

Introduzione a C

Table of Contents

- [Introduzione a C](#)
 - [Table of Contents](#)
 - [Tipi di dato](#)
 - [Regole di scrittura](#)
 - [Sintassi per dichiarazione delle variabili](#)
 - [Assegnamento variabili](#)
 - [Operatori aritmetici e di casting](#)
 - [Operatori aritmetici](#)
 - [Operatore di casting](#)
 - [Operatori Logici Booleani](#)
 - [Astrazione procedurale](#)
 - [Main function](#)
 - [Scanf](#)
 - [Printf](#)
- [Astrazione sul controllo](#)
 - [If](#)
 - [Caso 1](#)
 - [Caso 2](#)
 - [Caso 3](#)
 - [Loops](#)
 - [For](#)
 - [While](#)
 - [Do-while](#)
 - [Numeri random](#)
 - [Rand](#)
 - [Srand](#)
 - [Vettori](#)
 - [Matrici](#)
- [Puntatori](#)
 - [Dichiarazione di puntatori](#)
 - [Operatore &](#)
 - **[Operatore *](#)**
 - [Esempio](#)
 - [Puntatori con vettori](#)
 - [Stringhe \(Array di caratteri\)](#)
 - [strlen\(\)](#)
 - [gets\(\)](#)
 - [puts\(\)](#)
 - [strcpy\(\)](#)
 - [strncpy\(\)](#)
 - [strcat\(\)](#)

- `strncat()`
- `strcmp()`
- Funzioni
 - Definire e chiamare una funzione
 - Return nelle funzioni
 - Passaggio di parametri
 - Passaggio per valore
 - Passaggio per riferimento
- Operazioni su sequenze
 - Ricerca completa
 - Ricerca Binaria
 - Inserimento all'inizio
 - Inserimento alla fine
 - Inserimento diretto
 - Cancellazione
 - Fusione
- Algoritmi di ordinamento
 - Ordinamento per inserimento diretto
 - Ordinamento per selezione
 - Bubble Sort
- Strutture
 - Definizione di una struttura
 - Inizializzazione di una struttura
 - Struct e puntatori
- Esercizi da svolgere:

Tipi di dato

- `int`: Integer type (%d)
- `short int`: Short integer type (%hd)
- `long`: Long integer type (%ld)
- `long long`: Long long integer type (%lld)
- `float`: Single-precision floating-point type (%f)
- `double`: Double-precision floating-point type (%lf)
- `long double`: Extended-precision floating-point type (%Lf)
- `char`: Character type (%c)

Unsigned Types

- `unsigned int`: Unsigned Integer type (%u)
- `unsigned short`: Unsigned Short integer type (%hu)
- `unsigned long`: Unsigned Long integer type (%lu)

- `unsigned long long`: Unsigned Long long integer type (%llu)

Regole di scrittura

- Indentazione
- Nome significativo delle variabili
- Commenti nel caso fossero necessari
- Evitare l'uso di codici superflui
- Utilizzo graffe
- Salto delle righe tra dichiarazione e corpo del codice

Sintassi per dichiarazione delle variabili

Specificatore tipo (int, float, char) + nome variabile (o lista di variabili separate da una virgola) + ;

```
int x, y;
```

Assegnamento variabili

Specificatore tipo (int, float, char) + nome variabile (o lista di variabili separate da una virgola) +

```
int x = 2;  
int y = 5;  
  
y = x;
```

Operatori aritmetici e di casting

Operatori aritmetici

- `+`: somma;
- `-`: differenza;
- `*`: moltiplicazione;
- `/`: diviso;
- `%`: modulo (ritorna il resto di una divisione).

Operatore di casting

Operatore che mi permette di cambiare il tipo di dato in un altro.

```
#include <stdio.h>  
  
int main() {  
    int a = 5, b = 4;  
    float c;  
  
    c = (float) (a+b)/2;  
    printf("La media è : %f", c );  
}
```

Nonostante sia un'operazione tra interi riusciamo ad ottenere i decimali risultanti da quella operazione.

Operatori Logici Booleani

- **AND** &&
- **OR** ||
- **NOT** !

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

Esiste anche il tipo di dato bool che permette di memorizzare "vero" o "falso" all'interno delle variabili; per poterlo utilizzare è necessario include la libreria `#include <stdbool.h>`.

Astrazione procedurale

Il programma viene suddiviso in funzioni.

Main function

La funzione main è l'unica funzione che viene eseguita autonomamente e la sua presenza è indispensabile per il funzionamento del programma.

```
#include <stdio.h>

int main(void) {

    return 0;
}
```

- void: indica che nella funzione non sono presenti dei parametri
- return 0;: se viene eseguito tutto correttamente la funzione ritorna il valore 0 (il valore è un intero perchè la funzione è di tipo int).

Scanf

Funzione di input, che serve per inserire dei dati, tramite tastiera, all'interno del programma

```
#include <stdio.h>

main() {
    int a = 0;
```

```
scanf("%identificatore", &a);  
}
```

In questo caso sto prendendo in input un valore da tastiera e lo sto assegnando alla variabile `a`, questa cosa non avviene direttamente, ma mediante l'operatore `&` che serve per indicare l'indirizzo della variabile.

Printf

Questa funzione serve a mostrare a video una frase o una variabile, anche in questi casi si ricorre all'utilizzo degli identificatori.

```
#include <stdio.h>  
  
main() {  
    int a = 7;  
  
    printf("La variabile a, ha valore: %d", a);  
}
```

Output del programma:

La variabile a, ha valore: 7

Astrazione sul controllo

If

Il blocco if serve a eseguire un'operazione nel caso venga verificata o meno una condizione. Letteralmente il blocco if significa "se la condizione è vera, fai questo, altrimenti non farlo o fai qualcos'altro".

Caso 1

```
if(condizione){  
    "fai qualcosa";  
}
```

In questo caso il blocco if esegue solo l'operazione solo se la condizione viene verificata.

Caso 2

```
if(condizione){  
    "fai qualcosa";  
}else{  
    "fai altro";  
}
```

In questa situazione se la condizione non viene rispettata viene svolta un'altra operazione.

Caso 3

```
if(condizione){  
    "fai qualcosa";  
}else if(condizione 2){  
    "fai altro";  
}else{  
    "fai altro ancora";  
}
```

In questo caso se la prima condizione non viene verificata, controlla la seconda, se anche questa non viene verificata esegue l'operazione nel blocco `else`.

Loops

Blocchi che permettono di iterare delle operazioni. Esistono 3 differenti loops:

- for
- while
- do-while

For

Il for è il ciclo con contatore.

```
for (int i = 0; i < 100; i++) {  
    "operazione da eseguire fino a quando i >= 100";  
}
```

Il ciclo for è costituito da un header e un body. All'interno dell'header c'è l'inizializzazione della variabile contatore, la condizione di permanenza e l'incremento (o decremento) della variabile. Nel body vengono eseguite le operazioni da ripetere.

While

```
while (condizione) {  
    "fai qualcosa dopo aver controllato la condizione";  
}
```

Il while controlla sempre prima la condizione prima di eseguire l'operazione al suo interno.

Do-while

Il do-while è un loop a condizione finale, questo garantisce che il codice al suo interno venga eseguito almeno una volta; questo perchè se la condizione viene verificata il loop continua, fino a quando non viene più rispettata.

```
int x = 0;  
do{  
    scanf("%d", &x);  
}while(x<100);
```

In questo do-while l'operazione `scanf` viene ripetuta fino a quando il valore di x non supera 99.

