POLITECNICO DI MILANO

Scuola di Ingegneria Industriale

Laurea in Ingegneria Energetica, Meccanica e dei Trasporti





# Informatica B – Sezione D

## Richiamo alle funzioni principali di C

# Premessa

**NOTE INTRODUTTIVE:**

Le seguenti slide sono pensate come strumento complementare alle lezioni.
Potrete trovare indicazioni sui principali argomenti affrontati riguardanti il linguaggio di programmazione C.

Sarà cura dello studente **NON** utilizzare le seguenti slide come materiale principale del corso o senza aver prima studiato e compreso gli argomenti che verranno trattati.

Ogni argomento sarà suddiviso in quattro parti principali:

1. Breve spiegazione
2. Sintassi
3. Esempi
4. Principali errori che si possono commettere

# Premessa

Per tutti gli argomenti trattati nelle seguenti slide viene riportato, nell'indice, il riferimento alla prima pagina delle slide delle lezioni in cui viene trattato il medesimo argomento.

Nel caso in cui non fosse possibile stabilire un riferimento preciso al numero di pagina delle slide delle lezioni, viene riportato solo il nome del file in cui l'argomento viene trattato.

Per una trattazione più approfondita si rimanda ai file delle lezioni, disponibili sul sito:

http://www.bioinformatics.deib.polimi.it/masseroli/infoB/#Lezioni

# Indice

# Indice

# Assegnamento (=)



L'assegnamento in C viene fatto utilizzando il simbolo **=** ed assegna alla variabile a sinistra il risultato dell'espressione (o il contenuto della variabile) a destra.

**SINTASSI:**

<variabile_destinazione> **=** <costante>;

**Oppure:**

<variabile_destinazione> **=** <variabile2>;

**Oppure:**

<variabile_destinazione> **=** <espressione>;

**Oppure:**

<variabile_destinazione> **=** <funzione>;

**NOTA:**
A sinistra del simbolo di assegnamento **NON** può esserci un'espressione!

# Assegnamento (=)



Dove:

- <*costante*> può essere un valore numerico (intero o reale), un carattere, (…)  a seconda del tipo della <variabile_destinazione>.
- <*variabile2*> deve essere una variabile compatibile con <variabile_destinazione> (deve essere dello stesso tipo, oppure deve essere possibile effettuare un cast implicito tra i due tipi).
- <*espressione*> indica una serie di operazioni tra variabili e/o costanti.
- <*funzione*> può essere una qualsiasi funzione delle libreie di C o definita dall'utente (eg. sqrt()).

# Assegnamento (=)



**ESEMPI:**

- ***a = 10;*** //La variabile ***a*** assumerà il valore *10*.
- ***a = 'b';*** //Alla variabile ***a*** (di tipo char) sarà assegnata la lettera *'b'*.
- ***a = b;*** //La variabile ***a*** assumerà lo stesso valore della variabile ***b***.
- ***a = b*(c + 10);*** //La variabile ***a*** assumerà il valore dell'espressione ***b*(c + 10)*.**
- ***a = sqrt(b/10);*** //La variabile ***a*** assumerà il valore restituito dalla funzione ***sqrt(b/10)*** *(ovvero radice quadrata di b/10)*.

E' anche possibile combinare le "modalità" di assegnamento elencate nelle slide precedenti.

**ESEMPI:**

- ***a = sqrt(c) - b;***
- ***a = sqrt(c) – sqrt(b) + d*(c – b);***

# Assegnamento (=)



**ERRORI COMUNI:**

- Confondere l'assegnamento (**=**) con il confronto (**==**).
- Utilizzare l'assegnamento tra variabili di tipo diverso o tra variabili per le quali non è previsto un cast implicito.
- Sbagliare l'ordine delle variabili (***b = a*** al posto che ***a = b***).
- Mettere a sinistra del simbolo **=** un'espressione o qualsiasi cosa che non sia una singola variabile.
- Dimenticarsi di fare il cast nei casi in cui è necessario (ad esempio se si vuole assegnare ad una variabile intera il valore numerico corrispondente della tabella ASCII di una variabile char).

# Operatore di uguaglianza (==)



L'operatore di uguaglianza, in C, è **==** e serve per confrontare se il valore della variabile a sinistra coincide con quello della variabile (o dell'espressione) a destra.

Il confronto restituisce il valore ***1*** se i valori a sinistra e a destra del simbolo **==** coincidono, ***0*** altrimenti.

**SINTASSI:**

<espressione_1> **==** <espressione_2>;

**NOTE:**

**NON** è possibile utilizzare l'operatore di uguaglianza con le variabili di tipo *stringa*. Per confrontare due stringhe si usa la funzione *strcmp()* contenuta nella libreria *string.h*.

# Operatore di uguaglianza (==)



Dove:

- <*espressione_1*> è un'espressione tra variabili (di tipo int, char, *etc*.), valori costanti o funzioni. NON può assumere un valore costante.
- < *espressione_2* > è un'espressione tra variabili, valori costanti o funzioni. L'espressione deve restituire un valore confrontabile con il valore restituito da <*espressione_1*>. Può anche essere un valore costante.

# Operatore di uguaglianza (==)



E’ utilizzato principalmente nelle condizioni degli ***if*** per decidere se eseguire o meno una certa porzione di codice.

**ESEMPIO:**

- ***if(a == sqrt(b))*** // Se la variabile ***a*** assume lo stesso valore di ***sqrt(b)***
  ***c = 10;*** // Allora assegna ***10*** alla variabile ***c***

E’ anche possibile combinare le “modalità” di assegnamento elencate nelle slide precedenti.

**ESEMPIO:**

- ***if(a == (b + sqrt(2))/10 + 1)*** // Se la variabile ***a*** assume lo stesso valore di ***((b + sqrt(2))/10 + 1)***
  ***c++;*** // Allora incrementa la variabile ***c***

# Operatore di uguaglianza (==)



Altro utilizzo frequente di questo operatore si ha all'interno dei cicli (eg. ***do – while***).

