, του είπε πως δεν ήταν η καλύτερη ιδέα του κόσμου.

Λίγες δεκαετίες μετά, ο **Tony Hoare** δήλωσε ...

Το λάθος των δισεκατομμυρίων $$

I call it my billion-dollar mistake… Εκείνη την εποχή, σχεδίαζα το πρώτο συνεκτικό σύστημα τύπων για αναφορές πάνω σε μια αντικειμενοστραφή γλώσσα. Ο στόχος μου ήταν να διασφαλίσω ότι η χρήση των αναφορών στην μνήμη θα ήταν απολύτως ασφαλής, με τον έλεγχο να γίνεται αυτόματα από τον μεταγλωττιστή.

Αλλά δεν μπορούσα να αντισταθώ στον πειρασμό να προσθέσω μια μηδενική αναφορά NULL, απλώς και μόνο επειδή ήταν τόσο εύκολο να υλοποιηθεί. Αυτό έχει οδηγήσει σε αναρίθμητα **σφάλματα**, **ευπάθειες** και **καταρρεύσεις** συστημάτων, τα οποία πιθανώς προκάλεσαν ένα δισεκατομμύριο δολάρια **πόνο** και **ζημιά** τα τελευταία σαράντα χρόνια.

Ένα meme που κυκλοφορεί

Non-zero value

0

null

undefined

- Σε κάποιες άλλες γλώσσες προγραμματισμού τα πράγματα είναι χειρότερα (βλέπε πχ Javascript)
- Εκεί υπάρχει διαφορά μεταξύ μιας μεταβλητής που δεν έχει οριστεί ακόμα, μιας μεταβλητής που έχει οριστεί και δεν έχει τιμή ακόμα, καθώς και της τιμής **μηδέν**.

Τα references της C++

- Είναι ψευδώνυμα μιας μεταβλητής και μοιάζουν με τους δείκτες.
- Ένα ψευδώνυμο μοιάζει με δείκτη, αλλά δεν μπορεί να είναι ποτέ NULL και θα πρέπει να αρχικοποιηθεί κατά τον ορισμό.
- Από την στιγμή που θα οριστεί δεν μπορεί να αλλάξει και να δείχνει σε άλλη μεταβλητή.
- Μοιράζονται τον ίδιο χώρο μνήμης με την αρχική μεταβλητή.
  - Αν περάσουν σε συνάρτηση, μοιάζουν με δείκτες, αλλά είναι ποιο εύκολα
- Λύνουν το πρόβλημα του NULL σε κάποιες περιπτώσεις, αλλά δεν έχουν την ίδια δύναμη με τους pointers

Παράδειγμα με references

```
#include <assert.h>

void fpointer(int* val) {
  *val = 42;
}



int main() {
  int x;
  fpointer(&x);


  assert(x==42);
}
```

```
#include <cassert>

void freference(int& val) {
  // no need to dereference !

 // no need to check for NULL !
  val = 42;
}

int main() {
  int y;
  // no special symbols
  freference(y);
  assert(y==42);
}
```

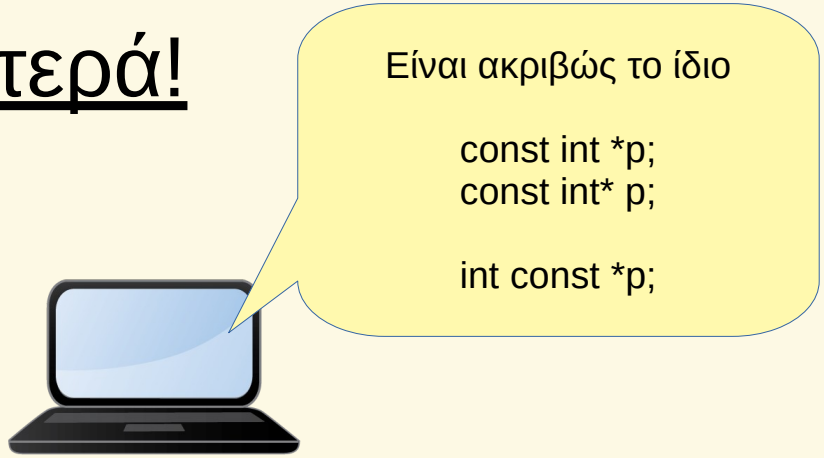
- Ο κώδικας που καλεί δεν χρησιμοποιεί ειδική σύνταξη, κάνει την συνάρτηση ευκολότερη στην χρήση.
- Μέσα στην συνάρτηση επίσης η σύνταξη είναι απλή
- Δεν είναι ανάγκη να γίνει έλεγχος για **NULL**

Κόλπο: Διαβάζοντας δηλώσεις

Από τα δεξιά προς τα αριστερά!

const int *pci;

1. pci is a variable	`const int *pci;`
2. pci is a pointer variable	`const int *pci;`
3. pci is a pointer variable to an integer	`const int *pci;`
4. pci is a pointer variable to a constant integer	`const int *pci;`



int * const p;

The **p** is a constant pointer to an int

Παραδείγματα δηλώσεων

`int *ptr;`	ptr is a pointer to an int
`const int* const ptr;`	ptr is a constant pointer to const int
`const int *ptr;` `int const * ptr;`	ptr is pointer to int const (constant int)
`int * const ptr;`	ptr is const pointer to int
`int * const * const ptr;`	ptr is const pointer to a const pointer to an int
`int **const ptr;`	ptr is const pointer to pointer to an int

Pointers with Constants

Declaration Syntax	Name	Can reassign?	Can modify pointee?
`const Type* myPtr`	Pointer-to-`const`	Yes	**No**
`Type const* myPtr`	Pointer-to-`const`	Yes	**No**
`Type* const myPtr`	`const` pointer	**No**	Yes
`const Type* const myPtr`	`const` pointer-to-`const`	**No**	**No**
`Type const* const myPtr`	`const` pointer-to-`const`	**No**	**No**

Τι συμβαίνει με το const παράδειγμα

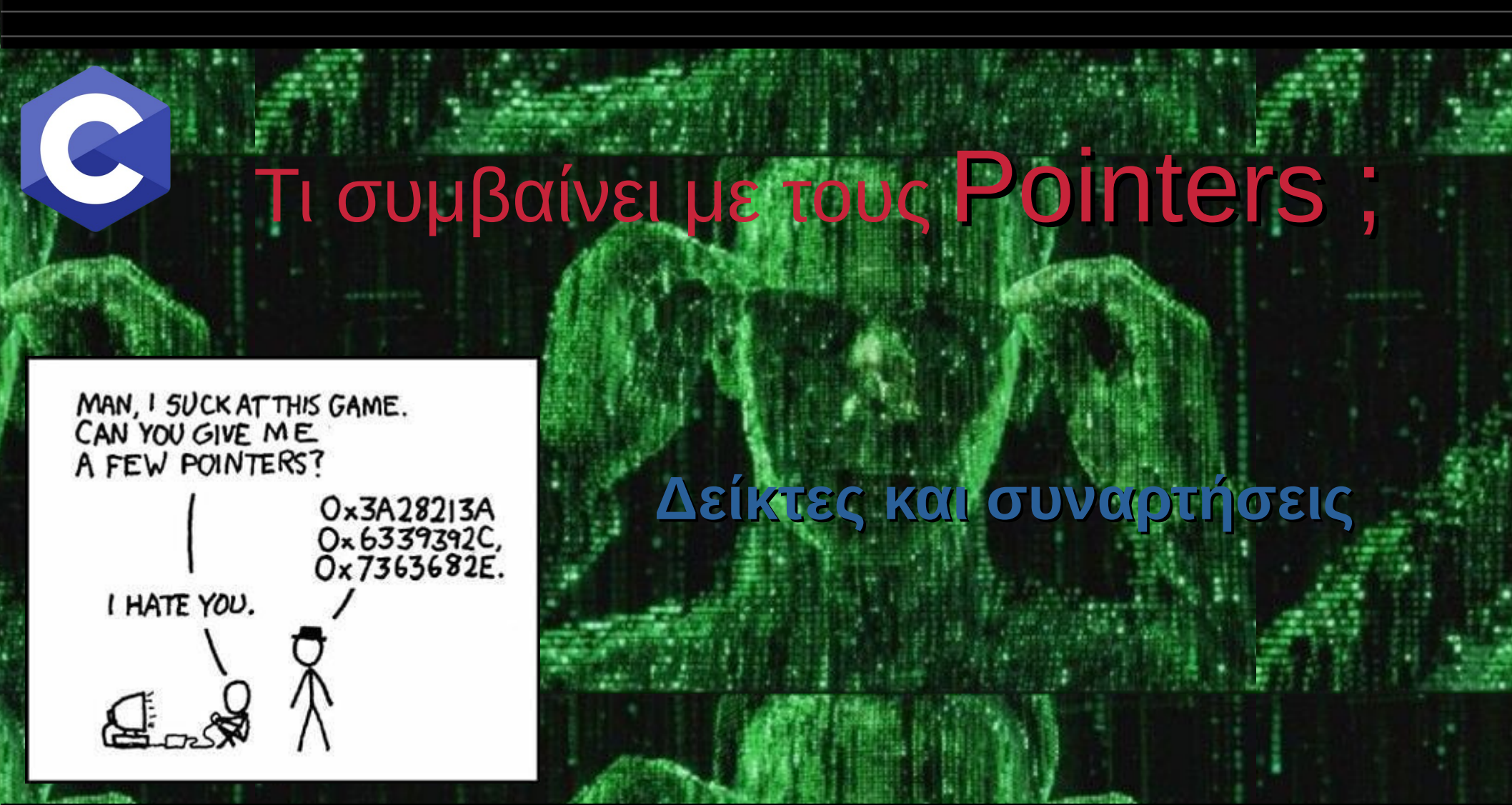
```
char a ='A';
char b ='B';

char *ptr = &a;
*ptr = b;  // change value A1
ptr = &b;  // change pointee A2

const char *ptr1 = &a;
*ptr1 = b;   // B1
ptr1 = &b;   // B2

char *const ptr2 = &a;
*ptr2 = b;   // Γ1
ptr2 = &b;   // Γ2

const char *const ptr3 = &a;
*ptr3 = b;   // Δ1
ptr3 = &b;   // Δ2
```

- Το **const** μας ήρθε από την **C++**
  - Υπάρχουν μικρές διαφορές, αλλά δεν είναι σημαντικές.
- Το **const** δεν προστατεύει την μνήμη, απλά μας βοηθά να μην κάνουμε λάθη.
  - Είναι ένα συμβόλαιο με μια συνάρτηση πως δεν θα πειράξει τα δεδομένα που περνάμε σε μια συνάρτηση με δείκτη.
  - Μπορούμε να αλλάξουμε μια τιμή ακόμα και αν είναι const, αν κάνουμε aliasing με κάποιο pointer (δηλαδή αν ένας άλλος pointer δείχνει στην ίδια θέση μνήμης)
- Αντικαθιστά κάποια από τα **#define** του προεπεξεργαστή.
- Ποιες από τις γραμμές του δίπλα κώδικά είναι λάθος;

Τι συμβαίνει με το const (λύση)

```
char a ='A';
char b ='B';

char *ptr = &a;
*ptr = b;  // change value A1
ptr = &b;  // change pointee A2

const char *ptr1 = &a;
*ptr1 = b;   // B1
ptr1 = &b;   // B2

char *const ptr2 = &a;
*ptr2 = b;   // Γ1
ptr2 = &b;   // Γ2

const char *const ptr3 = &a;
*ptr3 = b;   // Δ1
ptr3 = &b;   // Δ2
```

- Ποιες από τις γραμμές του δίπλα κώδικά είναι λάθος;
  - Διαβάζω τον κώδικα και βλέπω αν προσπαθώ να αλλάξω τιμή ή διεύθυνση μνήμης
  - Ελέγχω αν επιτρέπετε
  - Οι λάθος γραμμές είναι σημειωμένες



Τι συμβαίνει με τους Pointers ;

Δείκτες και συναρτήσεις

Πέρασμα τιμών σε συναρτήσεις

- Όταν περνάμε μια τιμή σε μια συνάρτηση τα ορίσματα αντιγράφονται στην στοίβα
- Αν τα δεδομένα είναι μεγάλα (πχ ένας πίνακας) έχουμε σπατάλη χώρου και χρόνου.
- Επειδή τα δεδομένα είναι αντίγραφο, δεν μπορεί μια συνάρτηση να αλλάξει ένα όρισμα της, ή να επιστρέφει πολλές τιμές.
- Η χρήση δεικτών λύνει αυτά τα προβλήματα.

