

Programmazione in C



Intro

C è un linguaggio di programmazione *general purpose* che ha origine nei primi anni '70

The C Programming Language - 1978 - K&R

Ad oggi, il linguaggio **C** ha subito diverse modifiche, dettate da successive standardizzazioni imposte da diverse organizzazioni (**ANSI, ISO**)

- 1989 :: C89
- 1999 :: C99
- 2011 :: C11
- 2018 :: C18



Applicazioni

Sistemi Embedded:

- Sistemi Domotici
- PLC
- Automobili
- Apparecchiature mediche
- Telecamere e sistemi di sicurezza
- Sistemi di automazione
- Elettrodomestici
- Industria

Linguaggio C

Imperativo !

Permette di avere il controllo del dispositivo tramite diverse *istruzioni*, intese dunque come dei veri e propri ordini

Procedurale

Cioè è costituito da blocchi di codice identificati da un nome e racchiusi in particolari delimitatori { Funzioni }

Alto Livello

Significativa astrazione dai dettagli di funzionamento di uno specifico calcolatore e dalle caratteristiche del linguaggio macchina

```
1 #include <stdlib.h>
2 #include <stdio.h>
3
4 /* The main thing that this program does. */
5 int main(void) {
6     // Declarations
7     double A[5] = {
8         [0] = 9.0,
9         [1] = 2.9,
10        [4] = 3.E+25,
11        [3] = .00007,
12    };
13
14    // Doing some work
15    for (size_t i = 0; i < 5; ++i) {
16        printf("element %zu is %g, \ tits square is %g\n",
17              i,
18              A[i],
19              A[i]*A[i]);
20    }
21
22    return EXIT_SUCCESS;
23 }
```

Compilazione ed Esecuzione

Del semplice testo, anche se apparentemente molto specifico dettagliato, non può essere compreso da un calcolatore.

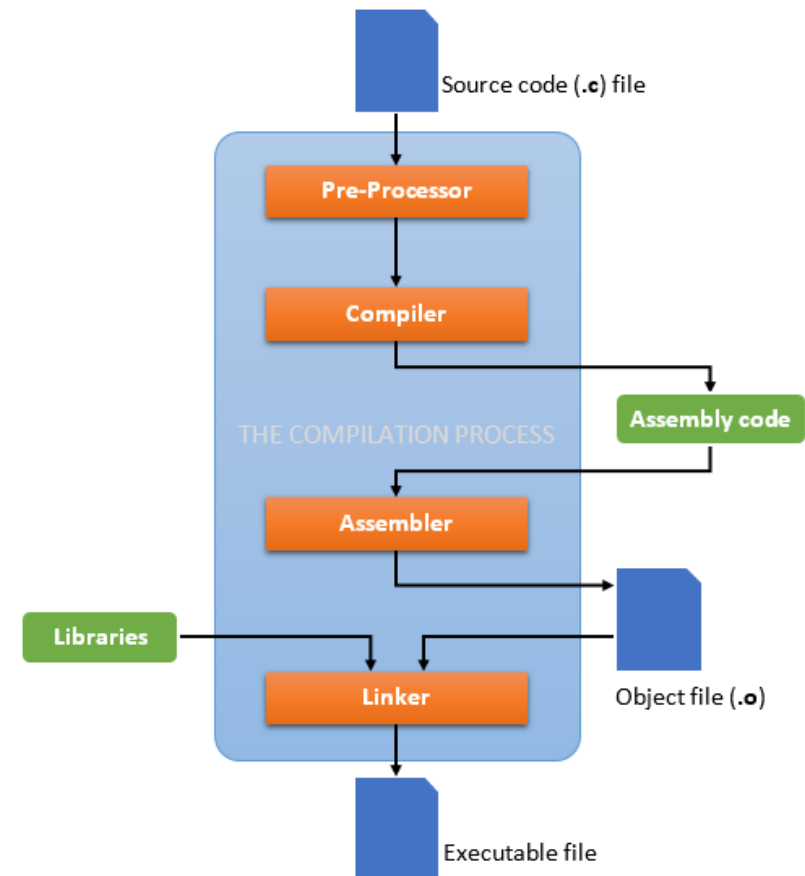
C fornisce un livello di astrazione che si può adeguare a diverse piattaforme, grazie a

differenti **compilatori** e **linker**

Il nome del **compilatore**, il suo utilizzo e i suoi *argomenti*, sono specifici della piattaforma sulla quale il programma dovrà essere eseguito

```
gcc -Wall -o program_name source_code.c -lm
```

```
arm-linux-gnueabi-gcc -std=c99 -O1 -g3 -c -  
fmessage-length=0 -pthread -MMD -MP program -o  
source.c
```



Struttura di un programma

Aspetti **Sintattici**

Come scrivere correttamente, in maniera comprensibile dal compilatore?

