# Puntatori

Salvatore Filippone
salvatore.filippone@uniroma2.it

# Puntatori

Un puntatore in C è una variabile che contiene *un indirizzo*.

```
1  int main(int argc, char *argv[])
2  {
3    int *a, b;
4
5    a = &b;
6  }
```

L'operatore & consente di accedere all'indirizzo di un'altra variabile, e quindi di assegnarlo ad un puntatore.

I puntatori hanno normalmente un tipo associato, ma esistono anche puntatori "generici"

```
1  {
2    void *p;
3  }
```

# Puntatori

Come si usano i puntatori:

- Dichiarare una variabile

```
type *ptr;
```

- Accedere al valore "puntato" (ossia, all'indirizzo corrispondente)

```
val = *ptr;
```

- Accedere al valore "puntato" (ossia, all'indirizzo corrispondente)

```
val = *ptr;
```

- Accedere ad un indirizzo

```
type *ptr = &val;
```

- Aritmetica dei puntatori

```
val = *(ptr+i);
ptr++;
```

# Aritmetica dei Puntatori

- Che vuol dire incrementare un puntatore?

  ```
  ptr++;
  ```

  Aumentare il valore dell'indirizzo di un numero di byte pari alla dimensione del tipo base!

- ```
  int *ptr=1000; /* Se sizeof(int) == 4 allora */
  (ptr+2) == 1008

  char *cp=2000; /* Se sizeof(char) == 1 allora */
  (cp+3) == 2003
  ```

- Operazioni valide: somma e sottrazione con interi, sottrazione tra puntatori, confronto per uguaglianza di puntatori.

N.B.: incrementare un puntatore `void *` è una operazione illegale.
  ```

# Puntatori e memoria dinamica

Come gestire necessità di memoria di dimensioni non prevedibili al momento della scrittura del programma? Ovvero, come si può ottenere memoria durante l'esecuzione di un programma?

- Ottenere memoria con `malloc`:

```
void *malloc(size_t size);
```

- Rilasciare la memoria con

```
void free(void *ptr);
```

Esempio: se abbiamo bisogno di un array di `K` numeri interi, possiamo eseguire:

```
int  *a;
a = (int *) malloc(K*sizeof(int));
```

# Errori con puntatori e memoria dinamica

Quali errori vanno evitati?

- Controllare sempre che `malloc` sia andata a buon fine

```
if ((a=(int *) malloc(K*sizeof(int)) == NULL) {
  fprintf(stderr,"Failed allocation for %d integers\n",K); exit(1);
}
```

- Tentare di rilasciare un puntatore non allocato

```
int *a, *b;
a = (int *) malloc(K*sizeof(int));
b = a;
free(a);
free(b); /* FAIL */
```

- Creare una *memory leak*

```
int *a, i,
for (i=0; i<10; i++) {
  a = (int *) malloc(K*sizeof(int)); /* memory leak */
}
```

# Errori con puntatori e memoria dinamica

Attenzione all'uso delle stringhe:

- Quando si alloca una stringa, ricordare sempre il carattere di terminazione `'\0'`;
- Usare le funzioni di copia `strcpy`, `strncpy` etc.
- Concatenazione e "tokenizzazione" di stringhe sono una fonte di errori

# Il passaggio dei parametri

Cosa succede quando si esegue questo codice?

```
void foo(int a)
{
  a = a+1;
}

main()
{
  int n=10;
  foo(n);
  printf("Value of n:%d\n",n);
}
```

# Il passaggio dei parametri

Cosa succede quando si esegue questo codice?

```
void foo(int a)
{
  a = a+1;
}

main()
{
  int n=10;
  foo(n);
  printf("Value of n:%d\n",n);
}
```

Nel linguaggio C i parametri vengono passati per *valore*, ossia la funzione ne riceve una copia.

# Il passaggio dei parametri

Per risparmiare il tempo della copia, ovvero per modificare l'argomento, occorre passare il parametro per *riferimento*, ovvero passarne un puntatore

```
void foo(int *a)
{
  *a = *a+1;
}

main()
{
  int n=10;
  foo(&n);
  printf("Value of n:%d\n",n);
}
```

Analogamente se si vuole *modificare* un puntatore in una funzione:

```
void foo(int *v)
{
  v = (int *)malloc(10*sizeof(int)); /* WRONG */
}
void foo(int **v)
{
  *v = (int *)malloc(10*sizeof(int)); /* CORRECT */
}
main()
{
  int *v;
  foo(&v);
  v[0]=1;
  printf("Value of v[0]:%d\n",v[0]);
}
```