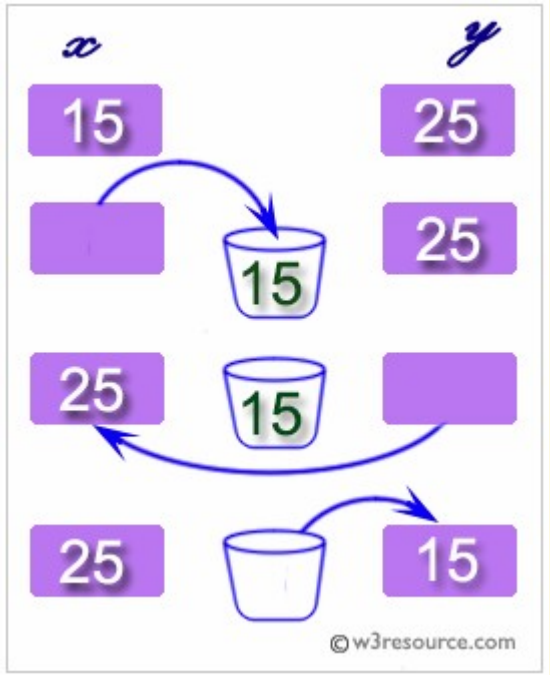
Πέρασμα με τιμή ή με αναφορά

in	void f(int x)	Με Τιμή	ΟΧΙ
in	void f(const int *x)	Με Αναφορά	ΟΧΙ
out	void f(int *x)	Με Αναφορά	ΝΑΙ
inout	void f(int *x)	Με Αναφορά	ΝΑΙ

- Ένα όρισμα μπορεί να είναι μόνο για είσοδο (in) και δεν θέλουμε να μπορεί η συνάρτηση να αλλάξει την τιμή του, να είναι για έξοδο (out) ή και τα δύο (inout).
- Αν θέλουμε να μπορεί να αλλάζει την τιμή (out,inout) θα πρέπει να το περάσουμε σαν δείκτη.
- Αν δεν θέλουμε να μπορεί να αλλάζει την τιμή, αλλά τα δεδομένα είναι μεγάλα και θέλουμε να αποφύγουμε την αντιγραφή θα το περάσουμε σαν σταθερό (const) δείκτη.
- Η C++ υποστηρίζει και τα λεγόμενα references που μοιάζουν με τους δείκτες. Αντί για δείκτες μπορούμε να περάσουμε αναφορές και η σύνταξη είναι: **void f(int &x)**

Μια συνάρτηση swap

```c
void swap(int *p, int *q) {
    int tmp;
    tmp = *p;
    *p = *q;
    *q = tmp;
}

int n1 = 15;
int n2 = 25;
swap(&n1,&n2)
```



Η συνάρτηση **swap** αντιμεταθέτει τις τιμές δύο μεταβλητών. Πρέπει να περάσουμε τις τιμές σαν δείκτες και δουλεύει μόνο με ορίσματα τύπου δείκτη σε int.

Τι συμβαίνει με τους Pointers ;

Πίνακες και δείκτες

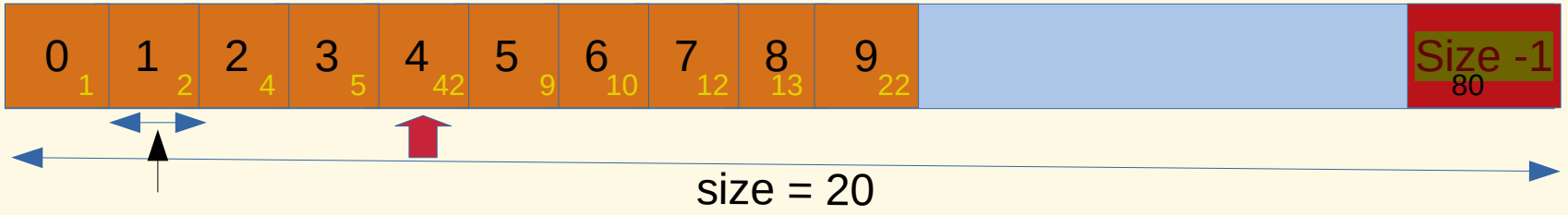
μια βαθιά σχέση αγάπης

Ένας πίνακας στην μνήμη

Το index (**δείκτης θέσης**) **4**, η τιμή **42**

Η **διεύθυνση μνήμης** **(array+4)** ή **&array[4]**

Κάπου στην στοίβα ο πίνακας **int array[size]**



- Σύνολο στοιχείων: **20,** size == 20
- Πρώτη τιμή : **array[0] → 1**
- Τελευταία τιμή : **array[size-1] → 80**
- Δεν “υπάρχει” το : **array[size]** !
- Συνολικό μέγεθος σε bytes: **size * sizeof(int)**

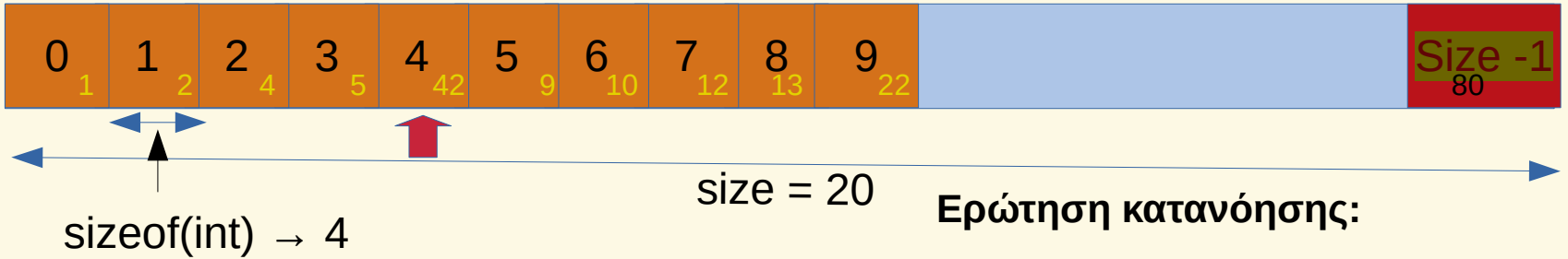
```
assert(array == &array[0]);

array[4] = 42;
assert(array[4] == *(array + 4));
assert(&array[4] == array+4);

*(array+4) = 24;
assert(aarray[4]==24);
```

Ένας πίνακας στην μνήμη

Το index (δείκτης θέσης) **4**, η τιμή **42**

Η διεύθυνση μνήμης **(array+4)** ή **&array[4]**

Κάπου στην μνήμη ένας πίνακας **int array[size]**



- Σύνολο στοιχείων: **20,** size == 20
- Πρώτη τιμή : **array[0] → 1**
- Τελευταία τιμή : **array[size-1] → 80**
- Δεν “υπάρχει” το : **array[size]** !
- Συνολικό μέγεθος σε bytes: **size * sizeof(int)**

Ερώτηση κατανόησης:

Έστω ότι το πρώτο στοιχείο (**δείκτης θέσης 0**) είναι στην θέση 100 της μνήμης.

Σε ποια θέση θα βρίσκετε το 5 στοιχείο (**δείκτης θέσης 4**) του πίνακα και τι τιμή έχει;

α. 100, β. 104, γ. 105, δ. 120

Πως βρίσκω το μέγεθος ενός πίνακα

```c
#include <stdio.h>

int main() {

    int arr[] = {10, 0, -10};
    printf("Size of int datatype is: %ld byte\n", sizeof(arr[0]));
    printf("Size of int array in bytes : %ld bytes\n", sizeof(arr));

    size_t size = sizeof(arr) / sizeof(arr[0]);

    printf("Size of int array is: %ld integers", size);

    return 0;
}
```

- Size of int datatype is:
  4 bytes
- Size of int array in bytes:
  12 bytes
- Size of int array is:
  3 integers

Προσοχή: Αν περάσω ένα πίνακα σε μια συνάρτηση χάνω την πληροφορία για το μεγέθους του πίνακα (περισσότερα στην συνέχεια)

Αριθμητικές πράξεις

- Ο τελεστής '++'/'--' θα προσθέσει/αφαιρέσει στο δείκτη το μέγεθος του στοιχείου στο οποίο δείχνει.
- Δείτε στο παράδειγμα πως μετέτρεψα το δείχτη σε ακέραιο με **cast** για να δω την διαφορά.
  - Σε 64bit Intel το μέγεθος του pointer είναι 8bytes. Άρα θέλω ένα αντίστοιχου μεγέθους ακέραιο, χωρίς πρόσημο (unsigned long int).
  - Θα μπορούσα επίσης να χρησιμοποιήσω τον τύπο **[u]intptr_t**, το πρόγραμμα μου θα είχε φορικότητα στις διάφορες αρχιτεκτονικές CPU.
  - Η C99 τον ορίζει σαν προαιρετικό (optional) και μπορεί να μην υποστηρίζετε στον compiler μας.
  - Η μετατροπή (cast) ενός δείκτη σε ακέραιο είναι πάντα μια πολύ κακή ιδέα.

```c
#include <assert.h>
#include <stdint.h>
#include <stddef.h>

int main() {
    int var = 42;
    int *ip = &var;

    int *ip2 = ip;
    ip2++;
                                cast
    uintptr_t np =  (uintptr_t) ip;
    uintptr_t np2 = (uintptr_t) ip2;

    ptrdiff_t ptrdiff = ip2 - ip;
    assert( ptrdiff== 1 );
    assert( (np2 - np) == sizeof(int) );
}
```

Παράδειγμα αύξησης δείκτη

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

const int MAX = 3;

int main() {
   int var[] = {10, 42, 50};
   int *ptr = &var;
   for (int i = 0; i < MAX; ++i) {
      printf("Address of var[%d] = %p, ", i, ptr);
      printf("Value of var[%d] = %d\n", i, *ptr);


      assert(*ptr == var[i]);
      ptr++;
   }
}
```

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

const int MAX = 3;

int main() {
   int var[] = {10, 42, 50};
   int *ptr = &var[MAX-1];

   for (int i = MAX; i > 0; --i) {

      printf("Address of var[%d] = %p, ", i, ptr);
      printf("Value of var[%d] = %d\n", i, *ptr);

      assert(*ptr == var[i-1]);
      ptr--;
   }
}
```

Pointers * και ++

`*p++`	Η τιμή που δείχνει ο *p. Μετά αύξησε τον p
`*(p++)`	
`(*p)++`	
`*++p`	Αύξησε τον p. Το αποτέλεσμα είναι εκεί που δείχνει ο p+1
`*(++p)`	
`++*p`	
`++(*p)`	

Ισότητες και ανισότητες

- Ισχύουν οι γνωστοί τελεστές:
  ==, !=, <, >, <=, >=
- Να προσέχουμε η σύγκριση να έχει νόημα, πχ δείκτες στον ίδιο πίνακα.

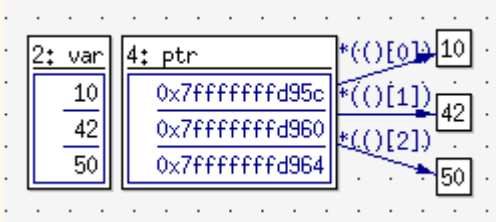
```c
#include <stdio.h>
#include <assert.h>

const int MAX = 3;

int main() {
   int var[] = {10, 42, 50};
   int *ptr= var; // same as &var
   int i=0;

   while( ptr <= &var[MAX-1]  ){

      printf("Address of var[%d] = %p, ", i, ptr);
      printf("Value of var[%d] = %d\n", i, *ptr);

      assert(*ptr == var[i]);
      ptr++;
      i++;
   }
}
```

Ένας πίνακας με pointers

- Η δήλωση

```
int *ptr[MAX];
```

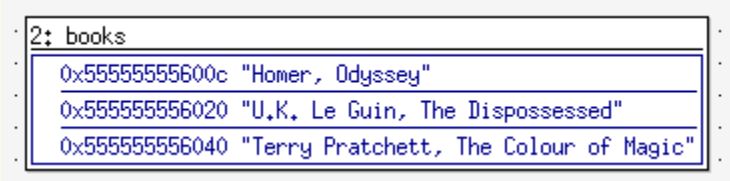
είναι ένας **πίνακας** με **δείκτες** σε **int**




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

const int MAX = 3;

int main() {
   int var[] = {10, 42, 50};
   int *ptr[MAX]; // A table full of pointers
   int i=0;

   for(int i=0; i< MAX; ++i){
      ptr[i] = &var[i]; // Assign the ADDRESS of int
      printf("Value of var[%d] = %d, ", i, var[i]);
      printf("Value of ptr[%d] -> %d\n", i, *ptr[i]);
      assert(*ptr[i] == var[i]);
   }
}
```

Ένας άλλος πίνακας με δείκτες

```c
#include <stdio.h>

const int MAX = 3;

int main() {
   char *books[] = {
         "Homer, Odyssey",
         "U.K. Le Guin, The Dispossessed",
         "Terry Pratchett, The Colour of Magic"
   };

   for (int i = 0; i < MAX; ++i) {
      printf("Value of books[%d] = %d\n", i, books[i]);
   }
}
```



Τι συμβαίνει με τους Pointers ;

Πίνακες και δείκτες

ξετυλίγοντας το κουβάρι

Ένα παράδοξο πρόγραμμα

```c
#include <stdio.h>
int main() {
    int arr[4] = {5, 1, 8, 9};

    for (int i = 0; i < 4; ++i) {
        printf("%d, ", arr[i]);
    }
    printf("\n");

    for (int i = 0; i < 4; ++i) {
        printf("%d, ", i[arr]);
    }
    printf("\n");
}
```

- Το πρώτο for διασχίζει και τυπώνει όλα τα στοιχεία του πίνακα.
- Αλλά τι κάνει το δεύτερο for ;
  - Η λύση στο παράδοξο είναι στην αριθμητική διευθύνσεων
    - `arr[i]` → `*(arr +i)`
    - `i[arr]` → `*(i + arr)`
  - Άρα είναι ισοδύναμα

Είναι το ίδιο πρόγραμμα;

```c
#include<stdio.h>

int main() {
    char hello[] = "Hello World";
    hello[6] = 'Z';
    printf("%s", hello);
}
```

```c
#include<stdio.h>

int main() {
    char *hello = "Hello World";
    hello[6] = 'Z';
    printf("%s", hello);
}
```

Είναι το ίδιο πρόγραμμα; **ΟΧΙ**

```c
#include<stdio.h>

int main() {
    char hello[] = "Hello World";
    hello[6] = 'Z';
    printf("%s", hello);
}
```

```c
#include<stdio.h>

int main() {
    char *hello = "Hello World";
    hello[6] = 'Z';
    printf("%s", hello);
}
```

