Appunti di informatica

Generali

- Vedremo come passare dalla specifica di un problema alla sua soluzione automatica attraverso l'uso di un algoritmo eseguibile da un calcolatore. La specifica è una descrizione semi-formale del problema
- Qualità:
  - Correttezza: risolve il problema e prende in considerazione tutti i casi possibili
  - Efficienza: usa con parsimonia le risorse (es. tempo)
- La correttezza è fondamentale ma difficile da verificare, l'efficienza è desiderabile e facile da misurare

Codifica binaria

- Per convertire da base 2 a base 10 (qui con m = 5):
  - $11011_2 = (1 \cdot 24+1 \cdot 23+0 \cdot 22+1 \cdot 21+1 \cdot 20)_{10} = 27_{10}$
- Posso rappresentare i numeri nell'intervallo discreto: $[0 , 2^{m-1}]$
- Per convertire da base 10 a base 2: metodo delle divisioni successive
  - $13_{10} = 1101_2$ —> 13/2=6 resto=1—> 6/2 =3 resto=0 —> 3/2 =1 resto=1—> 1/2 =0 resto=1
  - leggere i resti dal basso vs l'alto: 1101
- In caso di numero con segno, bisogna codificare anche il segno, per il quale si usa un bit di memoria: 1=numero negativo, 0=numero positivo. Esempio con m=3:
  - $-3_{10} = 111_2$ (con segno)
  - $+3_{10} = 011_2$ (con segno)
- Di solito nei computer si usa la **codifica CPL2**
  - con CPL2 si possono rappresentare i numeri nell'intervallo $[-2^{m-1}, 2^{m-1}-1]$
  - asimmetria tra negativi e positivi

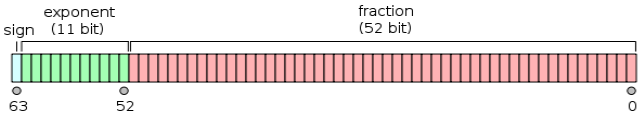
- Esempio (m=8): [-128, +127], perché $-2^7 = -128$ e $2^7 - 1 = +127$
- Tutti i numeri negativi cominciano con il bit più significativo posto a “1”, mentre tutti i positivi e lo zero iniziano con uno “0”

- Codifica di –N da base 10 a complemento a 2
  - Rappresentare $2^m – N$
  - Rappresento N, complemento tutti i bit e sommo 1
- Somma: come per i naturali
- Sottrazione: $N_1 - N_2 = N_1 + (-N_2)_{CPL2}$
- Carry: Il carry non viene considerato!
- Overflow: Se, sommando due interi di m bit dotati di segno concorde, ottengo un risultato di segno discorde (sempre considerando m bit), allora si ha un overflow (il risultato non è codificabile su m bit) e l’operazione è errata
  - L’overflow non può verificarsi se gli operandi sono di segno discorde

- Parte frazionaria di un numero
  - In base due, un numero frazionario N, composto da n cifre, si esprime come:
  - $N_2 = a_{-1} \cdot 2^{-1} + a_{-2} \cdot 2^{-2} + \ldots + a_{-n} \cdot 2^{-n} = \sum(\text{per i che va da -n a -1}) a_i \cdot 2^i$
  - Esempio con n=3: $0{,}101_2 = (1\cdot 2^{-1} + 0\cdot 2^{-2} + 1\cdot 2^{-3})10 = 0{,}625_{10}$
  - Date n cifre in base p=2, posso rappresentare numeri nell’intervallo continuo: [0 , 1-2-n]
  - L’errore di approssimazione sarà minore di 2-n
- **Virgola fissa**
  - Uso m bit e n bit per parte intera e frazionaria
  - Esempio (m=8, n=6, tot. 14 bit): $-123{,}21_{10}$
  - $-123_{10} = 10000101_2$
  - $0{,}21_{10} \approx 001101_2$
  - $-123{,}21_{10} \approx 10000101{,}001101_2$

- **Virgola mobile** (floating point)
  - Il numero è espresso come: r = m·bn
    - m e n sono in base p
    - m: mantissa (numero frazionario con segno)
    - b: base della notazione esponenziale (numero naturale)
    - n: caratteristica (numero intero)
    - Esempio (p=10, b=10): -331,6875 = -0,3316875 • 10^3 dove m = -0,3316875; n = 3

- **Standard IEEE 754-1985**



  - Il numero è espresso come: [S]M·2^n
  - 1 bit per il segno S
  - Mantissa M normalizzata tra 1.0000.. e 1.11111...
  - La parte intera (sempre 1) della mantissa viene omessa
  - L'esponente viene memorizzato in eccesso K
  - E=n+K
  - K = 2^{m-1} -1 (se m=8 K=127)

Campo	Precisione singola	Precisione doppia	Precisione quadrupla
Ampiezza totale	32	64	128
S	1	1	1
E	8	11	15
M	23	52	112
K	127	1023	16383

Valori speciali e segni

Campo	0	NaN	+	-
S	0/1	0/1	0	1
E	0	$2^{\|E\|-1}$	$2^{\|E\|-1}$	$2^{\|E\|-1}$
M	0	≠0	0	0

- Dove |E| è il numero di bit usati per rappresentare l’esponente.

