5 - Stile di programmazione

Finora abbiamo creato codici offuscati, incomprensibili agli altri una volta passati i file di lavoro, una buona programmazione oltre alla conoscenza del linguaggio utilizzato è proprio renderlo snello e il più **autoesplicativo** possibile.
Per questo un codice in C sarà più comprensibile di un codice scritto in **assembler**.

Motivazioni dello stile

Un programma viene considerato comprensibile per:

- la **macchina** $\rightarrow$ è comprensibile se può essere **compilato**
- per l'**uomo** $\rightarrow$ è comprensibile quando:
  - Ha una logica immediata
  - Usa espressioni vicine al linguaggio umano
  - Usa forme convenzionali, come:
  - **nomi significativi**
  - **formattazione pulita**
  - **commenti significativi che aiutino la comprensione**

  Il problema che il nostro codice andrà a svolgere deve avere una risoluzione semplice ed efficace, non solamente funzionante, in maniera tale da poterlo documentare da più persone in modo semplice e poter essere compreso da tutti.

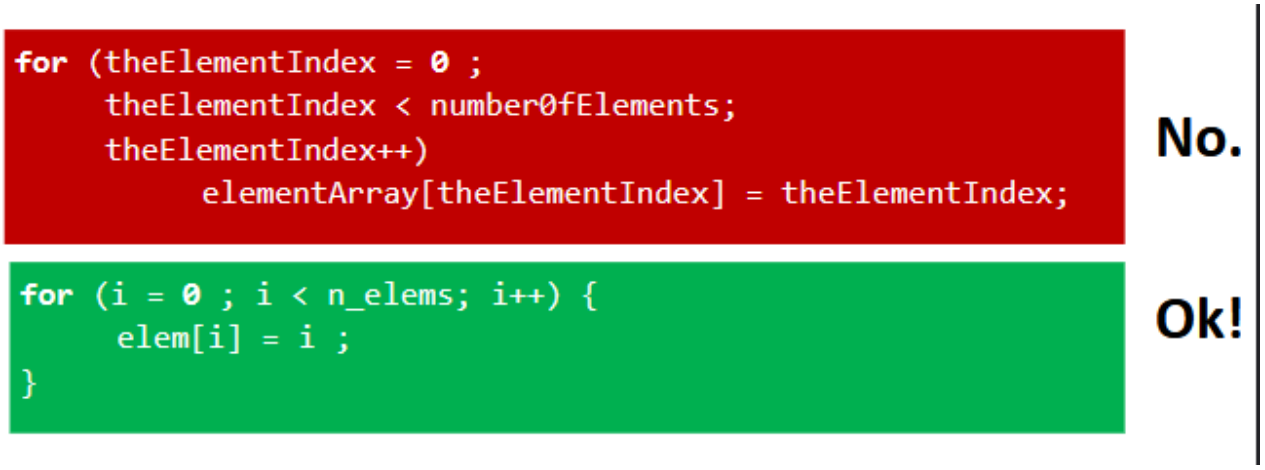
Nomi significativi

Uno stile di programmazione deve partire dalla scelta dei **nomi**; la scelta del nome si basa sull'informazione dello **scopo** del dato/funzione (salvo direttive su offuscamento del codice).
Il nome deve seguire queste quattro caratteristiche formali:

- Chiarire lo scopo
- Conciso
- Mnemonico
- Pronunciabile

  Più lo scopo di un nome è ampio, **maggiore sarà l'informazione** che deve essere convogliata dal nome

  Più una variabile è importante, più attenzione bisogna riporre nella definizione del suo nome, senza andare contro agli standard di utilizzo (come `int i=0`).



Il codice in rosso crea un **sovraccarico cognitivo** per il programmatore, non è coerente sia per il codice stesso che per chi lo leggerà.