- Parole Chiave Speciali
 - #include, int, void, double, for, return
- Punteggiatura
 - { ... }, (...), [...], /* ... */, < ... >
- Valori Alfa-Numerici
- Identificativi Specifici
 - main, printf, size_t, EXIT_SUCCESS, A, i
- Funzioni
 - main, printf
- Operatori
 - =, *, <, ++, -

Aspetti **Semantici**

Come esprimere con il linguaggio ciò che vogliamo venga eseguito?

```
1 #include <stdlib.h>
2 #include <stdio.h>
3
4 /* The main thing that this program does. */
5 int main(void) {
6     // Declarations
7     double A[5] = {
8         [0] = 9.0,
9         [1] = 2.9,
10        [4] = 3.E+25,
11        [3] = .00007,
12    };
13
14    // Doing some work
15    for (size_t i = 0; i < 5; ++i) {
16        printf("element %zu is %g, \ tits square is %g\n",
17            i,
18            A[i],
19            A[i]*A[i]);
20    }
21
22    return EXIT_SUCCESS;
23 }
```

Struttura di un programma

Dichiarazioni e Definizioni

Prima di utilizzare un qualsiasi particolare identificativo, è necessario fornire al compilatore una **dichiarazione** che specifica ciò che l'identificativo rappresenta.

```
int main(void);  
double A[5];  
size_t i;
```

- **main**, seguito da una coppia di parentesi tonde, è dichiarato come *funzione* di *tipo* **int**
- **i** è dichiarato come una *variabile* numerica di *tipo* **size_t**
- **A** è dichiarato come un *array* di valori di *tipo* **double**

```
int printf(char const format[static 1], ...);  
typedef unsigned long size_t;  
#define EXIT_SUCCESS 0
```

Generalmente le **dichiarazioni** specificano il tipo di oggetto a cui un identificativo si riferisce, ma non attribuiscono alcun valore a tale oggetto: questo compito è affidato alle procedure di **definizione**

```
size_t i = 0;
```

Questa **inizializzazione** definisce l'oggetto con un corrispondente nome; questa istruzione per il compilatore significa allocare un'area di memoria, cui ci si riferirà con un particolare nome, e conservare al suo interno (inizialmente) il valore 0

Struttura di un programma

Iterazioni e chiamate a Funzioni

Eseguire una o più operazioni diverse volte

```
for (loop-variable; loop-condition; loop-post-exec)
    operations ...
```

```
while (condition)
    operations ...
```

```
do
    operations ...
while(condition);
```

Delegare l'esecuzione di parti di codice a blocchi che si trovano da un'altra parte

Una funzione viene chiamata passando ad essa dei particolari **argomenti**, e aspettando che essa svolga le sue funzioni restituendo dei valori

- Call by values
- Call by reference

In **C** non è implementato questo meccanismo, ma si possono ottenere risultati equivalenti utilizzando i **puntatori**