**ESEMPIO:**

- ```
  do
  {
      <altre_istruzioni>
     scanf("%c", &risposta);
  } while(risposta == 's');
  ```
  // Continua ad eseguire il ciclo finché ***risposta*** assume il valore ***'s'***
  ```

# Operatore di uguaglianza (==)



**ERRORI COMUNI:**

- Confondere il confronto (**==**) con l'assegnamento (**=**).
- Eseguire il confronto tra variabili dello stesso tipo, ma per il cui tipo non sia definita una relazione di ordinamento. Sono ad esempio confrontabili variabili intere (***int***), reali (***float***) e caratteri (***char***).
- Eseguire il confronto tra variabili di tipo diverso per le quali non è previsto un cast implicito, o senza effettuare un cast esplicito, ove possibile (ad esempio un ***char*** con un ***int***, senza prima effettuare un cast).
- Eseguire il confronto tra stringhe.

# Operatore di disuguaglianza (!=)



L'operatore di disuguaglianza, in C, è **!=** e serve per confrontare se il valore della variabile a sinistra è diverso da quello della variabile (o dell'espressione) a destra.

Il confronto restituisce il valore ***1*** se i valori a sinistra e a destra del simbolo **!= NON** coincidono, ***0*** altrimenti.

**SINTASSI:**

<espressione_1> **!=** <espressione_2>;

**NOTE:**

**NON** è possibile utilizzare l'operatore di disuguaglianza con le variabili di tipo *stringa*. Per confrontare due stringhe si usa la funzione *strcmp()* contenuta nella libreria *string.h*.

# Operatore di disuguaglianza (!=)



Dove:

- <*espressione_1*> è un'espressione tra variabili (di tipo int, char, *etc*.), valori costanti o funzioni. NON può assumere un valore costante.
- < *espressione_2* > è un'espressione tra variabili, valori costanti o funzioni. L'espressione deve restituire un valore confrontabile con il valore restituito da <*espressione_1*>. Può anche essere un valore costante.

# Operatore di disuguaglianza (!=)



**E' utilizzato principalmente**

1. Negli ***if*** per decidere se eseguire o meno una certa porzione di codice.

**ESEMPIO:**

- ***if(a != b)*** // Se la variabile ***a*** assume un valore diverso dalla variabile ***b***
  ***c = 10;*** // Allora assegna ***10*** alla variabile ***c***

2. Nei cicli ***while*** e ***do-while*** per decidere quando terminare il ciclo.

**ESEMPIO:**

- ***while(a != b*10 - 1)***
  ***a++;*** // La variabile ***a*** viene incrementata finché non raggiunge il valore ***b*10 – 1***

E' anche possibile combinare le "modalità" di assegnamento elencate nelle slide precedenti.

**ESEMPIO:**

- ***if(a != sqrt(2) + b)*** // Se la variabile ***a*** assume un valore diverso da ***(sqrt(2) + b)***
  ***c--;*** // Allora decrementa la variabile ***c***

# Operatore di disuguaglianza (!=)



**ERRORI COMUNI:**

- Scambiare l'ordine dei simboli (**=!** al posto di **!=**).
- Eseguire il confronto tra variabili dello stesso tipo, ma per il cui tipo non sia definita una relazione di ordinamento. Sono ad esempio confrontabili variabili intere (***int***), reali (***float***) e caratteri (***char***).
- Eseguire il confronto tra variabili di tipo diverso per le quali non è previsto un cast implicito, o senza effettuare un cast esplicito, ove possibile (ad esempio un ***char*** con un ***int***, senza prima effettuare un cast).
- Eseguire il confronto tra stringhe.

# Disuguaglianze (>, <, >=, <=)



In C le disuguaglianze possono essere effettuate seguendo il seguente schema:

- ***Maggiore (>):*** la variabile di sinistra deve essere strettamente maggiore di quella di destra.
- ***Minore (<):*** la variabile di sinistra deve essere strettamente minore di quella di destra.
- ***Maggiore – uguale (>=):*** la variabile di sinistra deve essere maggiore o uguale a quella di destra.
- ***Minore – uguale (<=):*** la variabile di sinistra deve essere minore o uguale a quella di destra.

# Disuguaglianze (>, <, >=, <=)



**SINTASSI:**

<espressione_1> <operatore> <espressione_2>;

Dove:

- <*espressione_1*> è un'espressione tra variabili (di tipo int, char, *etc*.), valori costanti o funzioni. NON può assumere un valore costante.
- < *espressione_2* > è un'espressione tra variabili, valori costanti o funzioni. L'espressione deve restituire un valore confrontabile con il valore restituito da <*espressione_1*>. Può anche essere un valore costante.
- <*operatore*> è uno degli operatori di disuguaglianza: **>**, **<**, **>=**, **<=**.

**NOTE:**

**NON** è possibile concatenare più simboli di disuguaglianza (eg. b < a < c). Per fare ciò bisogna ricorrere agli operatori logici (AND, OR e NOT).

**NON** è possibile utilizzare gli operatori di disuguaglianza con le variabili di tipo *stringa*. Per confrontare due stringhe si usa la funzione *strcmp()* contenuta nella libreria *string.h*.

# Disuguaglianze (>, <, >=, <=)



**ERRORI COMUNI:**

- Scambiare l'ordine nel maggiore/minore – uguale **(=>** al posto di **>=** o **<=** al posto di **<=**).
- Usare il maggiore – uguale/minore – uguale in condizioni in cui servirebbe il maggiore/minore.
- Eseguire il confronto tra variabili dello stesso tipo, ma per il cui tipo non sia definita una relazione di ordinamento. Sono ad esempio confrontabili variabili intere (***int***), reali (***float***) e caratteri (***char***).
- Eseguire la disuguaglianza tra variabili di tipo diverso per le quali non è previsto un cast implicito, o senza effettuare un cast esplicito ove possibile (ad esempio un ***char*** con un ***int***, senza prima effettuare un cast).
- Concatenare più operatori di disuguaglianza (ad esempio chiedere che una variabile sia compresa tra due valori in questo modo: 10 < a < 20).
- Eseguire il confronto tra stringhe.

# Operatori logici: AND (&&), OR (||) e NOT (!)



**SINTASSI (AND):**

espressione1 **&&** espressione2;

**SINTASSI (OR):**

espressione1 **||** espressione2;

**SINTASSI (NOT):**

**!**espressione1;

Dove **espressione** è una serie di operazioni matematiche tra variabili o numeri, che può essere **VERA** (assume valore **1**) o **FALSA** (assume valore **0**).

**Espressione** può anche essere una singola variabile.

# Operatori logici: AND (&&), OR(||) e NOT (!)



In C, gli operatori logici **AND, OR** e **NOT** si indicano rispettivamente con i simboli **&&, ||** e **!**.

Di seguito sono riportate le tabelle della verità dei tre operatori:

| a | b | a AND b |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

| a | b | a OR b |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

| a | NOT a |
|---|---|
| 0 | 1 |
| 1 | 0 |

Dove **1** sta per **VERO** e **0** sta per **FALSO**.

# Operatori logici: AND (&&), OR(||) e NOT (!)



**ERRORI COMUNI:**

- Utilizzare il simbolo ***&*** al posto del simbolo ***&&*** per l'**AND**.
- Utilizzare il simbolo **|** al posto del simbolo **||** per l'**OR**.
- Utilizzare l'**AND** al posto dell'**OR** o viceversa.
- Applicare gli operatori a condizioni che non assumono valore binario (**0** o **1**).
- Mettere il simbolo di negazione ***!*** dopo la condizione da negare.

# Operatori binari: AND (&), OR (|)



A differenza degli operatori logici booleani, questi operatori servono per fare operazioni binarie.

**SINTASSI (AND):**

<variabile> = <espressione1> **&** <espressione2>;

**SINTASSI (OR):**

<variabile> = <espressione1> **|** <espressione2>;

Dove **espressione** è una serie di operazioni matematiche tra variabili o numeri, che può essere **VERA** (assume valore **1**) o **FALSA** (assume valore **0**).

Possono essere utili, ad esempio, per convertire un numero salvato in una variabile ***char*** in intero (si veda l'esempio nella slide successiva).

# Operatori binari: AND (&), OR (|)



**ESEMPIO:**

Si consideri la seguente porzione di codice:

***char a = '5';***
***int b;***

***b = a & 15;***

Questo codice, non fa altro che convertire il **carattere '5'** nel corrispondente valore **intero 5**.
Infatti, il codice **ASCII** del **carattere '5'** è **65** (in **binario**: **011 0101**), mentre il **numero 5**, in **binario**, è **0101**.
Si nota facilmente che i 4 bit meno significativi del codice ASCII corrispondono proprio al numero intero a cui si riferiscono (questo vale per tutti i numeri, da 0 a 9).
Quindi, per convertire in numero il **carattere '5'**, è sufficiente svolgere la seguente operazione binaria:

**(65) 011 0101 AND**
**(15) 000 1111 =**
**----------**
**(05) 000 0101**

# Resto (%)



L’operatore che restituisce il resto di una divisione, in C, è il %.

Si utilizza come tutti gli altri operatori, ovvero si frappone tra due variabili (o tra una variabile e un numero).

**SINTASSI:**

*<variabile> % <variabile>;*

**Oppure:**

*<variabile> % <numero>;*

E’ utilizzato **principalmente** per:

- Verificare se un numero è pari (num***%***2 == 0) o dispari (num***%***2 != 0).
- Verificare i divisori interi di un numero.
- Verificare se un numero è primo oppure no.
- Conversione di un numero in una base diversa *(eg. base 2)* (vedi metodo dei resti trattato a lezione).

# Incremento (++) e decremento (--)



**SINTASSI:**

Esistono due forme contratte differenti per effettuare l'incremento o il decremento di una variabile in C:

*<variabile> <operatore>;*

*<operatore> <variabile>;*

*<operatore>* può essere: **++** o **--**

Esiste una differenza nell'uso dell'uno o dell'altro metodo. Tale differenza verrà illustrata nelle prossime slide.

# Incremento (++) e decremento (--)



Come illustrato nella slide precedente, per incrementare di uno il valore di una variabile esiste in C una forma contratta che consiste nel far seguire o precedere la variabile dal simbolo **++**.

**ESEMPIO:**

- ***while(i < 10)*** // Esegui il ciclo finché ***i < 10***

  ***i++;*** // Incrementa ***i*** di 1 (è equivalente a scrivere ***i = i + 1*** o ***++i***)

Allo stesso modo, per decrementare di uno il valore di una variabile esiste in C una forma contratta che consiste nel far seguire o precedere la variabile dal simbolo **--**.

**ESEMPIO:**

- ***while(i < 10)*** // Esegui il ciclo finché ***i < 10***

  ***i--;*** // Decrementa ***i*** di 1 (è equivalente a scrivere ***i = i – 1*** o ***--i***)

# Incremento (++) e decremento (--)



## DIFFERENZA TRA LE DUE FORME CONTRATTE:

Il risultato delle due forme contratte è identico. L'unica cosa che cambia è il momento in cui viene fatto l'incremento (o decremento, a seconda del caso).
Bisogna prestare attenzione nel caso in cui si utilizzino le forme contratte all'interno delle condizioni di un ***if***, di un ***while***, di un ***do – while*** o di un ***for***, oppure in un assegnamento.

Infatti, scrivere

***a = ++i;***

Oppure

***a = i++;***

Porta a due risultati differenti.

Infatti, nel primo caso, prima viene incrementato il valore della variabile ***i***, poi viene assegnato il valore di ***i*** ad ***a***.

Nel secondo caso, invece, prima viene effettuato l'assegnamento e successivamente l'incremento.

# Incremento (++) e decremento (--)



In altre parole, scrivere

***a = ++i;***

Equivale alle seguenti righe di codice (in questo ordine):

***i = i + 1;***

***a = i;***

Mentre, scrivere

***a = i++;***

Equivale alle seguenti righe di codice (in questo ordine):

***a = i;***

***i = i + 1;***

# Incremento (++) e decremento (--)



## ESEMPI:

- *c = 10;* // Viene assegnato il valore ***10*** a ***c***
  *a = ++c;* // Alla fine di questa istruzione sia ***a*** che ***c*** valgono ***11***

- *num = 121;* // Viene assegnato il valore ***121*** a ***num***
  *num_2 = num++;* // Alla fine di questa istruzione ***num_2*** vale ***121***, mentre ***num*** vale 122

- *i = 0;*
  *while(++i < 10)* // Ad ogni ciclo prima incrementa ***i*** e poi verifica la condizione
  *{*
  *printf("%d ", i);* // Alla fine del ciclo verrà stampato a schermo: *1 2 3 4 5 6 7 8 9*
  *}*

- *i = 0;*
  *while(i++ < 10)* // Ad ogni ciclo prima verifica la condizione e poi incrementa ***i***

  *{*
  *printf("%d", i);* // Alla fine del ciclo verrà stampato a schermo: *1 2 3 4 5 6 7 8 9 10*
  *}*

# Abbreviazioni



In C esistono delle forme contratte per esprimere somme, sottrazioni, moltiplicazioni e divisioni.

La sintassi è la seguente:

*<variabile1> <operatore> = <espressione>;*

Ed equivale a scrivere:

*<variabile1> = <variabile1> <operatore> <espressione>;*

*<operatore>* può essere: **+**, -, ***, /**, *<etc.>*

*<espressione>* può essere una qualsiasi espressione matematica (eg. (***(a + b + c) *d/e***)

# Abbreviazioni



## ESEMPI:

- *a = 150;* // Viene assegnato il valore ***150*** ad ***a***
  *c = 10;* // Viene assegnato il valore ***10*** a ***c***
  *a / = c;* // Alla fine di questa istruzione ***a*** vale 15 (è come scrivere ***a = a/c;***)

- *i = 0;*
  *i + = 10;* *// Alla fine di questa istruzione i vale 10*

- *num = 121;* // Viene assegnato il valore ***121*** a ***num***
  *num - = num%10;* // Alla fine di questa istruzione ***num*** vale 120 (è come scrivere ***num = num – num%10***)