- **Codifica dei caratteri**
  - ASCII (American Standard Code for Information Interchange) utilizza 7 bit (estesa a 8 bit)
  - L’ASCII codifica:
    - I caratteri alfanumerici (lettere maiuscole e minuscole e numeri), compreso lo spazio
    - I simboli (punteggiatura, @, #, ...)
    - Alcuni caratteri di controllo che non rappresentano simboli visualizzabili (TAB,LINEFEED, RETURN, BELL, ecc)

ASCII TABLE

Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	[NULL]	32	20	[SPACE]	64	40	@	96	60	`
1	1	[START OF HEADING]	33	21	!	65	41	A	97	61	a
2	2	[START OF TEXT]	34	22	"	66	42	B	98	62	b
3	3	[END OF TEXT]	35	23	#	67	43	C	99	63	c
4	4	[END OF TRANSMISSION]	36	24	$	68	44	D	100	64	d
5	5	[ENQUIRY]	37	25	%	69	45	E	101	65	e
6	6	[ACKNOWLEDGE]	38	26	&	70	46	F	102	66	f
7	7	[BELL]	39	27	'	71	47	G	103	67	g
8	8	[BACKSPACE]	40	28	(	72	48	H	104	68	h
9	9	[HORIZONTAL TAB]	41	29	)	73	49	I	105	69	i
10	A	[LINE FEED]	42	2A	*	74	4A	J	106	6A	j
11	B	[VERTICAL TAB]	43	2B	+	75	4B	K	107	6B	k
12	C	[FORM FEED]	44	2C	,	76	4C	L	108	6C	l
13	D	[CARRIAGE RETURN]	45	2D	-	77	4D	M	109	6D	m
14	E	[SHIFT OUT]	46	2E	.	78	4E	N	110	6E	n
15	F	[SHIFT IN]	47	2F	/	79	4F	O	111	6F	o
16	10	[DATA LINK ESCAPE]	48	30	0	80	50	P	112	70	p
17	11	[DEVICE CONTROL 1]	49	31	1	81	51	Q	113	71	q
18	12	[DEVICE CONTROL 2]	50	32	2	82	52	R	114	72	r
19	13	[DEVICE CONTROL 3]	51	33	3	83	53	S	115	73	s
20	14	[DEVICE CONTROL 4]	52	34	4	84	54	T	116	74	t
21	15	[NEGATIVE ACKNOWLEDGE]	53	35	5	85	55	U	117	75	u
22	16	[SYNCHRONOUS IDLE]	54	36	6	86	56	V	118	76	v
23	17	[ENG OF TRANS. BLOCK]	55	37	7	87	57	W	119	77	w
24	18	[CANCEL]	56	38	8	88	58	X	120	78	x
25	19	[END OF MEDIUM]	57	39	9	89	59	Y	121	79	y
26	1A	[SUBSTITUTE]	58	3A	:	90	5A	Z	122	7A	z
27	1B	[ESCAPE]	59	3B	;	91	5B	[	123	7B	{
28	1C	[FILE SEPARATOR]	60	3C	<	92	5C	\	124	7C	|
29	1D	[GROUP SEPARATOR]	61	3D	=	93	5D	]	125	7D	}
30	1E	[RECORD SEPARATOR]	62	3E	>	94	5E	^	126	7E	~
31	1F	[UNIT SEPARATOR]	63	3F	?	95	5F	_	127	7F	[DEL]

I linguaggi di programmazione

- Livelli di astrazione dei linguaggi (vicinanza al linguaggio umano):
  - Pseudo-codice: alto
  - C, C++, Java: medio
  - Assembly, Linguaggio Macchina: basso
- Sintassi e semantica
  - **La sintassi definisce come si scrive il programma** (forma e struttura)
    - Esempio: <variabile> = <espressione>
  - **La semantica definisce come si interpretano le istruzioni** contenute nel programma (significato)
    - Esempio: <variabile> = <espressione> => “calcola il valore dell’espressione e assegna al contenuto della variabile il valore calcolato”
  - Un programma sintatticamente corretto non è necessariamente corretto
  - La sintassi può essere verificata automaticamente dal compilatore, la semantica no!
- Compilare e interpretare
  - Utilizzando un **compilatore**, un programma traduce i programmi di alto livello in codice macchina
  - Utilizzando un **interprete**, un programma interpreta direttamente le istruzioni di alto livello e le esegue
  - Entrambi gli approcci offrono vantaggi e svantaggi
    - Portabilità
    - Efficienza
    - Controlli
    - Semplicità
  - Il C è un linguaggio compilato
  - Passaggi: Scrittura—>Traduzione—>Collegamento—>Caricamento—>Esecuzione

