Lezione 8: I Puntatori

Laboratorio di Elementi di Architettura e Sistemi Operativi

2 Maggio 2012

Esercizi della lezione scorsa

Esercizio 3: la calcolatrice

1. Realizzare un insieme di funzioni per gestire una pila (stack) di valori floating point. In particolare, si implementino le operazioni di inserimento (`pop`) ed estrazione (`push`) di un valore nella pila. Si assuma che la pila possa contenere al massimo 10 valori.
2. Utilizzare le funzioni di gestione della pila per implementare una calcolatrice che esegua le quattro operazioni aritmetiche (+, −, ⋆, /) usando la notazione polacca inversa (RPN). La calcolatrice riceve le operazioni da eseguire dalla tastiera, leggendo un operatore o operando per ogni riga. Dopo aver letto un numero o un operatore (ed aver eseguito l'operazione) mostra i valori contenuti nello stack. Il programma termina quando l'utente inserisce `q`.

Esercizio 3: la calcolatrice

```
float stack[10];
int cima = -1;

void push(float f) {
  cima++;
  stack[cima] = f;
}

float pop() {
  if(cima >= 0) {
    cima--;
    return stack[cima+1];
  } else {
    printf("Errore! Lo stack è vuoto!\n");
    return 0;
  }
}

main()
{
    char str[20], *res;
    int op;

    printf("Calcolatrice Polacca inversa\n");
    op = NUM;
    res = gets(str);
    while(res != NULL) {
        op = getop(str);
        switch(op) {
          case NUM:
            push(getnum(str)); break;
          case OP:
            esegui(str[0]); break;
          case Q:
            printf("Arrivederci!\n"); return;
          default:
```

```
            printf("Istruzione non valida!\n");
        }
        printstack();
        res = gets(str);
    }
}

int getop(char *str) {
    int n = strlen(str);
    float f;

    if(n == 1 && (str[0] == '+' || str[0] == '-'
                || str[0] == '*' || str[0] == '/'))
        return OP;
    if(n == 1 && str[0] == 'q')
        return Q;
    n = sscanf(str, "%f", &f);
    if(n == 1)
        return NUM;
    return ERR;
}

float getnum(char *str) {
    float f;
    sscanf(str, "%f", &f);
    return f;
}

void esegui(char c) {
    float a,b;

    if(cima < 1) {
        printf("ERRORE: numero insufficiente di operandi nello stack!\n");
        return;
    }
    b = pop();
    a = pop();
    switch(c) {
      case '+':
        push(a+b); break;
      case '-':
        push(a-b); break;
      case '*':
        push(a*b); break;
      case '/':
        push(a/b); break;
      default:
        printf("Operazione sconosciuta!\n");
        push(a); push(b); break;
    }
}
```

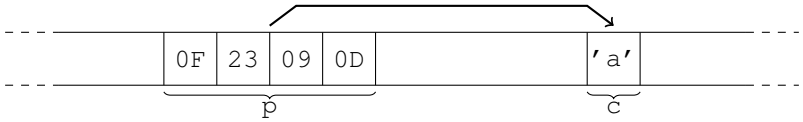
I Puntatori

Puntatori ed indirizzi

- Semplificando, la memoria di un computer può essere vista come un *vettore di celle* numerate in modo consecutivo
  - tipicamente, le celle hanno la dimensione di un byte
- ogni tipo di dato occupa un certo numero di celle di memoria:
  - `char`: un byte
  - `short int`: due byte
  - `long int`: quattro byte
  - ...

