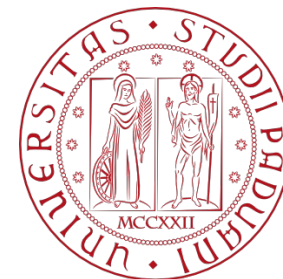


# Argomenti Vari



UNIVERSITÀ  
DEGLI STUDI  
DI PADOVA

- Variabili e Allocazione Dinamica della Memoria
- Numeri Casuali
- Puntatori a Funzione
- Operatori di Incremento e Decremento
- Argomenti da linea di comando
- Switch e break

## Variabili Globali

- create all'inizio dell'esecuzione
- mantengono il loro valore per tutta l'esecuzione
- sono accessibili per tutta l'esecuzione\*

## Variabili Locali

- create quando il blocco dove sono definite viene eseguito
- mantengono il loro valore finché si esegue il blocco
- sono accessibili all'interno dello stesso blocco\*



\*se non viene ridefinita una variabile con lo stesso nome in un blocco più interno

## Variabili Globali

- create all'inizio dell'esecuzione
- mantengono il loro valore per tutta l'esecuzione
- sono accessibili per tutta l'esecuzione\*

## Variabili Locali

- create quando il blocco dove sono definite viene eseguito
- mantengono il loro valore finché si esegue il blocco
- sono accessibili solo all'interno del blocco dove sono definite\*

## Variabili Statiche

- create all'inizio dell'esecuzione
- mantengono il loro valore per tutta l'esecuzione
- sono accessibili solo all'interno del blocco dove sono definite\*

- le variabili dichiarate statiche mantengono il valore tra due invocazioni della funzione dove sono definite
- L'inizializzazione della variabile viene fatta solamente una volta
- il programma alla destra stampa

1

2

```
int contatore(void) {  
    static int c = 0; //eseguita una volta sola  
    c += 1;  
    return c;  
}  
  
int main (void) {  
    printf("%d\n", contatore());  
    printf("%d\n", contatore());  
}
```

- La visibilità di una variabile static è limitata al file dove è definita
- Se il qualificatore static viene usato per una variabile globale, questa non potrà essere riferita in un altro file (tramite extern)
  - se si usa extern, verrà creata una variabile
- si può utilizzare static con una funzione per renderla visibile solamente nel file dove è definita



- Motivazione: la dimensione delle variabili globali e locali non può variare a tempo di esecuzione.
  - Anche gli array a lunghezza variabile (VLA), una volta dichiarati di una certa dimensione non possono essere allungati o accorciati
  - i VLA non fanno parte delle moderne versioni del C
- È possibile allocare memoria per variabili a tempo di esecuzione utilizzando una zona di memoria dedicata: l'heap



- L'allocazione dinamica della memoria è sotto il controllo del programmatore ed avviene mediante funzioni apposite
- Così come il rilascio della memoria utilizzata (la memoria rimane allocata/"utilizzata" finché non viene esplicitamente deallocata)
- La dimensione del blocco è in buona sostanza un argomento della funzione di allocazione
- Il nucleo centrale del sistema di allocazione dinamica della memoria in C è costituito dalle funzioni:
  - malloc() → alloca un blocco di memoria libera
  - free() → libera la memoria precedentemente allocata
  - entrambe le funzioni sono definite in <stdlib.h>



- `void *malloc(size_t <num_byte>)`
- malloc riceve come parametro un numero che indica la quantità di byte da allocare per la variabile
  - malloc non inizializza la memoria allocata
- malloc restituisce un puntatore (di tipo void) al primo byte della memoria allocata
  - Se la memoria nello HEAP non è sufficiente restituisce NULL
- `int *p; p = (int*)malloc(sizeof(int));` //per leggibilità si fa un casting esplicito del puntatore al tipo void
- Il compilatore considererà i `sizeof(int)` byte puntati da p come un intero

- `int *p; p = (int*)malloc(sizeof(int)*20);`
- Concettualmente abbiamo appena dichiarato un vettore di 20 interi, in modo analogo alla dichiarazione:
  - `int p[20];`
- Nel primo caso il compilatore non conosce l'estensione della variabile `p` e non può effettuare gli stessi controlli che riesce a fare nel secondo caso
- Il programmatore deve gestire correttamente il primo caso

- Non utilizzare la free per rilasciare la memoria allocata da p quando si è terminato di utilizzare p, genera uno spreco di memoria
- Per programmi grandi o in casi particolari porta ad esaurire tutta la memoria a disposizione. In ogni caso è un errore logico da evitare.
- Se la funzione f restituisse p, non sarebbe una dangling reference, ma si dovrebbe ricordarsi di utilizzare la free dopo che si è utilizzata la memoria puntata da p

```
int f(int n) {  
    int *p, x;  
    p = (int*)malloc(sizeof(int)*20);  
    x = p[0];  
    //free(p);  
    return x+f(n-1);  
}
```

- Oltre a malloc(), ci sono altre funzioni di allocazione dinamica della memoria:
- calloc: ha prototipo `void *calloc(size_t <num>, size_t <size>)` (notate che il tipo restituito è void \*, puntatore a void, non void) ed alloca una quantità di memoria pari a num\*size (ossia tali da contenere un array di num elementi, ciascuno di size bytes) ed inizializza tutti i bit di memoria a 0
- realloc: ha prototipo `void *realloc(void* , size_t <size>)` e ridimensiona a byte il blocco di memoria puntato da ptr (creato con una malloc/calloc), preservandone il contenuto

- Il calcolatore non è in grado di generare una sequenza casuale di numeri
- E' però in grado di generare in modo deterministico una sequenza di numeri pseudocasuale, ovvero deterministica ma che abbia le stesse proprietà di numeri casuali (es. distribuzione uniforme)
- le funzioni per generare numeri casuali sono definite in `<stdlib.h>`
- `rand()` restituisce un intero  $x$ .  $0 \leq x \leq \text{RAND\_MAX}$  (definita in `stdlib.h`, generalmente almeno  $2^8$ )
- la sequenza di numeri casuali dipende dal valore passato alla funzione `srand()`
- `unsigned int seed; srand(seed);`
- se utilizziamo lo stesso valore di `seed`, otterremo la stessa sequenza di valori casuali

- Scegliere casualmente un numero tra 1 e 10 inclusi

```
srand(0);  
printf("%d\n", ... );
```

- Scegliere casualmente un numero tra 1 e 10 inclusi

```
srand(0);  
printf("%d\n", rand()%10+1);
```

- il nome di un array è in realtà l'indirizzo in memoria del primo elemento dell'array
- il nome di una funzione è in realtà l'indirizzo in memoria di partenza del codice della funzione stessa
- possiamo assegnare ad un puntatore una funzione, possiamo passare funzioni come parametro
- come indichiamo un puntatore a funzione?
- `tipo_restituito (* nome_funzione)(parametri)`
- Es. `int (*compare)(int a, int b)`



- Consideriamo la funzione che ordina un array di interi selezionando ripetutamente l'elemento minore ed inserendolo ad inizio array
- Alla seconda iterazione seleziona il minimo tra gli elementi dell'array che vanno dal secondo all'ultimo e lo inserisce in seconda posizione
- In letteratura l'algoritmo prende il nome di Selection Sort.

```
void selectionSort(int *X, int size) {  
  
    int indice_min;  
    for(int i=0; i<size-1; i+=1) {  
        indice_min =  
        trova_indice_minimo(X+i, size-i);  
        scambia(X+i, &X[indice_min+i]);  
    }  
}
```

- Se vogliamo generalizzare selectionSort in modo da poter ordinare l'array in modo crescente o decrescente, possiamo
- passare un puntatore a funzione come parametro, assumendo che tale funzione calcoli il minimo o il massimo di un array:
- `void selectionSortGeneral(int *X, int size, int (*seleziona_elem)(int *A, int size));`

```
void selectionSortGeneral(int *X, int size,  
int (*seleziona_elem)(int *A, int size)) {  
  
    int elem;  
    for(int i=0; i<size-1; i+=1) {  
        elem = (*seleziona_elem)(X+i, size-i);  
        scambia(X+i, &X[elem+i]);  
    }  
}
```

