Riassunto Fondamenti di Programmazione

Cosa si intende per informatica?

Il termine “informatica” fu introdotto per la prima volta nel 1962 dall’ingegnere francese Philippe Dreyfus. L’etimologia deriva dalla parola “informatique”, composizione delle parole “**informa**tion” ed “automa**tique**” (”information eletronique ou automatique”, in italiano “informazioni elettroniche o automatiche”).

L’informatica é quindi la **scienza che studia la rappresentazione e l’elaborazione dell’informazione** in maniera automatica mediante l’utilizzo di un elaboratore elettronico (noto anche come computer o calcolatore) o non (per esempio un elaboratore meccanico).

L’informatica d’altro canto, é anche la **scienza dell’astrazione**. L’astrazione é quel procedimento che consente di semplificare (creare un modello) un sistema apparentemente complesso nei suoi costituenti fondamentali (scomporre un sistema complesso in qualcosa di piú semplice ma altrettanto preciso). La scienza dell’astrazione si occupa quindi di **creare il giusto modello per un problema** e di individuare le tecniche corrette per risolverlo in maniera automatica.

L’obiettivo dell’informatica é quindi quello di ***creare delle astrazioni (dei modelli) ai problemi del mondo reale in modo tale che possono essere rappresentate ed elaborate all’interno di un sistema informatico il quale si occupa di risolverli in maniera automatica.*** L’informatica quindi cerca di semplificare i problemi quotidiani rappresentandoli ed elaborandoli all’interno dei calcolatori i quali hanno il compito di risolvere tali problemi in maniera automatica.

Cosa si intende per informazione?

L’informazione quindi gioca un ruolo chiave… ma cos’é un’informazione? Per informazione si intende qualunque conoscenza in grado di ridurre in qualche modo il grado di incertezza (cit. *“Shannon”*). É un qualcosa che, se fornita dissipa i dubbi aumentando la certezza.

Possiamo quindi parlare di informazione solamente in caso di una scelta, se non esiste scelta non c’é informazione. L’informazione in particolare puó essere vista come una terna:

$I = \{Attributo, Tipo, Valore\}$. Dove:

- **l’Attributo** é il nome associato all’informazione,
- il **Tipo** é l’insieme finito di tutti i possibili valori che l’informazione puó assumere.
- il **Valore** rappresenta la scelta di uno specifico elemento all’interno del Tipo

Per esempio considerando l’informazione “ la luce é accesa “:

$I = \{Attributo = \ 'luce', \ Tipo = \{accesa, spenta\}, \ Valore =' \ accesa'\}$

⚠️⚠️LA CARDINALITÁ DEL TIPO DEVE ESSERE FINITA ALTRIMENTI NON POTRÁ ESSERE CORRETTAMENTE RAPPRESENTATO IN UN CALCOLATORE ELETTRONICO O SU UN SUPPORTO FISICO⚠️⚠️

La rappresentazione binaria dell'informazione

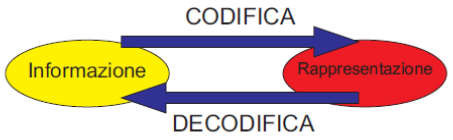
Affinché le informazioni possano essere comprese dal calcolatore, é necessario che quest'ultime siano "tradotte" (codificate) in un linguaggio a lui comprensibile, il **linguaggio binario**.

Con "Rappresentazione dell'informazione" si intende un processo, composto da un insieme di regole, volto alla **ricerca di una codifica (non ambigua) dell'informazione** in un particolare linguaggio.

Le informazioni possono essere rappresentate in un'infinitá di modi diversi (basta pensare a quante lingue si parlano nel mondo) ma, affinché due interlocutori diversi possano comprendersi é necessario che entrambi conoscano il metodo di rappresentazione utilizzato. Per esempio il numero 5, decimale, puó essere rappresentato in diversi modi: V, 5.00, 101, five....

I calcolatori elettronici utilizzano la **rappresentazione binaria**, basata sulle cifre $(0, 1)$ dette digit (in italiano "cifra". La parola deriva dal latino "digĭtus", dita) da cui deriva anche il termine "rappresentazione digitale".

Il procedimento che consente di assegnare una determinata rappresentazione ad una informazione prende il nome di "**codifica**".



Un sistema di codifica utilizza un **alfabeto** (un insieme di simboli) e quindi ogni informazione é rappresentata da una combinazione di questi simboli.

Data un'informazione del tipo T di cardinalitá n, e un alfabeto di k simboli $A = \{S_1, S_2, ..., S_k\}$, inoltre sia S l'insieme di tutte le stringhe o configurazioni composte da m simboli di A. La codifica é una funzione $C(T)$ che ad ogni valore $v \in T$ possibile dell'informazione, associa una stringa $\sigma \in S$; ovvero:

$$C: \ \forall v \in T, \ v \to \sigma \in S$$

Particolarmente importanti sono le codifiche che a **valori diversi associano diverse stringhe codificate**:

$$C: \forall \ v_1, v_2 \in T, v_1 \to \sigma_1 \in S, \ v_2 \to \sigma_2 \in S, \ v_1 \neq v_2 \implies a_1 \neq a_2$$

Il sistema binario e le conversioni di base

Come il sistema numerico decimale, anche quello binario é un sistema **posizionale e pesato** con l'unica differenza che la base utilizzata é la base 2 e si basa solamente su 2 simboli: 0 e 1.

Essendo un sistema posizionale e pesato, ogni numero si esprime come la somma dei prodotti di ciascuna cifra per la base elevata all'esponente che rappresenta la posizione della cifra:

$$10_{10} = 0 \cdot 10^0 + 1 \cdot 10^1$$

$$1010_2 = 0 \cdot 2^0 + 1 \cdot 2^1 + 0 \cdot 2^2 + 1 \cdot 2^3 = 10_{10}$$

Generalmente, per convertire un qualunque numero T in una base b occorre dividere il numero T per la base b e recuperare i resti della divisione letti al contrario:

Dato un numero T vogliamo ottenere la sequenza di cifre in base b: $c_k c_{k-1} c_{k-2} ... c_0$ tale che:

$$T = c_k \cdot b^k + c_{k-1} \cdot b^{k-1} + ... + c_1 \cdot b^1 + c_0 \cdot b^0$$

Effettuando la divisione si ottiene un quoziente Q_0 ed un resto $r = c_0$ che costituisce la prima cifra della sequenza.

Dividendo T/b si ottiene: $T = Q_0 \cdot b + r$ per cui:

$$T = c_k \cdot b^k + c_{k-1} \cdot b^{k-1} + ... + c_1 \cdot b^1 + c_0 \cdot b^0 = (c_k \cdot b^{k-1} + c_{k-1} \cdot b^{k-2} + ... + c_1) \cdot b + c_0$$

Dove quindi $Q_0 = (c_k \cdot b^{k-1} + c_{k-1} \cdot b^{k-2} + ... + c_1)$ rappresenta il quoziente della divisione e c_0 il resto. A questo punto si puó continuare ad applicare il medesimo procedimento ai vari quozienti fino ad ottenere un quoziente nullo:

$$Q_1 = (c_k \cdot b^{k-2} + c_{k-1} \cdot b^{k-3}) \cdot b + c_1$$

ESEMPIO: Convertire il numero 35_{10} in base 2

Divisione	Quoziente	Resto
$35_{10} : 2_{10}$	17_{10}	1
$17 : 2$	8	1
$8 : 2$	4	0
$4 : 2$	2	0
$2 : 2$	1	0
$1 : 2$	0	1

$$35_{10} = 100011_2 = 1 \cdot 2^0 + 1 \cdot 2^1 + 0 \cdot 2^2 + 0 \cdot 2^3 + 0 \cdot 2^4 + 1 \cdot 2^5$$

ESEMPIO: Convertire il numero 100011_2 in decimale

$$100011_2 = 1 \cdot 10^0 + 1 \cdot 10^1 + 0 \cdot 10^2 + 0 \cdot 10^3 + 0 \cdot 10^4 + 1 \cdot 10^5 = 35_{10}$$

Le basi multiple di 2: ottale ed esadecimale

Un caso particolare si ha quando si vuole convertire un numero T dalla base 2 in una qualunque base che é multipla di 2 (o viceversa). $b = 2^k \leftrightarrow k = \log_2 b$.