```
Process finished with exit code 139
(interrupted by signal 11: SIGSEGV)
```

Τι έγινε;

- Το πρώτο πρόγραμμα έτρεξε κανονικά
- Το δεύτερο πρόγραμμα κατέρρευσε
- Γιατί;
  - Ένας πίνακας δεν είναι ισοδύναμος με ένα δείκτη;
  - Ας διερευνήσουμε το ζήτημα ...

Άλλο πίνακας και άλλο pointer

- Τόσο ο τύπος όσο και το μέγεθος τους στην μνήμη δεν είναι ίδια.
  - Ένας πίνακας ξέρει πόσο χώρο καταλαμβάνει στην μνήμη.
  - Ένας δείκτης πιάνει πάντα τον ίδιο χώρο στην μνήμη (σε συνήθεις αρχιτεκτονικές).
- Τα κείμενα υπάρχουν σε διαφορετικούς “χώρους μνήμης”
  - Με ονόματα **.text** και **.rodata**.
  - Αλλά μπαίνουμε πολύ βαθιά και δεν θα πούμε περισσότερα για τώρα.
  - Η συμπεριφορά μπορεί να διαφέρει ανάλογα με τον compiler και την CPU.

	Τύπος		Χώρος μνήμης
`hello1`	`char *`	`pointer`	`8 bytes`
`hello2`	`char[12]`	`table`	`n(12) bytes`

```
char *hello1 = "Hello World";  hello1: "Hello World"
char hello2[] = "Hello World";  hello2: char [12]

size_t size1= sizeof(hello1);  size1: 8    hello1: "Hello World"
size_t size2= sizeof(hello2);  size2: 12    hello2: char [12]
```

Ο τρόπος της C++

- Η **C++** είναι ποιο αυστηρή και πιάνει τέτοια λάθη.
  - Αλλάζοντας την κατάληξη του αρχείου σε '.cpp'

```
main.cpp: In function 'int main()':
main.cpp:6:20: warning: ISO C++ forbids converting a string constant to 'char*' [-Wwrite-strings]
    6 |     char *hello1 = "Hello World";
      |                    ^~~~~~~~~~~~~
```

  - Το '**string constant**' ή '**string literal**' δείχνει πως το κείμενο είναι μέσα σε κλειδωμένη για εγγραφή μνήμη.
- Είναι καλύτερο να χρησιμοποιούμε τον compiler της **C++** ακόμα και αν γράφουμε απλή **C.**
- Πρέπει να ορίσουμε τον δείκτη σαν const, για να δείξουμε πως δεν αλλάζει τιμή

```
const char *hello1 = "Hello World";
char hello2[] = "Hello World";
```

Τιμές, διευθύνσεις και δείκτες

`char hello[] = "Hello World";`	`const char *hello = "Hello World";`
Είναι ένας πίνακας	Ένας pointer στο "Hello World"
`assert( sizeof(hello) == 12 )`	`assert( sizeof(hello) == sizeof(void*) )`
Τα hello και &hello είναι **ίδια**	Τα hello και &hello είναι **διαφορετικά**
Η τιμή ενός πίνακα είναι η διεύθυνση μνήμης του πρώτου του στοιχείου. Η τιμή του δείκτη ~~είναι~~ δείχνει μια άλλη διεύθυνση μνήμης από την δική του διεύθυνση στην μνήμη.	
hello = "Geloo Zorld"; Το hello είναι μια διεύθυνση στην μνήμη, και το "Geloo Zorld" είναι μια άλλη διεύθυνση στην μνήμη, άρα **δεν είναι δυνατό**.	hello = "Geloo Zorld"; Ναι γιατί όχι; Μια χαρά. Απλά αλλάζουμε την διεύθυνση μνήμης που δείχνει ο pointer. Μα είναι const! Τι μας λες; **const** είναι εκεί που δείχνει, **και μπορεί άνετα** να δείχνει κάπου αλλού.
`hello++;` **Δεν έχει νόημα**, είναι πίνακας	**`hello++;`** **Ναι έχει νόημα**, αυτά κάνουμε με τους pointers

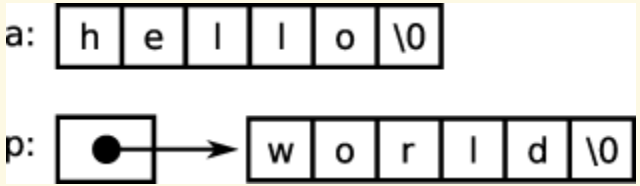
Initialization vs Assignment

```c
// initialization
int a = 3;
// Assignment
a = 4;

// initialization
char hello[]="Hello";
// Assignment INVALID
hello = 'Zello';
```

- Οι πράξεις τις **αρχικοποιήσης (initialization)** μιας τιμής και της **ανάθεσης (assignment)** μιας τιμής μοιάζουν αλλά είναι εντελώς διαφορετικές πράξεις.
- Μπορούμε να κάνουμε μια αρχική ανάθεση πάντα, αλλά δεν μπορούμε να αλλάξουμε την αρχική τιμή σε πολλές περιπτώσεις.

Διευθύνσεις και τιμές στην μνήμη

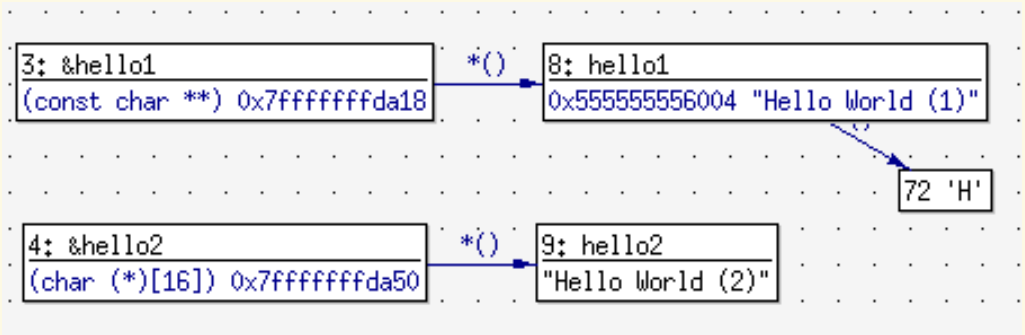
```
char a[] = "hello";
char *p = "world";
```



- Είναι χρήσιμο να συνειδητοποιήσουμε ότι μια αναφορά όπως το x[3] δημιουργεί **διαφορετικό κώδικα μηχανής** ανάλογα με το αν το x είναι ένας πίνακας ή ένας δείκτης.
- Λαμβάνοντας υπόψη τις παραπάνω δηλώσεις, όταν ο μεταγλωττιστής βλέπει την έκφραση a[3], δημιουργεί κώδικα για να ξεκινήσει από τη θέση "a", να μετακινηθεί τρία πέρα από αυτήν και να πάρει τον χαρακτήρα εκεί.
- Όταν βλέπει την έκφραση p[3], δημιουργεί κώδικα για να ξεκινήσει από τη θέση "p", να πάρει την τιμή του δείκτη εκεί, να προσθέσει τρεις στον δείκτη και τελικά να πάρει τον χαρακτήρα που δείχνει.
- Με άλλα λόγια, το a[3] είναι τρία μέρη μετά (την αρχή) του αντικειμένου που ονομάζεται a, ενώ το p[3] είναι τρία μέρη μετά το αντικείμενο που δείχνει το p.
- Στο παραπάνω παράδειγμα, τόσο το a[3] όσο και το p[3] είναι ο χαρακτήρας ‘l’, αλλά ο μεταγλωττιστής φτάνει εκεί διαφορετικά.

Διευθύνσεις και τιμές (ξανά)

```
const char *hello1 = "Hello World (1)";
char hello2[] = "Hello World (2)";
```



- Η τιμή ενός πίνακα είναι η διεύθυνση μνήμης του πρώτου του στοιχείου.

  Ο πίνακας είναι στην διεύθυνση μνήμης **0x7...da18** και καταλαμβάνει 16 συνεχή bytes.

- Η τιμή του δείκτη ~~είναι~~ δείχνει μια άλλη διεύθυνση μνήμης από την δική του διεύθυνση στην μνήμη.

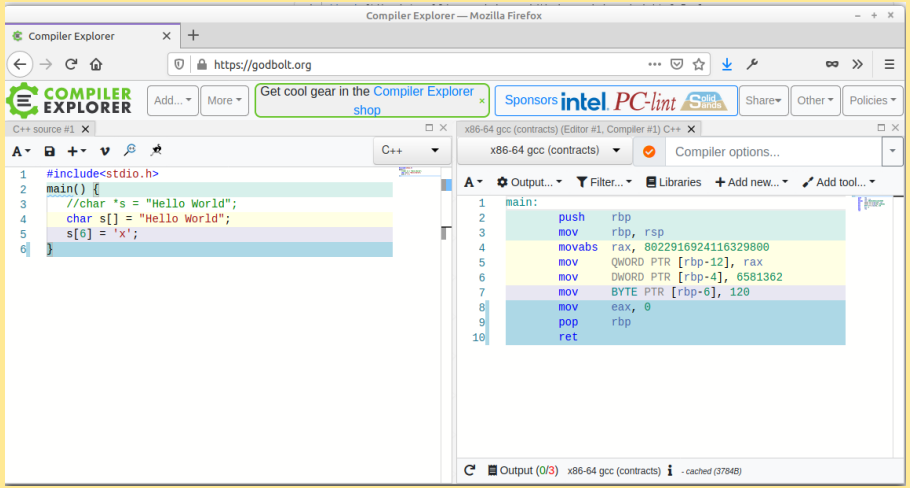
  Ο δείκτης βρίσκετε στην διεύθυνση μνήμης **0x7ff...da50** πολύ κοντά στην διεύθυνση μνήμης του πίνακα, κάτι που περιμένουμε μιας και τα δυο είναι κοντά στην ίδια στοίβα.

  Αλλά δείχνει στην διεύθυνση μνήμης **0x555...004** πολύ χαμηλά στον χώρο διευθύνσεων. Σε αυτή την θέση θα βρούμε το κείμενο **“Hello World (1)”**.

- Δεν είναι ανάγκη να ξέρουμε καλά το δεκαεξαδικό σύστημα αρίθμησης για να διαπιστώσουμε πόσο κοντά είναι οι τιμές.

- Οι ακριβείς τιμές των διευθύνσεων δεν έχουν καμία σημασία και δεν θα είναι αναγκαστικά ίδιες σε κάθε μηχανή.

Σε επίπεδο κώδικα μηχανής

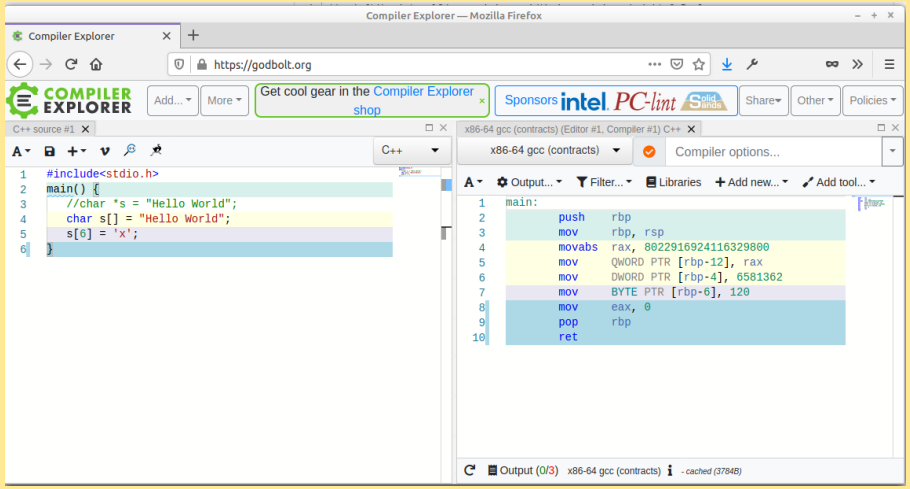


- Δείτε τις διαφορές στον κώδικα μηχανής με την βοήθεια του

  https://godbolt.org/

- Με την βοήθεια του `google` προσπαθήστε να καταλάβετε τι περίπου κάνουν οι εντολές.
  - Προσέξτε τον χρωματικό κώδικα.
  - Μόνο το κίτρινο μας ενδιαφέρει εδώ.

Σε επίπεδο κώδικα μηχανής



- Είναι καλή ιδέα να κοιτάζεις μια φορά τον μήνα τον κώδικά μηχανής
  - Ακόμα και αν δεν τον καταλαβαίνεις ούτε κατά 10%
- Καταλαβαίνεις καλύτερα την γλώσσα και πως το πρόγραμμα θα γίνει καλύτερο
  - Θα δεις πως οι compiler σήμερα είναι **διαβολικά** έξυπνοι και θα γράψουν καλύτερο κώδικά από εσένα αν ήξερες κώδικά μηχανής.
  - Αρκετά πράγματα στην C όμως έχουν μείνει στην εποχή που είχαμε 8KB μνήμη.
    - Για παράδειγμα η δεσμευμένη λέξη **register**

Διαφορά πίνακα και δείκτη

- Ο πίνακας είναι μια σειρά από συνεχείς θέσεις στην μνήμη.
  Το πλήθος των στοιχείων του είναι γνωστό και σταθερό.
- Ο δείκτης είναι μια μεταβλητή που περιέχει μια διεύθυνση μνήμης και δείχνει σε αυτήν
- Είναι λοιπόν δυο διαφορετικά πράγματα.
  Γιατί όμως πολλοί λένε πως είναι το ίδιο πράγμα;
  - **Αριθμητική των δεικτών** και προσπέλαση των στοιχείων του πίνακα με “δείκτες θέσης”, δηλαδή το `array[3]`, που είναι ισοδύναμο με το `*(array + 3)`.
  - **Συναρτήσεις**. Όταν ένας πίνακας περνάει σε συνάρτηση έχουμε την λεγόμενη **κατάρρευση τύπου (decay)**. Ο πίνακας εκφυλίζετε σε ένα δείκτη στο πρώτο του στοιχείο.
    - Επειδή χάνουμε την πληροφορία για το μέγεθος του πίνακα θα πρέπει να περάσει και αυτό σαν όρισμα στην συνάρτηση. Ο δείκτης σε κάποιο στοιχείο μαζί με ένα μήκος καλείτε **span**.

Περισσότερα για το **span**

- Το span είναι ένας συνδυασμός ενός δείκτη σε πίνακα (που δείχνει σε κάποιον συγκεκριμένο τύπο και δεν είναι void) και ενός μήκους.
  - Δεν είναι ανάγκη να ξεκινά από το πρώτο στοιχείο
  - Τυπικά περνάμε ένα span σε συναρτήσεις που παίρνουν πίνακες σαν είσοδο.
- Ένας εναλλακτικός τρόπος είναι με χρήση **φρουρών (sentinel)**. Δηλαδή ειδικών τιμών που μπαίνουν στο τέλος του πίνακα και το σημαδεύουν
  - Τυπικό παράδειγμα η χρήση του **‘\0’** για το τέλος των συμβολοσειρών.
  - Δεν είναι πάντα εύκολο να βρεθεί μια τέτοια τιμή.
  - Συχνά δεν είναι αποτελεσματικό, αν πρέπει να γνωρίζουμε από τα πριν το μήκος

Ανέκδοτο

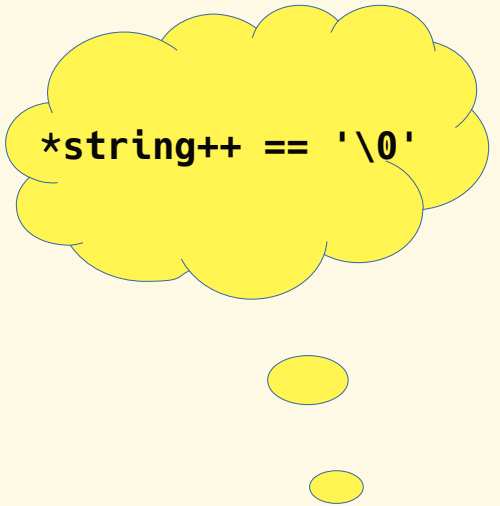
- Ο Μητσάρας πιάνει δουλεία σε έναν εργολάβο. Του δίνει ένα κουτί μπογιά και μια βούρτσα και του λέει να βάψει τις διακεκομμένες γραμμές ενός δρόμου. Την πρώτη μέρα βγάζει ένα κουτί βαφής στο δρόμο και τελειώνει 300 μέτρα από το δρόμο. "Αυτό είναι πολύ καλό!" λέει το αφεντικό του, "είσαι γρήγορος εργαζόμενος, ο καλύτερος!" και τον πληρώνει παραπάνω.
- Την επόμενη μέρα ο Μητσάρας βάφει μόνο 150 μέτρα. "Λοιπόν, αυτό δεν είναι τόσο καλό όσο χθες, αλλά είσαι ακόμα γρήγορος εργαζόμενος. 150 μέτρα είναι μια χαρά", και του πληρώνει το μεροκάματο.
- Την επόμενη μέρα ο Μητσάρας βάφει 30 μέτρα από το δρόμο. "Μόνο 30!" φωνάζει το αφεντικό του. “Αυτό είναι απαράδεκτο! Την πρώτη μέρα μου έβγαλες δέκα φορές περισσότερη δουλειά! Τι συμβαίνει"
- “Δεν μπορώ αφεντικό, όσο και να τρέχω δεν μπορώ να ξεκινάω από την αρχή του δρόμου να βάζω μπογιά στο πινέλο και μετά να βάφω στο τέλος”

- Αστείο; Κάπως έτσι δουλεύουν οι συναρτήσεις χειρισμού συμβολοσειρών όπως η **stlen()** και ή **strcat()** στην γλώσσα C που χρησιμοποιούν φρουρούς.

https://www.joelonsoftware.com/2001/12/11/back-to-basics/

Ο Αλγόριθμος του Μητσάρα