- ```
  i = 0;
  a = 5;
  while(i++ < 5)
  {
     a * = i; // Alla fine del ciclo, a vale 600
  }
  ```
  ```

# if – else



**SINTASSI:**

```
if(condizione)
{
        istruzione 1;
        istruzione 2;
        …
        istruzione n;
}

else
{
        istruzione 1;
        istruzione 2;
        …
        istruzione n;
}
```

# if – else



Le istruzioni ***if – else*** sono utilizzate in C per *decidere* se eseguire o meno una porzione di codice.

La sintassi prevede che la porzione di codice racchiusa tra parentesi graffe *(non strettamente necessarie se si tratta di una sola riga di codice)* dopo l'istruzione ***if*** venga eseguita se la condizione dell'***if*** risulti **VERA**.

L'***else*** *(opzionale)* rappresenta, invece, la porzione di codice da eseguire se la condizione dell'***if*** risulta **FALSA**.

**ESEMPIO:**

```
if(a < b) // Se a < b incremento di 1 il valore di a
    a++;
else // Se a NON è < b incremento di 1 il valore di b
    b++;
```

# if – else



**ERRORI COMUNI:**

- Utilizzare più di un ***else*** per un solo ***if***.
- Inserire un ***else*** senza che vi siano ***if***.
- Inserire un ***else*** prima che vengano chiuse le parentesi graffe corrispondenti all'***if***.
- Inserire righe di codice tra la fine del blocco di istruzioni relative all'***if*** e l'***else*** corrispondente.

# if – else if



**SINTASSI:**

```
if(condizione 1)
{
        istruzione 1;
        istruzione 2;
        …
        istruzione n;
}

else if(condizione 2)
{
        istruzione 1;
        istruzione 2;
        …
        istruzione n;
}
```

# if – else if



L'istruzione ***if-else if*** rispettano le stesse regole dell'istruzione ***if-else*** illustrata nelle slide precedenti, con la differenza che, mentre nel caso precedente il blocco di istruzioni racchiuse nell'***else*** è eseguito in ogni caso in cui la condizione dell'***if*** non viene rispettata, in questo caso è possibile esprimere una seconda condizione da rispettare affinché le istruzioni racchiuse nel blocco ***else if*** vengano eseguite.

**ESEMPIO:**

- ```
  if(a < b) // Se a < b incremento di 1 il valore di a
    a++;
  else if(b < a) // Se b < a incremento di 1 il valore di b
    b++;
  ```

**NOTA:**

in questo caso, se ***a*** e ***b*** sono uguali, non viene incrementato né il valore di ***a***, né quello di ***b***.
  ```

# if – else if



**ERRORI COMUNI:**

- Utilizzare più di un ***else*** per ogni ***if*** (l'else può poi essere seguito da un ulteriore ***if***, ma la regola da rispettare è che per ogni ***if*** può esserci uno ed un solo ***else***).
- Inserire un ***else if*** senza che vi siano ***if***.
- Inserire un ***else if*** prima che vengano chiuse le parentesi graffe corrispondenti all'***if***.
- Inserire righe di codice tra la fine dell'***if*** e l'***else if*** corrispondente.
- Inserire righe di codice tra la fine del blocco di istruzioni relative all'***if*** e l'***else*** corrispondente.

# Condizioni *if – else/if else* concatenate



**NOTA:**

E’ possibile concatenare più di due ***if – else if – else*** in questo modo:

```
if(condizione 1)
{
        <istruzioni>
}

else if(condizione 2)
{
        <istruzioni>
}

else
{
        <istruzioni>
}
```

L’unica regola da rispettare è che all’inizio ci sia un ***if*** e che l’unico ***else*** (senza ***if***) può essere messo **SOLO** alla fine della sequenza.

# Condizioni *if – else/if else* annidate



**NOTA:**

E’ possibile annidare più di due ***if – else / if – else if*** in questo modo:

```
if(condizione 1)
{
        <istruzioni>
        if(condizione 2)
        {
                <istruzioni>
        }
        else
        {
                <istruzioni>
        }
}
else
{
        <istruzioni>
}
```

# Condizioni *if – else/if else* annidate 2/2

**NOTA:**

Nell'esempio riportato nella slide precedente sono stati annidati solo degli ***if – else***.

E' tuttavia possibile (seguendo lo stesso schema) annidare anche degli ***if – else if*** o degli ***if – else if*** concatenati.

Ovviamente bisogna prestare attenzione ed attenersi alle regole dei costrutti, come riportate nelle relative slide.

E' altrettanto permesso annidare più livelli di ***if – else*** o di ***if – else if***.

**ATTENZIONE:**

Bisogna prestare molta attenzione alla chiusura delle parentesi graffe che racchiudono i blocchi di istruzioni, in quanto, un'errata chiusura delle parentesi può essere sintatticamente corretta (non vìola nessuna regola di C), ma può portare a risultati errati.

# Switch



Nelle slide precedenti sono stati presentati i costrutti ***if – else*** e ***if – else if*** per poter eseguire un certo blocco di istruzioni in base a condizioni espresse sulle variabili.

Nel particolare caso in cui si volesse eseguire un blocco di istruzioni differente in base al valore assunto da una singola variabile, è possibile utilizzare il comando di selezione multipla ***switch***.

Il comando ***switch*** è utilizzato nel linguaggio C per gestire più casistiche dipendenti dai valori che una data variabile può assumere.

E' composto da una serie di etichette **case** e di un caso opzionale **default**.

# Switch



**SINTASSI:**