Bisogna suddividere il numero in n gruppi formati OGNUNO da k cifre, aggiungendo eventualmente degli 0 a sinistra per colmare l'ultimo gruppo.

Un esempio sono la base **ottale** e quella **esadecimale,** le quali utilizzano rispettivamente 8 $(0, 1, 2, 3, 4, 5, 6, 7)$ e 16 $(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F)$ simboli.

Essendo $8 = 2^3$ (una cifra ottale rappresenta esattamente 3 cifre decimali) si divide il numero in gruppi di 3 cifre aggiungendo tanti 0 a sinistra fino a riempire il gruppo. Successivamente si converte ogni gruppo nella cifra ottale corrispondente.

ESEMPIO: Convertire il numero 100011_2 in base 8

$100011_2 = [100][011] = [0 \cdot 2^0 + 0 \cdot 2^1 + 1 \cdot 2^2][1 \cdot 2^0 + 1 \cdot 2^1 + 0 \cdot 2^2] = 43_8$

ESEMPIO: Convertire il 43_8 in binario

L'operazione inversa prevede invece la conversione di OGNI CIFRA ottale con la corrispondente cifra in base 2:

$43_8 = [100][011] = 100011_2$

ESEMPIO: Convertire il numero 100011_2 in esadecimale

Analogamente con la base esadecimale:

$100011_2 = [0010][0011] = [1 \cdot 2^1][1 \cdot 2^0 + 1 \cdot 2^1] = 23_{16}$

La base ottale e quella esadecimale vengono utilizzate in ambito informatico per rappresentare **sinteticamente** lunghe stringhe di bit (come le posizioni in memoria).

Perché i calcolatori utilizzano "solamente" la base 2?

I calcolatori utilizzano il sistema binario in quanto é il sistema piú **semplice** e meno costoso in termini economici e di risorse vista l'architettura interna del calcolatore. La matematica binaria é molto piú semplice.

Essendo il calcolatore un dispositivo elettronico tutto si riconduci a quelli che sono dei **segnali elettrici** i quali, inizialmente erano molto difficili da misurare e controllare, era molto piú semplice distinguere solamente uno stato "acceso" e uno stato "spento". Aggiungendo un'ulteriore cifra occorrerebbe distinguere, oltre agli stati "on" e "off", anche tra diversi livelli di corrente del tipo "un po", "molto" , "poco".

Sono proprio questi due stati che costituiscono l'informazione piú elementare che un computer é in grado di comprendere. Tale informazione prende il nome di **bit**. Il bit non é altro che **l'unitá di misura dell'informazione.**

A dire il vero é errato affermare che i computer "riescono ad utilizzare solamente il sistema binario"... ne sono un esempio l'informatica quantistica e quella fotonica.

La rappresentazione dei numeri interi

Per quanto riguarda la **rappresentazione dei numeri interi** (o interi relativi), bisogna tener presente che essi sono dei numeri naturali preceduti da un segno $+$ o $-$.

Occorre quindi codificare anche il **segno**. Possono essere rappresentati in 2 modi diversi:

- Rappresentazione in segno e modulo
- Rappresentazione in complementi alla base

La rappresentazione in segno e modulo (SM)

La **rappresentazione in segno e modulo** é la piú semplice e consiste nel rappresentare separatamente il segno ed il modulo del numero. Sostanzialmente un bit (detto **MSB**, "Most Significative Bit") "viene sacrificato" per il segno. Generalmente, per il segno si utilizza il bit piú a sinistra.

Segno	Modulo

Tale rappresentazione é poco utilizzata in quanto:

- prevede una **doppia rappresentazione dello zero** $(+0, -0)$;
- le operazioni sono complicate;
- rimane il dubbio sul posizionamento del segno

La rappresentazione in complementi alla base (CB)

I computer sfruttano la **rappresentazione in complementi alla base** in quanto risolve la maggior parte dei problemi della rappresentazione in segno e modulo:

- **Unica rappresentazione per lo** 0;
- Le operazioni sono semplici e si svolgono direttamente sulle rappresentazioni:
  $Rap(x + y) = Rap(x) + Rap(y)$

Tale rappresentazione consente di trasformare i numeri negativi in numeri positivi, detti per l'appunto "complemento a due". I numeri positivi invece rimangono invariati.

Considerando un numero z che deve essere rappresentato su n bit, si definisce complemento a due $\overline{z}$ il numero:
$\overline{z} = 2^n - z$

$$Rap(z) = \begin{cases} x & \rightarrow \quad se\ x \geq 0 \\ 2^n - z & \rightarrow \quad se\ x < 0 \end{cases}$$

Range di rappresentazione

Generalmente con n bit é possibile rappresentare:

- **Naturali (Unsigned):** $[0 \quad , \quad 2^m - 1]$
- **Interi (Signed - SM):** $[-2^{n-1} + 1 \quad , \quad 2^{n-1} - 1]$
- **Interi (Signed - CB):** $[-2^{n-1} \quad , \quad 2^{n-1} - 1]$

La rappresentazione dei tipi non numerici: i caratteri (charset)

Una delle codifiche di caratteri piú famose all'interno del mondo informatico, é la codifica ASCII la quale consente di codificare 128 caratteri differenti dell'alfabeto latino (di cui 33 sono caratteri di controllo), utilizzando 7 bit: $2^7 = 128$.

Una versione estesa, detta per l'appunto "ASCII Esteso" prevede l'utilizzo di 8 bit e quindi la possibilitá di codificare fino a 256 caratteri.

Binary	Oct	Dec	Hex	Glyph
010 0000	040	32	20	
010 0001	041	33	21	!
010 0010	042	34	22	"
010 0011	043	35	23	#
010 0100	044	36	24	$
010 0101	045	37	25	%
010 0110	046	38	26	&
010 0111	047	39	27	'
010 1000	050	40	28	(
010 1001	051	41	29	)
010 1010	052	42	2A	*
010 1011	053	43	2B	+
010 1100	054	44	2C	,
010 1101	055	45	2D	-
010 1110	056	46	2E	.
010 1111	057	47	2F	/
011 0000	060	48	30	0
011 0001	061	49	31	1
011 0010	062	50	32	2
011 0011	063	51	33	3
011 0100	064	52	34	4
011 0101	065	53	35	5
011 0110	066	54	36	6
011 0111	067	55	37	7
011 1000	070	56	38	8
011 1001	071	57	39	9
011 1010	072	58	3A	:
011 1011	073	59	3B	;
011 1100	074	60	3C	<
011 1101	075	61	3D	=
011 1110	076	62	3E	>
011 1111	077	63	3F	?