Consistenza

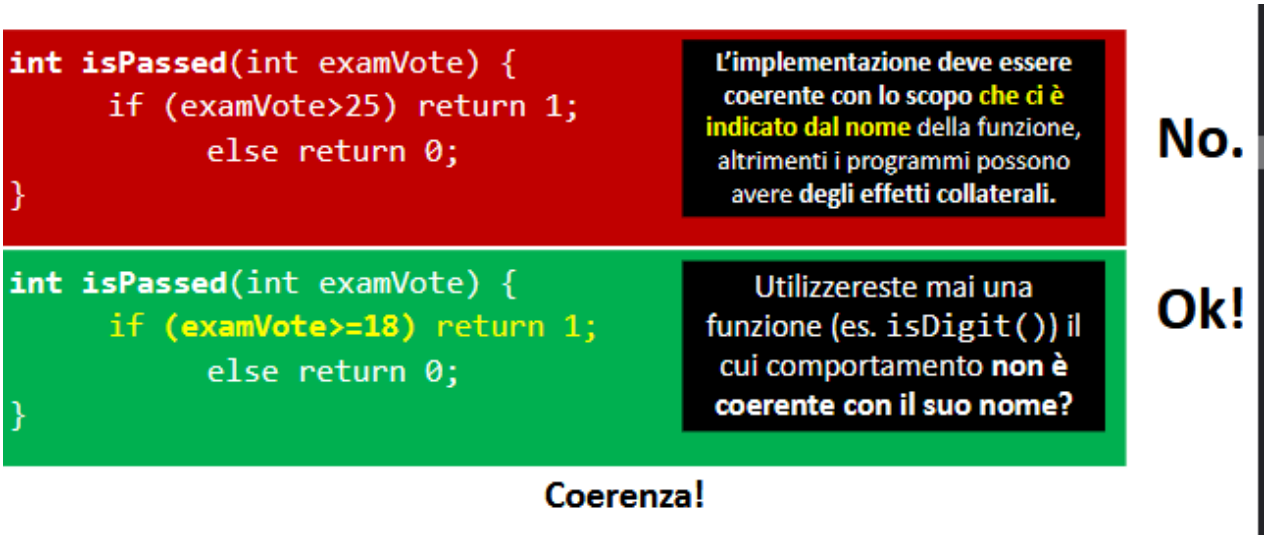
Un altro elemento importante nei nomi è la **consistenza**, ovvero se due variabili diverse si riferiscono allo stesso **concetto**, bisogna utilizzare lo **stesso stile di nomenclatura** e quest'ultima sarà anche automaticamente auto-esplicativa.
Ex:

```
float max_bmi=0;
float min_bmi=0;
```

Convenzioni obbligatorie

Alcune convenzioni sono divenute così diffuse da essere diventate ormai dogmatiche, come:

- Le costanti si scrivono in **maiuscolo**
- Nomi di variabili lunghi devono essere **suddivisi**:
  - **NO** → `int maxbmimaggiorenni`
  - **SI** → `int maxBMIMaggiorenni`
    Oltre alla lettera maiuscola come separatore possono essere usati, in base alla propria preferenza, altri tipi di separatori
- Le funzioni devono avere uno stile **imperativo**, indicando i nomi dell'azione che esegue la sua implementazione
- Le funzioni booleane devono includere la dicitura **is** nel loro nome, per far capire che restituiscono un tipo **true/false**



Il corpo di una funzione (ciò che la funzione fa) e il nome della funzione (il nome dello scopo che svolgerà) devono **corrispondere**.

Espressioni

`if(a>b&&c!=0&&d<1)` è un' espressione NON **trasparente**, questo comporta ad eseguire un tokenaizer difficoltoso anche per un programmatore, costringendolo a un'analisi mentale faticosa.
Bisogna scegliere una chiave chiara durante la scrittura delle espressioni:

- Non sempre la più chiara è la più breve
- Bisogna inserire degli **spazi** tra gli operatori per suggerire i raggruppamenti che eseguono le varie operazione nell'espressione.
- L'espressione dev'essere formattata per aumentarne la **leggibilità**.
  Ex:
  Un caso di un' ottima espressione è il seguente:
  `if ( a>b && c!=0 && d<1)`

Indentanzioni

L'indentazione mostra la struttura espressiva di un programma e deve seguire delle **precise convenzioni** per strutturare correttamente un programma.
L'**Indentazione** è l'allineamento del codice con spazi o tabulazioni per indicare la struttura dei blocchi logici e migliorarne la leggibilità.
L'inserimento di punteggiature, spazi e lasciare qualche riga vuota, rende perfettamente il concetto di ottima indentazione.

