Lezione 30 C e assembly 8

giovedì 29 febbraio 2024 10:33

Vediamo ora come si manifesta la relazione tra vettori e puntatori in C:

```c
#include <stdio.h>

int main()
{
        int nums[]={10,12,13,14,20};
        //notare che il '\0' e carattere terminatore
        char name[]={'A','l','e','s','s','a','n','d','r','o','\0'};
        char *name_ptr=name;

        printf("the size of an int: %ld\n",sizeof(int));
        printf("The size of nums (int[]): %ld\n", sizeof(nums));
        printf("The number of ints in nums: %ld\n\n", sizeof(nums) / sizeof(int));
        printf("The size of a char: %ld\n", sizeof(char));
        printf("The size of name (char[]): %ld\n", sizeof(name));
        printf("The number of chars: %ld\n\n", sizeof(name) / sizeof(char));
        printf("The size of name_ptr: %ld\n", sizeof(name_ptr));
        //assegnazione di vettore a puntatore -> autocast
        //vettore quindi decade a puntatore
        //indirizzo primo elemento viene castato ad indirizzo
        //perdo info sul tipo -> name vettore di certo numero di caratteri
        printf("Bugged number of chars: %ld\n", sizeof(name_ptr) / sizeof(char));

        return 0;
}
```

Con output il seguente :

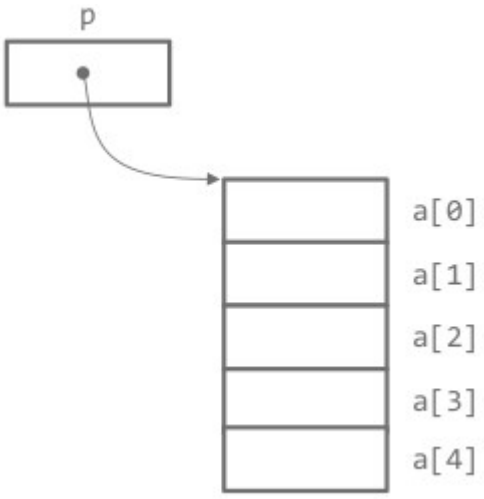
```
the size of an int: 4
The size of nums (int[]): 20
The number of ints in nums: 5

The size of a char: 1
The size of name (char[]): 11
The number of chars: 11

The size of name_ptr: 8
Bugged number of chars: 8
```

Ricordiamo qui che il "4" sta ad indicare 32 bit!!. Invece nell'ultima stampa (bugged) stampa comunque 8 (indirizzo a 64 bit) , ma viene convertito grazie ad autocast -> **si perdono informazioni sul tipo di dato**

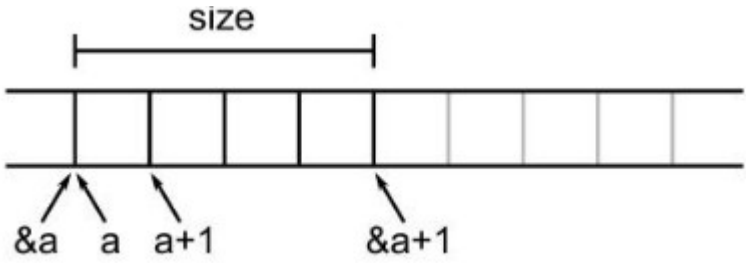
Quindi con questa assegnazione (autocast) da vettore a puntatore , come già detto si perdono delle info e queste info sono la taglia del vettore. Quindi stesso comportamento , ma cambia il codice assembler . Esempio :

```c
int a[4] = {3, 2, 1, 0};
int *p = a;
```

Da notare che qui entrambi generano l'indirizzo dove elemento 2 è memorizzato . Quindi in generale e riassumendo : un puntatore può essere deferenziato utilizzando operatore spiazzamento ([ ]). Quindi in generale : *Prendi un puntatore, deferenzialo , spiazzati da quel punto come se avessi un vettore* :



Analogamente ai puntatori , andiamo a vedere il duale : **L'indirizzo** di una variabile : oggetto duale del puntatore . **Questo oggetto rappresenta effettivamente la cella di memoria dove si trova quella variabile**.



Vediamo un esempio :

file1.c:

```c
#include <stdio.h>

extern int *array;

int main(void)
{
   printf("Third element of the
     array is %d\n", array[2]);
   return 0;
}
```

file2.c:

```c
int array[] = {0, 1, 2, 3, 4};
```

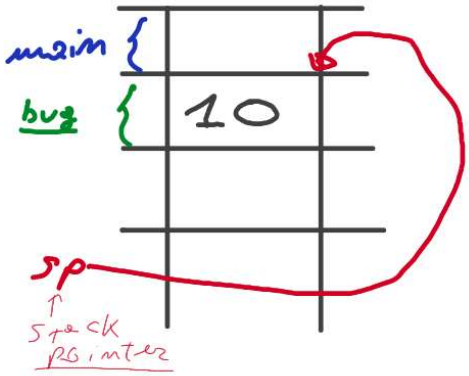
- Questi due file compilano in un programma correttamente
- Non generano nemmeno un warning
- Il programma è corretto?