Binary	Oct	Dec	Hex	Glyph
100 0000	100	64	40	@
100 0001	101	65	41	A
100 0010	102	66	42	B
100 0011	103	67	43	C
100 0100	104	68	44	D
100 0101	105	69	45	E
100 0110	106	70	46	F
100 0111	107	71	47	G
100 1000	110	72	48	H
100 1001	111	73	49	I
100 1010	112	74	4A	J
100 1011	113	75	4B	K
100 1100	114	76	4C	L
100 1101	115	77	4D	M
100 1110	116	78	4E	N
100 1111	117	79	4F	O
101 0000	120	80	50	P
101 0001	121	81	51	Q
101 0010	122	82	52	R
101 0011	123	83	53	S
101 0100	124	84	54	T
101 0101	125	85	55	U
101 0110	126	86	56	V
101 0111	127	87	57	W
101 1000	130	88	58	X
101 1001	131	89	59	Y
101 1010	132	90	5A	Z
101 1011	133	91	5B	[
101 1100	134	92	5C	\
101 1101	135	93	5D	]
101 1110	136	94	5E	^
101 1111	137	95	5F	_

Binary	Oct	Dec	Hex	Glyph
110 0000	140	96	60	`
110 0001	141	97	61	a
110 0010	142	98	62	b
110 0011	143	99	63	c
110 0100	144	100	64	d
110 0101	145	101	65	e
110 0110	146	102	66	f
110 0111	147	103	67	g
110 1000	150	104	68	h
110 1001	151	105	69	i
110 1010	152	106	6A	j
110 1011	153	107	6B	k
110 1100	154	108	6C	l
110 1101	155	109	6D	m
110 1110	156	110	6E	n
110 1111	157	111	6F	o
111 0000	160	112	70	p
111 0001	161	113	71	q
111 0010	162	114	72	r
111 0011	163	115	73	s
111 0100	164	116	74	t
111 0101	165	117	75	u
111 0110	166	118	76	v
111 0111	167	119	77	w
111 1000	170	120	78	x
111 1001	171	121	79	y
111 1010	172	122	7A	z
111 1011	173	123	7B	{
111 1100	174	124	7C	|
111 1101	175	125	7D	}
111 1110	176	126	7E	~

Gli algoritmi

Quel procedimento sistematico (giunge ad una conclusione seguendo delle rigide regole prestabilite) che consente di giungere alla soluzione di un problema, prende il nome di **algoritmo**. piú dettagliatamente si definisce algoritmo quel **procedimento sistematico**, costituito da una serie finita e di operazioni (ognuna delle quali é precisa ed eseguibile), da applicare a dei dati in ingresso perché possa fornire dei dati in uscita.

Un algoritimo presente diverse caratteristiche:

1. **FINITO**: é composto da un numero finito di passi (operazioni);
2. **NON AMBIGUO**: i passi devono essere precisi in maniera da essere interpretati correttamente dell'esecutore;
3. **ESEGUIBILE** (O REALIZZABILE): i passi devono essere eseguibili

4. **DETERMINISTICO**: deve produrre lo stesso identico risultato se eseguito a partire dalle stesse condizioni iniziali;
5. **GENERALE**: deve essere in grado di risolvere “problemi simili”.

La rappresentazione dell'algoritmo in un determinato linguaggio di programmazione, chiara ed eseguibile da un esecutore automatico é detta **PROGRAMMA**. È la formulazione testuale, in un determinato linguaggio di programmazione, di un algoritmo in grado di risolvere un determinato problema.

Quindi lo stesso algoritmo puó avere diverse implementazioni in diversi linguaggi.

Il programma, per essere eseguito, deve essere sottoposto ad un esecutore (umano o artificiale) il quale deve essere in grado di:

- **INTERPRETARE** la sequenza di comandi (la descrizione di un algoritmo deve essere indipendente dall’esecutore che dovrá eseguirlo);
- **ESEGUIRE** i comandi forniti;
- **MEMORIZZARE** le informazioni su opportuni supporti

Affinché il programma possa essere avviato, il sistema deve verificare a priori la disponibilità delle risorse necessarie all’esecuzione del programma e, qualora non fossero presenti, metterle a disposizione (se possibile) in maniera tale che il programma venga avviato. Un programma in esecuzione viene detto “**processo**”.

Tra programma e processo esiste una sostanziale differenza: se il programma é la descrizione di un determinato procedimento risolutivo, il processo é invece l’attuazione di tale procedimento.

Per definire correttamente un algoritmo é quindi necessario specificare le informazioni su cui esso deve lavorare e le operazioni da compiere. Le informazioni possono essere organizzate in diversi modi:

- **Variabile**: ente appartenente a un certo tipo che puó assumere qualunque valore del tipo. Il valore puó essere modificato. Concettualmente una variabile é assimilabile all’idea di un contenitore;
- **Costante**: oggetto appartenente a un certo tipo il cui valore rimane immodificato durante l’esecuzione del programma;
- **Espressione**: sequenza di operandi, operatori (variabili o costanti) e parentesi.

Un algoritmo puó essere rappresentato come pseudocodice o diagramma di flusso.

L’assegnazione

L’operazione di **assegnazione** permette di attribuire un valore a una variabile. Si indica con il simbolo $=$ oppure con $\leftarrow$.

Il formato standard di assegnazione é: $variabile = espressione$

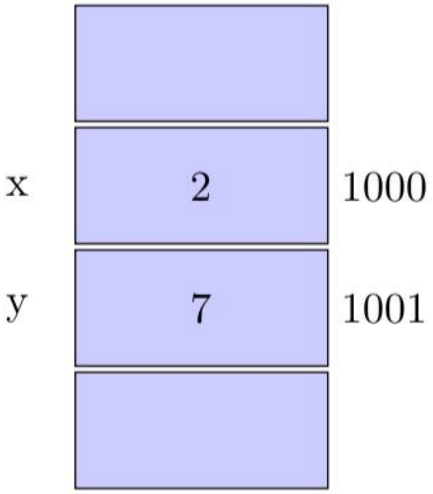
ESEMPIO

$x = 10$ “Il valore costante 10 viene assegnato alla variabile di nome x” .

$x = y$ “Il valore assunto dalla variabile x viene assegnato alla variabile di nome y.

Ció che si trova alla destra dell’uguale deve essere quindi interpretato com il “contenuto della variabile”, il cosiddetto “**Right Value** (RV)”. Nel primo esempio il $RV(x) = 10$ deve essere assegnato alla variabile di nome x, x in questo caso assume il valore simbolico di contenitore ed é detto **Left Value** (LV). Il Left Value coincide con la posizione della variabile in memoria (es. $LF(x) = 1010$).

ESEMPIO



$x = 2$ $”assegnazione”$
$RV(x) = 2$
$LV(x) = 1000$

$y = 7$ $”assegnazione”$
$RV(y) = 7$
$LV(y) = 1001$

Ricapitolando l’assegnazione pone il $RV(x)$ nel contenitore di posizione $LV(x)$.

I costrutti di programmazione

Il **flusso di esecuzione** consente di specificare “se, quando, in quale ordine e quante volte” devono essere eseguite le istruzioni del programma. I costrutti di programmazione permettono di **controllare** tale flusso costruendo istruzioni composte a partire da istruzioni semplici.

Tutte le strutture sono del tipo **one-in-one-out** (singolo ingresso, singola uscita) e sono pertanto considerate come una **macro-istruzione** con un unico punto di ingresso ed un unico punto di uscita. Tali strutture di controllo possono essere inserite una all’interno dell’altra (nesting, innestate o annidate).

Secondo il teorema di **Bohm Jacopini** del 1966, le 3 strutture di controllo bastano a scrivere qualunque algoritmo. Esse sono:

- **Sequenza**
- **costrutti selettivi**: operare delle scelte
- **costrutti iterativi:** ripetere alcune operazioni

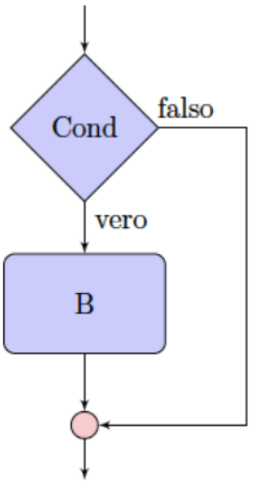
I costrutti selettivi

Consentono di instradare il flusso di controllo dell'algoritmo su **piú strade** in base al valore assunto da una condizione. I costrutti selettivi principali sono 2: "if-then", "if-the-else" e la generalizzazione "case".