- Un modo compatto per creare un menu per un'applicazione
  - nel quale l'utente indica con un numero, tra 0 e n, la funzionalità desiderata
- è quello di creare un array di puntatori a funzioni e poi richiamare la funzione dell'indice nell'array corrispondente alla scelta dell'utente:

```
void function1(int a);
```

```
void function2(int b);
```

```
void function3(int c);
```

```
void (*f[3])(int) = { function1, function2, function3 };
```

# Esempio: Menu



```
void function1(int a);  
void function2(int b);  
void function3(int c);
```

```
void (*f[3])(int) = { function1, function2, function3 };  
scanf("%d", &choice);  
(*f[choice])(choice);
```

```
void function1(int a) {  
    printf("You entered %d so function1 was called\n\n", a);  
}
```

# Operatori di Incremento/Decremento



- Forniscono un modo compatto per esprimere espressioni come  $x=x+1$  e  $x=x-1$

Esempio di espressione	Spiegazione
<code>++a</code>	Incrementa <code>a</code> di 1, quindi usa il nuovo valore di <code>a</code> nell'espressione in cui si trova <code>a</code> .
<code>a++</code>	Usa il valore corrente di <code>a</code> nell'espressione in cui si trova <code>a</code> , quindi incrementa <code>a</code> di 1.
<code>--b</code>	Decrementa <code>b</code> di 1, quindi usa il nuovo valore di <code>b</code> nell'espressione in cui si trova <code>b</code> .
<code>b--</code>	Usa il valore corrente di <code>b</code> nell'espressione in cui si trova <code>b</code> , quindi decrementa <code>b</code> di 1.

- La differenza tra `++a`, `a++` diventa significativa se sono usati all'interno di un'espressione o come parametri di una funzione

# Operatori di Incremento/Decremento



```
int c = 5;
```

```
printf( "%d\n", c ); // stampa 5
```

```
printf( "%d\n", c++ ); // stampa 5 e poi incrementa c
```

```
printf( "%d\n\n", c ); // stampa 6
```

```
c = 5;
```

```
printf( "%d\n", c ); // stampa 5
```

```
printf( "%d\n", ++c ); // incrementa c e poi (quindi) stampa 6
```

```
printf( "%d\n", c ); // stampa 6
```

Gli operatori di incremento e decremento

- si applicano solamente a nomi di variabili: non si può scrivere `++(x + 1)`
- hanno priorità rispetto agli altri operatori aritmetici:
  - `*dim--` NON decrementa di uno il valore puntato da `dim`, ma
  - `dim` viene fatto puntare alla cella di memoria precedente a quella attuale e DOPO si effettua la dereferenziazione, ovvero ci si reca alla cella di memoria di indirizzo `dim-1`
  - se `dim` puntava al primo elemento di un array, `*dim--` si spera di ottenere un segmentation fault

- permettono di scrivere codice compatto:

```
char a[10] = "hello";
```

```
int i=0;
```

```
while(i<5)
```

```
    printf("%c ", a[i++]);
```

- ma, a volte, meno leggibile

```
int i=2; printf("%d\n", 3*i+++3); //stampa 9, i==3 dopo la printf.
```

- gli operatori ++ -- non dovrebbero essere utilizzati in un'espressione se la stessa variabile è presente più di una volta nella stessa espressione (anche a sinistra dell'uguale):

```
int i=2; int x=3; x = i++ * i++;
```

- è un comportamento non definito dallo standard: i compilatori hanno libertà di decidere quando effettuare le i++, per cui può succedere che dopo la valutazione del primo i la variabile venga incrementata subito, oppure può succedere che l'incremento avvenga dopo la valutazione del secondo i:
  - $x=2*3$  e  $i=4$  oppure  $x=2*2$ ,  $i=4$



- E' possibile passare argomenti ad un programma C direttamente da linea di comando al momento dell'esecuzione (invece di leggerli da tastiera)

```
gcc -o palindroma palindroma.c
```

```
palindroma abba
```

```
stampa "la stringa abba è palindroma"
```