Esempio di codice correttamente indentato (senza commenti per ora):

```
for(n=n+1; n<100; n++){
    field[n] = '\0';
}
```

```
*i = '\0';
return ('\n');
```

Il principio da seguire durante la scrittura di una corretta espressione è quello di scriverlo come se fosse un **pensiero pronunciato ad alta voce**, infatti quando pensiamo non usiamo delle forme negate o delle espressioni troppo lunghe, dove è necessaria una certa lunghezza, sarebbe ottimo spezzarle in sotto-espressioni.

Parentesi

Le parentesi sono di ottimo aiuto per il miglioramento della **leggibilità e comprensibilità** di un espressione, le parentesi chiariscono il significato anche quando non sono obbligatoriamente necessarie.
In una espressione del genere: `a!=0&&b+1==0`

- è un'espressione comprensibile per il compilatore, ma la stessa semantica è più leggibile trascrivendola in questo modo: `(a!=0) && (b==1)`

Non sempre la semplicità e la riduttività rende leggibile un codice, conviene suddividere le espressioni, la struttura del codice deve essere orientata alla comprensibilità e non alla compattibilità, la compattibilità rende il codice difficile ed un codice difficile non è sinonimo di bravura.

```
child =
  (!LC&&!RC)?0:(!LC?RC:LC)
```

```
if ((LC==0) && (RC==0))
   child = 0;
else if (LC==0)
   child = RC;
else
   child = LC;
```

29/04/2025

Un buon programmatore capisce che entrambe le espressioni si devono utilizzare, ma in un espressione complicata è preferibile usare la sintassi a destra che la medesima sintassi scritta a sinistra.
L'operatore ternario: `<condizione> ? <valore_vero>:<valore_falso>` dev'essere usato con parsimonia e con espressioni **semplici**.

Effetti collaterali di un codice scarso

Gli effetti collaterali di un codice non chiaro non porta ad errori sintattici ma si possono ricevere comportamenti **inaspettati**, ovvero dei risultati indefiniti e non premeditati.
Ex:

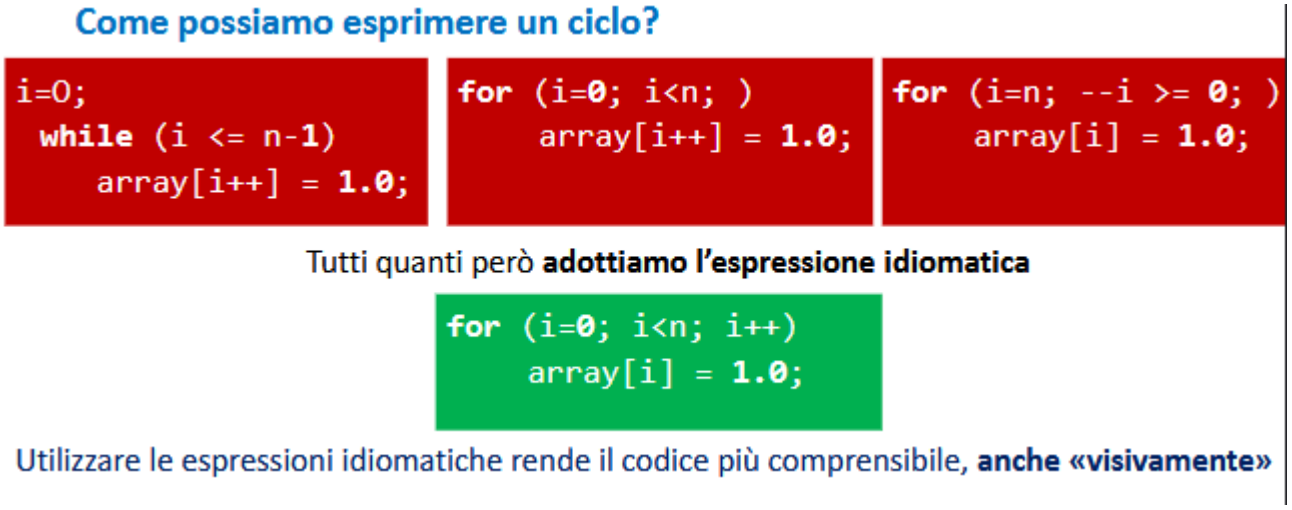
```
printf("Indica la posizione del vettore da modificare e il nuovo
valore:");
scanf("%d %d",&yr,&profit[yr]);
```