Il costrutto "IF-THEN" é detto anche **selezione unaria** (a singola via).

```
if (20>18){
printf("20 is greater than 18");
}
```

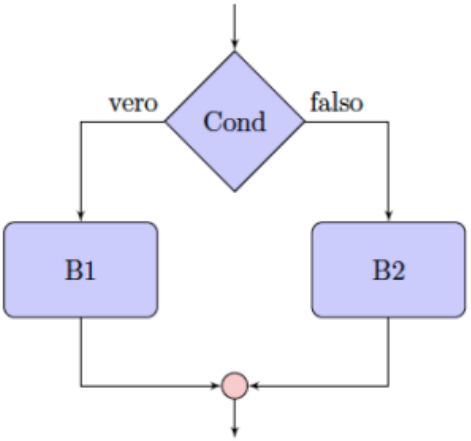
```
if b > a:
   print ("b is greater than a")
```



Se la condizione $(Cond)$ é verificata viene eseguito il blocco B, altrimenti si procede secondo il flusso d'esecuzione.

Il costrutto selettivo "IF-THEN-ELSE" é anche detto "**selezione binaria**" (a due vie).

Se la condizione assume il valore "true" viene eseguito il blocco di codice B_1, altrimenti viene eseguito il blocco B_2.



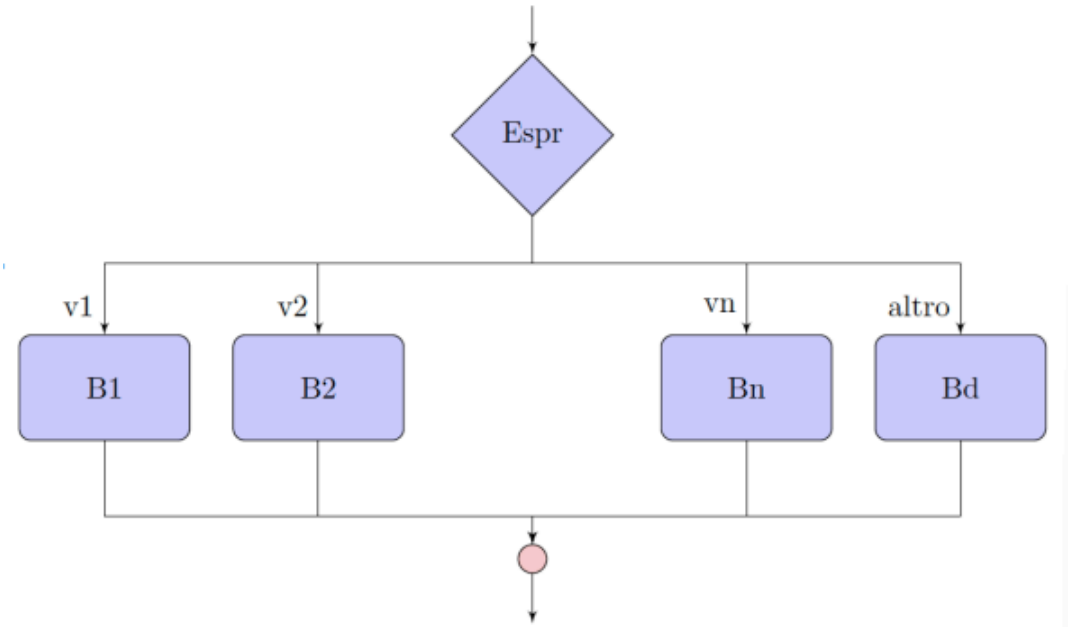

```
if (time < 18) {
  printf("Good day.");
} else {
  printf("Good evening.");
}
```

```
if b > a:
  print("b is greater than a")
else:
  print("a is greater than b")
```

Il “case” é una generalizzazione del costrutto selettivo dove sono presenti piú strade distinte.

In questo caso il controllo avviene su un’**ESPRESSIONE** e non una variabile. Se l’espressione assume valore ”v_1” viene eseguito il blocco ”B_1”"... se assume valore ”v_2”, viene eseguito il blocco ”B_2” e cosí via.

Nel caso in cui l’espressione assumesse valori che non coincidano con le n-strade... si intraprende la strada di default B_d.switch (a) { case 0: printf("il numero e' zero\n"); break; //causa l'uscita immediata dallo switch case 1: printf("il numero e' uno\n"); break; default: printf("numero non compreso\n"); break;




```
switch (a)
  {
     case 0:
     printf("il numero e' zero\n");
```

```
    break; //causa l'uscita immediata dallo switch
    case 1:
    printf("il numero e' uno\n");
    break;}
```

I costrutti iterativi

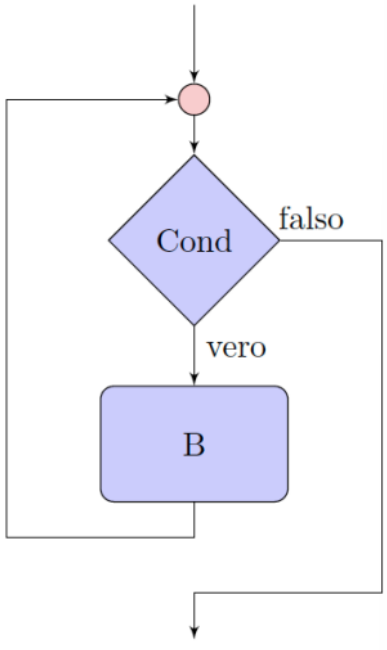
Comunemente dette “**cicli**”, consentono di eseguire ripetutamente un’istruzione o un blocco di istruzioni. Un costrutto iterativo si compone di 3 elementi:

- **Inizializzazione**: le variabili utilizzate devono avere un valore iniziale;
- **Condizione di permanenza nel ciclo**: deve essere valutata un’espressione per determinare la ripetizione o la terminazione del ciclo;
- **Modifica**: affinché si possa uscire dal ciclo almeno una delle variabile della condizione deve essere modificata.

Possiamo distinguere 2 tipologie di costrutti iterativi:

- i cicli predeterminati, si usano quando si conosce a priori il numero di iterazioni da effettuare: **for**.
- i cicli non predeterminati: **while**, **do-while**, repeat-until (poco usato).

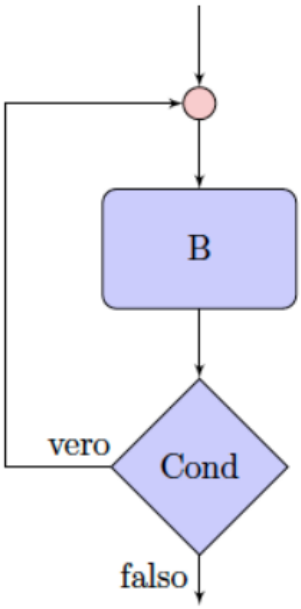
Il ciclo **while** é anche detto “ciclo a condizione iniziale” o con controllo in testa. Il blocco di codice viene ripetuto finché la condizione risulta “true”.




```
while (i != 100){
   printf("%d n", i);
}
```

```
while n < 5:
   somma_totale += n
   n += 1
```