Numeri random

Rand

La funzione rand genera un numero pseudo-casuale nell'intervallo da 0 a RAND_MAX. Per ottenere numeri in un intervallo specifico, è possibile utilizzare l'operazione modulo (%). Prima di utilizzare rand, è necessario inizializzare il generatore di numeri casuali con srand.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main() {
    // Generazione di un numero casuale compreso tra 0 e RAND_MAX
    int numeroCasuale = rand();

    // Generazione di un numero casuale compreso tra 1 e 100
    int numeroCasualeLimitato = rand() % 100 + 1;

    printf("Numero casuale: %d\n", numeroCasuale);
    printf("Numero casuale tra 1 e 100: %d\n", numeroCasualeLimitato);

    return 0;
}
```

Srand

La funzione srand viene utilizzata per inizializzare il generatore di numeri casuali. Di solito, si utilizza come argomento il valore restituito dalla funzione time(NULL). Ciò consente di generare sequenze di numeri casuali diverse ad ogni esecuzione del programma.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main() {
    // Inizializzazione del generatore di numeri casuali
    srand(time(NULL));

    // Resto del codice

    return 0;
}
```

Vettori

I vettori sono strutture dati che consentono di memorizzare più elementi dello stesso tipo in una singola variabile. La dichiarazione di un vettore avviene specificando il tipo degli elementi e la dimensione del vettore.

```
#include <stdio.h>

int main() {
    // Dichiarazione e inizializzazione di un vettore di interi
    int numeri[5] = {1, 2, 3, 4, 5};

    // Accesso agli elementi del vettore
    printf("Elemento 0: %d\n", numeri[0]);
    printf("Elemento 1: %d\n", numeri[1]);

    return 0;
}
```

Matrici

Una matrice è una struttura bidimensionale che può essere vista come un vettore di vettori. La dichiarazione di una matrice avviene specificando il tipo degli elementi e le dimensioni della matrice.

```
#include <stdio.h>

int main() {
    // Dichiarazione e inizializzazione di una matrice di interi
    int matrice[3][3] = {
        {1, 2, 3},
        {4, 5, 6},
        {7, 8, 9}
    };

    // Accesso agli elementi della matrice
    printf("Elemento in posizione (1, 1): %d\n", matrice[1][1]);

    return 0;
}
```

Puntatori

I puntatori sono variabili che contengono l'indirizzo di memoria di un'altra variabile. Consentono di manipolare direttamente la memoria, migliorando l'efficienza e la flessibilità del codice. I puntatori sono variabili che contengono l'indirizzo di memoria di un'altra variabile. Consentono di manipolare direttamente la memoria, migliorando l'efficienza e la flessibilità del codice.

Dichiarazione di puntatori

I puntatori vengono dichiarati utilizzando il simbolo * seguito dal tipo di variabile a cui puntano. Ad esempio, la seguente dichiarazione dichiara un puntatore a una variabile di tipo int:

```
int *p;
```

Questa dichiarazione alloca spazio per una variabile di tipo int, ma non assegna un indirizzo di memoria a tale variabile. Per assegnare un indirizzo di memoria a un puntatore, è necessario utilizzare l'operatore &.

Operatore &

L'operatore & viene utilizzato per ottenere l'indirizzo di memoria di una variabile. Ad esempio, la seguente istruzione assegna l'indirizzo di memoria della variabile x al puntatore p:

```
int x = 10;  
int *p = &x;
```

Dopo questa istruzione, il puntatore p contiene l'indirizzo di memoria della variabile x. Questo indirizzo può essere utilizzato per accedere al valore di x utilizzando l'operatore *.

Operatore *

L'operatore * viene utilizzato per accedere al valore a cui punta un puntatore. Ad esempio, la seguente istruzione stampa il valore della variabile x utilizzando il puntatore p:

```
int x = 10;  
int *p = &x;  
  
printf("%d\n", *p);
```

Questa istruzione stampa il valore 10.