Il programma assolutamente sbagliato !! Core dump creato .

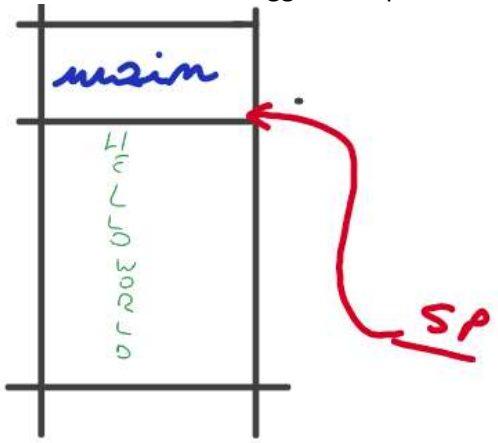
Vediamo altro esempio :

```c
#include <stdio.h>

int *buggy_return(void)
{
        int a_variable=10;
        //ritorna indirizzo contenente variabile nello stack
        return &a_variable;
}

void using_stack(void)
{
        char a_string[]="Hello World!";
        printf("%s\n",a_string);
}

int main(void)
{
        int *dummy=buggy_return();
        //deferenzio il puntatore , accedendo al valore
        printf("%d\n",*dummy);
        using_stack();
        printf("%d\n",*dummy);
        return 0;
}
```

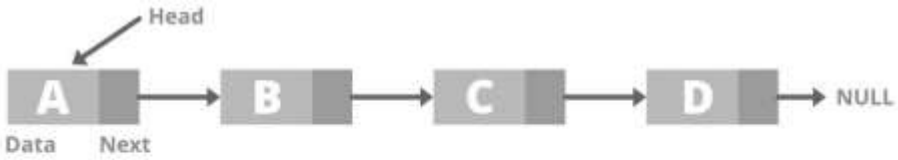
Il programma va in crash : core dump. Invalida la finestra di stack. **Quindi nel caso del codice sopra, si hanno le finestre di stack sovrapposte**. Vediamolo in dettaglio cosa succede alla finestra di stack : Nel primo disegno si ha la situazione nella funzione BUGGY_RETURN:



Mentre nel prossimo sia ha la situazione dello stack della funzione STACK_PRINTER : da notare che l'area dello stack allocata è maggiore in quanto è diverso il parametro :

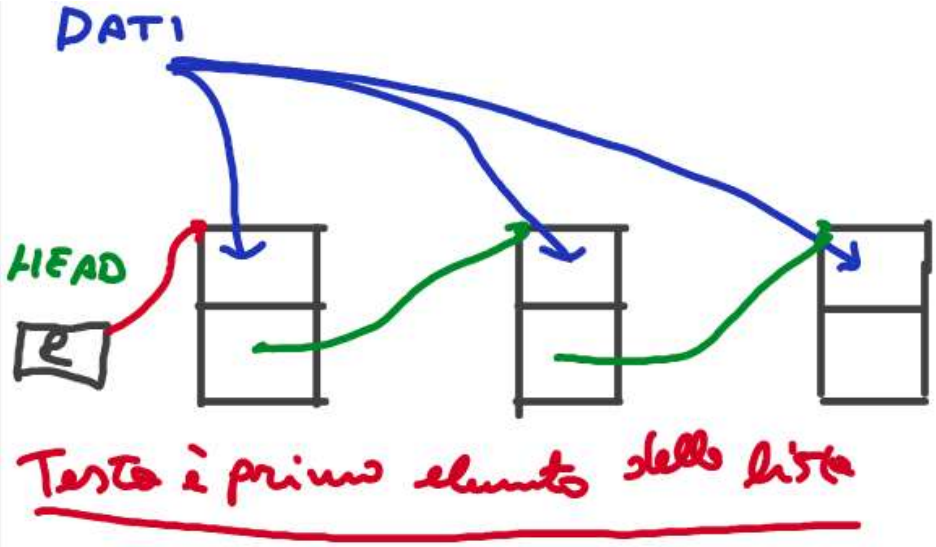


Vediamo ora un altro concetto : **le liste** : codice compatto ed efficiente . Si parla di quelle singolarmente collegata : insieme ordinati di nodi :




```
struct node_t {
  int data;
  node_t * next;
};
```

Concettualmente è corretto , ma praticamente no : sia per il fatto che si punta a null e secondo le frecce (puntatori a strutture) : dovrebbero contenere indirizzo del primo byte del nodo successivo . Quindi dal punto di vista pratico :