- Scrittura
  - Il programma, costituito da una sequenza di caratteri, viene composto e modificato usando un qualsiasi editor
  - Così otteniamo un codice sorgente memorizzato in memoria di massa in un file di testo (es. XYZ.c)
- Traduzione
  - Il compilatore si occupa della traduzione dal linguaggio di alto livello al linguaggio macchina
  - Durante questa fase si riconoscono i simboli, le parole e i costrutti del linguaggio: Eventuali messaggi diagnostici segnalano errori di sintassi
  - Viene generato il codice macchina in forma binaria: a partire dal codice sorgente si genera il codice oggetto, cioè in un file binario
- Collegamento (linking)
  - Il collegatore (linker) deve collegare fra loro il file oggetto ed altre librerie utilizzate (es. librerie di I/O)
  - Si rendono globalmente coerenti i riferimenti agli indirizzi dei vari elementi collegati
  - Si genera un programma eseguibile, un file binario che contiene il codice macchina del programma eseguibile completo, di nome XYZ.exe
  - Messaggi di errore possono essere dovuti ad errori nel citare i nomi delle funzionalità di librerie esterne da collegare
  - Il programma sarà effettivamente eseguibile solo dopo che il contenuto del file sarà stato caricato nella memoria di lavoro (centrale) del calcolatore
- Caricamento (loading)
  - Il caricatore (loader) individua una porzione libera della memoria di lavoro e vi copia il contenuto del programma eseguibile
  - Eventuali messaggi rivolti all’utente possono segnalare che non c'è abbastanza spazio in memoria
- Esecuzione

- Per eseguire il programma occorre fornire in ingresso i dati richiesti e in uscita riceveremo i risultati (su video o file o stampante)
- Durante l'esecuzione possono verificarsi degli errori (detti "errori di run-time"), quali:
  - calcoli con risultati scorretti (per esempio un overflow)
  - calcoli impossibili (divisioni per zero, logaritmo di un numero negativo, radice quadrata di un numero negativo,....)
  - errori nella concezione dell'algoritmo (l'algoritmo non risolve il problema dato)
- Tutti gli esempi citati si riferiscono ai cosiddetti errori semantici

---

C: basi

- Struttura di base (da copiare e incollare sempre)

```
**********************************************************************************************
#include <stdio.h>
int main()
{
	//codicecodicecodice
	return 0;
}
**********************************************************************************************
```

- Struttura di un programma (versione descrittiva)

```
**********************************************************************************************
Inclusione di librerie
Dichiarazione di variabili globali (che valgono sia nel main che in tutte le funzioni)
int main()
{
```

```
    dichiarazione di variabili locali
    istruzione1;
    istruzione2;
    . . .
    return 0;
}
```

****

- **Librerie**
  - `stdio.h` —> essenziale, standard input output
  - `stdlib.h` —> per la funzione rand(), abs() e la costante NULL
  - `string.h` —> per le funzioni che lavorano con le stringhe
  - `math.h`—> per le operazioni matematiche in virgola mobile
  - `time.h` —> per la funzione time()
- **Input/Output**
  - `scanf("%d", &n);` —> legge da input una variabile
  - `printf("hello %c%c", a, b);` —> stampa in output testo e/o variabili
    - aggiungendo un * (es. “%*c”) verrà letta anche quella variabile e subito cancellata dalla memoria del computer. Da usare quando si vuole leggere dei caratteri subito dopo aver letto dei numeri, per evitare sbagli di lettura da parte della funzione scanf (scrivere quindi “%d%*c” per leggere un numero ed interpretare l’andata a capo come carattere da buttare)
  - possibili **variabili** da leggere/stampare con “%x”:
    - d —> int
    - ld —> long int
    - u —> unsigned int
    - lu —> unsigned long int
    - c —> char

- s —> string (char pointer)
- f —> float (numeri con virgola, singola precisione)
- lf —> double (numeri con virgola, doppia precisione)

- si possono stampare come argomenti numeri, espressioni aritmetiche (es. printf(3*4) stampa “12”) e variabili (es. printf(“%d”, num))
- Valori speciali:
  - \n —> newline; andata a capo
  - \t —> tabulazione; utile per stampare tabelle
  - \b —> backspace

- **Assegnazione**
  - a = b; —> a assume il valore di b
  - area = 3 * funzione(a, b); —> si possono assegnare operazioni matematiche e funzioni
  - a++; —> aggiunge 1 al valore di a
  - a--; —> toglie 1 al valore di a
  - a += 3; —> aggiunge 3 al valore di a (si può fare anche con -, * e /)
  - f = 1.045;
  - f = 4.5567e3 —> equivalente di 4556.7
  - Attenzione: se si usano variabili di tipo float, indicare numeri interi come xxx**.0,** anche nelle operazioni (es. 5.0/9.0)
  - c = ‘a’; —> i caratteri si mettono tra apici
  - c = ‘\n’ —> vale; manda a capo
  - c = 20; —> assegna il carattere numero 20 secondo la tabella ASCII; si possono anche fare operazioni matematiche, che alla fine verrano convertite in un carattere