```c
#include <assert.h>
#include <string.h>

size_t mitsilength(char * string);

int main(void) {
    char *text="We are the word";
    assert( strlen(text) ==  mitsilength(text) );
}

size_t mitsilength(char *string){
    int length =0;
    while(*string++) length++;
    return length;
}
```



Assymetric bounds, or C++ ranges

- Είναι ένας δείκτης στον πρώτο στοιχείο του πίνακα μαζί με ένα δείκτη στο επόμενο μετά το τέλος στοιχείο του πίνακα.
  - Πλήθος στοιχείων: `end-start`
  - Αν `end==start` έχουμε άδειο πίνακα
  - Πάντα `end≥start`

Έλεγχος ορίων στους πίνακες

- Σε αντίθεση με άλλες γλώσσες η C δεν κάνει έλεγχο για τα όρια των πινάκων. Η ALGOL το είχε
  - Αυτό αυξάνει κατά πολύ την ταχύτητα εκτέλεσης
  - Αλλά το 50% των προβλημάτων ασφάλειας έχουν να κάνουν με τα λεγόμενα buffer overflow attacks
  - Χρησιμοποιήστε ειδικά εργαλεία για εντοπισμό προβλημάτων
  - Χρησιμοποιήστε της ασφαλείς εκδόσεις της βιβλιοθήκης της **C**.
    - Και ποτέ την συνάρτηση `gets()`.
      Είναι ανασφαλής και θα καταργηθεί στην επόμενη έκδοση της **C**.

Ασφαλείς συναρτήσεις

Table 2.7 Truncating Copy Functions

	Standard/TR	Buffer Overflow Protection	Guarantees Null Termination	May Truncate String	Allocates Dynamic Memory	Checks Runtime Constraints
`strncpy()`	C99	Yes	No	Yes	No	No
`strlcpy()`	OpenBSD	Yes	Yes	Yes	No	No
`strndup()`	TR 24731-2	Yes	Yes	Yes	Yes	No
`strncpy_s()`	C11	Yes	Yes	No	No	Yes

Table 2.8 Truncating Concatenation Functions

	Standard/TR	Buffer Overflow Protection	Guarantees Null Termination	May Truncate String	Allocates Dynamic Memory	Checks Runtime Constraints
`strncat()`	C99	Yes	No	Yes	No	No
`strlcat()`	OpenBSD	Yes	Yes	Yes	No	No
`strncat_s()`	C11	Yes	Yes	No	No	Yes

Η τυπική βιβλιοθήκη είχε πολλές “προβληματικές” συναρτήσεις. Η C11 παρέχει καλύτερες εναλλακτικές, αλλά πολλά βιβλία και παραδείγματα δεν έχουν ενημερωθεί.

Μάθετε και χρησιμοποιήστε τις ασφαλείς εκδόσεις.

Καλή αναφορά στο τι υπάρχει

- με το zeal
- ή στο https://en.cppreference.com/w
- Δείτε και την συνάρτηση `strdup()`

Πηγή πίνακα: Robert Seacord, “Secure Coding in C and C++”

Προβλήματα με την **strcmp**

```c
int main(void) {
    const char* str1 = NULL;
    const char* str2 = "";
    const char *str3="Hello";
    const char *str4="Hello1";

    // Segmentation fault
    // assert( strcmp(str1, str3) == false);
    // assert( strcmp(str3, str1) == false);
    // assert( strcmp(str2, str3) == false);
    // assert( strcmp(str3, str2) == false);
    // assert( strcmp(str2, str3) == false);
    // assert( strcmp(str3, str2) == false);

    /* Neither true or false */
    // assert( strcmp(str3, str4) == false);
    // assert( strcmp(str4, str3) == false);
    // assert( strcmp(str3, str4) == true);
    // assert( strcmp(str4, str3) == true);

    assert( strcmp(str3, str3) == true);
}
```

- Τα ανενεργά assetions είτε προκαλούν κατάρρευση του προγράμματος, είτε δεν μπορούν να περάσουν ούτε με **true** ούτε με **false**.
  - Προσπαθήστε να βρείτε το γιατί.
  - Τι ακριβώς επιστρέφει η συνάρτηση **strcmp()**;

Η συνάρτηση **strcmp**

```c
#include <stdio.h>

//#include <string.h>
int strcmp( const char *lhs, const char *rhs );

void demo(const char *lhs, const char *rhs) {
    int rc = strcmp(lhs, rhs);

    const char *rel = rc < 0 ? "precedes" : rc > 0
            ? "follows" : "equals";

    printf("[%s] %s [%s]\n", lhs, rel, rhs);
}

int main(void) {
    const char *string = "Hello World!";
    demo(string, "Hello!");
    demo(string, "Hello");
    demo(string, "Hello there");
    demo("Hello, everybody!" + 12, "Hello, somebody!" + 11);
}
```

```
[Hello World!] precedes [Hello!]
[Hello World!] follows [Hello]
[Hello World!] precedes [Hello there]
[body!] equals [body!]
```

Επιστρέφει **int** και όχι **bool**.

Το πρόσημο του αποτελέσματος είναι το πρόσημο της διαφοράς μεταξύ των τιμών του πρώτου ζεύγους χαρακτήρων (ως **unsigned char)** που διαφέρουν στις συμβολοσειρές που συγκρίνονται.

https://en.cppreference.com/w/c/string/byte/strcmp

Μια πίσω πόρτα σε **Intel** επεξεργαστές

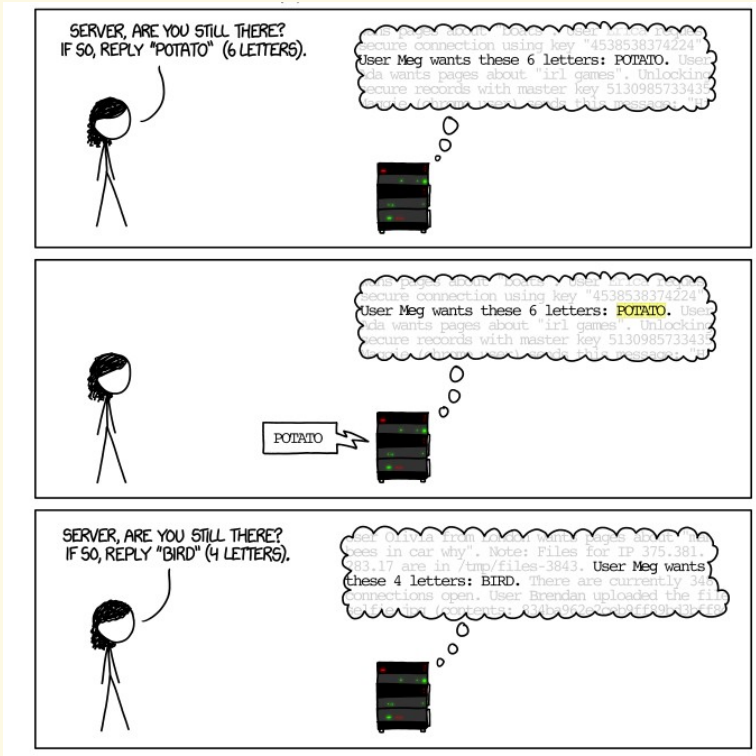
- Μέσα στους επεξεργαστές της Intel υπάρχει ένας κρυφός επεξεργαστής που τρέχει το δικό του λειτουργικό και είναι ανεξάρτητος από τον επεξεργαστή και ο προγραμματιστής δεν μπορεί να το ελέγξει.
  - Αυτό είναι χρήσιμο για απομακρυσμένη διαχείρηση, όταν κάποιος πρέπει να διαχειριστεί χιλιάδες **servers**.
  - Η πρόσβαση γίνετε μέσα από ένα **web intreface**.
- Ανακαλύφθηκε κάποιο κενό ασφαλείας όπου οποιοσδήποτε μπορεί να συνδεθεί αν απλά στείλει ένα κενό κωδικό χρήστη. Σήμερα το πρόβλημα έχει βελτιωθεί με ένα ενημερωμένο **microcode**.
- Το πρόβλημα ήταν στην “ασφαλή” συνάρτηση **strncp**. Οι παρακάτω δυο γραμμές κώδικά είναι ακριβώς ίδιες, γιατί η **strncmp** σταματά στην πρώτη διαφορά που θα βρει.

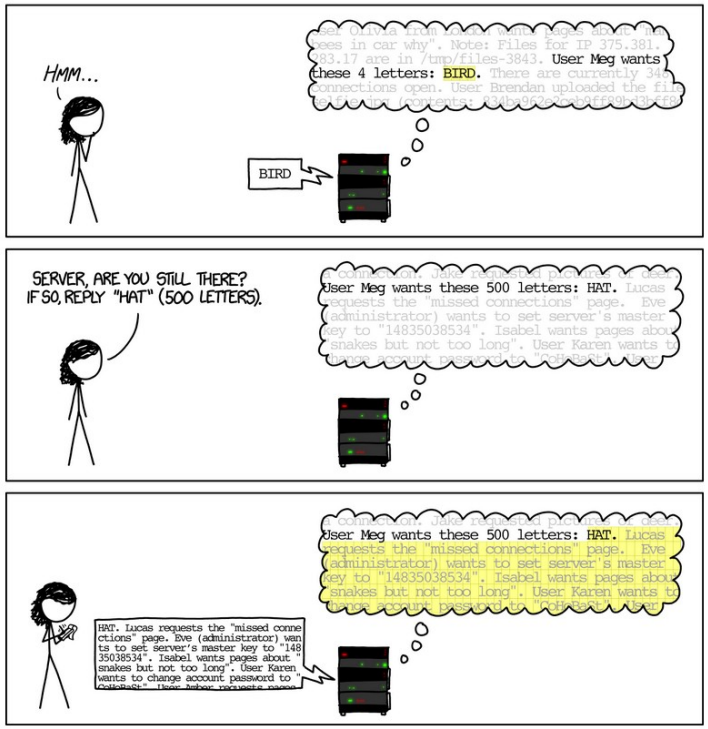
```
strncmp("6629fae49393a05397450978507c4ef1","",0);
strncmp("","",0);
```

Μια υλοποίηση της **strncmp**

- Ξέρουμε ότι χρειάζεται για να φτιάξουμε την δική μας συνάρτηση **strncp**.
  - Υλοποιήστε μια συνάρτηση **mystrncmp** που να αναπαράγει το bug της Intel.
  - Πώς μπορεί να γίνει ασφαλής;
  - Αποδείξτε το με τα κατάλληλα asserts.
  - Θα επιλέγατε να επιστρέφει **bool** ή **int** και γιατί;
  - Ελέγξτε αν η συνάρτηση της βιβλιοθήκης του μεταγλωττιστή σας είναι ασφαλής.

HOW THE HEARTBLEED BUG WORKS:





Τι συμβαίνει με τους Pointers ;

Πίνακες και δείκτες

διάφορα θέματα

Διαφορά “array”, “&array” για “array[5]”