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

/* The main thing that this program does. */
int main(void) {

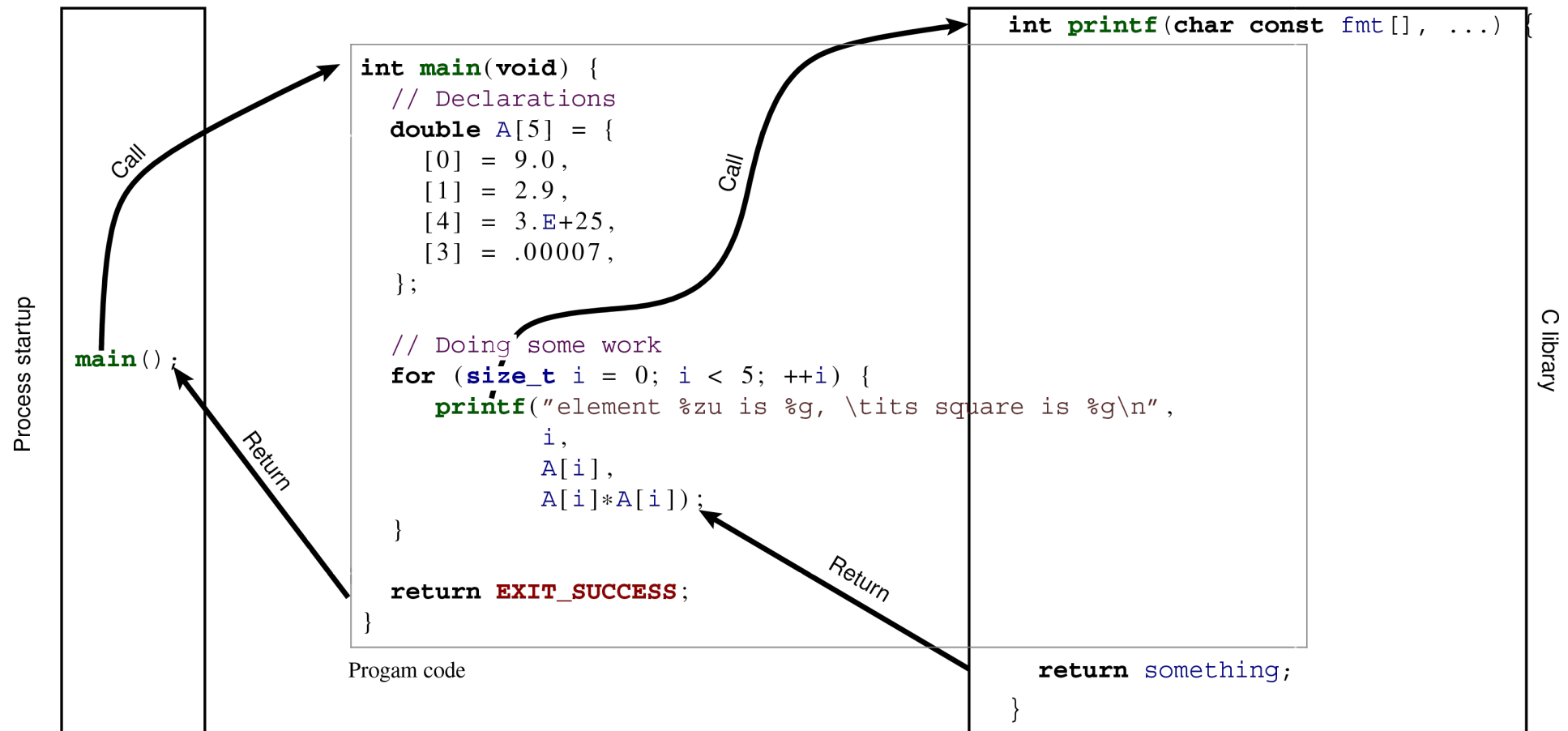
    double A[5] = {
        [0] = 9.0,
        [1] = 2.9,
        [4] = 3.E+25,
        [3] = .00007,
    };

    // Doing some work
    for (size_t i = 0; i < 5; ++i) {
        printf("element %zu is %g, \tit's square is %g\n",
            i,
            A[i],
            A[i]*A[i]);
    }

    return EXIT_SUCCESS;
}
```

Struttura di un programma

Chiamate a Funzioni



Struttura di un programma

Controllo di flusso

Controllare il flusso di un programma significa, in generale, gestirne il comportamento

Sebbene la chiamata a funzione sia già un meccanismo di controllo di flusso, un costrutto come il **for** fornisce qualcosa in più

Controllo Condizionale

```
if (condition)
    operations ...
else if
    operations ...
else
    operations ...
```

```
switch (arg) {
    case 'arg1':
        operations ...
        break;