- un *puntatore* è un gruppo di celle (di solito 4 o 8) che può contenere un *indirizzo di memoria*

Esempio:
`c` è un `char` e `p` un puntatore che *punta* a `c`, cioè *contiene l'indirizzo di memoria di* `c`



- Dichiarazione di un puntatore: `tipo *nome`
  - ogni puntatore è vincolato a puntare un certo tipo di dato
  - *eccezione:* un puntatore a `void` punta ad un dato generico
  - un puntatore a `void` *non può essere dereferenziato*
- L'operatore unario `&` fornisce l'indirizzo di un oggetto in memoria:
  - `p = &c;`
- L'operatore unario `*` *dereferenzia* un puntatore, ossia permette di *accedere al contenuto* dell'oggetto puntato:
  - `*p = 'a';`

Esempio di uso degli operatori `&` e `*`:

```
int x = 1, y = 2, z[10];
int *ip, *iq;  /* ip e iq sono puntatori ad int */

ip = &x;       /* ora ip punta a x */
y = *ip;       /* ora y vale 1 */
*ip = 0;       /* ora x vale 0 */
ip = &z[0];    /* ora ip punta a z[0] */
iq = ip;       /* ora iq punta a z[0] */
```

Puntatori e parametri di funzione

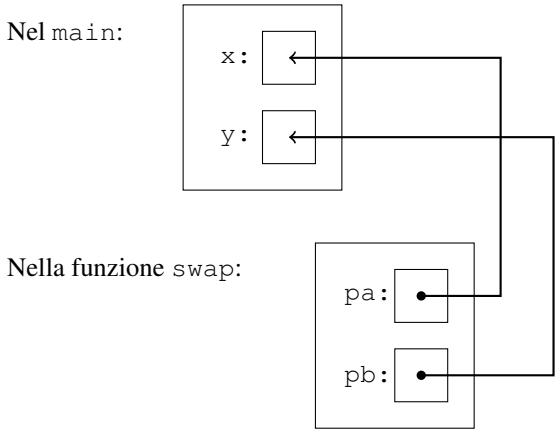
- In C, il passaggio dei parametri avviene per *valore*:
  - come abbiamo già visto, questo impedisce alle funzioni di modificarne il valore
- Usando i *puntatori* posso passare i parametri *per indirizzo* (by reference):
  - Parametri formali = puntatori al tipo corrispondente dei parametri
- Concetto
  - Passando gli indirizzi dei parametri attuali posso modificarne il valore

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

void swap(int *pa, int *pb) {
  int tmp;
  tmp = *pa;
  *pa = *pb;
  *pb = tmp;
  printf("swap: a=%d
          b=%d\n",*pa,*pb);}
```

```c
int main()
{
  int x,y;
  printf("Inserire due
          numeri: ");
  scanf("%d %d",&x,&y);
  printf("main: x=%d
          y=%d\n",x,y);
  swap(&x,&y);
  /* ORA x e y VENGONO
     MODIFICATI */
  printf("main: x=%d
          y=%d\n",x,y);}
```

```
Terminal
File Edit View Terminal Go Help
prava@mas:~/teaching/LabSO/examples$ ./par_by_ref.x
Inserire due numeri: 10 20
main: x=10 y=20
swap: a=20 b=10
main: x=20 y=10
prava@mas:~/teaching/LabSO/examples$
```



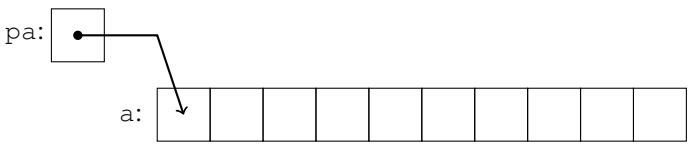
Puntatori e vettori

- In C c'è un legame molto stretto tra puntatori e vettori:
  - *nome del vettore = puntatore al primo elemento*

Esempio:

```c
int a[10], *pa;

pa = &a[0];
```

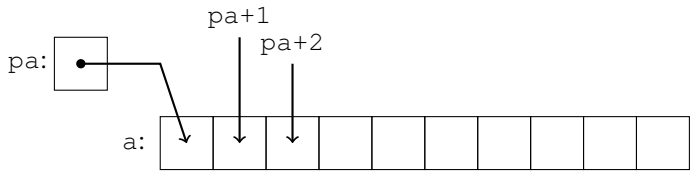


- Se `pa` punta ad un elemento di un vettore, allora:
  - `pa + 1` punta all'elemento successivo

- `pa + i` punta ad `i` elementi dopo `pa`
- `pa - i` punta ad `i` elementi prima di `pa`

```
int a[10], *pa;

pa = &a[0];
```



- Indicizzare un vettore corrisponde ad incrementare il puntatore corrispondente:

`int a[10]`		`int *pa`		
`a`	`&a[0]`	`pa`	`&pa[0]`	puntatore al primo elemento
`&a[i]`	`(a+i)`	`&pa[i]`	`(pa+i)`	puntatore all'`i`-esimo elemento
`a[i]`	`*(a+i)`	`pa[i]`	`*(pa+i)`	valore dell'`i`-esimo elemento

- Ci sono però delle differenze:
  - un puntatore è una *variabile* che può essere modificata:

    `* pa = a` e `pa++` sono operazioni permesse
  - il nome di un vettore non è una variabile, e non si può modificare:

    `* a = pa` e `a++` *non sono operazioni permesse!*
  - la definizione di un puntatore *non riserva spazio di memoria* per il contenuto
- Passare un vettore ad una funzione equivale a passare il *puntatore al primo elemento*

  Esempio: la funzione `strlen`

```
int strlen(char *s)
{
  int n;

  for (n = 0; *s != '\0'; s++)
  {
    n++;
  }
  return n;
}
```

- all'interno della funzione `s` è una variabile locale: l'operazione `s++` non ha nessun effetto sulla stringa passata alla funzione.

Puntatori e costanti stringa

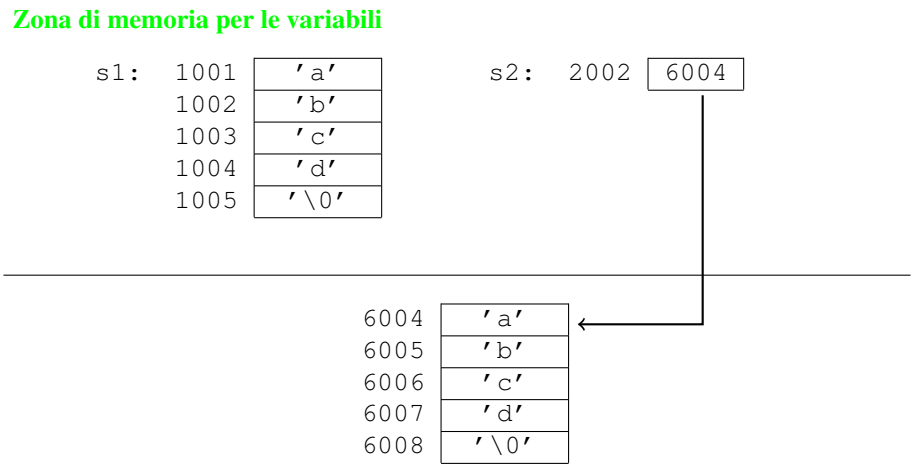
- Differenza sostanziale nel trattamento delle costanti stringa:

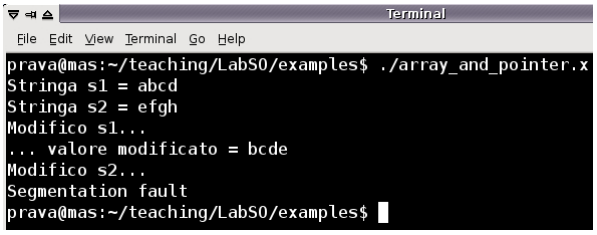
```
char s1[] = "abcd";
char *s2 = "abcd";
```

  - `s1` è un vettore:

    - i caratteri che lo compongono possono cambiare
    - si riferisce sempre alla stessa area di memoria
  - `s2` è un puntatore:
    - punta ad una stringa costante (`"abcd"`)
    - si può far puntare altrove (es. scrivendo `s2 = ...`) ma ...
    - *... la modifica del contenuto di `s2` ha risultato NON DEFINITO*

- *`s1` e `s2` puntano a "zone" di memoria diverse!!!*