```c
int main() {
   int array[5];

   printf("array=%p : &array=%p\n", array, &array);
   printf("array+1 = %p : &array + 1 = %p\n", array + 1, &array + 1);

   printf("\narray=%u : &array=%u\n", array, &array);
   printf("array+1 = %u : &array + 1 = %u\n", array + 1, &array + 1);

}
```

```
array=0x7fffa8dbdc50 : &array=0x7fffa8dbdc50
array+1 = 0x7fffa8dbdc54 : &array + 1 = 0x7fffa8dbdc64

array=2832981072 : &array=2832981072
array+1 = 2832981076 : &array + 1 = 2832981092
```

- Η τιμή του **array** και του **&array** είναι η ίδια (0x7fffa8dbdc50 στο δεκαεξαδικό ή 2832981072 στο δεκαδικό χωρίς πρόσημο)
- Μπορεί να μπείτε στον πειρασμό να πείτε ότι είναι ίδια, αλλά οι pointers έχουν διαφορετικούς τύπους. Το **array** είναι ένας δείκτης στο πρώτο στοιχείο του πίνακα, ενώ το **&array** είναι ένας δείκτης σε ένα πίνακα 5 στοιχείων και αυτά τα στοιχεία είναι int.
- Άρα η αριθμητική των δεικτών θα είναι διαφορετική. Θα προστεθεί 4 στην πρώτη περίπτωση, ενώ θα προστεθεί 20 στην δεύτερη περίπτωση.

Διαφορά ‘char a’ και ‘char a[1]’

```c
#include <stdio.h>

int main ()
{
   char a1 = 'A';
   char a2[1] = {'A'};

   printf("%d  %p %d", a1, a2, *a2);

   return 0;
}
```

```
65  0x7ffdca11fc47 65
```

- Το ‘char a’ είναι ένας χαρακτήρας.
  Η τιμή του είναι ο αριθμός 65 (από τον πίνακα ASCII).
- Το ‘char a[1]’ είναι ένας πίνακας.
  Η τιμή του είναι μια διεύθυνση μνήμης.
- Θα πρέπει να χρησιμοποιήσουμε έμμεση αναφορά για να πάρουμε την τιμή του χαρακτήρα.
  - Ισοδύναμα: `*a2, a2[0], *(a2+0)`.

Διαφορά char[] και char[n]

```
char hello1[] = "Hello";

char hello2[5] = "Hello";
```

Ισοδύναμοι ορισμοί:

```
char hello1[] = {'H','e','l','l','o', '\0'};

char hello2[5] = {'H','e','l','l','o'};
```

- Οι δύο ορισμοί δεν είναι ισοδύναμοι καθώς ο πρώτος θα προσθέσει μαγικά το σύμβολο τερματισμού
- Άρα ο πρώτος πίνακας θα έχει 6 θέσεις και όχι 5 που είναι τα γράμματα της λέξης.
- Δεν τολμώ να σκεφτώ τι θα γίνει αν πάει κάποιος να τυπώσει το δεύτερο string.

Τι είναι το undefined behavior

EXAMPLE 8: The declaration

```
char s[] = "abc", t[3] = "abc";
```

defines "plain" char array objects `s` and `t` whose elements are initialized with character string literals.

This declaration is identical to

```
char s[] = { 'a', 'b', 'c', '\0' },
t[] = { 'a', 'b', 'c' };
```

The contents of the arrays are modifiable. On the other hand, the declaration

```
char *p = "abc";
```

defines `p` with type "pointer to char" and initializes it to point to an object with type "array of char" with length 4 whose elements are initialized with a character string literal. If an attempt is made to use `p` to modify the contents of the array, the behavior is undefined.

Απόσπασμα από το C99 πρότυπο.

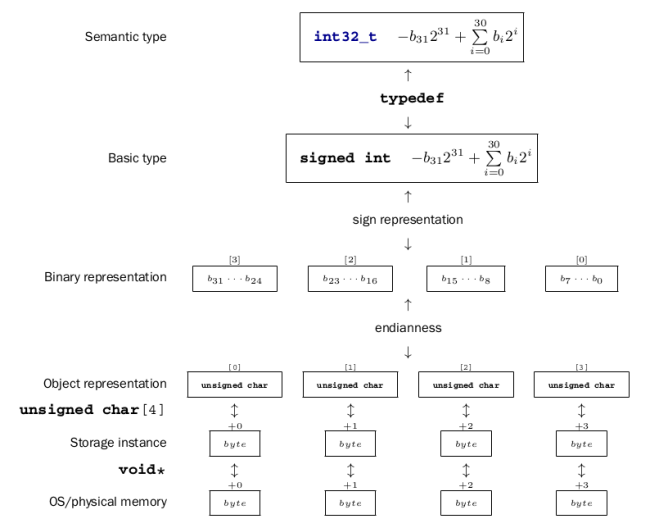
- Αν κάτι είναι **undefined behavior** ο compiler μας μπορεί να κάνει κυριολεκτικά ότι θέλει !
  - Είναι συχνά πηγές λαθών και πρέπει να ελέγχουμε την ύπαρξη τους. Ο compiler μπορεί συχνά να τα εντοπίσει πχ ο gcc με την σημαία -fsanitize.
- Το **unspecified behavior** είναι αν δεν υπάρχει καν πρόβλεψη από το πρότυπο της γλώσσας για το τι πρέπει να κάνει.
  - Και το πρόγραμμα μπορεί να έχει άλλη συμπεριφορά, ανάλογα με τον compiler που χρησιμοποιήθηκε
  - Παράδειγμα: σειρά υπολογισμού των ορισμάτων μιας συνάρτησης `fun(fun1(), fun2());`
    - `Πότε θα προξενήσει προβλήματα αυτό;`
- Το **implementation-defined** είναι όψεις που κάθε compiler **επιλέγει** πως θα το κάνει και το **τεκμηριώνει** στα βιβλία του
  - Παράδειγμα `sizeof(int)`.

Οι δυναμικοί πίνακες (VLA) της C99

```
float read_and_process(int n) {
    float values[n];

    for (int i = 0; i < n; ++i)
        values[i] = read_val();

    return process(n, values);
}
```

- Στην C99 μπορούμε να ορίσουμε πίνακες δυναμικά χωρίς την χρήση της `malloc/free`!
- Προβλήματα
  - Οι τιμές του πίνακα είναι στην στοίβα (και όχι στον σωρό) και η στοίβα δεν μπορεί να έχει μεγάλο μέγεθος.
  - Ο κώδικάς δεν είναι αποδοτικός
  - Μπορεί να μην το υποστηρίζει ο compiler της C++
- Προσέξτε την κατάρρευση τύπου και πως περνάμε τον πίνακα σε μια συνάρτηση σαν ένα **span**.
  - Εδώ το πέρασμα είναι ασφαλές, αλλά ο δείκτης θα παύσει να είναι έγκυρος μόλις τερματίσει η συνάρτηση `read_and_process()`.
  - Με χρήση της malloc, ο δείκτης θα είναι έγκυρος μέχρι την κλήση της free.
  - Αλλά προσοχή. Δεν πρέπει να χάσουμε τον δείκτη, γιατί τότε πως θα καλέσουμε την free();
  - Σε αυτή την περίπτωση θα έχουμε ένα **memory leak**. Υπάρχουν εργαλεία που τα ανιχνεύουν.

Το μοντέλο μνήμης της C

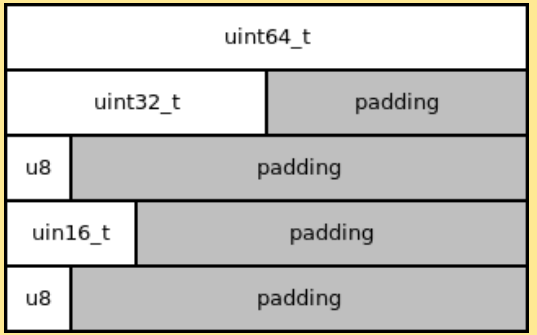


- Ένας τύπος όπως ο int32_t είναι ένα typedef στον signed int.
- Και είναι αποθηκευμένος στην μνήμη εδώ σαν συμπλήρωμα του 2, δηλαδή κάποια δυαδική αναπαράσταση.
- Θα πάρει 4 θέσεις στην μνήμη (στην συγκεκριμένη CPU), και η μνήμη μπορεί να ιδωθεί σαν ένας πίνακα 4 χαρακτήρων. Γενικά για τον A : unsigned char[sizeof A]
- Εξ ορισμού και πάντα sizeof(char) == 1
- Η σειρά αποθήκευσης στην μνήμη μπορεί να είναι διαφορετική από την αναπαράσταση στην CPU (Endianness).
- Η πρόσβαση σε αυτό το επίπεδο στην μνήμη μαζί με χαρακτηριστικά όπως τα **unions** και τα **bit fields** κάνουν την **C** μια γλώσσα για να γράφουμε λειτουργικά συστήματα όπως το **UNIX**.
- Μια εποχή το μοντέλο της μνήμης της C ταίριαζε με την πραγματική αρχιτεκτονική του υπολογιστή, αλλά σήμερα τα πράγματα είναι λίγο ποιο πολύπλοκα με τα caches, τους πολλούς πυρήνες την ιδεατή μνήμη και πολλά νήματα να τρέχουν ταυτόχρονα.

Padding και alignment

```c
#include <stdint.h>

typedef struct structA {
    uint64_t bit64;
    uint32_t bit32;
    uint16_t bit16;
    uint8_t  byteA;
    uint8_t  byteB;
} packed;

typedef struct structB {
    uint64_t bit64;
    uint32_t bit32;
    uint8_t  byteA;
    uint16_t bit16;
    uint8_t  byteB;
} unpacked;
```

- Οι διάφοροι τύποι μπορούν να υπάρχουν σε συγκεκριμένες θέσεις στην μνήμη (alignment).
- Σε ένα struct η σειρά των στοιχείων δεν αλλάζει και έχει σημασία.
  - Τι θα μπορούσε να συμβεί αν κάθε compiler άλλαζε την σειρά και δυο προγράμματα προσπαθούσαν να μιλήσουν μεταξύ τους;
- Το μέγεθος που καταλαμβάνει στην μνήμη μια δομή μπορεί να είναι περισσότερο από τον χώρο των επιμέρους στοιχείων της.