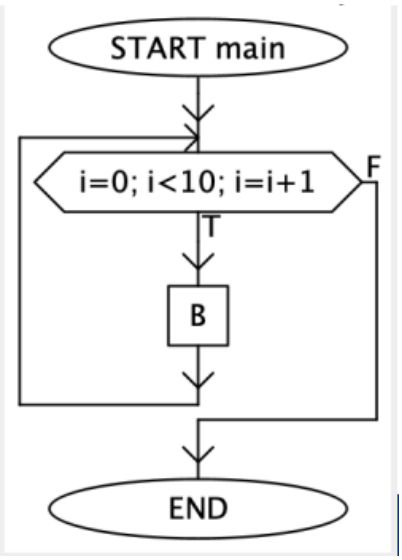
La logica di funzionamento del ciclo **do-while** é identica a quella del ciclo while solo che il controllo é posto in coda (quindi si entrerá sicuramente almeno una volta nel ciclo). Per questo motivo é detto anche ciclo a condizione finale.




```
do
{
   y = f( x );
   x--;
} while ( x > 0 );
```

```
while True:
  n = int(input("Digita un valore: "))
  if n > 0:
    break
```

Il ciclo **for** invece, essendo un ciclo predeterminato, si utilizza quando si sa giá in partenza il numero di ripetizioni da effettuare (es. countdown 10 a 0). Si usa una variabile **contatore** che conta il numero di iterazioni. Puó essere sostituito da 2 cicli while innestati.




```
for(i=0;i<10;i++)
   {
      printf("inserisci il %d numero: ", i+1);
      scanf("%d", &n);
   }
```

```
>>> for n in seq:
...     print('Il quadrato di', n, 'è', n**2)
```

Il ciclo **repeat-until** é un ciclo poco utilizzato (si usa in Pascal). Sostanzialmentne si tratta di un ciclo while con una condizione di “permanenza nel ciclo”. Questo significa che si rimane nel ciclo finché la condizione resta “false”.

I linguaggi di programmazione

Linguaggi a basso livello (LLL) e linguaggi ad alto livello (HLL)

Rientrano a far parte dei "**linguaggi a basso livello**", tutti quei linguaggi di programmazione che differiscono ben poco dal linguaggio macchina e che garantiscono quindi un elevato controllo sull'hardware. Essendo dei linguaggi simili al linguaggio macchina, presentano una sintassi complessa.

Per la progettazione del software si utilizzano i linguaggi di programmazione ad **alto livello** in quanto essi presentano una sintassi "piú semplice" (ma comunque rigida e precisa) rispetto ai linguaggi di basso livello, con parole riconducibili spesso alla lingua inglese. Presentano quindi un alto livello di astrazione per cui si avvicinano molto alla descrizione "umana" di un algoritmo.

I linguaggi ad alto livello:

- permettono una gestione completa dei tipi fondamentali con la possibilitá di definire tipi strutturati;
- offrono un'implementazione immediata dei costrutti di programmazione;
- presentano costrutti precisi per la definizione di dati ed operazioni

I linguaggi ad alto livello quindi agevolano significativamente la programmazione permettendo di scrivere programmi riutilizzabili e modificabili. Questo perché, i linguaggi HLL **non dipendono direttamente dallo specifico processore** per cui possono essere eseguiti su macchine differenti dal momento che, il programma scritto con un linguaggio ad alto livello qualunque sará poi tradotto nel corrispettivo in linguaggio macchina, il quale é strettamente dipendente dal processore.

Il linguaggio Macchina

Dal momento che il computer comprende solamente le informazioni fornite in formato binario, anche le operazioni vanno codificate in binario. Una generica istruzione (di addizione per esempio) si comporrà quindi di una sequenza di bit divisa in 2 parti: un **codice operativo** (OP Code) e due operandi (linguaggio macchina).

Il processore é in grado di eseguire solamente i programmi scritti in **linguaggio macchina** per cui occorre una traduzione dal linguaggio ad alto livello.

Il linguaggio macchina é diverso da macchina a macchina per cui é proprio tale linguaggio a determinare l'insieme delle istruzioni fondamentali che un processore é in grado di compiere (**instruction set**).

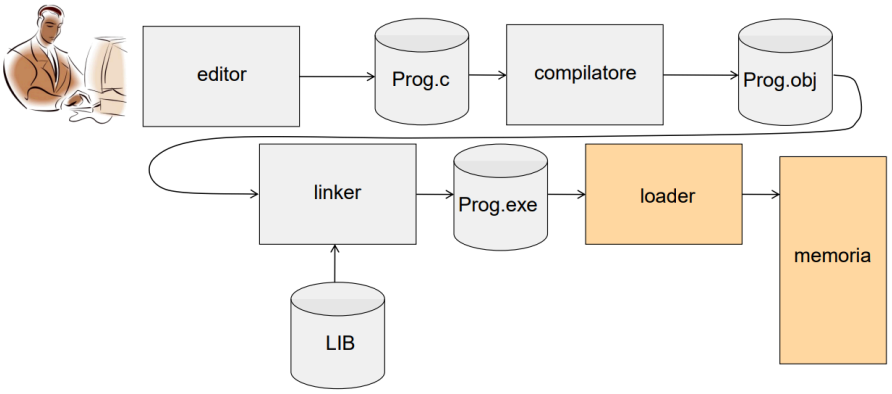
In particolar modo il processore é in grado di compiere solamente operazioni basilari, le operazioni piú complesse sono implementate come programmi. I costrutti per esempio non sono disponibili direttamente come operazioni eseguibili dal processore. Inoltre i tipi di dati non vengono gestiti direttamente.

Compilatori, interpreti e strumenti di sviluppo

Il ruolo di traduzione da alto livello a linguaggio macchina é affidato ai **compilatori** o agli interpreti.

Il compilatore traduce il programma (sorgente) nella sua interezza, generando un **file oggetto** (.obj) che corrisponde alla traduzione del codice sorgente in **linguaggio macchina**. Tale file non é ancora eseguibile dal momento che occorre effettuare l'operazione di **linking**. Il linker si occupa di completare il programma oggetto integrando i vari moduli (sottoprogrammi o **librerie**) necessari all'esecuzione del programma.

Una volta ottenuto il programma eseguibile (.exe) interviene il loader il cui compito é quello di caricarlo in memoria.



Gli **interpreti** invece effettuano iterativamente la traduzione di un'istruzione e l'immediata esecuzione di quest'ultima. Una soluzione intermedia prevede la traduzione del codice sorgente in un linguaggio intermedio detto **bytecode** il quale verrá poi interpretato da una Virtual Machine che emula una CPU. Un esempio é Java.

Gli **IDE** sono degli ambienti di sviluppo in grado di facilitare la programmazione di un software in quanto sono delle soluzioni all-in-one. Presentano al loro interno: un editor, un compilatore, un linker, un loader e spesso un debugger, un programma in grado di eseguire il programma istruzione per istruzione...

Il sistema dei tipi

Essendo ogni informazione una terna del tipo: $I = \{Attributo, Tipo, Valore\}$, per ogni informazione occorre specificare il **tipo** a cui appartiene. Ogni linguaggio di programmazione mete a disposizione un insieme di tipi detto **sistema dei tipi.**

In **arancione** i tipi disponibili in C.

Tipi numerici:

- **intero (int);**
- reale **(float)**
- reale doppia precisione **(double)**.

Tipi non numerici:

- carattere **(char)**;
- logico **(bool)**;
- stringa ;
- enumerativo
- **void**

Tipi strutturati (dipendono dal linguaggio):

- **arrays**
- strutture

Avendo il calcolatore un **supporto di memoria limitato**, non é possibile rappresentare nella loro interezza gli insiemi numerici (infiniti). I tipi numerici sono pertanto dei **sottoinsiemi limitati** (e/o discreti) degli insiemi numeri matematici.

