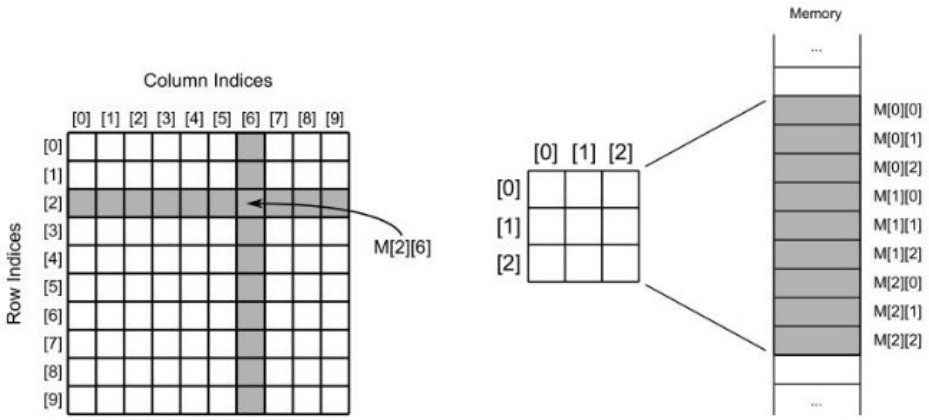
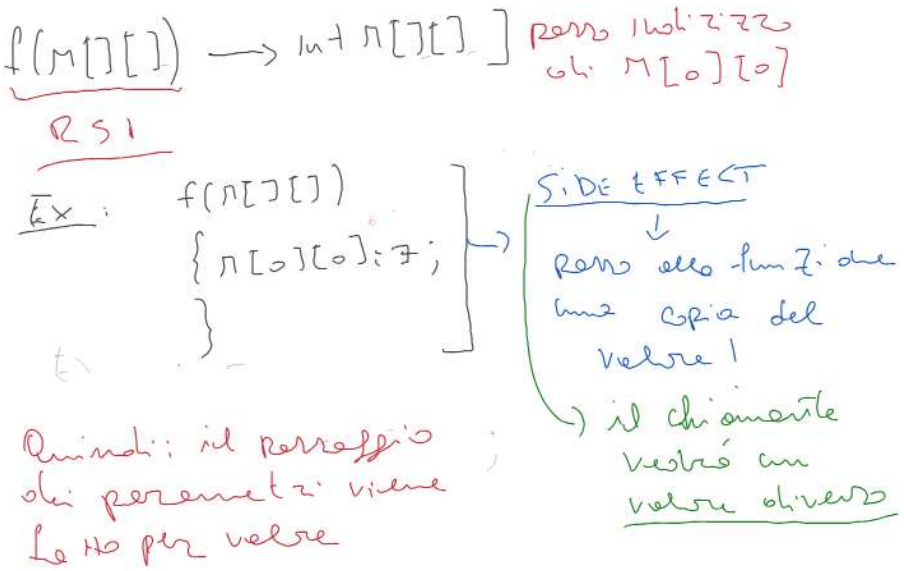
Lezione 28 C e assembly 6

venerdì 1 dicembre 2023 12:31

Andiamo a vedere un'altra rappresentazione dei dati : **matrici** , ovvero strutture dati a più dimensioni : ma dato il modello di memoria flat , lo dobbiamo linearizzare . Quindi si deve memorizzare linea e/o colonna in modo contiguo. Quindi per accedere allo spiazzamento ([][]) si usa la seguente formula : **address ⟺ i ∗ N ∗ size + j ∗ size ,** dove n è il numero di elementi nella riga. Vediamo concettualmente come sono organizzate :



Vediamo ora come passare una matrice come parametro :



Mentre dal punto di vista del codice :

```c
#include<stdio.h>
/*
notiamo che la prima matrice viene dichiarata normalmente
nella seconda invece il numero delle righe le calcola il compilatore
nell'ultimo caso dichiariamo una matrice vuota
*/
int A[2][3]={{1,3,0},{-1,5,9}};
int B[][3]={{2,7,-4},{3,-2,7}};
int C[2][3]={0,0,0,0,0,0};

void print_matrix(int m[2][3])
{
        int i,j;
        for (i=0;i<2;i++)
        {
                for (j=0;j<3;j++)
                {
                        printf("%d\t",m[i][j]);
                }
                puts("");
        }
}
int main(int argc, char *argv[])
{
        int i, j;
        puts("Matrix A:");
        print_matrix(A);
        puts("\nMatrix B:");
        print_matrix(B);
        for(i = 0; i < 2; i++) {
                for(j = 0; j < 3; j++) {
                        C[i][j] = A[i][j] + B[i][j];
                        }
                }
        // stampa matrice C come la somma delle due
        puts("\nMatrix C:");
        print_matrix(C);
}
```