```
switch(<variabile>)
{
        case <valore_1>:
                istruzione 1;
                …
                istruzione n;
                break;
        …
        case <valore_m>:
                istruzione 1;
                …
                istruzione n;
                break;
        default:
                istruzione 1;
                …
                istruzione n;
}
```

# Switch



Dove:

- *<variabile> è la variabile alla quale è applicato lo switch*. Può essere di tipo **int**, **short**, **double** o **char.**
- *<valore_i>* indica il valore che *<variabile>* deve assumere per eseguire il blocco di istruzioni relativo a quel case *(il blocco di istruzioni va dal simbolo **:** fino all'istruzione **break**)*.
- *default* indica il blocco di istruzioni che devono essere eseguite nel caso in cui *<variabile>* assuma valori che non rientrano tra i *<valori_i>*.

# Switch



**ESEMPIO:**

```
Switch(mese)
{
        case 1:
                printf("Gennaio.\n");
                break;
        …
        case 12:
                printf("Dicembre.")
                break;
        default:
                printf("Mese non riconosciuto.\n");
}
```

# Switch



**ERRORI COMUNI:**

- Mettere le parentesi graffe per racchiudere le istruzioni relative a un ***case <valore_i>***.
- Dimenticarsi le parentesi graffe per racchiudere i ***case <valore_i>*** relativi ad uno ***switch***.
- Dimenticarsi l'istruzione ***break***.
- Utilizzare lo ***switch*** con variabili che non siano ***int***, ***short***, ***long*** o ***char***.

# Ciclo while



Permette di eseguire un blocco di istruzioni (racchiuso tra parentesi graffe nel caso si tratti di più di una riga di codice) fintantoché la condizione espressa nel ***while*** risulta vera.

La condizione **deve** essere un'espressione logica che può assumere valore **VERO** o **FALSO**.

**SINTASSI:**

```
while(condizione)
{
        istruzione 1;
        istruzione 2;
        …
        istruzione n;
}
```

# Ciclo while



## ESEMPI:

- ***while(a< b)*** // Se ***a < b*** incremento di ***1*** il valore di ***a***

  ***a++;*** // E' una sola riga di codice, le parentesi graffe **non** sono necessarie

- ***while(i< 10)*** // Se ***i < 10*** assegno ad ***a a/2***, incremento ***i*** di uno

  ***{***

  ***a = a/2;*** // Sono più righe di codice, le parentesi graffe sono necessarie

  ***i++;***

  ***}***

- ***while(a)*** // Esegue il ciclo finché ***a*** è diversa da ***0***

  ***{***

  ***…***

  ***}***

- ***while(!a)*** // Esegue il ciclo finché ***a*** è uguale a ***0***

  ***{***

  ***…***

  ***}***

# Ciclo while



**NOTA:**

Nel ciclo ***while*** **PRIMA** viene verificata la condizione e **DOPO** vengono eseguite le istruzioni racchiuse nel blocco di codice relativo al ***while***. Questo significa che se la condizione risulta falsa fin dall'inizio, il blocco di codice racchiuso tra parentesi graffe non viene **MAI** eseguito.

**ERRORI COMUNI:**

- Sbagliare la condizione, creando così cicli infiniti (che non terminano mai) oppure cicli che non vengono mai eseguiti (quindi inutili).
- Sbagliare a decidere se usare il ciclo **while** o il ciclo **do-while**. Le differenze tra i due cicli sono poche, ma fondamentali.
- Inserire un punto e virgola dopo la condizione relativa al **while**.

# Ciclo do – while



Permette di eseguire un blocco di istruzioni (**SEMPRE** racchiuso tra parentesi graffe) fintantoché la condizione espressa nel ***while*** risulta vera.

La condizione **deve** essere un'espressione logica che può assumere valore **VERO** o **FALSO**.

**SINTASSI:**

```
do
{
        istruzione 1;
        istruzione 2;
        …
        istruzione n;
} while(condizione);
```

# Ciclo do – while



**ESEMPI:**

- ```
  do
  {
     a++;
  } while(a< b);
  ```
  // Se ***a*** **<** ***b*** incremento di ***1*** il valore di ***a***. Il valore di ***a*** verrà incrementato almeno una volta, anche se ***a*** **>** ***b***.

- ```
  i = 0;
  c = 10;
  do
  {
     c / = 2;
  } while(i);
  ```
  // Esegue il ciclo finché ***i*** è diverso da ***0***.

Nell'ultimo esempio, nonostante ***i*** valga ***0*** fin dall'inizio, il ciclo viene eseguito una volta. Questo perché la condizione del ***while*** viene verificata a posteriori (alla fine di ogni esecuzione del ciclo).

# Ciclo do – while



**NOTA:**

Come si è potuto osservare dagli esempi, nel ciclo ***do – while*** **PRIMA** viene eseguito il blocco di istruzioni e **DOPO** viene verificata la condizione, se risulta vera viene rieseguito il blocco di istruzioni, altrimenti si esce dal ciclo. Questo significa che il blocco di istruzioni viene **SEMPRE** eseguito almeno una volta.

**ERRORI COMUNI:**

- Sbagliare la condizione, creando così cicli infiniti (che non terminano mai) oppure cicli che non vengono mai eseguiti (quindi inutili).
- Sbagliare a decidere se usare il ciclo ***while*** o il ciclo ***do-while***. Le differenze tra i due cicli sono poche, ma fondamentali.
- Dimenticarsi il punto e virgola dopo la condizione del ***while***.

# Ciclo for



Il ciclo for, in C, permette di inizializzare una variabile, verificare una condizione ed incrementare (o decrementare) il valore di una variabile in modo più compatto.

Il principio di funzionamento del ciclo è il seguente:

1. Viene inizializzata la variabile.
2. Viene verificata la condizione (PRIMA di eseguire il ciclo). Se risulta VERA allora si esegue il ciclo, ALTRIMENTI si esce dal ciclo.
3. Viene incrementata (o decrementata) la variabile. L'incremento (o decremento) può anche essere NON unitario.

**SINTASSI:**