- **Variabili**
  - Oltre a quelle scritte sopra, si possono aggiungere delle precisazioni per l’utilizzo di memoria:

- short (int): numeri di piccole dimensioni
- long (int): numeri di grandi dimensioni
- unsigned (int): numeri senza segno, positivi. Permettono di rappresentare numeri più grandi con la stessa memoria

- sono combinabili (es. unsigned long int x;)

Definizioni particolari delle variabili

- **enum**
  - elencazione di valori simbolici che la variabile può avere; il computer lo vedrà come un valore numerico da 0 a N, serve solo al programmatore per facilitare il filo del pensiero
    - ```
      enum {falso, vero} condizione, condizione2;

      condizione = falso;      // il computer si segna 0

      condizione2 = vero;      // il computer si segna 1
      ```

- **typedef**
  - consente di ridefinire un tipo preesistente con un nuovo nome; anche qui serve solo al programmatore
    - ```
      typedef int colore;

      colore coloreMacchina;

      coloreMacchina = 5;
      ```
  - si possono combinare typedef ed enum per organizzare la proprie variabili
    - ```
      typedef enum {lun, mar, mer, gio, ven, sab, dom} giorno;

      giorno oggi;

      oggi = ven;
      ```

- **costanti**
  - due modi per definirle
    - ```
      const int x = 5;
      ```

- `#define VERO 1` (prima del main, per convenzione i nomi sono ALL-CAPS e separati da underscore _ dove andrebbero gli spazi)

- cambio di tipo: calcoli tra numeri di tipo diverso
  - se gli operandi sono di tipo diverso, il tipo inferiore viene convertito temporaneamente ad un tipo superiore per garantire che non i siano perdite di informazione, secondo l'ordine (per memoria occupata):
  - int < long < unsigned long < float < double

Strutture di controllo

- **if**
  - ```
    if (anni>65){
            printf("biglietto ridotto\n");
      }
    ```
  - semantica: le istruzioni dentro le graffe (opzionali se c'è solo una riga di istruzioni) vengono eseguite solo le condizioni sono vere
  - nel C non esiste il tipo booleano (binario: 0 o 1); se si inserisce nelle tonde qualunque valore diverso da 0, la condizione sarà interpretata come vera in ogni caso
    - `if (3)` —> vero
  - Operatori relazionali: >, <, >=, <=, == ('uguale a': mai usare un solo '='), != ('diverso da'). Si possono usare le parentesi tonde per effettuare diverse operazioni.
  - Operatori logici: && (and), || (or), ! (not)

A	B	A AND B	A OR B	NOT A
0	0	0	0	1
0	1	0	1	
1	0	0	1	0
1	1	1	1	

  - (ci sarebbe anche lo XOR, che in caso di A=B=1 darebbe 0, ma non serve)

- **if else**
    ```

- consente di scegliere tra due alternative nel flusso di esecuzione
- ```
  if (anni>18)
        printf("puoi votare");
  else
        printf("non puoi votare");
  ```
- semantica: il primo 'statement' (set di istruzioni) viene eseguito solo se le condizioni sono vere, il secondo solo se sono false

- **switch**
  - per scegliere tra molte alternative: un if—> else if—> else if… non è molto leggibile
  - ```
    switch (peso_moneta)
    {
        case 9:
        printf("5 centesimi\n");
        break;
        case 16:
        printf("5 centesimi\n");
        break;
        default: printf("moneta non riconosciuta"); break;
    }
    ```
  - il singolo caso si può anche mettere su una riga sola, se l'istruzione è singola (vedi esempio con 'default')
  - se più casi corrispondono alle stesse istruzioni da eseguire si possono scrivere tutti i casi diversi, le istruzioni in fondo e un singolo break alla fine di queste istruzioni, per non copiare ogni volta
  - il caso di default viene usato quando nessuno dei case è valido per lo switch
- **for**
  - ciclo di istruzioni da ripetere <u>un numero di volte già conosciuto</u>
  - ```
    for (int i=0; i<5; i++){
    ```
    ```

```
        //istruzioni
    }
```

- la variabile contatore può anche essere dichiarata insieme alle altre all'inizio del main, nel qual caso la prima parte è solo i=0

- **while**
  - ciclo da ripetere finché una data condizione rimane vera
  - ```
    while (i<5){
        //roba varia
        i++;
        }
    ```