In C:

- Il tipo “**carattere**” generalmente si basa sulla codifica ASCII e puó essere utilizzato anche come tipo numerico (di fatto sono permesse le operazioni artimetiche)
- i tipi enumerativi possono essere definiti dal programmatore.
- Consente l’utilizzo dei **modificatori di tipo**

⚠️⚠️Il Range di rappresentazione di un determinato tipo varia a seconda del compilatore che si utilizza⚠️⚠️

Esistono delle particolari istruzioni che consentono di “modificare” i tipi fondamentali forniti dal linguaggio:

- **signed** (default): il tipo contiene valori positivi e negativi
- **unsigned**: il tipo contiene solamente valori non negativi
- **short**: riduce il numero di byte impiegati per la rappresentazione del tipo (se per esempio per il tipo int si utilizzano 4 byte, anteponendo short se ne utilizzano 2)
- **long**: aumenta il numero di byte impiegati per la rappresentazione del tipo (se per int si usano 4 byte con long se ne utilizzano ALMENO 4)

Quando l’informazione che si vuole considerare é data dall’aggregazione di piú valori diversi, si possono utilizzare le “**variabili strutturate**” le quali consentono di definire l’insieme dei valori di interesse come una singola variabile (per esempio la data di nascita, formata da giorno, mese ed anno).

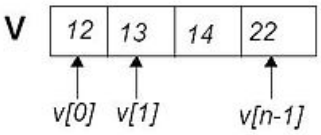
Per definire un **tipo strutturato** occorre:

- il **tipo** di ogni informazione che forma il dato;
- il **costruttore di tipo strutturato**: la forma sintattica che il linguaggio mette a disposizione per definire tali tipi;
- una **funzione di accesso** agli elementi: deve essere possibile accedere a tutte le singole parti che compongono l’informazione strutturata (es. giorno, mese, anno). Per far ció si realizza una funzione di accesso che corrisponde alla forma sintattica da utilizzare per accedere alle singole componenti dell’informazione strutturata.

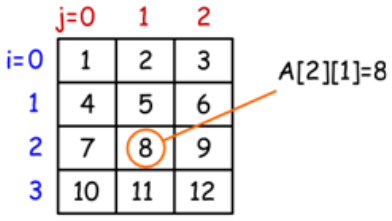
Gli arrays

Gli **array** consentono di aggregare dati omogenei, informazioni dello stesso tipo. Un array é strutturato come un insieme di variabili tutte dello stesso tipo, identificato da un nome unico.

Gli elementi dell’array sono **contigui** in memoria, questo significa che sono disposti in posizione consecutive. Aggiungendo il fatto che ogni cella dell’array presenta la stessa dimensione, il calcolatore memorizza l’indirizzo di memoria solamente della prima cella e si calcola gli indirizzi successivi. Questo é possibile solamente se le celle hanno la stessa dimensione e gli elementi sono vicini, consecutivi.



Gli **array monodimensionali**, comunemente detti **vettori**, consentono di ordinare gli elementi secondo una sola dimensione (i).



Gli **array bidimensionali** (**matrici**) consentono di ordinare gli elementi secondo 2 dimensioni (i, j).

Alcuni linguaggi di programmazione (come C) offrono la possibilità di dichiarare anche array **multidimensionali**, del tipo:

```
int vet [5][3][8]
```

Per definire un vettore occorre:

- **identificatore**;
- **tipo** degli elementi
- **cardinalità** dell'array

```
tipo identificatore[cardinalitá]
int vet [5]

\* Dichiarazione di un vettore di interi,
di cardinalitá 5 in C */
```

Per accedere ai singoli elementi dell'array é necessario specificare il nome della variabile e la posizione dell'elemento di interesse tramite un valore intero che prende il nome di **indice** (nel caso di matrici saranno presenti 2 indici). Gli indici vengono quindi utilizzati dalle **funzioni di accesso** per accedere a un particolare elemento della collezione.

```
vet[5]
```

```
matrix[5,7]
```

I record

A differenza degli array, le **strutture** o record, consentono di collezionare informazione di tipo diverso. In questo caso gli elementi sono identificati tramite un nome (**identificatore**) e non un numero (indice).

- il numero degli elementi é prefissato;
- gli elementi possono essere di tipo diverso;
- il tipo di ogni elemento, detto **campo**, é prefissato;
- l'accesso agli elementi avviene tramite identificatore (e non indice)

Il sistema degli operatori

Un **operatore** é un simbolo che specifica quale operazione occorre eseguire su uno o due operandi, andando effettivamente a definire un'espressione. Il **sistema degli operatori** é un insieme di regole e convenzioni che stabiliscono come valutare in maniera corretta le espressioni.

In base al numero di operandi che li accettano si classificano in:

- **operatori unari**, lavorano su un singolo operando;
- **operatori binari**, lavorano su due operandi;
- **operatori ternari**, lavorano su 3 operandi

In base all'operazione realizzata si classificano in:

- **Operatori Aritmetici** $(+, -, /, *, \%, \hat{})$. Possono essere sia unari che binari.
- **Operatori Relazionali** $(<, >, ==, \neq, \leq, \geq)$
- **Operatori Logici** $(\&\&, ||, !)$. $and(\&\&)$ e $or(||)$ sono binari, $not(!)$ é unario

Le operazioni il cui risultato é dello stesso tipo degli operandi coinvolti, si dicono **interne**. Le operazioni **esterne** invece restituiscono un risultato di tipo diverso da quello degli operandi coinvolti.

Nel caso di espressioni complesse occorre introdurre ulteriori regole per stabilire come valutarle senza ambiguità: precedenza e associatività.

Con **precedenza** si indicano un insieme di regole che stabiliscono la priorità di applicazione dei vari operatori. In particolare un operatore si applica solamente se tutti gli altri operatori piú prioritari sono stati giá applicati.

L'associativitá indica invece come vengono raggruppati gli operatori della stessa precedenza in assenza di parentesi. In particolare un operatore si dice associativo a sinistra se, a parità di priorità, viene applicato da sinistra verso destra.

Il linguaggio C

C è un linguaggio di programmazione ad **alto livello** creato da Dennis Ritchie nel 1972 al Bell Labs, come sviluppo di un linguaggio chiamato B. Tale linguaggio doveva servire per la stesura del sistema operativo **Unix**, sulla macchina PDP-7 ma era troppo pesante per l'hardware a disposizione.

Nel 1972 Denis Ritchie ottimizzó il linguaggio B inventando quello che oggi viene chiamato linguaggio C, il quale permise di riscrivere completamente il sistema Unix. Il linguaggio inizió a diffondersi a partire dal 1978 a seguito della stesura del "libro bianco" *("The C programming language")*. Successivamente venne standardizzato dall'ANSI (American National Standards Institute) per tappe successive, a partire dallo standard ANSI89 (C89) fino all'odierno **C99**.

> Storia del linguaggio C

TOP-DOWN: i sottoprogrammi

Con l'aumentare della dimensione del problema (all'aumentare del numero di informazioni) diventa complesso progettare l'algoritmo risolutivo in un unico passo. É una buona idea quella di decomporre un grande problema in diversi **sotto-problemi** piú piccoli e facilmente risolvibili.

Questo procedimento analitico prende il nome di decomposizione **TOP-DOWN** o step-wise refinement il quale consiste nell'approcciare il problema da una visione generale per poi addentrarsi nei dettagli (dall'alto verso il basso).

Una volta scomposto il problema occorre definire l'algoritmo risolutivo per ogni singolo sotto-problema.