    default:
        operations ...
}
```

1. **if**
2. **for**
3. **do**
4. **while**
5. **switch**

Inoltre in **C** sono presenti altri costrutti per il controllo condizionale particolari:

operatore ternario `condition ? A : B`

condizioni di pre-processor `#if/#ifdef/#ifndef/#elif/#else/#endif`

Computazione di espressioni

Una parte cruciale di molti programmi consiste nell'elaborazione di espressioni, che siano esse **aritmetiche**, **booleane** o miste.

Un linguaggio di programmazione fornisce una serie di *operatori* per trattare le espressioni

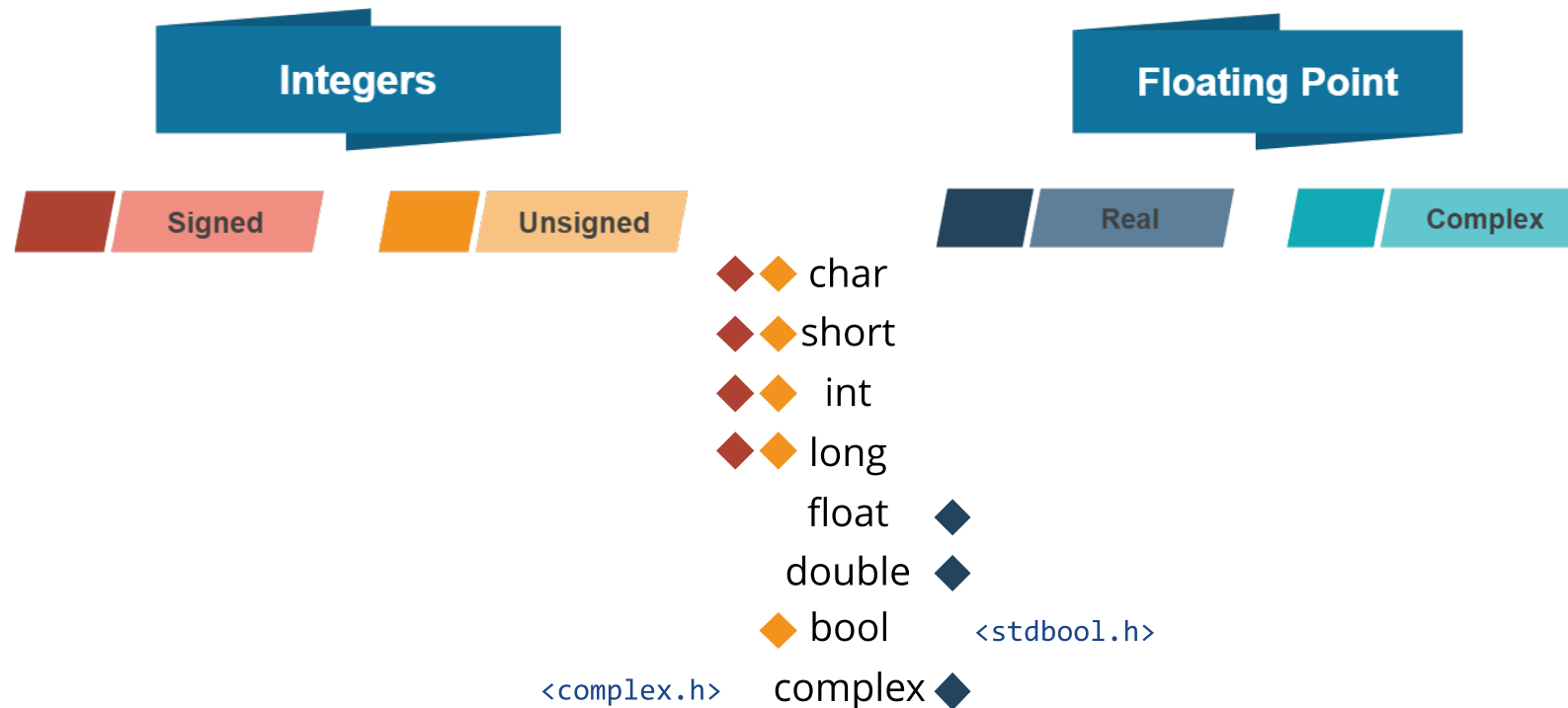
Operatori aritmetici	Operatori di confronto	Operatori Logici
+	==	&& (<i>and</i>)
-	!=	(<i>or</i>)
*	>	! (<i>not</i>)
/	<	
%	<=	
	>=	

Operatori per aritmetica binaria
~ (<i>complemento</i>)
& (<i>bit-and</i>)
(<i>bit-or</i>)
^ (<i>xor</i>)
>> , << (<i>shift</i>)

Tipi di dato

Un linguaggio di programmazione come **C** lavora con **dati** che, durante le elaborazioni, possono assumere differenti **valori**.

I valori che un dato può assumere dipendono dal **tipo di dato** che si sta trattando



Tipi di dato

La quantità di memoria occupata da un dato di un certo **tipo** dipende dalla piattaforma per la quale il codice viene compilato, sebbene ci siano delle direttive riguardo le possibili dimensioni che un tipo può avere

Tipi	Dimensione	Utilizzo
◆◆ char	8 bit	<code>char c = 'a'</code> <code>unsigned char cu = 254</code>
◆◆ short	16 bit	<code>short x = 3</code>
◆◆ int	16/32 bit	<code>int x = 350</code>
◆◆ long	32/64 bit	<code>long l = 123581321</code> <code>unsigned long long ll = 9223372036854775807</code>
◆ float	16/32 bit	<code>float f = 3.52</code>
◆ double	32/64 bit	<code>double d = 3.E+25</code>
◆ bool	8 bit	<code>bool b = true</code>

Ogni dato in memoria è conservato all'interno di una **variabile**

Tipi di dato

Valutare i valori range che possono assumere i diversi tipi

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <limits.h>
4 #include <float.h>
5
6 int main(int argc, char** argv) {
7
8     printf("CHAR_BIT      : %d\n", CHAR_BIT);
9     printf("CHAR_MAX      : %d\n", CHAR_MAX);
10    printf("CHAR_MIN      : %d\n", CHAR_MIN);
11    printf("INT_MAX       : %d\n", INT_MAX);
12    printf("INT_MIN       : %d\n", INT_MIN);
13    printf("LONG_MAX      : %ld\n", (long) LONG_MAX);
14    printf("LONG_MIN      : %ld\n", (long) LONG_MIN);
15    printf("SCHAR_MAX     : %d\n", SCHAR_MAX);
16    printf("SCHAR_MIN     : %d\n", SCHAR_MIN);
17    printf("SHRT_MAX      : %d\n", SHRT_MAX);
18    printf("SHRT_MIN      : %d\n", SHRT_MIN);
19    printf("UCHAR_MAX     : %d\n", UCHAR_MAX);
20    printf("UINT_MAX      : %u\n", (unsigned int) UINT_MAX);
21    printf("ULONG_MAX     : %lu\n", (unsigned long)
    ULONG_MAX);
22    printf("USHRT_MAX     : %d\n", (unsigned short)
    USHORT_MAX);
23
```