```c
size_t a= sizeof(packed);   a: 16
size_t b= sizeof(unpacked);   b: 24
```

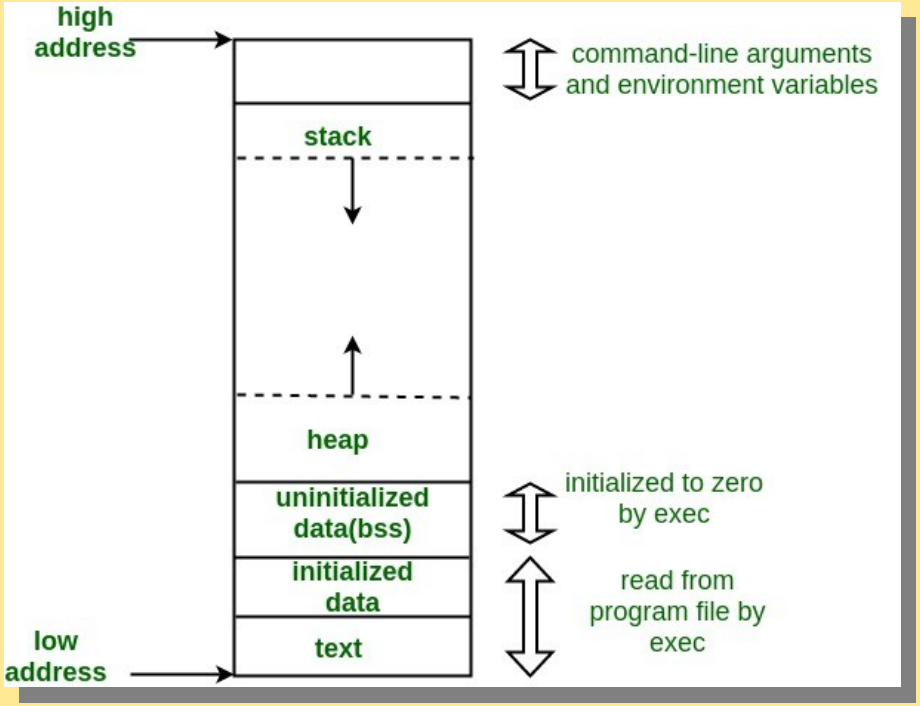
Ιδεατή μνήμη

- Σε ένα υπολογιστή τρέχουν πολλά προγράμματα ταυτόχρονα. Με χρήση ενός δείκτη μπορώ να διαβάσω, ή να γράψω σε οποιαδήποτε θέση μνήμης. Αλλά το ένα πρόγραμμα δεν μπορεί να επηρεάσει το άλλο. Πως γίνετε;
  - Όταν τρέχει το πρόγραμμα θεωρεί πως έχει πρόσβαση σε όλη την μνήμη του υπολογιστή, αλλά αυτό είναι μια ψευδαίσθηση. Το πρόγραμμα δεν γνωρίζει την πραγματική διεύθυνση μνήμης, αλλά μια ιδεατή. Το λειτουργικό σύστημα φροντίζει ώστε τα διάφορα προγράμματα να βλέπουν διαφορετικές περιοχές της φυσικής μνήμης.
- Σε ένα υπολογιστή το σύνολο της μνήμης που χρησιμοποιούν τα προγράμματα είναι μεγαλύτερο από το σύνολο της φυσικής μνήμης. Πως γίνετε;
  - Το λειτουργικό μεταφέρει τμήματα της μνήμης, που δεν χρησιμοποιούνται συχνά, από την φυσική μνήμη στον σκληρό δίσκο. Έτσι αυξάνετε η διαθέσιμη φυσική μνήμη που μπορεί να διατεθεί στα προγράμματα.

Linkers and Loaders

- Μετά τον μεταγλωττιστή τρέχει ο linker που αντιστοιχεί διευθύνσεις μνήμης στο **symbol table** και συνδέει το πρόγραμμα με τις **βιβλιοθήκες** που χρησιμοποιεί.
  - Υπάρχει τουλάχιστον μια βιβλιοθήκη η **libc** που περιέχει συναρτήσεις όπως η **printf**.
  - Η σύνδεση μπορεί να είναι **στατική σύνδεση(static linking)** όπου οι βιβλιοθήκες ενσωματώνονται στο κύριο πρόγραμμα ή **δυναμική (dynamic linking)** όπου το πρόγραμμα για να τρέξει θέλει εγκατεστημένες τις ίδιες εκδόσεις των βιβλιοθηκών. Στα Windows τις βιβλιοθήκες αυτές τις λέμε DLL.
- Όταν θέλουμε να τρέξει το πρόγραμμα θα αναλάβει ο **Loader** να διαβάσει το αρχείο από τον δίσκο και να το τακτοποιήσει σε περιοχές της μνήμης και να το συνδέσει με τις δυναμικές βιβλιοθήκες που χρησιμοποιεί
  - Στην μνήμη είναι πακεταρισμένο σε μια ειδική μορφή **(linker format)**
  - **PE** ή **DLL** στα Windows, **ELF** στο Linux και το MacOS.

Memory Layout of C Programs



- Ο Loader θα φορτώσει το πρόγραμμα σε διάφορες περιοχές μνήμης (**memory segments**).
  - **text**: Ο κώδικας του προγράμματος
  - **ınitialized data**: Γενικές μεταβλητές (global) που έχουν αρχική τιμή
  - **bss**: Γενικές μεταβλητές (global) που δεν έχουν αρχική τιμή. **Αυτή η περιοχή της μνήμης θα αρχικοποιηθεί με μηδενικά**.
  - **stack**: Η στοίβα
  - **heap**: Ο χώρος για δυναμική εκχώρηση μνήμης
  - Διάφορες άλλες περιοχές όπως το **symbol table** για να κάνουμε **debug** ή περιοχές για μεταβλητές που είναι μόνο για ανάγνωση (const).

https://www.geeksforgeeks.org/memory-layout-of-c-program/

Memory Layout : Παράδειγμα

- Στον κώδικα βλέπουμε τις δηλώσεις κάποιων μεταβλητών.
  - Βρείτε αυτές που έχουν αρχική τιμή
  - Βρείτε αυτές που έχουν αρχική τιμή ίση με το μηδέν.
  - Τι συμβαίνει με αυτές που δεν έχουν αρχική τιμή;
- Γιατί η **C** δεν βάζει σε όλες μια αρχική τιμή ίση με το μηδέν;
- Γιατί οι **static** μεταβλητές δεν είναι στην στοίβα;

```c
#include <stdlib.h>

int global;                  // BSS
double fpa=0.18;             // Initialized
const int answer = 42; // ??

int main() {
  static int stat;           // BSS
  static int istat = 42;  // Initialized
  int a;                     //  Stack

  // Heap
  int *ptr = (int*)malloc(answer * sizeof(int));
}
```

Ερωτήσεις κατανόησης

- Μπορούμε να προσθέσουμε/αφαιρέσουμε ένα αριθμό από ένα δείκτη; Ποια θα είναι η νέα τιμή;
- Έχει νόημα η **αφαίρεση** δεικτών; Τι ακριβώς σημαίνει το αποτέλεσμα;
- Έχει νόημα η **πρόσθεση** δεικτών;
- Ποια η διαφορά του **initialization** από το **assignment**;
- Πως θα σχεδιάζατε σήμερα τις συναρτήσεις για συμβολοσειρές;
- Γιατί είναι κακό να έχουμε **undefined behaviour** στα προγράμματα μας;
- Κάποιες μεταβλητές έχουν αρχική τιμή μηδέν. Ποιες είναι αυτές;

Τι συμβαίνει με τους Pointers ;

Δείκτες σε συναρτήσεις

Δείκτες σε συναρτήσεις

- Ένας **δείκτης σε μια συνάρτηση** (function pointer) είναι ένας δείκτης που δείχνει στην διεύθυνση **της μνήμης** που είναι **ο κώδικας της συνάρτησης**.
- Ένας τέτοιος δείκτης μπορεί να αποθηκευτεί, να γίνει όρισμα συνάρτησης, καθώς και να χρησιμοποιηθεί σαν να ήταν η ίδια η συνάρτηση.

Ένα παράδειγμα

```c
#include <stdio.h>

double squareFunc(double in) {
  return in * in;
}


int main() {
  double (*pFunc)(double);
  pFunc = squareFunc;

  double res = (*pFunc)(4.0);

  printf("res = %lf\n", res); // prints 16.00
  return 0;
}
```

Συνάρτηση
double -> double

Δείκτης σε συνάρτηση που
double-> double

Κλήση συνάρτησης
με χρήση δείκτη

Η δεσμευμένη λέξη **typedef**

- Δημιουργεί ψευδώνυμα για υπάρχοντες τύπους

```
typedef char *string_t;
typedef int money_t;

typedef struct {
  string_t title;
  string_t author;
  money_t cost;
} book_t;
```

```
string_t name = "Vasilakis";
money_t sum = 0;

book_t bestSciFiBooks[] = {
    {"The Dispossessed","Ursula K. Le Guin", 20},
    {"Dune", "Frank Herbert", 19},
    {"Stranger in a strange land", "R. Heinlein", 30},
    {"Foundation", "Isaac Asimov", 15},
    {"Solaris", "Stanislaw Lem", 12}
};
```

Image credit https://www.norwegiancreations.com/2017/10/getting-started-with-programming-part-8-typedef-and-structs/

Δείκτες σε συναρτήσεις (typedef)

```c
typedef double (*compute_fn)(double);

int main() {


  compute_fn pFunc = squareFunc;  // double (*pFunc)(double);

  double res = (*pFunc)(4.0);

  printf("res = %lf\n", res); // prints 16.00
  return 0;
}
```

Ορισμός τύπου δείκτη σε συνάρτηση που double -> double

Κλήση συνάρτησης με χρήση δείκτη

Πως να διαβάζω δηλώσεις

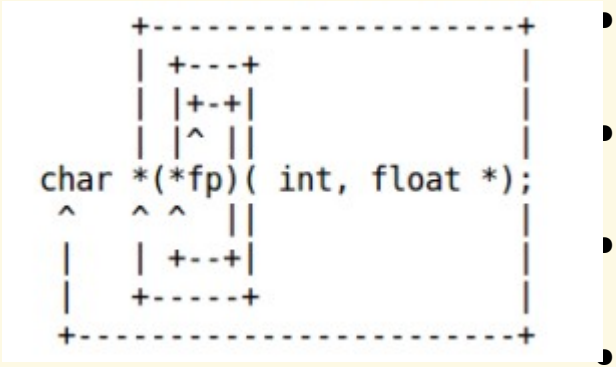
η πλήρης ιστορία

Η σπείρα των δηλώσεων

```
      +-------+
      | +-+   |
      | ^ |   |
char *str[10];
 ^    ^   |   |
 |    +---+   |
 +------------+
```

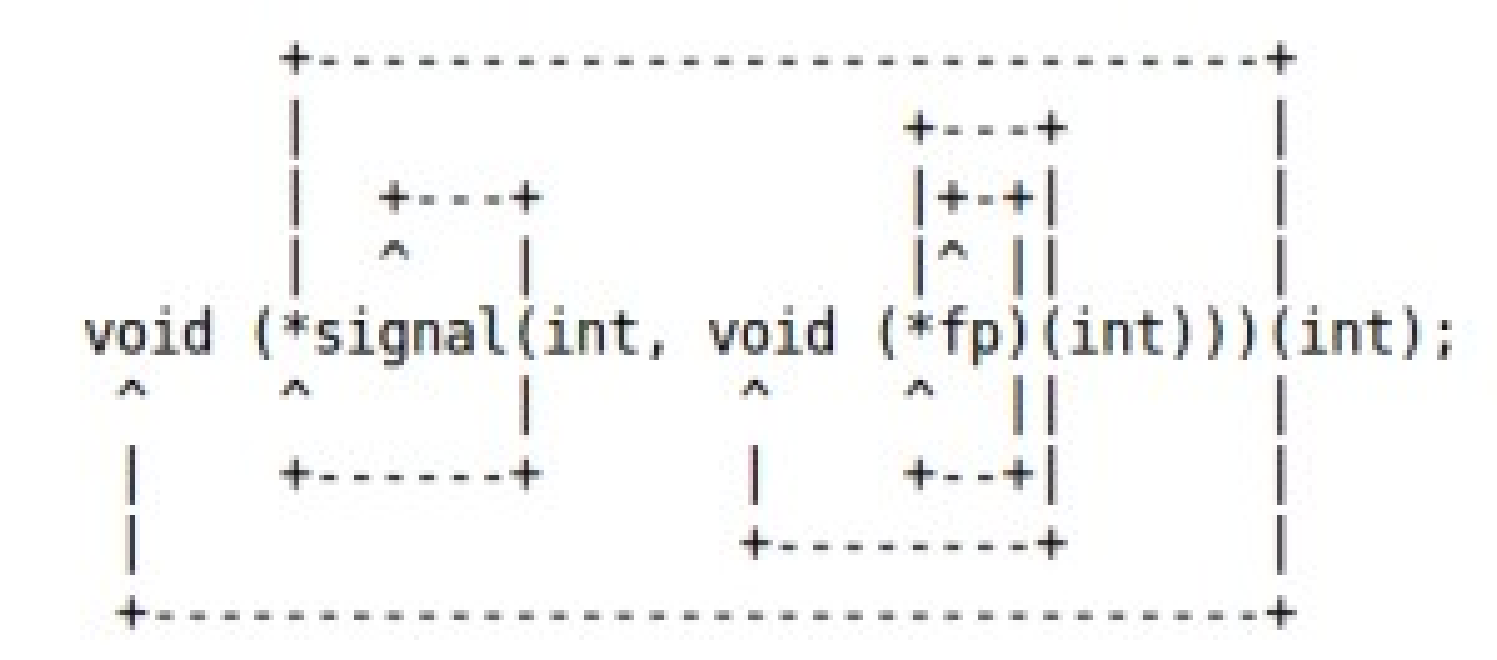
- str is
- str is an array 10 of
- … pointers to
- … pointers to char

Η σπείρα των δηλώσεων :2

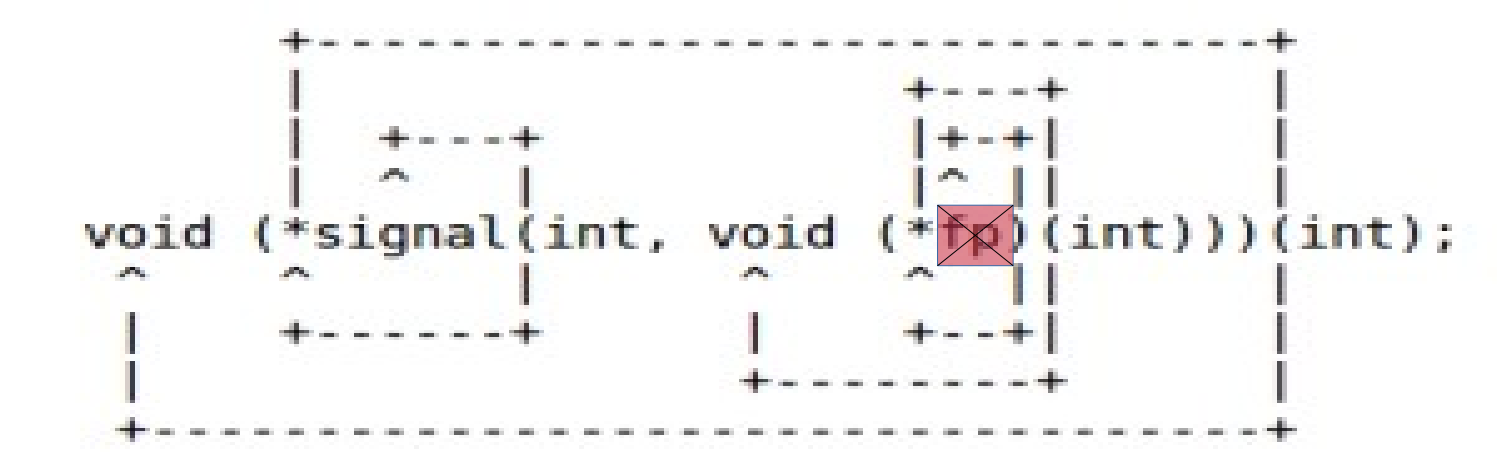


- fp is ...
- fp is a pointer to ...
- ... to a function ...
- ... passing an int and a pointer to float ..
- ... returning a pointer to char

Η σπείρα των δηλώσεων :3



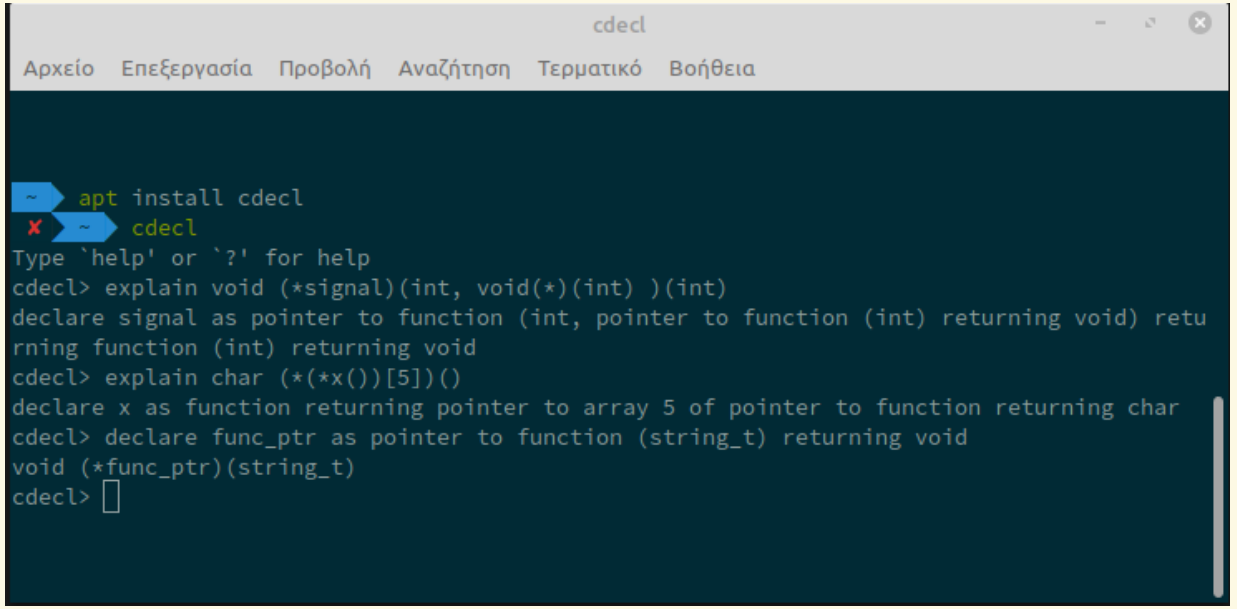
Η σπείρα των δηλώσεων :3



Είναι εντάξει αν δεν το καταλαβαίνεις

signal is a function, passing an int and a
  pointer to a function passing and int and return nothing
returning
  a pointer to a function passing and int returning nothing

Το πρόγραμμα cdecl



ή στο web στην διεύθυνση https://cdecl.org/

Ασκήσεις

```
int * const * p;
int * const * (* p)();
char ( * ( * x ( ) ) [ ] ) ( );
char ( * ( * x [3] ) ( ) ) [5];
```

- Με διαφορά το ποιο δύσκολο κομμάτι της σύνταξης της γλωσσάς C
  - αλλά δεν είναι τόσο δύσκολο όσο μοιάζει
- Διαβάστε
  https://cskill.wordpress.com/2010/06/09/the-clockwisespiral-rule-by-david-anderson/

The C++ keyword: using

```
// Old C
#define counter long

// Modern C Syntax
typedef long counter;
typedef void (*fPtrA)();
typedef void (*fPtrB)(int);

// New C++11 Syntax
using counter = long;
using fPtrA = void (*)();
using fPtrB = void(*)(int);

// Newer C++14 syntax
using FunctionPtr = auto (*)(int*) -> void;
```

Ποια διαβάζετε ευκολότερα;

(σημείωση: η δεσμευμένη λέξη ‘using’ έχει πολλές άλλες χρήσεις στην C++)

Παραδείγματα χρήσης

Δείκτες σε συναρτήσεις

Ομαλός τερματισμός με Ctrl+C

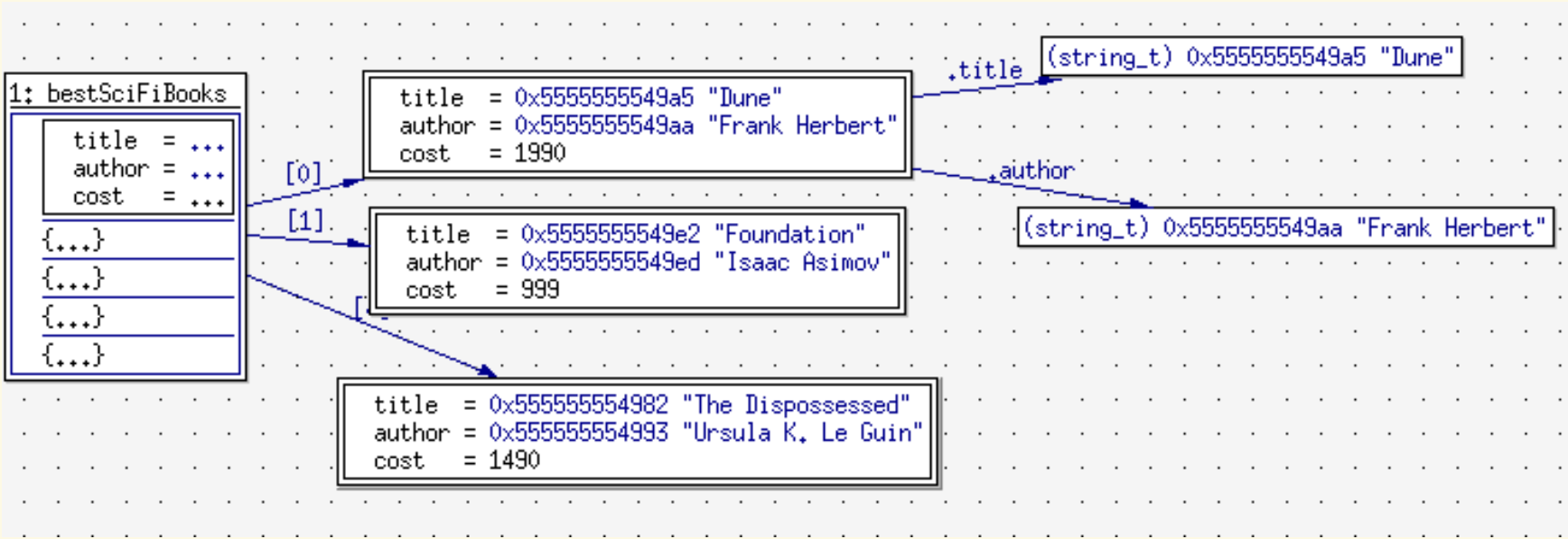
```c
#include <stdlib.h>
#include <stdio.h>
#include <signal.h>

void trapHandler(int dummy) {
  puts("Bye bye cruel word!");
  exit(1);
}

int main(void) {
  signal(SIGINT, trapHandler);

  puts("I take your computer forever!");
  // run forever
  for(;;) {
    // Do nothing
  }
  puts("You will never see this text");
}
```

- Τα **fuction pointer**s τα λέμε και **signal handlers** ή **callbacks**
- Όταν η ροή του προγράμματος δεν είναι χρονικά καθορισμένη (ασύγχρονος προγραμματισμός)
- Η ιδέα πίσω από τον προγραμματισμό με γραφικά περιβάλλοντα GUI.
- Παράδειγμα: Ασύγχρονος χειρισμός σημάτων στο UNIX

Παράδειγμα: Ταξινόμηση

- Να ταξινομηθεί ένας πίνακας που περιέχει μια δομή, με αύξουσα η φθίνουσα σειρά με βάση κάποιο στοιχείο του ή κάποιο συνδυασμό των στοιχείων της δομής
- Να ταξινομηθούν τα στοιχεία των βιβλίων
  - Κατά κόστος (αύξουσα σειρά ταξινόμησης)
  - Κατά όνομα συγγραφέα
  - Κατά έτος έκδοσής (τα ποιο καινούργια πρώτα)

Η δομή στην μνήμη



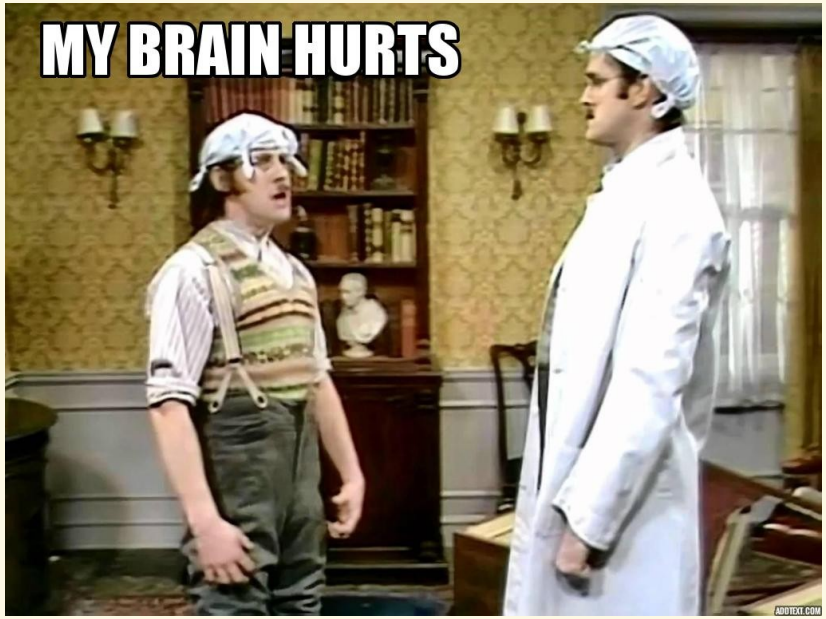
Ταξινόμηση πινάκων με δομές :1

```c
#include <stdlib.h>
#include <string.h>

// Compare 2 books by price
int compare_price(const void *a, const void *b){
  const book_t *pA = (const book_t *) a;
  const book_t *pB = (const book_t *) b;

  return (pA->cost)  < (pB->cost);
}

// Compare 2 books by author name
int compare_author(const void *a, const void *b){
  const book_t *pA = (const book_t *) a;
  const book_t *pB = (const book_t *) b;

  return strcmp(pA->author, pB->author);
}
```

Ταξινόμηση πινάκων με δομές :2

```c
int main() {
  size_t books_len = sizeof(bestSciFiBooks) / sizeof(book_t);

  qsort(bestSciFiBooks, books_len, sizeof(book_t), compare_author);

  // Print books
  for(int i=0; i != books_len; ++i) {
    book_t *book = &bestSciFiBooks[i];
    printf("%i: %s, cost: %.2f€ \n", i, book->title, book->cost/100.0 );
  }