Esempio

```
#include <stdio.h>

int main() {
    // Dichiarazione di una variabile e di un puntatore
    int x = 10;
    int \*puntatoreX;

    // Inizializzazione del puntatore con l'indirizzo di x
    puntatoreX = &x;

    // Accesso al valore di x tramite il puntatore
    printf("Valore di x: %d\n", *puntatoreX);

    // Modifica del valore di x tramite il puntatore
    *puntatoreX = 20;

    // Stampa del nuovo valore di x
    printf("Nuovo valore di x: %d\n", x);

    return 0;
}
```

I puntatori sono particolarmente utili nell'allocazione dinamica della memoria e nelle funzioni che devono modificare direttamente i valori delle variabili.

Puntatori con vettori

```
#include <stdio.h>
#define DIM 5

int main(){
    // dichiaro vettore
    int v[DIM];

    // dichiaro puntatore
    int* p;

    /* p = &v[0]: il valore del puntatore p è
    inizializzato al primo byte del vettore */
    for(p = &v[0]; p < v[DIM]; p++){
        // inizializzo il vettore tramite il puntatore
        *p = 0;
    }
}
```

Stringhe (Array di caratteri)

In C non esistono le "stringhe" in quanto le parole e le frasi vengono memorizzati come array di caratteri. Gli array vengono dichiarati mediante il tipo di dato `char`.

```
#include <stdio.h>
#define DIM 21

int main(){
    char parola[DIM];
    int len = strlen(parola);

    for(int i = 0; i < len; i++){
        parola[i] = '-';
    }
}
```

In questo esempio vediamo come inizializzare un char array; notiamo che nella dichiarazione inserisco nelle parentesi quadre la dimensione.

La dimensione DIM è 21, questo significa che possiamo accettare delle parole lunghe al massimo 20 caratteri in quanto nei char array l'ultima posizione è riservata per il carattere di fine stringa `\0`.

strlen()

La funzione `strlen` consente di trovare la lunghezza della stringa.

gets()

La funzione `gets()` ha la stessa funzione di `scanf()`, ma prende tutti i caratteri inseriti dall'utente, compresi gli spazi. Questo non avviene utilizzando `scanf()` in quanto lo spazio viene considerato come un "invio".

NB! `fgets()` migliore.

puts()

La funzione `puts()` ha la stessa funzione di `printf()`, ma solo per le stringhe.

strcpy()

```
char* strcpy(str2, str1);
```

Serve a copiare una stringa all'interno di un'altra. La funzione ritorna il puntatore della stringa in cui viene copiata la prima.

```
#include <stdio.h>
#include <string.h>

int main() {
    char str1[20] = "C programming";
    char str2[20];

    // copia str1 in str2
    strcpy(str2, str1);

    puts(str2);

    return 0;
}
```

strncpy()

Funziona come `strcpy()`, ma ammette un altro parametro di tipo intero che permette di specificare quanti char copiare.

```
char* strncpy(char*, char*, int);
```

strcat()

```
char* strcat(char*, char*);
```

Funzione che permette di concatenare due stringhe (quindi di "sommarle"). Il primo parametro corrisponde alla stringa di destinazione e il secondo alla stringa sorgente. Il valore che ritorna è la stringa di destinazione (posso concatenare la funzione).

```
char str1[4] = "abc";
char str2[7] = "def";
strcat(str2, str1);
```

```
risultato: "d e f a b c \0"
```

strncat()

```
char* strncat(char*, char*, int);
```

```
#include <stdio.h>
#include <string.h>

int main () {
    char src[50], dest[50];

    strcpy(src, "This is source");
    strcpy(dest, "This is destination");

    strncat(dest, src, 15);

    printf("Final destination string : %s", dest);

    return(0);
}
```

```
Final destination string : This is destinationThis is source
```

strcmp()

Funzione che viene utilizzata per comparare le due stringhe.