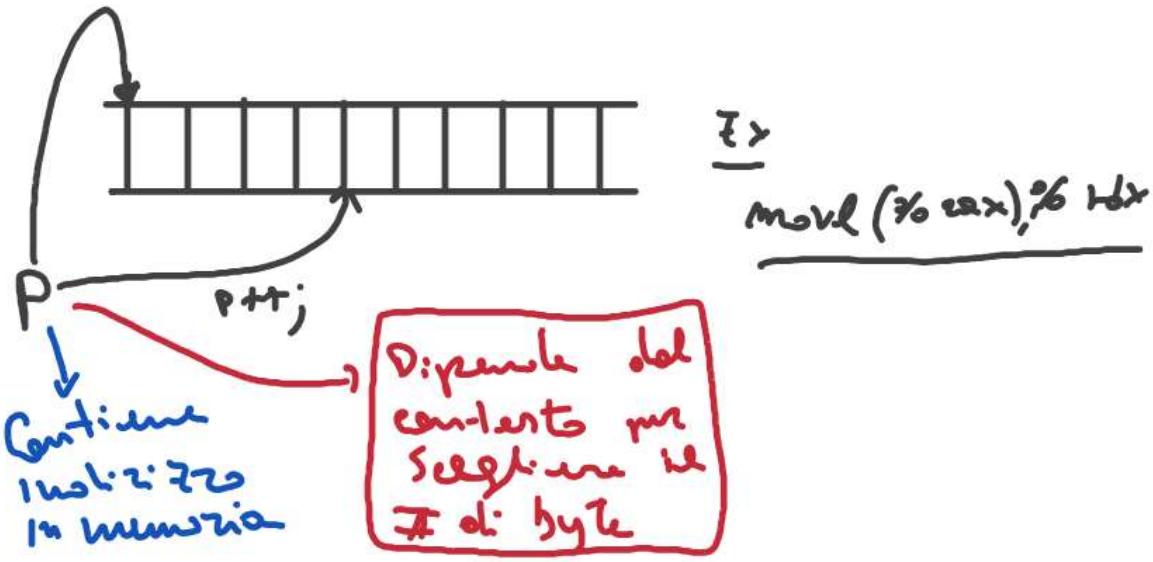
```
Matrix A:
1       3       0
-1      5       9

Matrix B:
2       7       -4
3       -2      7

Matrix C:
3       10      -4
2       3       16
```

```c
#include <stdio.h>
/*
esempio di marice a dimensione variabile
*/
int A[2][3] = {{1, 3, 0}, {-1, 5, 9}};
int B[][3] = {{2, 7, -4}, {3, -2, 7}};

void print_matrix1(int row, int col, int M[row][col])
{
        int i, j;
        for(i = 0; i < row; i++) {
        for(j = 0; j < col; j++) {
        printf("%d\t", M[i][j]);
                }
        }
        puts("");
}
void print_matrix2(int col, int M[][col])
{
        int i, j;
        for(i = 0; i < 2; i++) {
        for(j = 0; j < col; j++) {
        printf("%d\t", M[i][j]);
                }
        }
        puts("");

}
int main(int argc, char *argv[])
{
        /*
        qui è il compilatore/assemblatore che calcola
        l'indice dello spiazzamento
        */
        puts("Matrix A:");
        print_matrix1(2, 3, A);
        puts("\nMatrix B:");
        print_matrix2(3, B);
        return 0;
}
```