```
for(inizializzazione; condizione; incremento)
{
        istruzione 1;
        …
        istruzione n;
}
```

# Ciclo for



Il ciclo for, a differenza degli altri cicli illustrati precedentemente, non è formato solo da una condizione da rispettare, ma è formato da tre parti:

1. Inizializzazione di una variabile (normalmente un contatore). Viene effettuata solo la prima volta che viene eseguito il ciclo.
2. Condizione da rispettare (finché risulta vera si rimane all'interno del ciclo). **Deve** essere un'espressione logica che può assumere valore **VERO** o **FALSO**.
3. Incremento (o decremento) della variabile. Viene effettuata ogni volta che il ciclo viene rieseguito, quindi finché la condizione risulta vera. E' come se appartenesse al blocco di codice relativo al for.

Le tre parti sono separate da un **punto e virgola**.

Il blocco di istruzioni da eseguire finché la condizione risulta vera deve essere racchiuso tra parentesi graffe se è composto da più di una riga di codice.

Il ciclo for risulta particolarmente comodo nel caso in cui si lavora con i vettori o con le matrici.

# Ciclo for



## ESEMPI con incremento:

- ***for(i = 0; i < 10; i++)*** // Se **i < 10** incremento di 1 il valore di **a**

  ***a++;*** // E' una sola riga di codice, le parentesi graffe non sono necessarie

- ***for(i = 0; i < 10; i++)*** // Se **i < 10**

  ***{***

  ***a = a*2;*** // Moltiplico ***a*** per ***2***

  ***b = b/2;*** // Divido ***b*** per ***2***

  ***}***

- ***for(i = 0; ((num%2 > 0) && (i < 50)); i++)*** *// il ciclo viene eseguito finché valgono **ENTRAMBE** le condizioni*

  ***{***

  ***a[i] = num;***

  ***num = num/2;***

  ***}***

# Ciclo for



## ESEMPI con decremento:

- ***for(i = 10 - 1; i >= 0 0; i--)*** // Se **i > 0** incremento di 1 il valore di **a**

  ***a++;*** // E' una sola riga di codice, le parentesi graffe non sono necessarie

- ***for(i = 10-1; i >= 0; i--)***// Se **i > 0**

  ***{***

  ***a = a*2;*** // Moltiplico ***a*** per ***2***

  ***b = b/2;*** //Divido ***b*** per ***2***

  ***}***

- ***for(i = 50-1; ((num%2 > 0) && (i >= 0)); i--)***

  *// Il ciclo viene eseguito finché valgono **ENTRAMBE** le condizioni*

***{***

***a[i] = num;*** // ***a*** è un vettore così dichiarato ***int a[50];***

***num = num/2;***

***}***

# Ciclo for



**ERRORI COMUNI:**

- Dimenticare una delle tre parti di cui è composto il ***for***.
- Separare le tre parti con le virgole al posto che con i punti e virgola.
- Sbagliare ad inizializzare la variabile.
- Incrementare la variabile sbagliata.
- Sbagliare la condizione, generando così cicli infiniti o cicli che non vengono mai eseguiti.
- Sbagliare l'ordine delle tre parti di cui è composto il ***for***.

# Cicli annidati



In C è possibile inserire un ciclo all'interno di un altro ciclo. In questo caso si parla di **cicli annidati**.

I cicli possono dello stesso tipo, oppure di tipo differente (ad esempio è possibile inserire un ciclo ***while*** all'interno di un altro ciclo ***while***, come è possibile inserire un ciclo ***for*** all'interno di un ciclo ***while*** e così via).

# Cicli annidati



**ESEMPI:**

- ***do***
  ***{***
  ***while(num%i == 0)*** // Finchè il resto di ***num/i*** è ***0***, esegui ***num/i***
  ***num = num/i;*** // Mentre si esegue questo ciclo ***i*** è costante!
  ***i++;***
  ***} while(i < num);*** // Finché ***i < num*** esegui il ciclo esterno.

- ***for(i = 0; i < 10; i++)*** // Esegue il ciclo per 10 volte *(**i** = 0, …, 9)*
  ***{***
  ***while(i < 5)*** // Esegue il ciclo 5 volte *(**i** = 0, …, 4)*
  ***{***
  ***a[i] = i + 1;*** // Viene eseguito solo per i primi 5 elementi di ***a[]***
  ***}***
  ***a[i] = a[i]*a[i];*** // Viene eseguito per i primi 10 elementi di ***a[]***
  ***}***

# Cicli annidati



**NOTA:**

Nell'utilizzo di cicli annidati in cui la condizione è dettata da un indice, bisogna ricordarsi di utilizzare due indici differenti, uno per il ciclo esterno, uno per quello interno.
Bisogna fare attenzione anche ad incrementare l'indice del ciclo esterno tra le istruzioni del blocco relativo al ciclo esterno e quello del ciclo interno tra le istruzioni del blocco relativo al ciclo interno. Questo per evitare cicli infiniti.

**ESEMPIO:**

```
for(i = 0; i < 10; i++)
{
   for(j = 0; j < 10; j++) // Finchè j < 10
   // Mentre si esegue questo ciclo i è costante!
            if(vett[i] >= vett[j])
                     printf("%d", vett[i]);
}
```

# Errori nelle condizioni dei cicli



Tra gli errori più comuni che si commettono nel definire un ciclo c'è quello di sbagliare la condizione che determina l'uscita dal ciclo.

In alcuni casi questo errore porta solo a dei risultati errati, tuttavia vi sono due casi ricorrenti quando si sbaglia la condizione di un ciclo:

1. Creare un ciclo che non termina mai.
2. Creare un ciclo che non viene mai eseguito.

# Cicli che non terminano mai



Un ciclo che non termina mai non permette la corretta esecuzione del programma, in quanto si continua ad eseguire a tempo indeterminato il blocco di istruzioni relativo al ciclo.

In genere è un errore creare cicli che non terminano mai.

L'errore è dovuto normalmente ad un'errata condizione che determina l'uscita dal ciclo (la condizione risulterà sempre vera).

# Cicli che non terminano mai



**ESEMPI:**

- ```
  num_tmp = num;
  do
  {
    while(num%i == 0) // Il valore di num non cambia mai!
            num_tmp = num_tmp/i;
    i++;
  } while(i < num_tmp);
  ```
- ```
  for(i = 0; i < 10; j++) // Controllo che i < 10, ma incremento j !
    k = k/2;
  ```

# Cicli che non iniziano mai



Un ciclo che non inizia mai è sostanzialmente un ciclo inutile, in quanto la condizione per la quale si entra nel ciclo non è mai verificata, quindi la porzione di codice relativa a quel ciclo non sarà mai eseguita.

E’ un errore creare cicli che non iniziano mai.

L’errore è dovuto normalmente ad un’errata condizione che determina l’uscita dal ciclo (la condizione risulterà sempre falsa).

# Cicli che non iniziano mai



**ESEMPI:**

- ```
  num_tmp = num;
  do
  {
     while(num%i < 0) // Il valore del resto non potrà mai essere negativo!
                num_tmp = num_tmp/i;
     i++;
  } while(i < num_tmp);
  ```

- **for(i = 10; i < 0; i--)**
  *// i* viene inizializzata a 10, quindi non arriverà mai ed assumere valori negativi, perché la condizione non viene rispettata ed il ciclo non verrà mai eseguito**.**
  ```

# Array



Gli array in C sono utilizzati per definire un insieme di variabili dello stesso tipo (char, int, float, vettori, altri tipi dichiarati tramite ***typedef***, etc.).
Per poter effettuare una qualsiasi operazione utilizzando gli array è necessario riferirsi al singolo elemento.
Per riferirsi al singolo elemento dell'array si utilizza il nome dell'array e si indica tra parentesi quadre la posizione dell'elemento sul quale si vuole operare.

**NOTA:** Non è possibile eseguire operazioni (ad esempio, somme, sottrazioni, assegnamento, etc.) riferendosi all'intero array.

L'unica (e fondamentale) regola da rispettare in fase di dichiarazione di un array è che la dimensione dell'array deve essere nota in fase di dichiarazione e non può più essere modificata durante l'esecuzione.
Non è quindi possibile dichiarare un array di dimensione ***n*** dove ***n*** è una variabile.
Nel corso del programma è possibile utilizzare solo una porzione dell'array, come se si stesse utilizzando un array di dimensione minore, ma non si può mai eccedere la dimenzione massima.

# Array



**SINTASSI:**

***<tipo> <nome_array>[<dimensione>];*** // Dichiarazione

***<nome_array>[<posizione>]*** // Seleziono l'elemento ***<posizione>*** dell'array ***<nome_array>***

Nelle prossime slide vengono illustrati i tre differenti modi possibili per diciarare un array.
I metodi di seguito illustrati sono equivalenti dal punto di vista del risultato finale, ma sono appositamente elencati in ordine decrescente di efficienza e di chiarezza.

In ogni slide verrà brevemente spiegato il motivo per cui un metodo è preferibile rispetto ad un altro.

# Array



**PRIMO METODO:** dimensione dichiarata tramite ***#define***

```
#define <nome> <valore>
int main()
{
   <tipo> <nome_array>[<nome>]; // Dichiarazione di un array di
                                   dimensione <valore>
}
```

Questo metodo è, tra tutti, il migliore dal punto di vista sintattico e di efficienza.
Infatti permette di:

- L'uso della ***#define*** rende più semplice e veloce modificare il codice (si vedano le slide relative alla ***#define*** per maggiori approfondimenti).
- La ***#define*** non comporta occupazione di spazio in memoria. Attribuisce semplicemente un valore ad un nome (***<nome>*** non è né una variabile, né una costante!).