- **ritorna 0** se le stringhe sono identiche
- **ritorna un numero > 0** se il primo carattere non identico ha un valore ASCII maggiore del secondo
- **ritorna un numero < 0** se il primo carattere non identico ha un valore ASCII minore del secondo

Funzioni

Fino ad ora abbiamo scritto i nostri programmi utilizzando solo la funzione main: si può (e si deve), invece, suddividere il programma in blocchi di codice che risolvono specifiche task. Dividendo il programma in blocchi rende più facile la comprensione e la manutenzione del programma. Ci sono due tipi di funzioni, quelle standard, definite "dal linguaggio di programmazione" che già conosciamo, `scanf()`, `printf()` etc. e **quelle definite dall'utente**.

Definire e chiamare una funzione

```
#include <stdio.h>
int nomeFunzione("eventuali parametri")
{
```

```
    ... ..  
    ... ..  
}  
  
int main()  
{  
    ... ..  
    ... ..  
    // chiamata della funzione  
    nomeFunzione();  
    ... ..  
    ... ..  
}
```

Si può definire anche in modo diverso, creando prima un prototipo della funzione:

```
#include <stdio.h>  
int nomeFunzione("eventuali parametri");  
  
int main()  
{  
    ... ..  
    ... ..  
    nomeFunzione();  
    ... ..  
    ... ..  
}  
  
int nomeFunzione("eventuali parametri"){  
    // operazioni da svolgere  
}
```

Return nelle funzioni

Una volta che la funzione viene eseguita è previsto un **return** che permette di ritornare il risultato della funzione, appena eseguita, alla funzione main, per magari memorizzarla all'interno di una variabile. Il **tipo di dato** ritornato dalla funzione dipende dal **tipo della funzione**, se una funzione è di tipo **int** allora dovrà ritornare una variabile di tipo int e così via.

Passaggio di parametri

Per fare in modo che le funzioni siano effettivamente utili dobbiamo passare dei **parametri** in modo che la funzione possa lavorare su qualcosa:

```
#include <stdio.h>  
int somma(int a, int b);           // prototipo funzione  
  
int main()
```



```

{
    int n1, n2, risultato;

    printf("Enters two numbers: ");
    scanf("%d %d",&n1,&n2);

    // chiamata funzione e memorizzazione del valore nella variabile
    risultato
    risultato = somma(n1, n2);
    printf("somma = %d", risultato);

    return 0;
}

// i parametri n1 e n2 prendono il nome di a e b
int somma(int a, int b)          // definizione funzione
{
    int risultato;
    risultato = a+b;

    return risultato;            // return
}

```

How to pass arguments to a function?

```

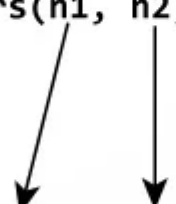
#include <stdio.h>

int addNumbers(int a, int b);

int main()
{
    ... ..
    sum = addNumbers(n1, n2);
    ... ..
}

int addNumbers(int a, int b)
{
    ... ..
    ... ..
}

```



The diagram shows two arrows originating from the arguments 'n1' and 'n2' in the function call `addNumbers(n1, n2);` within the `main` function. These arrows point to the parameters 'a' and 'b' in the function definition `int addNumbers(int a, int b)`, illustrating how the values of 'n1' and 'n2' are passed to 'a' and 'b' respectively.

In C, quando si passa un argomento a una funzione, si può passare il valore dell'argomento o il suo riferimento.

Passaggio per valore

Quando si passa un argomento per valore, la funzione riceve una copia del valore dell'argomento. Qualsiasi modifica apportata all'argomento all'interno della funzione non ha alcun effetto sull'argomento originale.