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <limits.h>
4 #include <float.h>
5
6 int main(int argc, char** argv) {
7
8     printf("Storage size for float : %d \n", sizeof(float));
9     printf("FLT_MAX       : %g\n", (float) FLT_MAX);
10    printf("FLT_MIN       : %g\n", (float) FLT_MIN);
11    printf("-FLT_MAX      : %g\n", (float) -FLT_MAX);
12    printf("-FLT_MIN      : %g\n", (float) -FLT_MIN);
13    printf("DBL_MAX       : %g\n", (double) DBL_MAX);
14    printf("DBL_MIN       : %g\n", (double) DBL_MIN);
15    printf("-DBL_MAX      : %g\n", (double) -DBL_MAX);
16    printf("Precision value: %d\n", FLT_DIG );
17
18    return 0;
19 }
```

Dati e Variabili

Qualificatori

I **qualificatori** si associano ai tipi in termini di attributi per la variabile cui sono associati

- ◆ **const** Associato ad un dato, specifica che non si hanno i diritti di modificarlo

```
const int x = 2;  
int y = x;
```

 ✓
- ◆ **volatile** Indica al compilatore che il valore così qualificato potrebbe cambiare in ogni momento anche a per mezzo di codice che non si trova "vicino" la dichiarazione della variabile
- ◆ **_Atomic** Si associa ad una variabile la cui lettura/scrittura deve avvenire in una singola istruzione: durante il cambio di valore della variabile non deve avvenire "altro"
- ◆ **restrict**

Vettori e Strutture

Esistono altri tipi di dato (apparentemente più complessi) che derivano comunque dai tipi di base presentati.

Due tra questi sono detti *tipi aggregati*, poiché nascono dalla combinazione di più oggetti (anche se non dello stesso tipo):

- **Vettori (*Arrays*)**
 - È un aggregazione di oggetti dello stesso tipo
- **Strutture (*Structures*)**
 - È una aggregazione di oggetti di tipo diverso

```
double a[4];  
double b[] = { [3] = 42.0, [2] = 37.0, };  
double c[] = { 22.0, 17.0, 1, 0.5, };
```

```
struct microcontroller{  
    char name[50];  
    int bitNo;  
    int ramKb;  
    float price;  
};  
struct microcontroller const my_micro = {  
    .name = "ARM",  
    .bitNo = 32,  
    .ramKb = 128,  
    .price = 2.87  
};
```

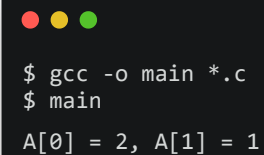
Vettori e Strutture

Esempi

```
#include <stdio.h>

void swap_double(double a[2]) {
    double tmp = a[0];
    a[0] = a[1];
    a[1] = tmp;
}

int main(void) {
    double A[2] = { 1.0, 2.0 };
    swap_double(A);
    printf("A[0] = %g, A[1] = %g\n", A[0], A[1]);
}
```