# Array



## ESEMPIO:

- ```
  #define MAX 100
  int main()
  {
    int vett[MAX]; // Dichiarazione di un array vett di 100 interi
  }
  ```
- ```
  #define MAX 10
  int main()
  {
    char str[MAX]; // Dichiarazione di un array str di 10 caratteri
  }
  ```
- ```
  #define MAX 10
  int main()
  {
    int mat[MAX][MAX]; // Dichiarazione di una matrice mat di 10x10 interi
  }
  ```
  ```

# Array



**SECONDO METODO:** dimensione dichiarata tramite ***costante***

```
int main()
{
   int const <nome> = <valore> ;
  <tipo> <nome_array>[<nome>]; // Dichiarazione di un array di
                                  dimensione <valore>
}
```

Questo metodo preserva solo una parte delle proprietà delle ***#define*** Infatti:

- L'uso della ***costante*** rende più semplice e veloce modificare il codice (come l'uso della #***define***).
- Al contrario della ***#define***, tuttavia, viene occupato spazio in memoria. Infatti ***<nome>*** è una variabile a tutti gli effetti, anche se assume valore costante che non può essere modificato all'interno del codice.

# Array



**ESEMPIO:**

- ```
  int main()
  {
    int const max = 100;
    int vett[max]; // Dichiarazione di un array vett di 100 interi
  }
  ```
- ```
  int main()
  {
    int const max = 10;
    char str[MAX]; // Dichiarazione di un array str di 10 caratteri
  }
  ```
- ```
  int main()
  {
    int const max = 10;
    int mat[MAX][MAX]; // Dichiarazione di una matrice mat di 10x10 interi
  }
  ```
  ```

# Array



**TERZO METODO:** dimensione dichiarata in modo diretto

```
int main()
{
   <tipo> <nome_array>[<valore>]; // Dichiarazione di un array di
                                  dimensione <valore>
}
```

Questo metodo è sconsigliato in quanto:

- Modificando la dimensione dell'array bisogna andare a modificare, a mano, tutti i punti del codice in cui questa dimensione viene utilizzata (ad esempio nei cicli ***for*** che scansionano l'array).
  Questo comporta, molto spesso, l'introduzione di errori causati dalla dimenticanza di sostituire alcune occorrenze della vecchia dimensione del codice.
  Tali errori non sono segnalati in fase di compilazione e quindi risulta più complicato identificarli e conseguentemente correggerli.

# Array

**ESEMPIO:**

- ***int main()***
  ***{***
  ***int vett[100];*** // Dichiarazione di un array ***vett*** di ***100*** interi
  ***}***
- ***int main()***
  ***{***
  ***char str[10];*** // Dichiarazione di un array ***str*** di ***10*** caratteri
  ***}***
- ***int main()***
  ***{***
  ***int mat[10][10];*** // Dichiarazione di una matrice ***mat*** di ***10x10*** interi
  ***}***

**ERRORI COMUNI:**

- Dichiarare un array di dimensione variabile (allocazione dinamica). Non si ottiene errore in fase di compilazione, ma è una procedura sbagliata e si ottiene un errore in fase di esecuzione del programma.
  In ANSI C è impedito dichiarare un array avente dimensione definita dal valore assunto da una variabile (allocazione dinamica).
- Effettuare operazioni tra array, senza accedere ai singoli elementi.
- Provare ad accedere ad elementi al di fuori della dimensione massima dell'array.

# Struct



La struct in C è utilizzata per creare aggregazioni di dati di tipo predefinito (char, int, float, vettori, etc.).

**SINTASSI:**

***struct{***
***<tipo_1> <variabili_tipo_3>;***
***<tipo_2> <variabili_tipo_2>;***
***…***
***<tipo_n> <variabili_tipo_n>***
***} <nome_record>;***

Dove:

- <*tipo_i*> è un tipo di dato predefinito (eg. char, int, double, float, etc.) oppure dichiarato dall'utente tramite una ***typedef***.
- <*variabili_tipo_i*> è la lista di variabili associate a <*tipo_i*>.
- <*nome_record*> è il nome che si vuole attribuire al record appena creato.

**NOTA:** la ***struct*** va dichiarata **PRIMA** del ***main***.

# Struct



In questo modo è possibile fornire un nome ad una serie di dati che si riferiscono, concettualmente, ad un'identità.

Ad esempio, se volessi scrivere un programma in cui viene richiesto di inserire i dati di un contatto telefonico, potrei seguire due approcci:

- Uno che non richiede l'uso della ***struct***, meno comprensibile
- Uno che richiede l'uso della ***struct***, più comprensibile

# Struct



**PRIMO MODO** *(senza **struct**)*:

Definisco, all'interno del ***main*** le seguenti variabili:

***char nome[20], cognome[20], email[20], tel[14], cell[14];***
***int eta;***

E successivamente vado a lavorare (eg. leggere, modificare, etc.) su queste variabili.

Come si può facilmente capire, il fatto che tali variabili si riferiscano ad un contatto telefonico è facilmente comprensibile da chi ha scritto il codice, ma lo è molto meno da una persona estranea che va a leggerlo.

# Struct



**SECONDO MODO** *(con **struct**)* **:**

Utilizzo la ***struct*** per dare un nome al record di dati che formano il contatto telefonico:

```
struct{
        char nome[20], cognome[20], email[20], tel[14], cell[14];
        int eta;
}contatto;
```

E successivamente (nel ***main***), per lavorare (eg. leggere, modificare, etc.) i singoli dati che compongono il record, si utilizza la notazione:

***contatto.<dato>***

Ad esempio, se volessi leggere il numero di telefono, scriverei:

***scanf("%s", contatto.tel);***

In questo modo è facilmente comprensibile da chiunque che si parla di un contatto telefonico.

# Definire nuovi tipi di dato



Tramite l'istruzione ***typedef*** è possibile, in C, definire un nuovo tipo di dato.

**SINTASSI:**

***typedef <tipo_dato> <nome_tipo> <[lunghezza]>;***

Dove:

- *<tipo_dato>* è un tipo di dato predefinito (eg. char, int, double, float, etc.) oppure dichiarato dall'utente a sua volta tramite un'altra ***typedef***.
- *<nome_tipo>* è il nome che si vuole attribuire al tipo di dato che si sta andando a creare.
- *<lunghezza>* serve **SOLO** nel caso in cui si vuole dichiarare un tipo di dato che sia un vettore di *<nome_tipo>*. Nel caso in cui il nuovo tipo di dato non fosse un vettore di *<nome_tipo>* questo campo va omesso.

**NOTA:** la ***typedef*** va dichiarata **PRIMA** del ***main***.

# Definire nuovi tipi di dato



In questo modo è possibile creare un nuovo tipo di dato ed utilizzarlo per definire variabili.

Tornando all'esempio usato per la ***struct***, possiamo creare un nuovo tipo di dato ***stringa*** e definire le variabili *nome*, *cognome* e *email* dichiarandole come variabili di tipo ***stringa***.

Per fare questo, come prima cosa dichiariamo il nuovo tipo di dato:

```
typedef char stringa [20];
```

A questo punto ho dichiarato un nuovo tipo di dato che avrà il nome ***stringa***. Tutte le variabili di tipo ***stringa*** che dichiarerò nel mio programma saranno delle stringhe di 20 caratteri.

Posso quindi modificare la ***struct*** del precedente esempio in questo modo:

```
struct{
        stringa nome, cognome, email;
        char tel[14], cell[14];
        int eta;
}contatto;
```

# Definire nuovi tipi di dato



E’ infine possibile combinare la ***struct*** e la ***typedef*** per creare nuovi tipi di dato che siano aggregazioni di dati di tipo predefinito (char, int, float, vettori, etc.) oppure di tipi di dato definiti in precedenza tramita la ***typedef***.

**SINTASSI:**

***typedef struct{***
***<tipo_1> <variabili_tipo_3>;***
***<tipo_2> <variabili_tipo_2>;***
***…***
***<tipo_n> <variabili_tipo_n>***
***} <nome_tipo>;***

**NOTA:** la ***typedef*** va dichiarata **PRIMA** del ***main***.

# Definire nuovi tipi di dato



Dove:

- <*tipo_dato*> è un tipo di dato predefinito (eg. char, int, double, float, etc.) oppure dichiarato dall'utente a sua volta tramite un'altra ***typedef***.
- <*variabili_tipo_i*> è la lista di variabili associate a <*tipo_i*>.
- <*nome_tipo*> è il nome che si vuole attribuire al tipo di dato (che in questo caso sarà una struttura) che si sta andando a creare.

# Definire nuovi tipi di dato



Prendendo in considerazione sempre lo stesso esempio dei contatti telefonici, si pensi di voler creare una rubrica.

Una rubrica sarà un elenco *(array)* di contatti telefonici.

E' quindi possibile definire un nuovo tipo di dato *contatto* in questo modo:

```
typedef struct{
        char nome[20], cognome[20], email[20], tel[14], cell[14];
        int eta;
}contatto;
```

Oppure, se si vuole prima definire il tipo di dato ***stringa***, come fatto nelle slide precedenti:

```
typedef char stringa [20];
typedef struct{
        stringa nome, cognome, email;
        char tel[14], cell[14];
        int eta;
}contatto;
```

# Definire nuovi tipi di dato



A questo punto, all'interno del main, è possibile definire un array di contatti, che sarà la nostra rubrica.

Ogni elemento della rubrica avrà, ovviamente, un *campo* nome, uno cognome, uno email, uno tel e uno cell.