Ad esempio, la seguente funzione somma due numeri:

```
int somma(int a, int b) {  
    int c = a + b;  
    return c;  
}
```

Se chiamiamo questa funzione con gli argomenti **a** e **b**, la funzione creerà una loro copia e la passerà come argomenti. All'interno della funzione, la somma dei due numeri verrà calcolata e il risultato verrà restituito. Tuttavia, la variabile **a** nella chiamata alla funzione rimarrà uguale a 1 e la variabile **b** rimarrà uguale a 2.

```
int main() {  
    int a = 1;  
    int b = 2;  
  
    int c = somma(a, b);  
  
    printf("a = %d\n", a);  
    printf("b = %d\n", b);  
    printf("c = %d\n", c);  
  
    return 0;  
}
```

L'output di questo programma è il seguente:

```
a = 1  
b = 2  
c = 3
```

Passaggio per riferimento

Quando si passa un argomento per riferimento, la funzione riceve un puntatore all'argomento. Qualsiasi modifica apportata all'argomento all'interno della funzione si rifletterà sull'argomento originale.

Per indicare che un argomento deve essere passato per riferimento, si usa l'asterisco davanti al tipo di dato dell'argomento. Ad esempio, la seguente funzione somma due numeri passandoli per riferimento:

```
void somma_per_riferimento(int *a, int *b) {  
    int c = *a + *b;
```

```
*a = c;  
*b = c;  
}
```

Se chiamiamo questa funzione con gli argomenti 1 e 2, la funzione riceverà un puntatore a 1 e un puntatore a 2. All'interno della funzione, la somma dei due numeri verrà calcolata e il risultato verrà memorizzato nell'argomento a. Quindi, il valore di a verrà aggiornato a 3 e il valore di b verrà aggiornato a 3.

```
int main() {  
    int a = 1;  
    int b = 2;  
  
    somma_per_riferimento(&a, &b);  
  
    printf("a = %d\n", a);  
    printf("b = %d\n", b);  
  
    return 0;  
}
```

L'output di questo programma è il seguente:

```
a = 3  
b = 3
```

Operazioni su sequenze

Ricerca completa

La ricerca completa è un algoritmo di ricerca che esamina ogni elemento di un array fino a trovare quello desiderato. È un algoritmo semplice da implementare, ma può essere inefficiente se l'array è grande o se l'elemento desiderato è raro.

Questa funzione esegue una ricerca lineare su un array per trovare la prima occorrenza di un elemento dato. Prende in input un array, la sua dimensione e l'elemento da cercare. Restituisce l'indice dell'elemento se trovato, o -1 se non trovato.

```
int ricerca_completa(int vet[], int n, int x) {  
    int i;  
    for (i = 0; i < n && vet[i] != x; i++);  
    if (i < n) return i;  
    else return -1;  
}
```

Ricerca Binaria

La ricerca binaria è un algoritmo di ricerca più efficiente della ricerca completa. Funziona suddividendo l'array a metà e quindi cercando l'elemento desiderato nell'emettitore metà. Questo processo viene ripetuto fino a trovare l'elemento desiderato o fino a quando l'array è vuoto.

Questa funzione esegue una ricerca binaria su un array per trovare un elemento dato. Prende in input un array, la sua dimensione e l'elemento da cercare. Restituisce l'indice dell'elemento se trovato, o -1 se non trovato.

```
int ricerca_binaria(int vet[], int n, int x) {
    int sx = 0, dx = n-1, cx;
    while (sx <= dx) {
        cx = (sx + dx) / 2;
        if (vet[cx] == x) return cx;
        else if (vet[cx] < x) sx = cx + 1;
        else dx = cx - 1;
    }
    return -1;
}
```

Inserimento all'inizio

L'inserimento all'inizio è un algoritmo di inserimento che inserisce un nuovo elemento all'inizio di un array. L'elemento esistente nella prima posizione viene spostato in avanti di una posizione.

Questa funzione inserisce un nuovo elemento all'inizio di un array. Prende in input un array, la sua dimensione e il nuovo elemento da inserire. Restituisce la nuova dimensione dell'array.

```
int inserimento_inizio(int vet[], int n, int nuovo) {
    if (n < DIM) {
        vet[n++] = vet[0];
        vet[0] = nuovo;
    }
    else printf("\nWarning: Array is full");

    return n;
}
```

Inserimento alla fine

L'inserimento alla fine è un algoritmo di inserimento che inserisce un nuovo elemento alla fine di un array. L'elemento viene inserito nella posizione successiva all'ultima posizione occupata.