- **do while**
  - praticamente uguale al while, ma garantisce che le istruzioni all'interno delle graffe vengano eseguite almeno la prima volta
  - ```
    do {
        //roba ancora più varia
        }while(i<5);
    ```

- **break e continue**
  - l'istruzione break all'interno di un ciclo lo interrompe immediatamente
  - l'istruzione continue passa direttamente all'iterazione seguente
  - **!** non so perché, ma quando ho provato ad usare un break è morto il codice, il computer e il mio gatto. Il tipo del laboratorio ha detto di evitare i break e i continue come la peste, poi fai come vuoi.

- **Array**
  - rappresentazione compatta di una collezione di variabili dello stesso tipo
  - ```
    int array[5];
    for(int i=0; i<5; i++){
      printf("%d", array[i]);
    }
    ```
  - vettore di cinque elementi con posizione che va da 0 a 4 (n-1) e sua stampa
  - ogni singolo elemento è del tutto analogo ad una variabile semplice
  - si può inizializzare un array in fase di dichiarazione, inizializzando ogni singolo elemento con un valore oppure **tutti** con un valore unico (ma non solo alcuni)
  - `float prezzo[3]={13.4, 11.1, 20.0, 30.4};`
  - `int macheneso[5]={0};  //tutti gli elementi = 0`
  - lettura e scrittura avvengono un elemento alla volta, mai tutti insieme. Servono dei cicli (di solito for)
  - il range di un vettore è lo spazio che occupa nella memoria. Se il programmatore cerca di occupare uno spazio al di fuori del vettore il computer non se ne accorge, e il comportamento del codice diventa impredicibile.
  - la copia tra due array va fatta anch'essa elemento per elemento, non per semplice assegnamento. Dati a e b vettori, a=b è un errore di sintassi.
  - allo stesso modo, non si possono confrontare due vettori con gli operatori == o !=, ma è necessario confrontare con un ciclo ogni elemento
  - è possibile usare gli array come parametri di una funzione in C, ma servono i puntatori
- **Stringhe**
  - Stringa = array di tipo char
  - `char nome[5];`
  - essendo molto usati, il C ha già molte funzioni nella libreria string.h per il loro uso più facilitato
    ```

- lettura e scrittura:
  - `scanf("%s", nome);`
  - `printf("ciao %s", nome);`
    - per la lettura delle stringhe non si deve aggiungere il simbolo & prima del nome della stringa
    - usando "%s" nella lettura di una stringa, questa verrà letta solo fino al primo spazio bianco. Per leggerla per intero quando include spazi si usa
  - `scanf("%[^\n]", nome);`
- il **carattere \0** indica la fine di una stringa. Quando la funzione printf lo individua, smette di stampare la stringa e passa all'istruzione successiva. Questo perché una stringa non ha mai elementi vuoti, ma viene riempita di caratteri "garbage" (a caso) se una sua parte non viene utilizzata.
- In genere la funzione scanf provvede automaticamente ad aggiungere alla fine il carattere di terminazione \0
- **copia e confronto** di stringhe: hanno le stesse limitazioni degli array, però la libreria *string.h* contiene delle funzioni apposta per il loro confronto
  - `strcpy(s1, s2);` —> copia il contenuto della stringa s2 nella stringa s1
  - `strcmp(s1, s2);` —> ritorna 0 solo se la stringa s1 è uguale alla stringa s2
- è possibile usare le stringhe come parametri di una funzione in C, ma come per gli array bisogna usare i puntatori

- **Matrici**
  - Strutture dati bidimensionali: array di array
  - `float m[3][3];`
    `m[1][2]=25.0;`

	m[x][0]	m[x][1]	m[x][2]
m[0][x]			
m[1][x]			25.0
m[2][x]			

  - lettura: come per gli array, richiede l'uso dei cicli per leggere ogni elemento singolarmente
  - `for(int i=0; i<3; i++){`
    `for(int j=0; j<3; j++){`

```
        scanf("%f", &m[i][j]);
        }
    }
```

- lo stesso procedimento si usa con la stampa (occhio alla tabulazione), le operazioni fra matrici come la somma e la copia di una matrice in un'altra.
- analogamente agli array, non si può confrontare una matrice con un'altra attraverso gli operatori == e !=, ma bisogna controllare un elemento per volta

- **struct**
  - rappresentazione compatta ed incapsulata di tipi di dati con una struttura complessa
  -
```
struct
    {
    int pagine;
    char titolo[20];
    }libro1, libro2;      //due struct di quel tipo, libro1 e 2
```
  -
```
typedef struct
    {
    int pagine;
    char titolo[20];
    }libro;
    . . .
    libro l;             //struttura chiamata libro con typedef
```
  -
```
struct punto
    {
    float x;
    float y;
    };                  //struttura chiamata punto senza typedef
```
  - per accedere ad un campo di una struct: `libro1.pagine = 394;`
  - la lettura avviene come per gli array, un elemento alla volta:

```
scanf("%d", &l.pagine);
```

- è spesso utile dichiarare un array di struct, per gestire una sequenza di elementi ognuno dei quali rappresenta un dato strutturato: `libro biblioteca[20];`
- il C permette di effettuare un assegnamento fra struct senza considerare un elemento alla volta: `libro1 = libro2;`
- invece non si può fare il confronto di una struct direttamente, ma bisogna analizzare ogni campo separatamente
- le struct possono essere usate nelle funzioni senza accorgimenti (o puntatori)

- **puntatori**
  - tipo di dato usato per dichiarare una variabile che deve contenere un indirizzo di una cella di memoria. È necessario dichiarare il tipo di dato contenuto nelle celle di memoria puntate dal puntatore.
  - `int *p = NULL;`