Questa metodologia porta numerosi benefici, fra cui:

- permette di semplificare la soluzione del problema, suddividendolo in problemi più semplici;
- permette di suddividere il lavoro tra persone diverse, che si possono occupare ciascuna di un sottoproblema (modularitá)
- permette di rendere più comprensibile il lavoro e quindi più facile la manutenzione successiva

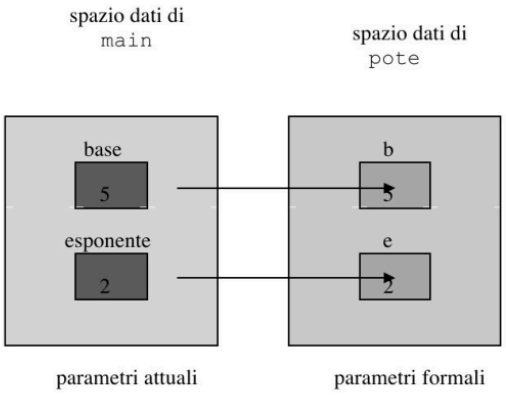
Il programma principale (**main**) sarà quindi composto da un'insieme di **sottoprogrammi** direttamente interfacciati. Una caratteristica dei singoli sottoprogrammi é il fatto che essi possono essere eseguiti solamente sotto richiesta di un altro programma (quello principale o un altro sottoprogramma), detto **chiamante**, che ha il compito di invocarlo.

Un sottoprogramma si caratterizza da:

- l'elaborazione che effettua (cosa fa?)
- i dati in ingresso
- i dati in uscita

L'esecuzione del sottoprogramma avviene mediante una specifica istruzione detta "**chiamata**" la quale invoca il nome del sottoprogramma desiderato. Il chiamante dovrá trasferirgli i dati necessari all'elaborazione, eseguire la chiamata, aspettare il termine dell'elaborazione ed infine gestire il trasferimento dei risultati prodotti.

Il programma chiamante puó quindi trasferire al sottoprogramma alcune informazioni e viceversa, il sottoprogramma, puó restituire al chiamante alcune informazione derivanti dalla sua eleborazione andando a definire un f**lusso di dati** tra chiamante e sottoprogramma.



In particolare il chiamante, tramite la chiamata, mette a disposizione al sottoprogramma i cosiddetti **parametri effettivi** (variabili in ingresso), i quali devono essere messi in corrispondenza (biunivoca) con i **parametri formali** del sottoprogramma i quali conterrano i valori dei parametri effettivi che il chiamante passa.

Ogni parametro formale (sottoprogramma) conterrá il valore del corrispettivo parametro effettivo (chiamante).

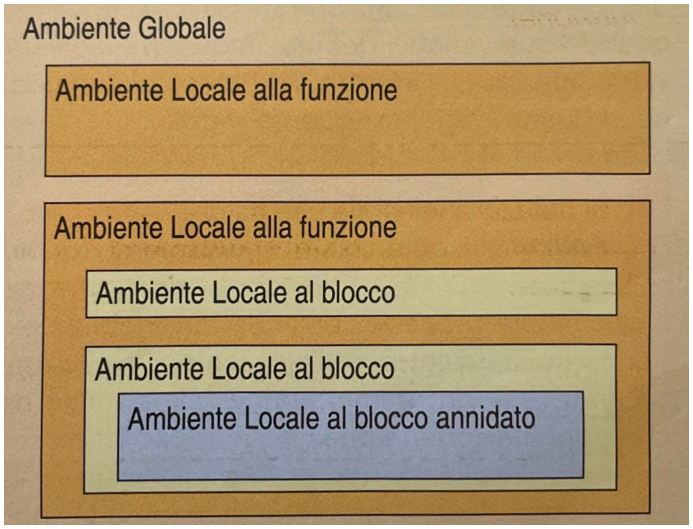
La modularitá é garantita anche dal fatto che il chiamante non ha la necessità di conoscere la struttura interna del sottoprogramma (il chiamante ignora in che modo sarà effettuata l'elaborazione).

Ambienti e visibilitá (scope)

Con ambiente di esecuzione si intende l'insieme di tutte le associazioni tra identificatori e posizioni di memoria (l'insieme delle variabili e costanti presenti in memoria al momento dell'esecuzione).

Si possono distinguere sostanzialmente 2 tipologie di ambienti:

- l'**ambiente locale** é l'insieme di variabili e costanti che sono dichiarate e visibili solamente a un determinato sottoprogramma... pertanto non possono essere utilizzate all'interno di altri sottoprogrammi. In base al "modulo" possiamo parlare di: "ambiente locale del chiamante", "ambiente locale della funzione", "ambiente del blocco". Ció consente di definire variabili con lo stesso identificatore in sottoprogrammi differenti in quanto, effettivamente esse saranno 2 variabili distinte e separate (in quanto si tratta di 2 ambienti diversi).
- l'**ambiente globale** invece comprende le variabili e le costanti che sono utilizzabili (visibili) da qualunque sottoprogramma. Le variabili globali sono definite al di fuori del $main$ e bisogna limitarne l'utilizzo per una serie di problematiche (rendono i sottoprogrammi dipendenti dal chiamante, funzioni diverse possono effettuare uno scambio di dati che non é visibile, ...)



Funzioni e Procedure

Esistono 2 tipologie di sottoprogrammi (anche se in alcuni linguaggi, come C, le procedure non sono altro che particolari funzioni con il tipo restituito “vuoto”):

```
tipo_restituito nome_funzione (lista_parametri_formali){
   Dichiarazioni
   Istruzioni
}
```

- **funzioni**: sulla base dei valori in ingresso calcola un unico valore, detto **valore di uscita**;

```
int massimo (int x);
```

- **procedure**: non prevede un valore di ritorno (void);

```
void potenze(int max);
```

L’ultima istruzione di un sottoprogramma é l’istruzione “**return**”, la quale consente di far terminare l’esecuzione del sottoprogramma e qualora presente, di restituire al chiamante il valore calcolato.

```
//Scrivere una funzione che restituisca il risultato (N1)^(N2)
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

float potenza (float N1, float N2);

int main() {
   float N1, N2, R;
   printf ("Inserire valore base ed esponente: \t");
   scanf ("%f %f", &N1,&N2);
   R = potenza (N1, N2);
   printf ("\n Il valore di %.2f ^ %.2f e\' %f", N1, N2, R);
   return 0;

}

float potenza (float N1, float N2){
   double R;
   if (N2==0)
      R = 1;
   else if (N2<0)
      R= pow (1/N1, N2);
   else
      R = pow (N1,N2);
```

```
    return R;
}
```

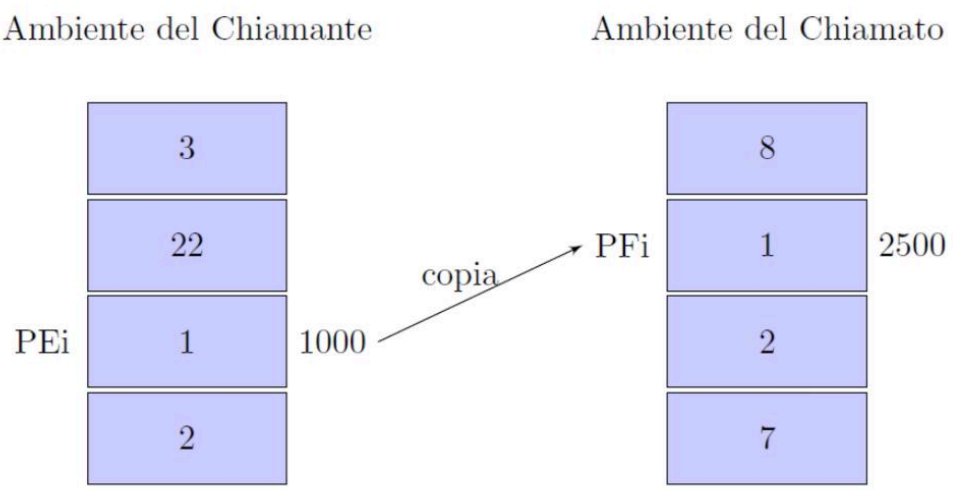
L'associazione dei parametri

Tramite la chiamata a funzione si crea **un'associazione** fra i parametri effettivi e quelli formali. Tale associazione avviene per **ordine** pertanto il numero dei parametri deve essere uguale; il parametro effettivo i-esimo é associato all'i-mo parametro formale.