Questa funzione inserisce un nuovo elemento alla fine di un array. Prende in input un array, la sua dimensione e il nuovo elemento da inserire. Restituisce la nuova dimensione dell'array.

```
int inserimento_fine(int vet[], int n, int nuovo) {
    if (n < DIM) vet[n++] = nuovo;
```

```
    else printf("\nWarning: Array is full");  
    return n;  
}
```

Inserimento diretto

L'inserimento diretto è un algoritmo di inserimento che inserisce un nuovo elemento in un array nella sua posizione corretta in base all'ordine dell'array. L'algoritmo inizia dalla prima posizione dell'array e sposta gli elementi maggiori di quello da inserire in avanti di una posizione. L'elemento viene quindi inserito nella posizione libera.

Questa funzione inserisce un nuovo elemento in un array nella sua posizione corretta in base all'ordine dell'array. Prende in input un array, la sua dimensione e il nuovo elemento da inserire. Restituisce la nuova dimensione dell'array.

```
int inserimento_diretto (int vet[], int n, int nuovo) {  
    int i, j;  
    for (i = 0; i < n && vet[i] < nuovo; i++);  
    for (j = n; j > i; j--) vet[j] = vet[j-1];  
    vet[i] = nuovo;  
    return ++n;  
}
```

Cancellazione

La cancellazione è un algoritmo di cancellazione che rimuove un elemento da un array. L'elemento viene rimosso spostando tutti gli elementi successivi indietro di una posizione.

Questa funzione rimuove un elemento da un array. Prende in input un array, la sua dimensione e l'elemento da rimuovere. Restituisce la nuova dimensione dell'array.

```
int cancellazione(int vet[], int n, int num) {  
    int pos = cerca(vet, n, num);  
    if (pos >= 0) {  
        vet[pos] = vet[n-1];  
        n--;  
    }  
    else printf("\nWarning: Number not found");  
    return n;  
}
```

Fusione

L'algoritmo di fusione permette di unire due array.

Questa funzione unisce due array in un terzo array. Prende in input due array, le loro dimensioni e il terzo array in cui unire i due. Restituisce la dimensione del terzo array.

```
int fusione(int v1[], int v2[], int v3[], int n1, int n2)
{
    int i=0, j=0, k=0;
    do {
        if (i < n1 && (j == n2 || v1[i] <= v2[j]))
            v3[k++] = v1[i++];
        else //if (j < n2 && (vet1[i] > vet2[j] || i == n1))
            v3[k++] = v2[j++];
    } while (i < n1 || j < n2);
    return k;
}
```

Algoritmi di ordinamento

Gli algoritmi di ordinamento sono algoritmi che permettono di ordinare gli elementi di un array in base al loro valore. Questa operazione si può effettuare con array di interi, ma anche con array di stringhe. I seguenti sono gli algoritmi di ordinamento più semplici, nonché i più dispendiosi in termini di tempo di esecuzione.

Ordinamento per inserimento diretto

Ordino una sequenza effettuando tanti inserimenti nella sequenza stessa. Inserisco l'i-esimo elemento nella 1°, 2°... i-esima posizione se esso è inferiore al 1°, 2°,..., i-esimo elemento.

Questa operazione si può effettuare un solo vettore ragionando per sottosequenze.

```
void ordinamento_diretto(int vet[], int n) {
    int i, j, x;
    for (i = 1; i < n; i++) {
        x = vet[i];
        for (j = i-1; j >= 0 && x < vet[j]; j--)
            vet[j+1] = vet[j];
        vet[j+1] = x;
    }
}
```

Questo algoritmo è molto dispendioso in termini di tempo di esecuzione, in quanto per ogni elemento devo scorrere tutto il vettore. La complessità di questo algoritmo è $O(n^2)$. Il caso peggiore è quando il vettore è ordinato in modo decrescente.