    ```
    int var=3;
    p = &var;
    ```

  - **& = "indirizzo di"**. Restituisce l'indirizzo di memoria di una variabile qualunque.
  - *** = "contenuto di"**. Tranne per la dichiarazione dei puntatori, indica il contenuto della variabile seguente. Se questa è un puntatore, indica la cella di memoria nella quale si trova la variabile puntata dal puntatore.
  - si può assegnare il valore di un puntatore ad un altro: `p2 = p1;`
  - si può assegnare ad una variabile il valore di un'altra variabile attraverso i loro puntatori: `*p2 = *p1;`
  - si possono eseguire operazioni matematiche (+ e -) con i puntatori:
    -
    ```
    float *p;
    float a;
    p=&a;
    *(p+1)=4.5;
    p=p+2;
    ```

- semantica: l'ultimo passaggio ha incrementato l'indirizzo contenuto nel puntatore di x posizioni: la dimensione di una posizione dipende dal tipo del puntatore (4 byte per un float, 1 byte per un char, e così via)
- puntatori a struct: se un puntatore è di tipo struct, allora `(*p).campo` —> `p->campo`
- Un array viene memorizzato come un blocco continuo a partire da un indirizzo di partenza —> indirizzo base
  - il nome della variabile array con posizione non specificata equivale all'indirizzo di partenza di quell'array
- si può accedere ad un array attraverso i puntatori. Se 'a' è una variabile di tipo array
  - `a[i] = *(a+i)`
- per sapere la dimensione di un array in numero di variabili, si può stampare la dimensione dell'array (attraverso sizeof()) e dividerla per la dimensione del tipo di variabile contenuta nell'array (es. sizeof(float))
- in C, i parametri di una funzione possono essere dei puntatori, per modificare la variabile originale anziché la copia fatta nella funzione (vedi dopo)
- anche le matrici vengono memorizzate, come gli array, in un blocco continuo di memoria a partire da un indirizzo base, riga dopo riga.
  - il nome di una matrice senza posizioni specificate è un puntatore come il nome di un array, ma le operazioni aritmetiche sui puntatori della matrice hanno effetti diversi a seconda della sintassi:
  - ```
    float *p1 = &m[0][0];
    float (*p2)[4] = m;
    ```
    - qui, `p1+i` è l'indirizzo dell'i-esimo elemento di m (letta per righe)
    - `p2+i` è l'indirizzo di partenza dell'i-esima riga di m
    - `p1+i` = &m[0][0]+i
    - `p2+i` = m+i
  - in generale, `m[i][j] == *(&m[0][0]+i*c+j)`
    ```

Funzioni

- servono a dare un nome ad una sequenza di istruzioni, per poi poterla richiamare facilmente in un programma.
- dichiarazione del prototipo (*signature*): specifica tipo, nome e parametri della funzione. Informa il compilatore che quella funzione verrà usata nel programma. Va messa prima del main
- definizione: specifica sia il prototipo che la sequenza di istruzioni della funzione. Può essere definita ovunque nel programma, ma di solito si mette dopo il main.

```
**********************************************************************************************

int somma(int addendo1, int addendo2);                    //prototipo

. . .

int main(){

     int a, b;

     scanf("%d%d", &a, &b);

     risultato = somma(a, b);       //richiamare la funzione: qui,
     //    'risultato' assumerà il valore che nella funzione ha
     //    assunto 'tot'

     . . .

}

int somma(int addendo1, int addendo2){                  //definizione

     int tot=0;

     tot=addendo1+addendo2;

     return tot;