Per definire l'array di tipo *contatto* si procede in modo analogo a tutti gli altri casi in cui si vuole definire un array:

***contatto rubrica[10];***

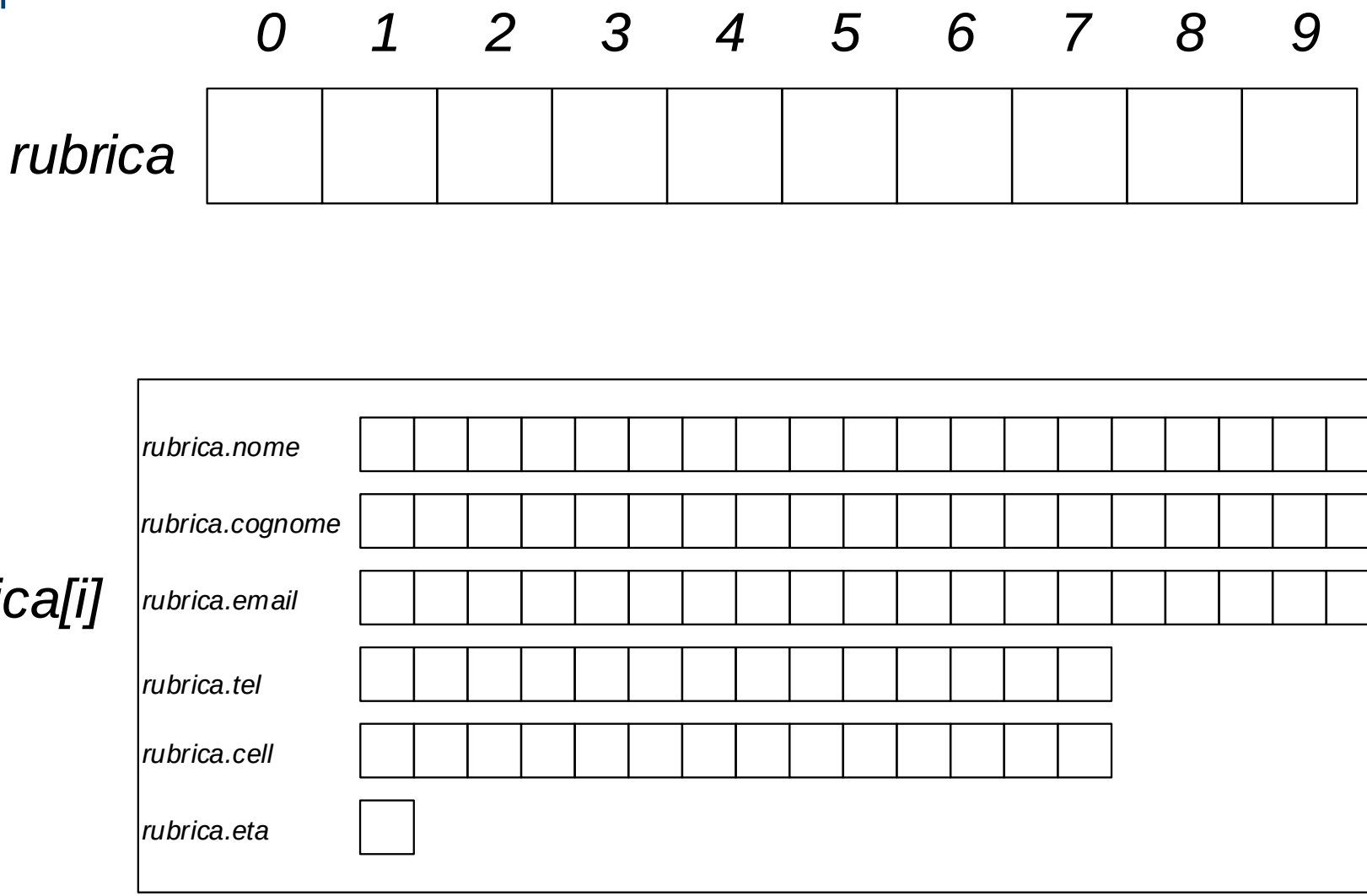
In questo modo ho definito un array di tipo contatto. L'array si chiama *rubrica* ed è lungo *10* (potrò memorizzare 10 contatti nella mia rubrica).

**NOTA:** La definizione dell'array va fatta all'interno del ***main***.

# Definire nuovi tipi di dato



Di seguito una rappresentazione di quello che è stato fatto nella slide precedente:



# Funzioni di input – Scanf



La scanf è una funzione di C, inclusa nella libreria ***stdio.h*** che serve per leggere input da tastiera e salvarlo in una variabile.

**SINTASSI:**

***scanf("%<tipo_dato>", &<nome_variabile>);***

Dove:

- <*tipo_dato*> varia in base al tipo di dato da leggere, seguendo il seguente elenco (vengono riportati solo i tipi principali):
  - ***d*** → int
  - ***c*** → char
  - ***f*** → float
  - ***s*** → stringhe

  <*tipo_dato*> può essere esclusivamente un tipo predefinito di C. Nel seguito verrà spiegato come utilizzare la ***scanf*** con tipi dichiarati dall'utente o con le ***struct***.
- <*nome_variabile*> è il nome di una variabile. Ovviamente deve essere dei tipo <*tipo_dato*>. La ***&*** va OMESSA nel caso in cui si utilizzi la scanf per leggere le stringhe.

# Funzioni di input – Scanf



**ESEMPI:**

- ***int a;***
  ***scanf("%d", &a);*** *//* Legge un intero da tastiera e lo salva nella variabile ***a*** (intera)

- ***char continua;***
  ***do***
  ***{***
  ***<istruzioni>***
  ***scanf("%c", &continua);*** *//* Legge un carattere da tastiera e lo salva nella variabile ***continua*** (char)
  ***} while(continua == 's');***

- ***float num;***
  ***scanf("%f", &num);*** *//* Legge un numero reale da tastiera e lo salva nella variabile ***num*** (float)

- ***int vet[20];***
  ***for(i = 0; i < 20; i++)***
  ***scanf("%d", vet[i]);*** *//* Legge un intero da tastiera e lo inserisce nella posizione i-esima dell'array ***vet*** (array di interi). NON è possibile leggere in altri modi gli array.

# Funzioni di input – Scanf



**ESEMPI:**

- ***char nome[20];***
  ***scanf("%s", nome);*** //Legge una sequenza di caratteri da tastiera, terminata da un invio o da uno spazio e la salva nella stringa ***nome***.

  **NOTA:** La lunghezza massima della sequenza deve essere al più 19, poiché l'array di ***char (stringa)*** è stato dichiarato di lunghezza massima pari a 20.
  Bisogna ricordarsi che, nelle stringhe, un carattere è necessario per terminare la sequenza, quindi una posizione è utilizzata dal così detto terminatore '\0'.
  Ad esempio, se inserissi la parola ***Fabio***, il contenuto della stringa sarebbe:

| F | a | b | i | o | \0 | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

# Funzioni di input – Scanf



Con la ***scanf*** è anche possibile leggere dati che abbiano un certo formato.

Supponiamo ad esempio di voler leggere una data.

Utilizzando la funzione ***scanf*** come visto fin ora, dovremmo usare una ***scanf*** per ogni variabile *(giorno, mese, anno)* in questo modo:

***scanf("%d", &giorno);***

***scanf("%d", &mese);***

***scanf("%d", &anno);***

L'utente dovrebbe inserire il giorno e premere invio, il mese e premere invio, l'anno e premere invio.

# Funzioni di input – Scanf



Come potremmo fare a leggere, invece, una data nel formato *gg/mm/aaa*?

Come si può vedere, quel formato ha una struttura del tipo:

*numero/numero/numero*.

Si può quindi utilizzare una sola scanf riproducendo la struttura appena vista in questo modo:

***scanf("%d/%d/%d", &giorno, &mese, &anno);***

Così facendo, il primo numero verrà salvato nella variabile ***giorno***, il secondo numero (dopo la prima barra /) verrà memorizzato nella variabile ***mese*** e l'ultimo numero (dopo la seconda /) verrà memorizzato nella variabile ***anno***.

In questo modo, l'utente potrà (e dovrà) inserire la data nel formato:

*15/11/2013*.

Ovviamente per effettuare controlli sulla coerenza della data (ad esempio non permettere all'utente di inserire date del tipo *30/25/-1000*) si ricorre alle condizioni ed ai cicli visti fino ad ora.

# Funzioni di input – Scanf



E’ da notare che la sintassi che si esprime nella ***scanf*** è molto restrittiva e conta anche gli spazi.