```
$ gcc -o main *.c
$ main
A[0] = 2, A[1] = 1
```


Vettori e Strutture

Esempi

```
#include <stdio.h>

float weighted_mean(int x[], int w[], int n) {
    int w_sum = 0, num = 0;

    for (int i = 0; i < n; i++){
        num = num + x[i] * w[i];
        w_sum = w_sum + w[i];
    }

    return (float)(num / w_sum);
}

int main(){

    int x[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    int w[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

    int n = sizeof(x)/sizeof(x[0]);
    int m = sizeof(w)/sizeof(w[0]);

    if (n == m){
        float result = weighted_mean(x, w, n);
        printf("Weigthed mean: %g\n", result);
    }else{
        printf("n!=m\n");
        return -1;
    }
    return 0;
}
```

```
$ gcc -o main *.c
$ main
Weigthed mean: 7
```

Vettori e Strutture

Esempi

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

bool leapyear(unsigned year) {
    return !(year % 4) && ((year % 100) || !(year % 400));
}

#define DAYS_BEFORE \
    (const int[12]){ \
        [0] = 0, [1] = 31, [2] = 59, [3] = 90, \
        [4] = 120, [5] = 151, [6] = 181, [7] = 212, \
        [8] = 243, [9] = 273, [10] = 304, [11] = 334, \
    }

struct tm time_set_yday(struct tm t) {
    t.tm_yday += DAYS_BEFORE[t.tm_mon] + t.tm_mday - 1;

    if ((t.tm_mon > 1) && leapyear(t.tm_year+1900)){
        t.tm_yday++;
    }
    return t;
}

int main() {
    struct tm today = {
        .tm_year = 2020-1900,
        .tm_mon = 2,
        .tm_mday = 19,
        .tm_hour = 10,
        .tm_min = 0,
        .tm_sec = 47,
    };
    printf("This year is %d, next year will be %d\n",
        today.tm_year+1900, today.tm_year+1900+1);
    today = time_set_yday(today);
    printf("Day of the year is %d\n", today.tm_yday);

    return EXIT_SUCCESS;
}
```

```
struct tm{
int tm_sec; /* secondi (0-59) */
int tm_min; /* minuti (0-59) */
int tm_hour; /* ora (0-23) */
int tm_mday; /* giorno del mese (1-31) */
int tm_mon; /* mese dell'anno (0-11) */
int tm_year; /* anno dal 1900 */
int tm_wday; /* giorno della settimana (0-6) */
int tm_yday; /* giorno dell'anno (0-365) */
int tm_isdst; /* Ora legale */
}
```



Puntatori

Un **puntatore** è una variabile che contiene l'indirizzo di memoria di un'altra variabile

È dunque una variabile che non contiene direttamente il dato cui siamo interessati ma *punta* al dato fornendo l'indirizzo di memoria in cui esso è conservato

Operatore di indirezione (deferenziazione) : *

Restituisce il contenuto dell'area di memoria puntata dal puntatore

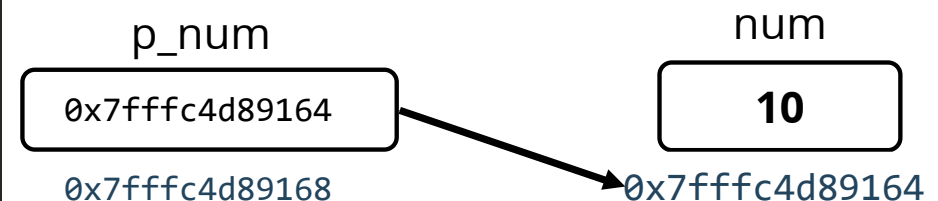
Operatore unario di referenziazione : &

Restituisce l'indirizzo cui si trova la variabile puntata

```
int *p1    /* Pointer to an integer */
double *p2 /* Pointer to a double */
char *p3   /* Pointer to a character */
```

```
double a = 10;
double *p;
p = &a; /* Get address of 'a' */
```

```
1 #include <stdio.h>
2
3 int main(){
4     int num = 10;
5     int *p_num = &num;
6     printf("Value of variable num is: %d", num);
7     printf("\nAddress of variable num is: %p", p_num);
8
9     return 0;
10 }
```