}

**********************************************************************************************
```

- se una funzione non deve restituire nulla il suo tipo è 'void'
  - se la funzione non è void, la funzione ritornerà un valore del tipo indicato attraverso il comando 'return' seguito dalla variabile il cui valore va restituito
- se una funzione non ha parametri in ingresso, le parentesi tonde vanno scritte comunque ma lasciate vuote (es. int nome_funzione**()**)

- nell'esempio, 'addendo1' e 2 sono parametri formali: tengono solo il posto alle variabili attuali (a e b)
- le variabili dichiarate nella funzione esistono e sono visibili solo all'interno della funzione stessa; richiamarle nel main causerebbe un errore
  - Questo vale per ogni blocco racchiuso da parentesi graffe; una variabile nel main non può essere letta da una funzione, a meno che quella non sia stata passata come parametro
  - allo stesso modo, una variabile dichiarata in un ciclo for (per esempio) esiste solo all'interno di quel for e viene distrutta una volta usciti dal ciclo
  - le variabili **globali** sono dichiarate fuori dai blocchi e vengono lette e condivise da tutti i blocchi
  - per questo motivo, si possono dichiarare diverse variabili con lo stesso nome, purché queste siano riconosciute da due blocchi diversi (dove uno non è interno all'altro); se un blocco è interno all'altro, la variabile locale oscura quella più globale
  - se una variabile viene dichiarata con la parola chiave 'static' questa mantiene il suo valore tra blocchi, ma è anche difficile da gestire correttamente
- i parametri di una funzione possono essere dei puntatori. Mentre normalmente verrebbe modificata la copia della variabile, usando il puntatore come parametro permette di modificare l'originale
- **sizeof(<arg>)**: funzione che ritorna
  - se <arg> è un tipo di dato (es. int), la quantità di memoria in byte occupata dal tipo
  - se <arg> è una variabile, la quantità di memoria occupata da quella variabile
  - se <arg> è un array, la quantità di memoria occupata dall'intero array
- **bubble sort:** la funzione bubble sort permette di ordinare in ordine crescente un array di n elementi, con n da input. Non è il metodo più rapido, ma è tra i più semplici.
  - si usano due cicli for, uno innestato nell'altro. Il contatore del primo (qui 'i') va da 0 a n, mentre il secondo ('j') va da 0 a n-i
  - nel caso in cui l'elemento i dell'array sia minore dell'elemento j, si effettua un'inversione con variabile temporanea tra gli elementi per scambiarli
  - alla fine dei due cicli for, l'array avrà i numeri ordinati in ordine crescente

  si passa l'indirizzo del vettore per modificare direttamente il vettore originale e non una sua copia; per questo la funzione non ritorna niente

Risorse veloci

Tipi	Rappresentazione	N. di byte[1]	Intervallo
char	Carattere	1 (8 bit)	[0,255]
int	Numero intero	4 (32 bit)	[-2147483648,2147483647]
unsigned int	Numero intero positivo	4 (32 bit)	[0,4294967295]
short int	Numero intero “corto”	4 (32 bit)	[-2147483648,2147483647]
long int	Numero intero “lungo”	4 (32 bit)	[-2147483648,2147483647]
long long int	Numero intero “molto lungo”	8 (64 bit)	[-9.223372e18 9.223372e18]
float	Numero reale	4 (32 bit)	[-3,4e38, +3,4e38] circa[2]
double	Numero reale “doppia precisione”	8 (64 bit)	[-1,8e308, +1,8e308] circa[2]

[1] ATTENZIONE: le dimensioni in byte dei tipi di dato non sono fissate univocamente dagli standard del c e potrebbero quindi variare in base alle impostazioni dei compilatori.

[2] ATTENZIONE: il tipo float e il tipo double non consentono di rappresentare OGNI numero nell’intervallo, ma prevedono un’approssimazione

specifier	**Descrizione**	**Characters extracted**
`i`	Integer	Any number of digits, optionally preceded by a sign (+ or -). Decimal digits assumed by default (0-9). *Signed* argument.
`d` *or* `u`	Decimal integer	Una serie di cifre decimali (0-9), può essere preceduto da segno (+ or -). d è per argomenti con segno, e u è per argomenti senza segno . %6d stampa un intero di 6 cifre ad esempio
`f`	Floating point number	Una serie di cifre decimali, può contenere il punto decimale, può contenere il segno (+ o -) e supporta la notazione esponenziale. %4.3f stampa un numero con 4 cifre prima della virgola e 3 dopo la virgola %.5f stampa un numero con 5 cifre dopo la virgola
`lf`	Double Long float	Qui non saprei cosa scrivere quindi lascio un testo a caso in modo che non disturbi la vista e lasci pensare che c’è qualche contenuto come in tutte le altre celle.
`c`	Character	The next character. If a *width* other than 1 is specified, the function reads exactly *width* characters and stores them in the successive locations of the array passed as argument. No null character is appended at the end.
`s`	String of characters	Qualsiasi numero di caratteri, durante la lettura si ferma al primo spazio A terminating null character is automatically added at the end of the stored sequence.
`p`	Pointer address	A sequence of characters representing a pointer. The particular format used depends on the system and library implementation, but it is the same as the one used to format %p in fprintf.
`%`	`%`	A % followed by another % matches a single %.

conditions	switch-case

x==y *(x uguale y)*
x!=y *(x diverso y)*
x<y *(x minore y)*
x>y *(x maggiore y)*
x<=y *(x minore uguale a y)*
x>=y *(x maggiore uguale a y)*
	switch (var) {
	case 1:
	//istruzioni
	break;
	case 2:
	//istruzioni
	break;
	default:
	//istruzioni
	break;
	}

if…else	**for**

if (condition){
// istruzioni se vero
}
else{
// istruzioni se falso
}
	for(int i=0; i<N; i++)
	{
	//istruzioni
	}

if	**while**

if (condition){
// istruzioni se vero
}
	while(expression){
	//istruzioni
	}

main	**struct**

#include <libreria.h>
int main()
{
//codice
return 0;
}
	struct
	{
	//variabili
	}nomestruct,nomestruct2;

bubble sort	**ricerca dicotomica***

int i,j;
 	
for(i=0; i<100; i++){
for(j=0; j<100-i; j++){
if(array[j]>array[j+1]){
temp=array[j];
array[j]=array[j+1];
array[j+1]=temp;
}
}
}
	m=-1;
	inf=0;
	sup=lunghezza-1;
	while(inf<=sup) {
	m = (inf+sup)/2;
	if(array[m]==x){
	found=m;
	break;}
	if(array[m]<x)
	inf=m+1;
	else
	sup=m-1;
	}

*x è l'elemento cercato, array[] è l'array in cui si cerca (ordinato), inf è l'estremo inferiore, sup è l'estremo superiore, m è la posizione di ricerca corrente. m sarà -1 se l'elemento non è stato trovato, conterrà la posizione dell'elemento se è stato trovato.

me@valerionappi.it

Questo codice è un esempio di applicazione di bubble sort e ricerca dicotomica.