- Comportamento **atteso**: il valore assegnato a yr è utilizzato per modificare l'elemento dell'array $profit$ in posizione yr
- Comportamento **effettivo**: crash, poiché si accede ad una posizione casuale dell'array a causa del valore non ancora inizializzato di yr
- **Problema**: $profit[yr]$ usa yr prima che `scanf` riesca a leggerlo in compilazione dal primo `%d`, poiché il compilatore legge da sinistra a destra gli argomenti prima di chiamare la funzione, il valore di yr non è ancora noto nel momento in cui si valuta `&profit[yr]`, perché l'ordine di valutazione degli argomenti nella chiamata a funzione non è garantito.
- **Conseguenza**: si accede a $profit[yr]$ con un indice non valido

Consistenza della medesima forma nel codice

Porzioni diverse del programma devono essere organizzate in modo **prevedibile** e immediatamente comprensibile
La Consistenza nel codice significa **usare la stessa forma** per snippet di codice che hanno un significato simile:
• Indentazione
• uso delle parentesi
• struttura dei cicli
• struttura delle decisioni
Di questi punti bisogna fare una scelta personale dall'inizio del programma e **seguire** questa scelta fino alla fine del codice. Nel caso si modifichi un codice altrui, bisogna **attenersi** all'idea di stile scelta da lui, anche se sono accorgimenti diversi dai nostri gusti personali.

Per quanto riguarda le parentesi, quest'ultime non sono sempre obbligatorie ma la loro rimozione deve essere consapevole e non introdurre bug, questo problema è detto **dangling else**.
Nella programmazione, come il linguaggio parlato, esistono le espressioni **idiomatiche**, ovvero dei concetti che si possono esprimere solo in un modo.
Ex:



Un'altra precisazione della consistenza, è che bisogna evitare la **forma diagonale** ed è preferibile la forma **verticale**, in maniera tale da capire il comportamento causato da ciascun

percorso degli if-else e non confonderci nella visualizzazione.
modo corretto (in verticale):

```
if(condition_1){
    //todo
}
else if(condition_2){
    //todo
}
else if...
else{
    //caso di default
}
```

modo non appropriato (in diagonale):

```
if (argc==3)
    if ((fin = fopen(argv[l], "r")) != NULL)
        if ((fout = fopen(argv[2], "w")) != NULL)
            while ((c = getc(fin)) != EOF) {
                putc(c, fout);
                fclose(fin );
                fclose(fout);
            }
        else
            printf("Can't open output file %s\n", argv[2]) ;
    else
        printf("Can't open input file %s\n", argv[l]);
else
    printf ("Usage: cp inputfile outputfile\n")
```

Numeri magici

In un programma tutti i numeri *"non ovvi"*, diversi da 0 e 1, dovrebbero essere sostituiti da **costanti** simboliche, un numero non fornisce informazioni per chi legge il programma, sarebbe meglio precisare con una costante perché sia importante e cosa vuole rappresentare, inoltre la sua modifica sarà più semplice ciò rende una facile **manutenzione**.

Commenti

I commenti sono stati ideati per aiutare il lettore di un programma, essi inoltre non devono:

- essere in **contraddizione** col codice
- **distrarre il lettore**
- **ripetere sempre gli stessi concetti del codice**
  Inoltre i commenti non sono obbligatori ovunque, infatti per delle variabili auto-esplicative

non serve inserire un commento, essi sono molto più utili ad individuare zone di codice quando lo andremo a leggere e aiutare a spiegare una determinata azione per quando sarà modificata per capirne il comportamento da lettori differenti.
I commenti quindi sono **utili** quando:

- indicano **cosa fa la funzione**
- indicano il signficato dei **parametri**
- aggiunge informazioni non **desumibili** dal nome della funzione
- chiarisce i procedimenti **fondamentali** nell'implementazione
  ATTENZIONE! → se il commento stesso diventa troppo lungo e intricato, sarebbe meglio riscrivere il codice!
  Ex. casi utili:

```
//controllo che il numero di under non sia zero per evitare valori errati
if (count_under>0) ...
---------------------------------------
unsigned char peso=0
//unsigned perché il valore non può essere negativo, char perché il peso è minore di 256 sicuramente
```

Non è vero che i commenti sono utili a spiegare la sintassi del linguaggio e a fornire informazioni aggiuntive.
Ex casi inutili:

```
num_under++ //incremento under
if (eta > SOGLIA_UNDER) // caso over
```