Puntatori

Esempi

```
1 #include <stdio.h>
2
3 void pointer_test(int val){
4     val = 20;
5 }
6
7 int main(){
8     int num = 10;
9     int *p_num = &num;
10    pointer_test(num);
11    printf("Value of variable num is: %d", num);
12    printf("\nAddress of variable num is: %p", p_num);
13
14    return 0;
15 }
```

```
1 #include <stdio.h>
2
3 void pointer_test(int *val){
4     *val = 20;
5 }
6
7 int main(){
8     int num = 10;
9     int *p_num = &num;
10    pointer_test(p_num);
11    printf("Value of variable num is: %d", num);
12    printf("\nAddress of variable num is: %p", p_num);
13
14    return 0;
15 }
```

```
$ gcc -o main *.c
$ main
Value of variable num is: 10
Address of variable num is: 0x0061FF18
```

```
$ gcc -o main *.c
$ main
Value of variable num is: 20
Address of variable num is: 0x0061FF18
```

Puntatori ed Array

I **Puntatori** e gli **Array** sono strettamente legati in C e spesso vengono erroneamente interscambiati

Se si utilizza solo il nome della variabile associata ad un **array**, senza alcun indice, si ottiene un **puntatore** al suo primo elemento

```
int array[2] = { 3, 5 };
```

```
array == &array[0]
```

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main(void){
5     char a[7]={'a', 'b', 'c', 'd', 'e', 'f', 'g'};
6     printf("a[3]: %c, *(a+3): %c", a[3], *(a+3));
7     return 0;
8 }
```

```
$ gcc -o main *.c
$ main
a[3]: d, *(a+3): d
```

Aritmetica dei puntatori

Le operazioni si effettuano implicitamente tenendo conto della dimensione delle variabili puntate

Puntatori ed Array

È possibile definire un **array** *dinamicamente* in memoria utilizzando i **puntatori**

Questa scrittura definisce un array di dimensione fissa nota al tempo di compilazione

```
int array[13];
```

Tramite i puntatori si può definire un array di dimensione nota a tempo di esecuzione

```
int *array = malloc(x * sizeof(int));
```

- ◆ **malloc** Alloca uno specifico numero di bytes in memoria
- ◆ **realloc** Aumenta o diminuisce la dimensione del blocco di memoria indicato
- ◆ **calloc** Alloca uno specifico numero di bytes in memoria inizializzandoli a 0
- ◆ **free** Libera specifico numero di bytes, lasciandoli nuovamente al sistema

Restituisce un tipo particolare: **void***
sono necessarie operazioni di **casting**

```
int *arr = (int*) malloc(2 * sizeof(int));  
arr[0] = 1;  
arr[1] = 2;  
arr = realloc(arr, 3 * sizeof(int));  
arr[2] = 3;
```

Puntatori ed Array

Esempi

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

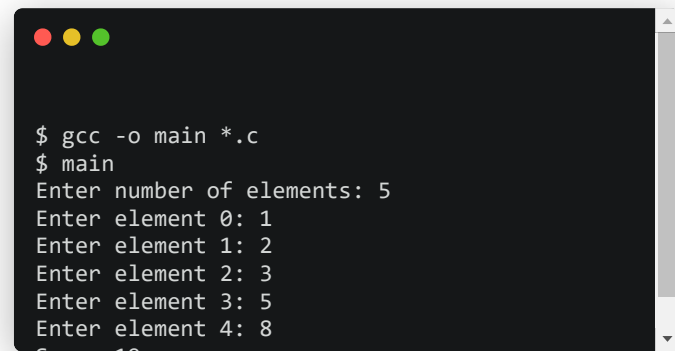
int main(){
    int n, i, *ptr, sum = 0;
    printf("Enter number of elements: ");
    scanf("%d", &n);

    ptr = (int*) calloc(n, sizeof(int));

    if(ptr == NULL){
        printf("Error! memory not allocated.");
        exit(0);
    }

    for(i = 0; i < n; ++i){
        printf("Enter element %d: ", i);
        scanf("%d", ptr + i);
        sum += *(ptr + i);
    }

    printf("Sum = %d\n", sum);
    free(ptr);
    return 0;
}
```



```
$ gcc -o main *.c
$ main
Enter number of elements: 5
Enter element 0: 1
Enter element 1: 2
Enter element 2: 3
Enter element 3: 5
Enter element 4: 8
Sum = 19
```

Programmazione

Convoluzione

In mathematics and, in particular, functional analysis, **convolution** is a mathematical operation on two functions f and g , producing a third function that is typically viewed as a modified version of one of the original functions