Da notare che all'interno di head vi è un indirizzo : è un puntatore ad indirizzo della struct. Vediamo ora un esempio di codice : rimozione di un nodo:

```
struct node_t {
  int data;
  node_t * next;
};

void remove_entry(struct node_t **const head, struct node_t const *const entry)
{
  struct node_t *prev = NULL;
  struct node_t *walk = *head;
  while(walk != entry) {
    prev = walk;
    walk = walk->next;
  }

  if(!prev) {
    *head = entry->next;
  } else {
    prev->next = entry->next;
  }
}
```

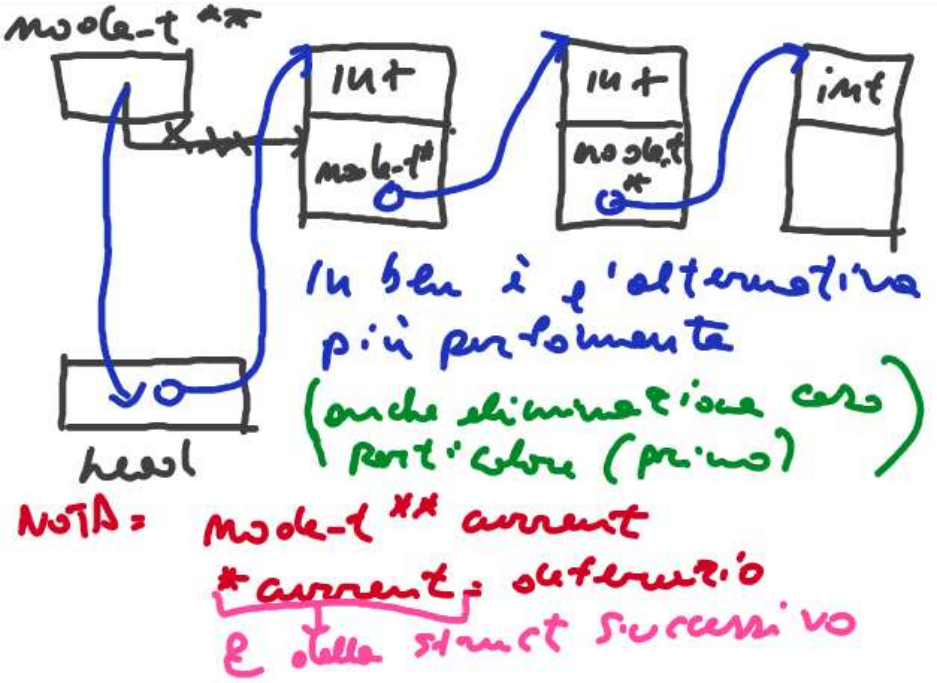
NOTA: null=0=false.

Si evince da qui che *walk* indica il puntatore che indica il cammino ; -> indica che leggo il contenuto. Inoltre non basta un solo puntatore , in quanto se devo eliminare il primo elemento , come si fa? Vediamo una versione migliorata :

```
void remove_entry(struct node_t **head, struct node_t const *const entry)
{
  while((*head) != entry) {
    head = &(*head)->next;
  }
  *head = entry->next;
}
```

Innanzitutto ingloba il caso particolare , poi evita di avere due puntatori per due elementi

adiacenti . **Quindi in questo caso si parla di puntatore a puntatore di struct.**
Vediamola ora in modo concreto:



Andiamo a vedere ora la precedenza degli operatori (ordine degli operatori in esecuzione): vengono scelti dal compilatore :

Precedenza (da maggiore a minore)

Operatori	Associatività
() [] -> .	da destra a sinistra
! - ++ -- + - * & (type) sizeof	da destra a sinistra
* / %	da sinistra a destra
+ -	da sinistra a destra
<< >>	da sinistra a destra
< <= > >=	da sinistra a destra
== !=	da sinistra a destra
&	da sinistra a destra
^	da sinistra a destra
|	da sinistra a destra
&&	da sinistra a destra
||	da sinistra a destra
?:	da destra a sinistra
= += -= *= /= %= &= ^= |= <<= >>=	da destra a sinistra
,	da sinistra a destra

Come per esempio :

Espressione	Equivale a	Significato
*p++	*(p++)	p viene incrementato dopo il suo utilizzo per accedere alla memoria.
*++p	*(++p)	p viene incrementato prima del suo utilizzo per accedere alla memoria.
++*p	++(*p)	p è utilizzato per l'accesso in memoria, ma il contenuto della locazione viene incrementato prima del suo utilizzo.
(*p)++	(*p)++	p è utilizzato per l'accesso in memoria; il contenuto della locazione viene incrementato dopo il suo utilizzo.

Vediamo ora come creare alias (nomi alternativi a tipi di dato esistenti) alle struct: **il typedef**

```
struct var {
        int data1;
        int data2;
        char data3;
};

struct var a;

typedef struct var newtype;

newtype a;
```