```
#include<stdio.h>

int main() {
  char s1[] = "abcd";
  char *s2 = "efgh";
  int i;

  printf("Stringa s1 = %s\n", s1);
  printf("Stringa s2 = %s\n", s2);

  printf("Modifico s1...\n");
  for(i=0;i<4;i++)
    s1[i] = s1[i]+1;
  printf("... valore modificato = %s\n", s1);

  printf("Modifico s2...\n");
  for(i=0;i<4;i++)
    s2[i] = s2[i]+1;
  printf("... valore modificato = %s\n", s2);
}
```

Vettori di puntatori e vettori multidimensionali

- I puntatori possono essere a loro volta memorizzati in vettori:

  - `char *line[MAXLINE];` crea un vettore di `MAXLINE` elementi, ognuno dei quali è un puntatore a carattere.
  - Se ogni `line[i]` punta al primo elemento di un ulteriore vettore di caratteri, il risultato complessivo sarà che il vettore `line` è un *vettore di righe di testo*.

- In C è possibile dichiarare vettori multidimensionali:
  - `int a[3][4];` dichiara una matrice di 3 righe e 4 colonne i cui elementi sono interi.
  - In C un vettore bidimensionale è trattato come un vettore (unidimensionale) di vettori.
  - Quindi l'espressione `a[i][j]` è corretta, mentre l'espressione `a[i,j]` non lo è.

- Le dichiarazioni

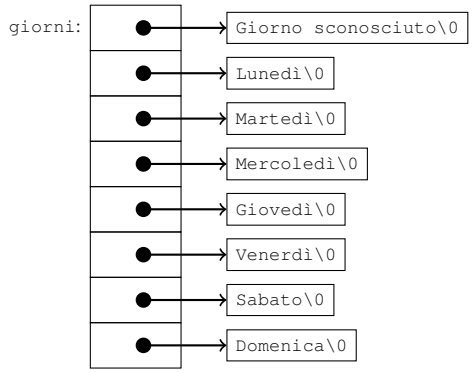
```
int a[10][20];
int *b[10];
```

  differiscono per i motivi seguenti:
  - nella prima dichiarazione vengono allocate le celle di memoria necessarie per contenere *200 interi* (10 x 20)
  - nella seconda vengono allocate le locazioni necessarie per contenere *10 puntatori ad interi*;
  - nel caso di `a` ogni elemento `a[i]` è un vettore di *lunghezza fissa* (20 elementi);
  - nel caso di `b` ogni elemento `b[i]` può puntare ad un vettore di *lunghezza arbitraria* (non necessariamente di 20 elementi);
    - l'inizializzazione degli elementi di `b` deve essere fatta esplicitamente!

- È conveniente utilizzare dei vettori di puntatori quando la lunghezza dei vettori è variabile.

Esempio di vettore di puntatori

```
char *giorni[] = { "Giorno sconosciuto", "Lunedì",
                   "Martedì", "Mercoledì", "Giovedì",
                   "Venerdì", "Sabato", "Domenica"};
```



Esempio di vettore bidimensionale

```
char giorni[][20] = { "Giorno sconosciuto", "Lunedì",
                      "Martedì", "Mercoledì", "Giovedì",
                      "Venerdì", "Sabato", "Domenica"};
```

giorni:	0	Giorno sconosciuto\0
	20	Lunedì\0
	40	Martedì\0
	60	Mercoledì\0
	80	Giovedì\0
	100	Venerdì\0
	120	Sabato\0
	140	Domenica\0