Ordinamento per selezione

Scambiare l'i-esimo elemento con l'i-esimo più piccolo.

```
void ordinamento_selezione(int vet[], int n)
{
    int i, j, k, x;
    for (i = 0; i < n - 1; i++)
    {
        k = i;
        x = vet[i];
        for (j = i + 1; j < n; j++)
        {
            if (vet[j] < x)
            {
                k = j;
                x = vet[j];
            }
        }
        vet[k] = vet[i];
        vet[i] = x;
    }

    for (i = 0; i < n; i++)
        printf("%d ", vet[i]);
}
```

Dopo $n-1$ iterazioni, il vettore è ordinato. Anche in questo caso il caso peggiore è se il vettore è ordinato in modo decrescente.

Numero di confronti: $n(n-1)/2$. Numero di scambi: $n-1$.

Bubble Sort

Con il bubble sort si procede scambiando due elementi adiacenti se non sono in ordine. Dopo $n-1$ iterazioni, il vettore è ordinato.

```
void bubble_sort(int vet[], int n)
{
    int i, j, x;
    for (i = 0; i < n - 1; i++)
    {
        for (j = n - 1; j > i; j--)
        {
            if (vet[j - 1] > vet[j])
            {
                x = vet[j - 1];
                vet[j - 1] = vet[j];
                vet[j] = x;
            }
        }
    }

    printf("\n");
}
```

```
for (i = 0; i < n; i++)  
    printf("%d ", vet[i]);  
}
```

Strutture

Le strutture sono un tipo di dato che permette di raggruppare più variabili di tipo diverso all'interno di una singola variabile. Le variabili all'interno di una struttura vengono chiamate campi o membri. Le strutture sono dei tipi di dato definiti dall'utente.

Definizione di una struttura

Per definire una struttura, si utilizza la parola chiave `struct` seguita dal nome della struttura e da una lista di campi racchiusi tra parentesi graffe. Ogni campo è definito specificando il tipo di dato e il nome del campo.

```
struct studente {  
    char nome[50];  
    int eta;  
    float media;  
} s1;
```

Inizializzazione di una struttura

Per inizializzare una struttura, si utilizza la parola chiave `struct` seguita dal nome della struttura e da una lista di campi racchiusi tra parentesi graffe. Ogni campo è definito specificando il tipo di dato e il nome del campo.

```
struct studente {  
    char nome[50];  
    int eta;  
    float media;  
} s1 = {"Mario", 20, 27.5};
```

Se effettuo l'inizializzazione dopo la dichiarazione posso procedere in due modi:

1. Inizializzo tutti i campi utilizzando la sintassi vista prima

```
struct studente s1 = {"Mario", 20, 27.5};
```


2. Inizializzo i campi uno alla volta utilizzando l'operatore `.` per accedere ai singoli campi:

```
struct studente s1;  
s1.nome = "Mario";  
s1.eta = 20;  
s1.media = 27.5;
```

Esempio completo:

```
struct studente {  
    char nome[50];  
    int eta;  
    float media;  
};  
  
int main(){  
    struct studente s1 = {"Mario", 20, 27.5};  
  
    struct studente s2;  
    s2.nome = "Luigi";  
    s2.eta = 21;  
    s2.media = 28.5;  
}
```

Struct e puntatori

Posso utilizzare i puntatori per accedere ai campi di una struttura.

```
struct studente {  
    char nome[50];  
    int eta;  
    float media;  
} s1, *p;  
  
p = &s1;  
  
(*p).nome = "Mario";  
(*p).eta = 20;  
(*p).media = 27.5;
```

Per evitare di scrivere `(*p).nome` posso utilizzare l'operatore `->`:

```
p->nome = "Mario";  
p->eta = 20;  
p->media = 27.5;
```

Esercizi da svolgere:

[Pdf esercizi Università della Calabria](#)