```c
#include<stdio.h>

#define ARRAY_LENGTH 13

int main() {
  int temp,p,u,m,x=9,found=-1; //cerca l'elemento x
  int array[ARRAY_LENGTH]={1,2,9,6,3,4,58,-3,0,10,4,7,3};

  for(int i=0; i<ARRAY_LENGTH; i++){
    for(int j=0; j<ARRAY_LENGTH-i; j++){
      if(array[j]>array[j+1]){
        int temp=array[j];
        array[j]=array[j+1];
        array[j+1]=temp;
      }
    }
  }

  for (int i=0; i<ARRAY_LENGTH; i++) {
    printf("%d ",array[i] );
  }
  inf=0;
  sup=ARRAY_LENGTH-1;


  while(inf<=sup) {
    m = (inf+sup)/2;
    if(array[m]==x){
      found=m;
      break;}
    if(array[m]<x)
      inf=m+1;
    else
      sup=m-1;
   }

   if (found==-1)
      printf("\nL'elemento %d non e' stato trovato", x);

   else
      printf("\nL'elemento trovato e' in posizione %d e si tratta del numero %d: ", found, array[found] );

  return 0;
}
```

ASCII TABLE

Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	*[NULL]*	32	20	*[SPACE]*	64	40	@	96	60	`
1	1	*[START OF HEADING]*	33	21	!	65	41	A	97	61	a
2	2	*[START OF TEXT]*	34	22	"	66	42	B	98	62	b
3	3	*[END OF TEXT]*	35	23	#	67	43	C	99	63	c
4	4	*[END OF TRANSMISSION]*	36	24	$	68	44	D	100	64	d
5	5	*[ENQUIRY]*	37	25	%	69	45	E	101	65	e
6	6	*[ACKNOWLEDGE]*	38	26	&	70	46	F	102	66	f
7	7	*[BELL]*	39	27	'	71	47	G	103	67	g
8	8	*[BACKSPACE]*	40	28	(	72	48	H	104	68	h
9	9	*[HORIZONTAL TAB]*	41	29	)	73	49	I	105	69	i
10	A	*[LINE FEED]*	42	2A	*	74	4A	J	106	6A	j
11	B	*[VERTICAL TAB]*	43	2B	+	75	4B	K	107	6B	k
12	C	*[FORM FEED]*	44	2C	,	76	4C	L	108	6C	l
13	D	*[CARRIAGE RETURN]*	45	2D	-	77	4D	M	109	6D	m
14	E	*[SHIFT OUT]*	46	2E	.	78	4E	N	110	6E	n
15	F	*[SHIFT IN]*	47	2F	/	79	4F	O	111	6F	o
16	10	*[DATA LINK ESCAPE]*	48	30	0	80	50	P	112	70	p
17	11	*[DEVICE CONTROL 1]*	49	31	1	81	51	Q	113	71	q
18	12	*[DEVICE CONTROL 2]*	50	32	2	82	52	R	114	72	r
19	13	*[DEVICE CONTROL 3]*	51	33	3	83	53	S	115	73	s
20	14	*[DEVICE CONTROL 4]*	52	34	4	84	54	T	116	74	t
21	15	*[NEGATIVE ACKNOWLEDGE]*	53	35	5	85	55	U	117	75	u
22	16	*[SYNCHRONOUS IDLE]*	54	36	6	86	56	V	118	76	v
23	17	*[ENG OF TRANS. BLOCK]*	55	37	7	87	57	W	119	77	w
24	18	*[CANCEL]*	56	38	8	88	58	X	120	78	x
25	19	*[END OF MEDIUM]*	57	39	9	89	59	Y	121	79	y
26	1A	*[SUBSTITUTE]*	58	3A	:	90	5A	Z	122	7A	z
27	1B	*[ESCAPE]*	59	3B	;	91	5B	[	123	7B	{
28	1C	*[FILE SEPARATOR]*	60	3C	<	92	5C	\	124	7C	|
29	1D	*[GROUP SEPARATOR]*	61	3D	=	93	5D	]	125	7D	}
30	1E	*[RECORD SEPARATOR]*	62	3E	>	94	5E	^	126	7E	~
31	1F	*[UNIT SEPARATOR]*	63	3F	?	95	5F	_	127	7F	*[DEL]*

me@valerionappi.it