Infatti la ***scanf*** dell’esercizio precedente accetterà ESCLUSIVAMENTE le date nel formato gg/mm/aaaa senza spazi!

Non accetterà date nel formato gg / mm / aaaa.

Bisogna quindi prestare attenzione nell’utilizzo della corretta sintassi all’interno della ***scanf***.

Da quanto scritto sopra si può comprendere che, anche le seguenti due sintassi sono differenti:

***scanf(“%d”, &a);***

***scanf(“%d ”, &a);***

Precisamente la seconda ***scanf*** presenta uno spazio dopo ***%d*** che introduce un comportamento indesiderato della ***scanf*** stessa (se provate vedrete che la seconda ***scanf*** vi porterà a dover inserire due volte il valore).

# Funzioni di input – Scanf



Vediamo ora come utilizzare la ***scanf*** per leggere valori da memorizzare in strutture di dati.

Come già visto nelle relative slide, per accedere ad un campo della struttura si utilizza la notazione puntata.

Consideriamo una struttura dati che descriva un punto nello spazio:

```
struct{
        int x, y, z;
}punto;
```

Per leggere le coordinate del punto non è ammesso utilizzare una scanf come di seguito riportato:

***scanf("%d", &punto);*** ***NO!***

Si deve, invece, leggere una coordinata per volta in questo modo:

```
scanf("%d", punto.x); // Leggo la coordinata x
scanf("%d", punto.y); // Legge la coordinata y
scanf("%d", punto.z); // Legge la coordinata z
```

# Funzioni di input – Scanf



**ERRORI COMUNI:**

- Utilizzare un tipo di dato incoerente con la variabile nel quale deve essere memorizzato (ad esempio *"%d"* e variabile *float*).
- Inserire la ***&*** quando si stanno leggendo delle stringhe.
- Dimenticare la ***&***.
- Sbagliare a leggere gli array (vanno letti elemento per elemento!)
- Inserire *"%d "* al posto di *"%d"* (al posto della d possono esserci lettere relative ad altri tipi a seconda del caso). Lo spazio in questo caso da problemi!
- Leggere strutture di dati senza fare uso della notazione puntata.
- Far confusione tra il nome della variabile e il suo contenuto. Il nome della variabile non ha nulla a che fare con il suo effettivo valore.

# Funzioni di output – Printf



La printf è una funzione di C, inclusa nella libreria ***stdio.h*** che serve per stampare a video dei messaggi.

**SINTASSI:**

***printf("<messaggio>");***

Dove:

- *<messaggio>* è il messaggio che si vuole stampare a video. Da notare che può contenere esclusivamente caratteri presenti nel codice ASCII (quindi NON lettere accentate).
  E' anche possibile inserire, ad un certo punto del messaggio, il valore di una variabile. Fare riferimento alla prossima slide per vedere come.

# Funzioni di output – Printf



Nel caso in cui, ad un certo punto del messaggio, si volesse inserire il valore di una variabile (o di operazioni tra variabili), si utilizza il carattere ***%*** seguito dalla lettera che indica il tipo di variabile (o il tipo del risultato dell'operazione) che si vuole inserire in quel punto del messaggio (si utilizza un ***%<tipo>*** per ogni variabile che si vuole stampare), seguendo il seguente elenco (vengono riportati solo i tipi principali):

- ***d*** → int
- ***c*** → char
- ***f*** → float
- ***s*** → stringhe

L'elenco delle variabili che si vogliono stampare a video va messo, in ordine *(la prima andrà dove viene posto il primo **%<tipo>**, la seconda dove viene messo il secondo, etc.)* dopo le ***“”*** che terminano il messaggio, separate da virgole. NON va messa la ***&*** prima della variabile.

**ESEMPIO:**

```
int a = 5, b = 3;
printf(“La somma tra %d e %d vale: %d”, a, b, a+b);
```

# Funzioni di output – Printf



Esistono delle sequenze di caratteri che, se inseriti in un qualsiasi punto del messaggio, consentono di formattare il messaggio stesso.

Tali sequenza sono dette ***sequenze di escape***.

Di seguito vengono elencati i più comuni:

- \n → per andare a capo
- \t → tabulazione orizzontale(serve per distanziare più di un singolo spazio. Si può utilizzare per allineare degli elementi per colonne).
- \v → tabulazione verticale (si può utilizzare per allineare degli elementi per righe).
- \’ → apostrofo

# Funzioni di output – Printf



**ESEMPI:**

- ***printf("Hello World!\n");*** // Stampa ***Hello World!*** e va a capo
- ***char continua = 's';***
  ***printf("Hai inserito %c.", continua);*** // Stampa ***Hai inserito s.***
- ***float pi = 3.14;***
  ***printf("pi = %f", pi);*** // Stampa ***pi = 3.14***
- ***char nome[20];***
  ***strcpy(nome, "Fabio");***
  ***printf("%s\n", nome);*** // Stampa ***Fabio*** e va a capo
- ***int vet[20];***
  ***for(i = 0; i < 10; i++)***
  ***{***
  ***vet[i] = i;***
  ***printf("%d ", vet[i]);*** // Stampa ***0 1 2 3 4 5 6 7 8 9***

  **NOTA**: Non è possibile stampare un array per intero.
  Bisogna per forza stampare singolarmente i suoi elementi!

  ***}***

# Funzioni di output – Printf



**ERRORI COMUNI:**

- Dimenticarsi di separare le variabili da virgole dopo il messaggio.
- Dimenticarsi di chiudere i doppi apici (**“”**) che delimitano il messaggio.
- Inserire le variabili prima di aver chiuso le **“”** (errore non segnalato dal compilatore. Tutto quello che viene inserito all’interno delle **“”** viene stampato così come viene scritto).
- Inserire la ***&*** prima dei nomi delle variabili.
- Dimenticarsi il *%<tipo>* dove si vogliono inserire le variabili.
- Provare a stampare un array per intero. Va stampato elemento per elemento.
- Provare a stampare una struttura di dati per intero, senza utilizzare la notazione puntata.
- Inserire il tipo sbagliato nel *%<tipo>* (ad esempio inserire *%d* nel punto in cui si vuole stampare una variabile di tipo ***float***).

# Elevamento a potenza



Per calcolare una potenza, in C, NON è possibile usare il simbolo ^ (che effettua un altro tipo di operazione, quindi se utilizzato non sempre il compilatore restituisce un errore, ma questo non vuol dire che il risultato del calcolo sia corretto!).

Esistono, invece, due metodi per effettuare l'elevamento a potenza:

1. Moltiplicare per se stesso il valore da elevare

   **ESEMPIO:**
   **a^3 → equivalente C → a*a*a;**

2. Utilizzare la funzione ***pow*** contenuta nella libreria ***math.h***
   **SINTASSI:**
   **pow(<elemento_da_elevare>, <potenza>);**

   **ESEMPIO:**
   **a^3 → equivalente C → pow(a, 3);**

# Define

*1/2*

La ***#define*** è un'istruzione di C che permette di assegnare un nome ad un valore.

**SINTASSI:**

**#define <NOME> <valore>**

Dove:

- <NOME> è il nome che si vuole attribuire al valore numerico (per convenzione si utilizzano nomi maiuscoli)
- <valore> è il valore che si vuole sostituire con il nome indicato.

NOTA: si possono utilizzare più define all'interno dello stesso codice, però:

- Vanno tutte messe PRIMA del main (e solitamente dopo le ***#include***)
- Ogni ***#define*** va su una riga diversa e può assegnare UN solo valore ad UN solo nome.
- All'interno del codice NON possono esserci variabili con lo stesso nome utilizzato per la ***#define*** (ricorda che C è *case sensitive,* ovvero distingue tra maiuscole e minuscole: ***a*** è diverso da ***A***).
- Alla fine della ***#define*** NON va mai messo il punto e virgola.

# Define



NON viene dichiarata una variabile costante e non viene occupato spazio in memoria.

Serve in tutti quei casi in cui si ha un valore ricorrente all'interno del codice che ha un certo significato.

Utilizzando la ***#define*** infatti è possibile aumentare la leggibilità del codice, dando dei nomi a valori significativi, e rendere più semplici eventuali modifiche future.
Infatti, nel caso si volesse sostituire un valore numerico molto ricorrende nel codice, senza l'uso della ***#define*** bisognerebbe procedere ad una sostituzione manuale di tutti i valori (con alta probabilità di errore), utilizzando la ***#define*** è invece sufficiente modificare il valore numerico associato ad un dato nome.

**ESEMPIO:**

```
#define MIN 0
#define MAX 10
```