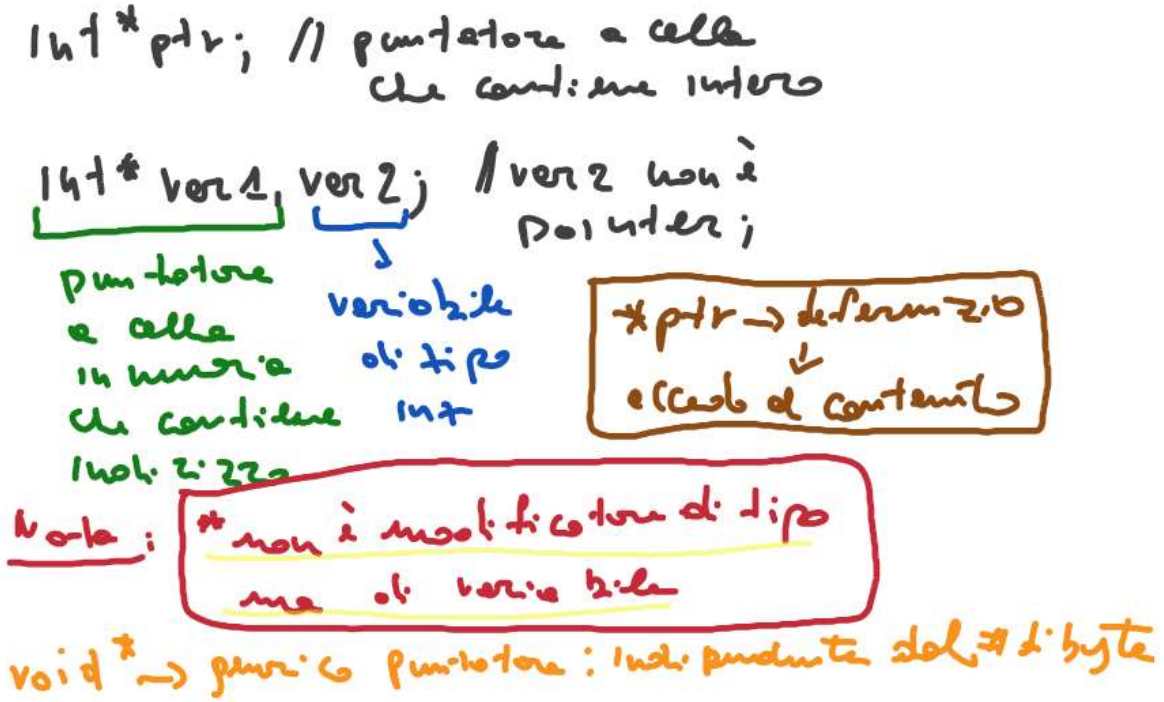
```
Matrix A:
1       3       0       -1      5       9

Matrix B:
2       7       -4      3       -2      7
```

Ricordiamo quindi che l'importante è che il numero di colonne sia definito, in quanto quello delle righe lo calcola il compilatore.

Andiamo ora a vedere la bestia nera del C : i **puntatori** . Queste variabili non contengono un valore , ma un indirizzo di memoria. In dettaglio :



In dettaglio (vediamolo in C):



Vediamo ora come è possibile "giocarci": ovvero **l'aritmetica dei puntatori** :

`type` `*ptr`: Un puntatore di tipo `type` chiamato `ptr`

`*ptr`: Il valore della variabile puntata da `ptr`, qualunque sia il tipo

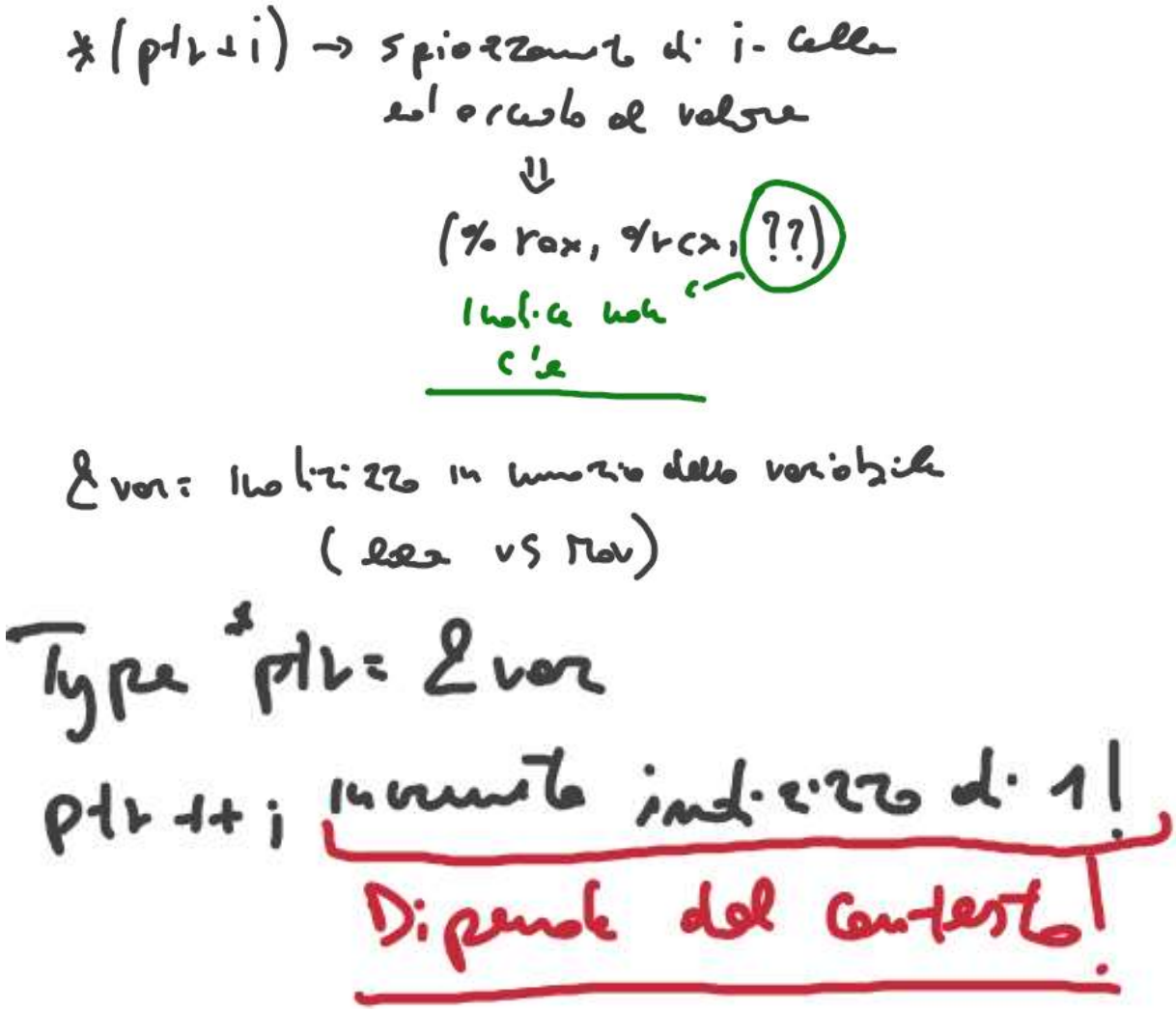
`*(ptr + i)`: Il valore di (qualsiasi cosa si trovi all'indirizzo `ptr + i`)