(from wikipedia.com)

$$f = \begin{bmatrix} 1 & 4 & 5 & 2 \end{bmatrix} \quad g = \begin{bmatrix} 3 & 4 & 1 \end{bmatrix} \quad h = \begin{bmatrix} f * g \end{bmatrix}$$

$$\begin{bmatrix} 1 & 4 & 3 \end{bmatrix} \quad \begin{bmatrix} 1 & 4 & 5 & 2 \end{bmatrix} \quad h[0] = 1 * 3$$

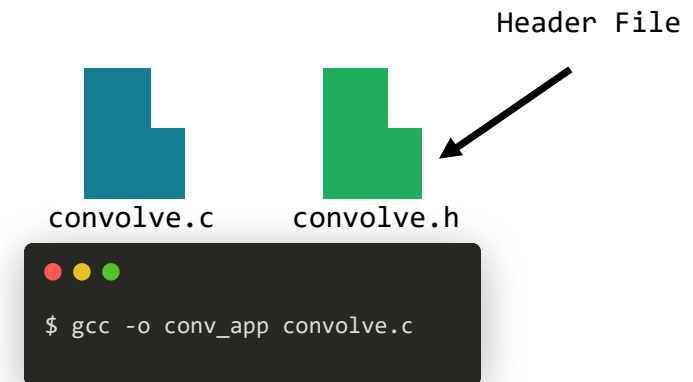
$$\begin{bmatrix} 1 & 4 & 3 \end{bmatrix} \quad \begin{bmatrix} 1 & 4 & 5 & 2 \end{bmatrix} \quad h[1] = 1 * 4 + 4 * 3$$

$$\begin{bmatrix} 1 & 4 & 3 \end{bmatrix} \quad \begin{bmatrix} 1 & 4 & 5 & 2 \end{bmatrix} \quad h[2] = 1 * 1 + 4 * 4 + 5 * 3$$

$$\begin{bmatrix} 1 & 4 & 3 \end{bmatrix} \quad \begin{bmatrix} 1 & 4 & 5 & 2 \end{bmatrix} \quad h[3] = 4 * 1 + 5 * 4 + 2 * 3$$

$$\begin{bmatrix} 1 & 4 & 3 \end{bmatrix} \quad \begin{bmatrix} 1 & 4 & 5 & 2 \end{bmatrix} \quad h[4] = 5 * 1 + 2 * 4$$

$$\begin{bmatrix} 1 & 4 & 3 \end{bmatrix} \quad \begin{bmatrix} 1 & 4 & 5 & 2 \end{bmatrix} \quad h[5] = 2 * 1$$



Programmazione

Media Mobile

A **moving average** is a form of a convolution often used in time series analysis to smooth out noise in data by replacing a data point with the average of neighboring values in a moving window: it operates by taking the arithmetic mean of a number of past input samples in order to produce each output sample.

$$\bar{y}_n = \frac{1}{N} \sum_{i=0}^{N-1} x_{n-i} \longrightarrow y_n = (y_{n-1} + x_n - x_{n-N+1})$$

$$x = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

$$\bar{y}_1 = \frac{1}{5}$$

$$x = \begin{bmatrix} 1 & 2 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

$$\bar{y}_2 = (1 + 2) \frac{1}{5}$$

$$x = \begin{bmatrix} 1 & 2 & 3 & 0 & 0 & 0 & 0 \end{bmatrix}$$

$$\bar{y}_3 = (1 + 2 + 3) \frac{1}{5}$$

$$x = \begin{bmatrix} 1 & 2 & 3 & 5 & 0 & 0 & 0 \end{bmatrix}$$

$$\bar{y}_4 = (1 + 2 + 3 + 5) \frac{1}{5}$$

$$x = \begin{bmatrix} 1 & 2 & 3 & 5 & 8 & 0 & 0 \end{bmatrix}$$

$$\bar{y}_5 = (1 + 2 + 3 + 5 + 8) \frac{1}{5}$$

$$x = \begin{bmatrix} 1 & 2 & 3 & 5 & 8 & 13 & 0 \end{bmatrix}$$

$$\bar{y}_6 = (2 + 3 + 5 + 8 + 13) \frac{1}{5}$$

$$x = \begin{bmatrix} 1 & 2 & 3 & 5 & 8 & 13 & 21 \end{bmatrix}$$

$$\bar{y}_7 = (3 + 5 + 8 + 13 + 21) \frac{1}{5}$$

```
$ gcc main.c moving_average_filter.c -o maf_app
$ maf_app
```