- i parametri sono separati da spazi: "palindromo abba 1221" indica due parametri stringa
- gli argomenti da linea di comando sono i parametri della funzione main:  

```
int main (int argc, char *argv[])
```
- i nomi delle variabili sono arbitrari, ma si usa argc e argv per tradizione

```
int main (int argc, char *argv[])
```

- una volta definiti i parametri della funzione main, ovvero

```
int main (void) -> int main (int argc, char *argv[])
```

- si avrà sempre almeno un argomento passato da linea di comando

argv[0] è il nome del programma,

quindi  $\text{argc} \geq 1$

- argc: numero di parametri passati dalla linea di comando (incluso il nome del programma)
- argv[i]: l'i-esimo argomento (che è sempre di tipo stringa)

- Scrivere un frammento di codice che stampi la lista dei parametri passati da linea di comando, escluso il nome del programma

- Scrivere un frammento di codice che stampi la lista dei parametri passati da linea di comando, escluso il nome del programma

```
int main (int argc, char *argv[]) {  
    for(int i=1; i < argc; i+=1)  
        printf("argv[%d]:= %s\n", i, argv[i]);  
}  
}
```

- Ma se volessimo passare dei numeri come argomenti da linea di comando?
- Non è possibile. Ciò che si può fare è convertire le stringhe in numeri
- `<stdlib.h>` fornisce le seguenti funzioni

Prototipo della funzione	Descrizione della funzione
<code>double strtod(const char *nPtr, char **endPtr);</code>	Converte la stringa nPtr in un double.
<code>long strtol(const char *nPtr, char **endPtr, int base);</code>	Converte la stringa nPtr in un long.
<code>unsigned long strtoul(const char *nPtr, char **endPtr, int base);</code>	Converte la stringa nPtr in un unsigned long.

# Conversione Stringa -> Double



- **double** strtod(**const char** \*nPtr, **char** \*\*endPtr);
  - nPtr è la stringa che contiene il nostro numero: i caratteri di spaziatura all'inizio della stringa vengono ignorati
  - la stringa può contenere altri caratteri dopo le cifre del numero, es. "12.4ciao", dopo l'esecuzione di strtod(), endPtr punta al primo carattere dopo il numero tradotto, nell'esempio 'c'.
  - Se nessun carattere viene tradotto endPtr punta al primo carattere di \*nPtr
- ```
char *endPtr;  
char *p = " 23.2abc";  
printf("%f -%s-\n", strtod(p, &endPtr), endPtr); // 23.20000 -abc-
```

# Conversione Stringa-> Long



- **long** strtol(**const char** \*nPtr, **char** \*\*endPtr, **int** base);
- nPtr, endPtr hanno lo stesso che hanno in strtod( )
- base è la base di decodifica del numero (binaria, decimale), solitamente base = 0

```
char *endPtr;
```

```
char *p = "-232abc";
```

```
printf("%f -%s-\n", strtol(p, &endPtr, 0), endPtr); // -232 -abc-
```

- in `<stdio.h>` sono presenti anche le seguenti funzioni:
- `int sprintf(char *s, const char *format, ...);`
  - equivalente alla `printf`, ma invece di stampare a video, memorizza nella stringa **sint** `sprintf(char *s, const char *format, ...);`
  - può essere utilizzata per simulare l'assegnamento ad una stringa
- `int sscanf(char *s, const char *format, ...);`
  - equivalente alla `scanf`, ma l'input è letto dalla stringa `s`
  - può essere utilizzata come alternativa alle funzioni `strtod()`, `strtol()`



- L'istruzione Switch è un'alternativa ad una serie di if
- si confronta grade con 'A', se sono uguali si esegue il codice dopo i : (in questo caso nessuna istruzione)
- si passa al case seguente: 'a', e così via
- il break sposta l'esecuzione al comando seguente allo switch (agli esami switch e break non devono essere utilizzati)
- default equivale ad un test sempre vero (viene sempre eseguita a meno di aver incontrato un break in precedenza)

```
char grade;  
int aCount=0, bCount=0;  
switch (grade) {  
    case 'A':  
    case 'a':  
        aCount+=1;  
        break;  
    case 'b':  
        bCount+=1;  
        break;  
    default:  
        printf("%s", "ERRORE");  
}
```