}
```

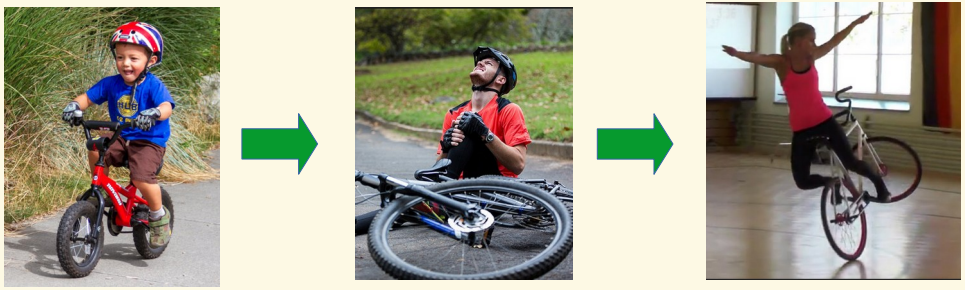
Μήπως θέλετε λίγο χαμομήλι;



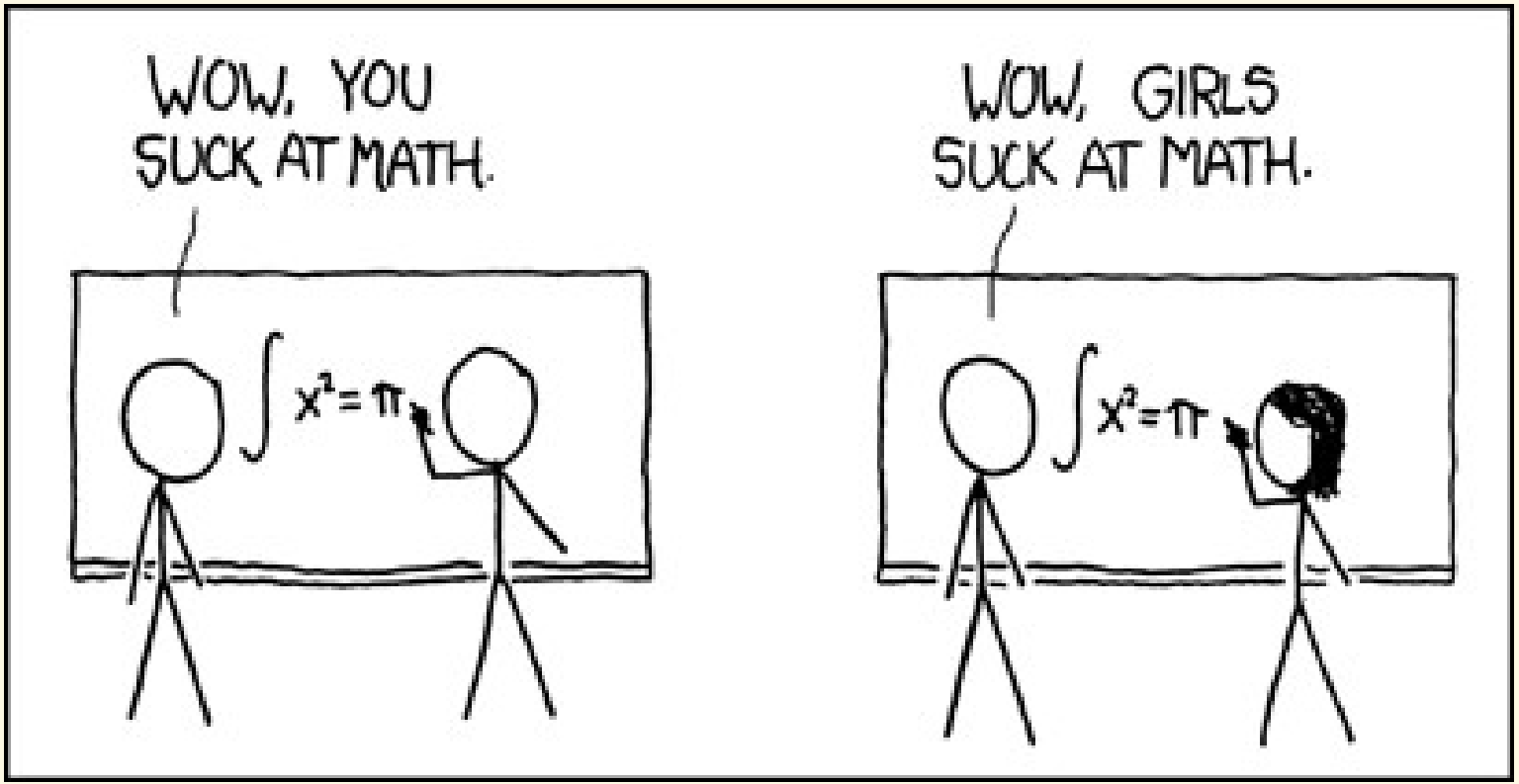
Οι δείκτες είναι ένα δύσκολο (και δυνατό) χαρακτηριστικό της γλώσσας C. Αλλά αν καταλάβεις τις βασικές έννοιες θα δεις πως είναι εύκολοι.

Μοιάζουν με το ποδήλατο ή το πατίνι. Την μια μέρα νομίζεις πως είναι αδύνατο να το μάθεις και την επόμενη κάνεις ακροβατικά.

Και το ποδήλατο δεν το ξεχνάς ποτέ.

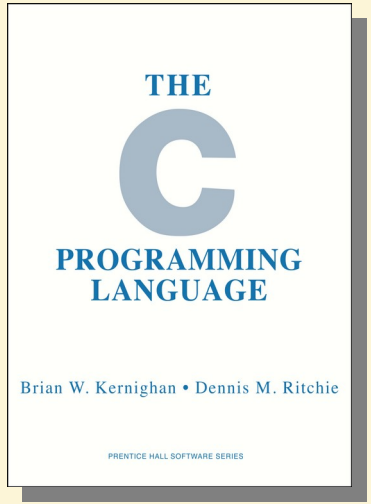


Ήταν δύσκολο;



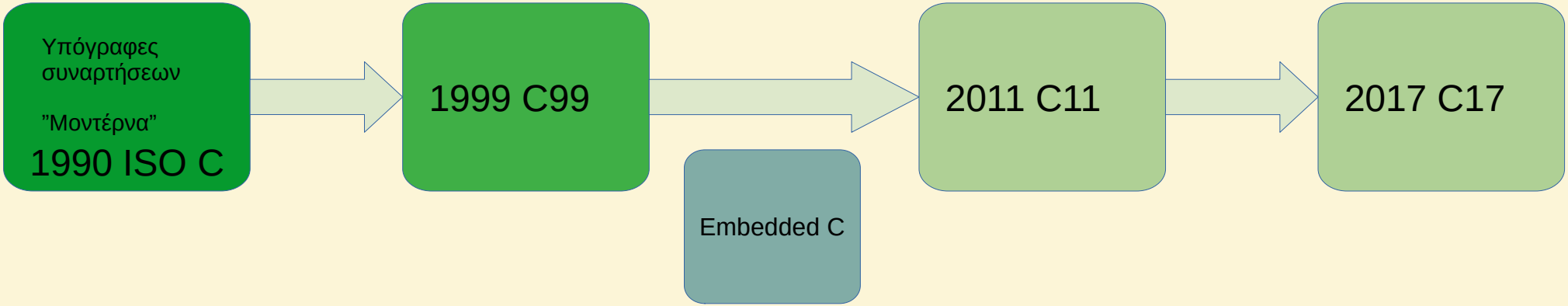
Οι **Monty Python** είναι το όνομα της ομάδας παραγωγής της τηλεοπτικής σειράς «Monty Python's Flying Circus», μιας κωμωδίας που άρχισε να εκπέμπεται στη Μεγάλη Βρετανία το 1969, αλλά και μιας σειράς εκπομπών, κινηματογραφικών έργων, ηχογραφήσεων και θεατρικών παραγωγών από το 1969 μέχρι το 1989. Έχουν δώσει το όνομα σε μια γνωστή άλλη γλώσσα προγραμματισμού.

Η γλώσσα C την νέα χιλιετία



- Η **C** ορίζετε από μια προδιαγραφή και μια επιτροπή, και όχι από μια υλοποίηση.
  - Αρχικά είχε οριστεί από το βιβλίο των K&R
- Είναι δύσκολο να αλλάξεις μια γλώσσα σαν την C, αλλά αλλαγές και προσθήκες συμβαίνουν
  - Συχνά με ιδέες δανεισμένες από την **C++**
- Η τελευταία έκδοση είναι η C18 με μικρές αλλαγές και βελτιώσεις από την C11. Περιμένουμε την **C2x** το 2021
- Πολλά βιβλία έχουν μείνει στην **C11** ή σε παλιότερες εκδόσεις

Σύντομο Ιστορικό



Άλλες σημαντικές χρονολογίες

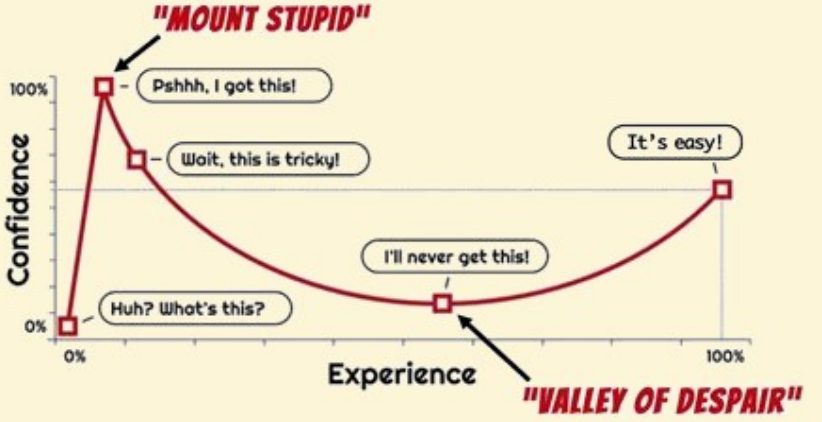
Γέννηση: 1972 K&R C: 1978 C++ 1985

Γιατί πάντα υπάρχουν περισσότερα

Google It

Φαινόμενο Ντάνινγκ-Κρούγκερ

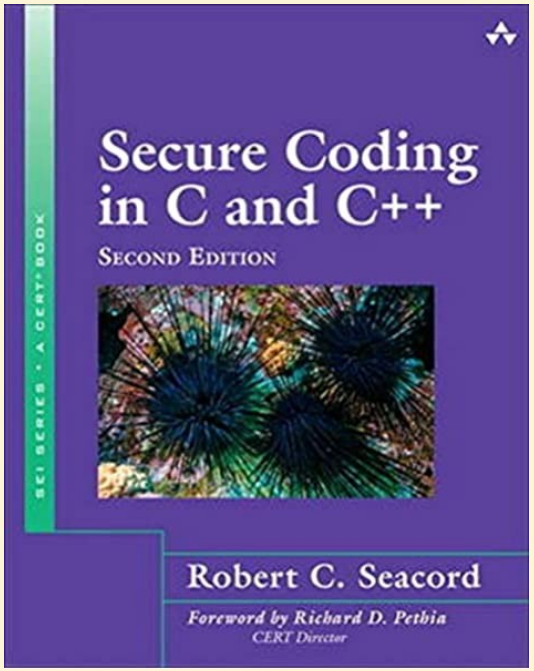
Το βουνό της ηλιθιότητας

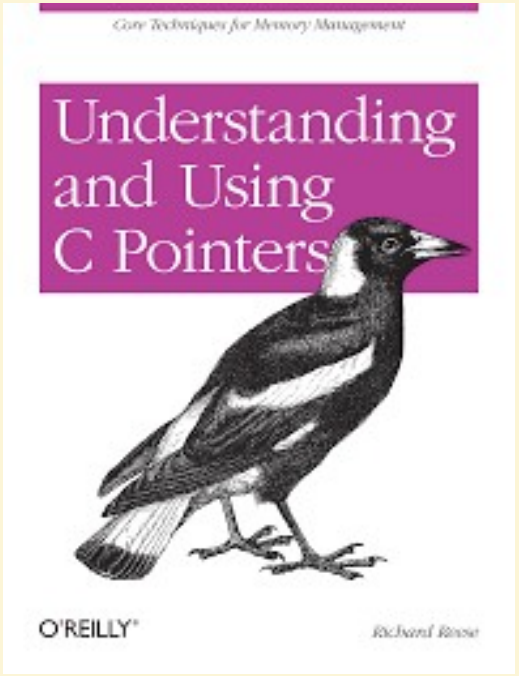


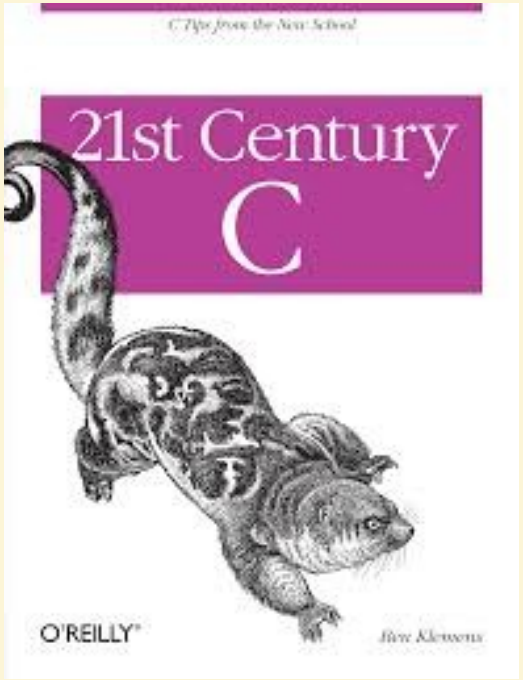
Η κοιλάδα της απόγνωσης

- Οι γλώσσες προγραμματισμού εξελίσσονται μαζί με τους υπολογιστές. Η **C** του 1970 δεν μπορεί να είναι ίδια με την **C** του 2020.
- Μην σταματάς ποτέ να διαβάζεις βιβλία, μην σταματάς πότε να κοιτάζεις για εξελίξεις στο διαδίκτυο.
  - Reddit, stackoverflow, podacasts, blogs, ...
- Για να μάθεις το οτιδήποτε θα πρέπει να ανέβεις το **βουνό της ηλιθιότητας** και να διαβείς την **κοιλάδα της απόγνωσης**.
- Δεν υπάρχει άλλος δρόμος, για να γίνεις ο ειδικός (expert).

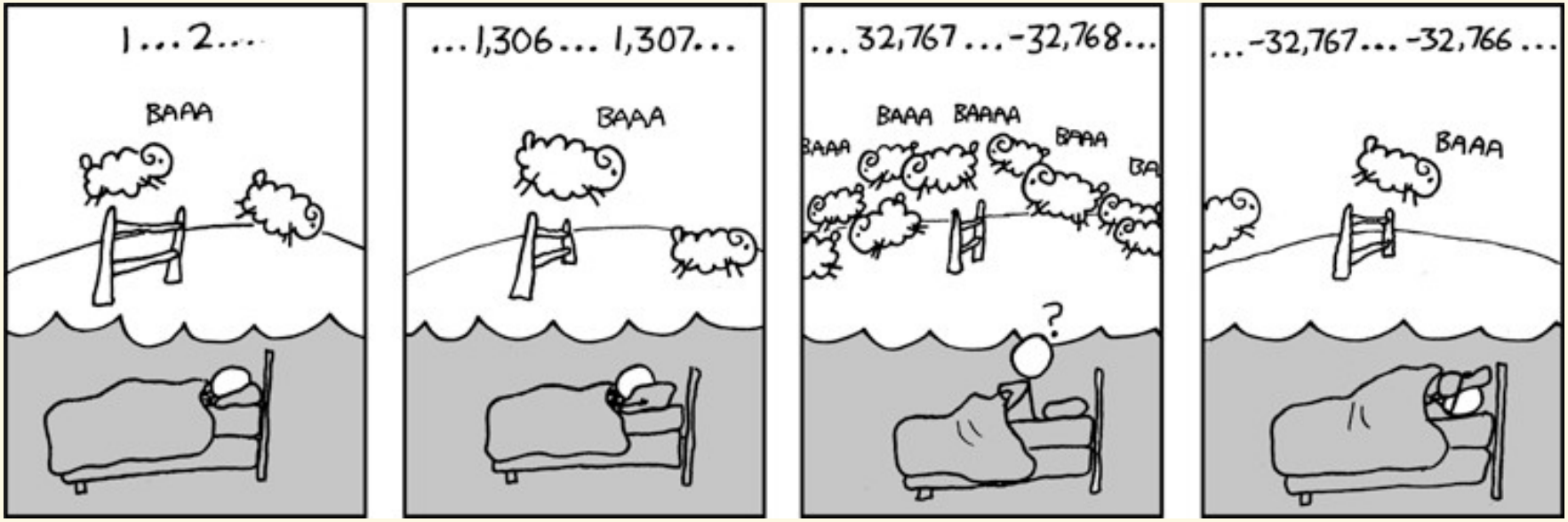
Βιβλιογραφία







Ερωτήσεις ;


1...2...
BAAA
...1,306... 1,307...
BAAA
...32,767...-32,768...
BAAA
BAAAA
BAAA
BA
?
...-32,767... -32,766...
BAAA