L'associazione puó avvenire in 2 modi:

- Valore (by value)

  il valore effettivo viene **copiato** nel parametro formale. Quindi le modifiche sul parametro formale (sottoprogramma) **non intaccano** anche il parametro effettivo (chiamante).



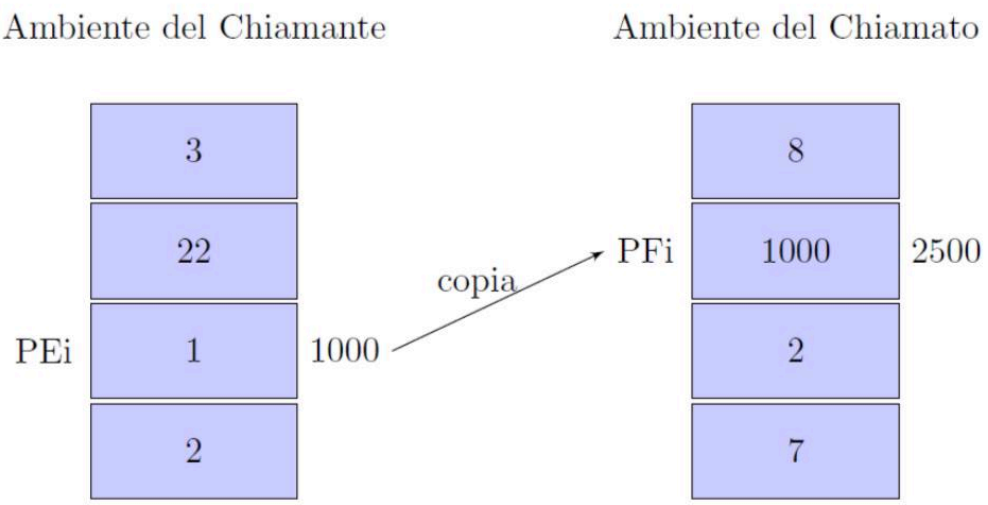
- Riferimento (by reference)

  al parametro formale viene assegnato il riferimento (**indirizzo** di memoria) del parametro effettivo. In questo caso le modifiche che vengono effettuate al parametro formale **avranno effetto** anche sul parametro effettivo corrispondente. Il passaggio per riferimento consente di far ritornare, seppure in maniera indiretta, piú di un parametro al chiamante... se si utilizza il passaggio per riferimento con una procedura, é possibile avere un parametro di ritorno (indiretto) anche con essa.

  Ideologicamente si puó pensare che, considerando x e y, rispettivamente un parametro effettivo e uno formale:

  $x = 5 \qquad LV(x) = 1010 \qquad RV(x) = 5$

  $*y = 5 \qquad LV(y) = 1111 \qquad RV(y) = LV(x)$



Design Pattern

Si tratta di **schemi risolutivi** a problemi giá esistenti che consentono di non reinventare da capo soluzioni che potrebbero non funzionare correttamente in specifiche condizioni.

Tali schemi consentono di ridurre i tempi di progettazione ottenendo contemporaneamente soluzioni stabili ed efficaci; oltre al fatto che, un codice scritto con design patter é piú semplice da leggere e comprendere e quindi piú facile da modificare (manutenibilitá).

Un **design pattern** puó essere definito *"una soluzione progettuale generale a un problema ricorrente"*. Progettare un design pattern significa definire un algoritmo abbastanza generale da poter essere utilizzato per tutti i "problemi simili".

In base al comportamento si classificano in:

- **pattern senza memorizzazione**
  - Generazione di sequenza
  - Visita di sequenza
    - Accumulazione su sequenza
    - Selezione elemento sequenza
- **pattern con memorizzazione (array)**
  - Lettura e memorizzazione
  - Visita sequenza
    - Accumulazione su sequenza
    - Selezione elemento sequenza

Con **sequenza** (S) si definisce un insieme di elementi che presentano una relazione d'ordine parziale: ogni sequenza presenta un elemento iniziale e, da ogni suo elemento (tranne l'ultimo), é possibile raggiungere il successivo: $S = x_1, x_2, ..., x_n$.

Esiste una relazione tra sequenze e vettori dal momento che quest'ultimi sono delle collezioni indicizzate di elementi dello stesso tipo.

Distinguiamo:

- sequenze con terminatore: la terminazione avviene mediante l'inserimento di un elemento fittizio, non appartenente alla sequenza stessa, detto **tappo**. Questo perché non si conosce a priori la lunghezza della sequenza.
- sequenze di lunghezza nota: la lunghezza é esplicitamente codificata;

Generazione di sequenze (senza memorizzazione)

La **generazione di sequenze** avviene mediante una funzione generatrice g che ha l'obiettivo di creare per passi successivi tutti gli elementi della sequenza stessa. In particolare, da un valore iniziale detto **seme**, la funzione generatrice ottiene il valore del secondo elemento $x_2 = g(x_1)$ e cosí via. La funzione generatrice va specializzate in base al problema che si vuole risolvere.

ESEMPI

La generatrice genera ogni elemento sulla base di quello precedente

$$x_i = g(x_{i-1})$$

La generatrice dipende dagli ultimi k valori.

$$x_i = g(x_{i-1}, x_{i-2}, ..., x_{i-k})$$

La generatrice dipende da tutti i valori precedentemente generati:

$$x_i = g(x_{i-1}, x_{i-2}, ..., x_1)$$

Lettura e memorizzazione (senza memorizzazione)

Il pattern consente di riempire un vettore di lunghezza $riemp$ (o di lunghezza non nota con l'ausilio di un "tappo"), con i valori forniti in input dall'utente.

Visita di una sequenza (con e senza memorizzazione)

La **visita a una sequenza** é un'operazione che consiste nel raggiungere (scorrere) tutti gli elementi della sequenza in un ordine prefissato. Nel caso del pattern con **memorizzazione** la sequenza é memorizzata all'interno di un array... il pattern **senza memorizzazione** invece non prevede il salvataggio della sequenza (non usa array). Il funzionamento é il medesimo.

Sono particolari tipologie di visite **l'accumulazione** su **sequenza e la selezione di un elemento** in una sequenza.

Accumulazione su sequenza (con e senza memorizzazione)

Se la visita si limita a raggiungere tutti gli elementi della sequenza, l'**accumulazione** ci permette:

- calcolare il numero di elementi (cardinalitá $nacc$) che soddisfano una certa proprietà (es. il numero degli elementi positivi della sequenza);
- totalizzare i valori degli elementi che soddisfano una certa proprietá (es. somma algebrica degli elementi);
- tenere conto della posizione dell'elemento nella sequenza (es. somma algebrica degli elementi di posizione pari).

É necessario memorizzare quante volte si effettua l'accumulazione dal momento che non é detto che venga effettuata un'accumulazione ad ogni iterazione.

Il pattern con **memorizzazione** consente anche di accedere al valore dell'elemento corrente $curr$.

Selezione su sequenza (con e senza memorizzazione)

La **selezione** invece consente di **individuare** un elemento all'interno di una sequenza effettuando una serie di controlli $(ConSel)$ iterativi sul valore corrente.

Nel momento in cui il valore viene trovato, si assegna il valore $curr$ ad una variabile sel che conterrá quindi il valore selezionato. É possibile anche memorizzare il posto occupato dall'elemento in una variabile pos. Si puó introdurre una variabile booleana $flag$ per memorizzare se la selezione é stata effettuata almeno una volta.

Nel caso di **pattern con memorizzazione** non é necessario salvare la posizione dell'elemento $curr$ in quanto, utilizzando array, si puó sfruttare l'indice del vettore.

Prontuario Pattern