`&var`: L'indirizzo della variabile `var`

`type *ptr = &var`: Un puntatore di tipo `type` chiamato `ptr` inizializzato all'indirizzo di `var`

`ptr++`: Incrementa l'indirizzo cui punta `ptr`

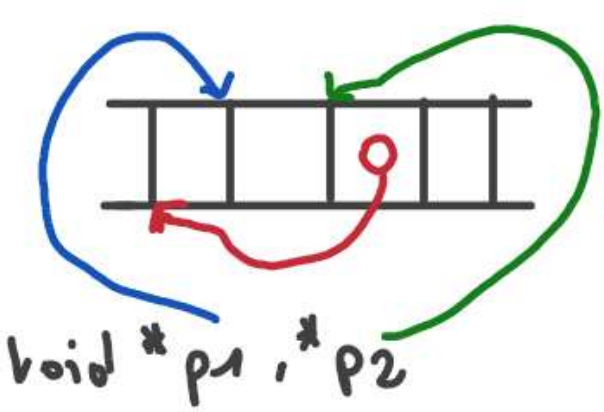
Nota : si usa solo la base , in quanto indice non ha senso. Esempio :



Come vengono usati i puntatori in assembler? Si usa un registro per contenere indirizzo di memoria , quel registro viene usato come registro base ed il contenuto può essere manipolato a piacimento :

```
int *intptr;                movq $intptr, %rax
*intptr = 42;               movl $42, (%rax)

int *ptr = &var;            movq $var, %rax
                            movq %rax, ptr

ptr++;                      addq $4, %rax
```

Nota : aggiungo costante 4 al registro perché stiamo usando interi.
Facciamo un discorso più generale :



In generale:
arbitrariamente grande di puntatori che puntano a stessa cella di memoria

Vediamo l'aritmetica:

Come detto in precedenza:
- `ptr++`: Incrementa l'indirizzo cui punta `ptr`

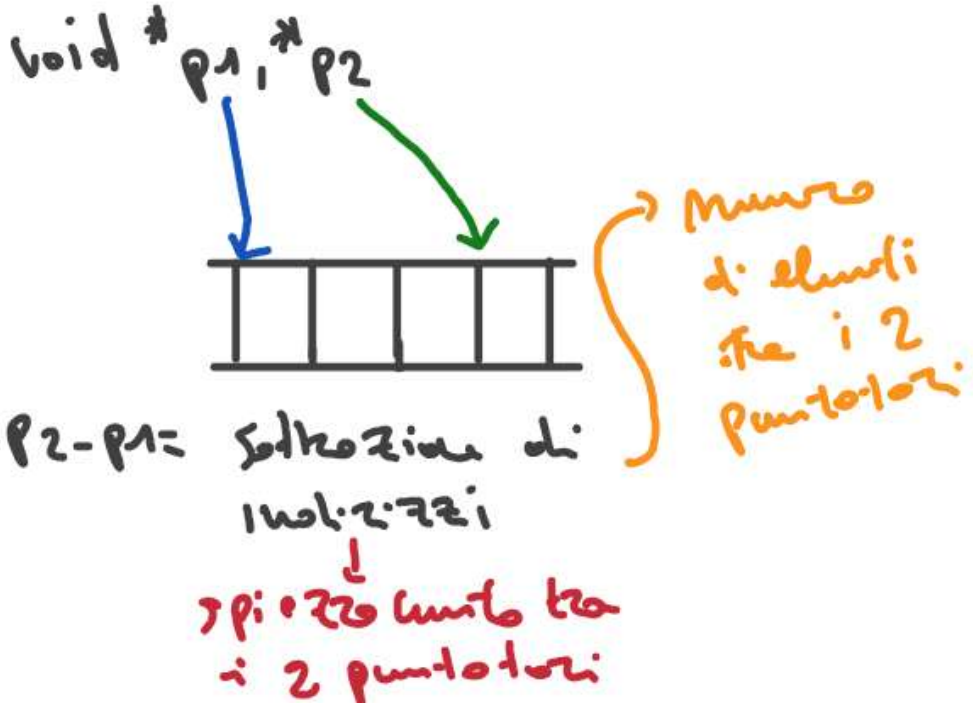
I puntatori tuttavia hanno un tipo, pertanto cosa vuol dire "incrementare l'indirizzo?"

L'aritmetica dei puntatori è tale per cui l'*indirizzo* contenuto nella variabile puntatore viene incrementato di un numero di byte pari alla *taglia* del tipo primitivo puntato

dato un `int *p = (void *)1000`, assumendo `sizeof(int) == 4`:
- `p+1 == 1004`
- `p+2 == 1008`
- `p+n == 1000+n*4`

Operazioni valide sono:
- incremento, decremento, somma, sottrazione
- sottrazione tra puntatori: determina il *numero di elementi* tra gli indirizzi

Inoltre è possibile fare la <u>sottrazione tra puntatori : ovvero la distanza in byte tra i due puntatori : dimensione in byte</u> :



Vediamo un esempio in C : stampo due vettori in quattro modi di differenza :

```c
#include<stdio.h>
int main(int argc, char *argv[])
{
        // The data we will be printing
        int ages[] = { 23, 43, 12, 89, 2 };
        char *names[] = {
        "Alan", "Frank",
        "Mary", "John", "Lisa"
        };
        int count = 5;
        int i = 0;
        // first way: indexing
        for (i = 0; i < count; i++) {
        printf("%s has %d years alive.\n", names[i], ages[i]);
        }
        puts("---");
        // second way: using pointers
        int *cur_age = ages;
        /*
        qui uso il doppio puntatore
        in quanto le stringhe sono puntatori a caratteri ,
        ma in realtà si intendono come puntatori ad indirizzo
        che punta ad indirizzo
        */
        char **cur_name = names;
        for (i = 0; i < count; i++) {
        printf("%s is %d years old.\n", *(cur_name + i), *(cur_age + i));
        }
        puts("---");
        // third way: "offsetting" pointers
        for (i = 0; i < count; i++) {
        printf("%s is %d years old again.\n", cur_name[i], cur_age[i]);
        }
        puts("---");
        // fourth way: let's make it unnecessarily complex
        for (cur_name = names, cur_age = ages;
        (cur_age - ages) < count; cur_name++, cur_age++) {
        printf("%s lived %d years so far.\n", *cur_name, *cur_age);
        }
        return 0;
}
```

```
Alan has 23 years alive.
Frank has 43 years alive.
Mary has 12 years alive.
John has 89 years alive.
Lisa has 2 years alive.
---
Alan is 23 years old.
Frank is 43 years old.
Mary is 12 years old.
John is 89 years old.
Lisa is 2 years old.
---
Alan is 23 years old again.
Frank is 43 years old again.
Mary is 12 years old again.
John is 89 years old again.
Lisa is 2 years old again.
---
Alan lived 23 years so far.
Frank lived 43 years so far.
Mary lived 12 years so far.
John lived 89 years so far.
Lisa lived 2 years so far.
```

Con questo esempio vediamo 2 cose : la prima è la commutatività della somma con i puntatori ed inoltre vediamo come sono dichiarate/interpretate le stringhe in C: