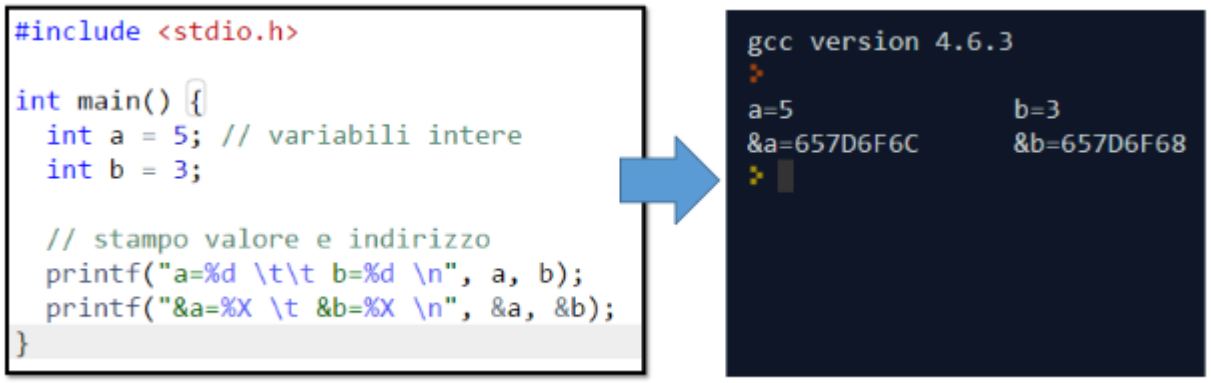
3 - Puntatori

Variabili

Si definisce **variabile** una porzione di memoria destinata a contenere dei valori acquisiti, elaborati o prodotti da un algoritmo.
Una variabile si descrive con:

- Nome
- Tipo
- Indirizzo di memoria → esso non viene esplicitamente dichiarato come i due precedenti
  Come si fa quindi a risalire all'indirizzo di memoria di una variabile?
  Nel *C* si usa l'**operatore d'indirizzo** `&`, il quale viene usato accanto a una variabile, restituendo l'indirizzo di memoria di quest'ultima.



  Il nome di una variabile viene quindi definito come **alias**, permettendoci di riferirci ad un indirizzo di memoria dell'area dati del programma.

Puntatori

I puntatori sono variabili *particolari*, la loro particolarità sta nella possibilità di memorizzare **esclusivamente** indirizzi di memoria.
Normalmente le variabili contengono un valore compatibile col tipo della variabile, mentre le variabili di tipo puntatore contengono proprio un indirizzo di memoria.

Dichiarazione

Una variabile puntatore si dichiara con:

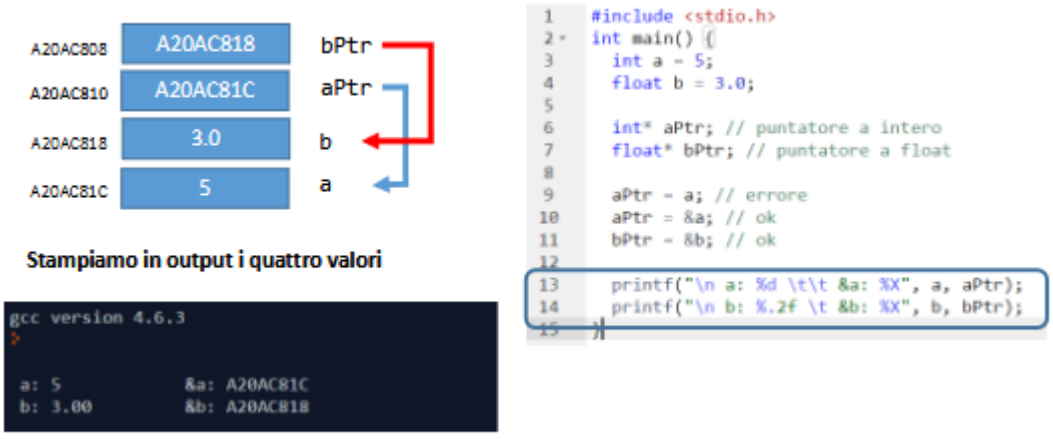
```
<tipo_primitivo>* <nome_variabile>
int* a
int *a
```

La posizione dell'asterisco nei due esempi è indifferente.
La variabile analizzata (*int *a*) detiene:

- **nome**: a
- **tipo**: puntatore ad intero
- **valore**: inizialmente casuale, successivamente deve essere un indirizzo di una cella di memoria in cui risiede un'altra variabile con un valore di quel tipo
- **indirizzo**: definito dal compilatore

Assegnazione

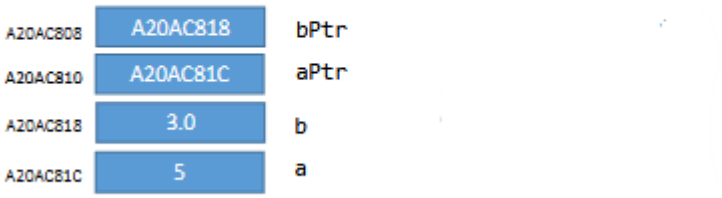
Come abbiamo capito per assegnare un valore ad una variabile puntatore dobbiamo assegnare un indirizzo, ma ciò non è possibile **direttamente**, poiché non possiamo conoscere direttamente gli indirizzi della macchina.
Per ovviare a questo problema si usa **l'operatore d'indirizzo**.



Operatore di indirezione

L'operatore di indirezione serve a **risalire** al valore memorizzato nella cella di memoria puntata dalla variabile.
Esso si rappresenta con `*` davanti al nome della variabile, attraverso esso si può risalire al valore memorizzato nella cella di memoria identificato dall'indirizzo.

- **In questo caso il valore di *bPtr è uguale a 3.0**



Note fondamentali

Bisogna prestare particolare attenzione per non confonderci con l'uso dei vari `*`:

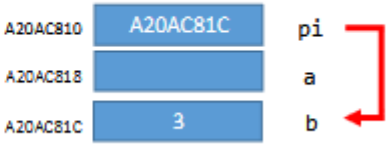
- **Asterisco per dichiarazione variabile**: `float* bPtr`, l'asterisco viene usato per dichiarare una variabile di tipo puntatore.
- **Asterisco per operatore di indirezione sul puntatore**: `*bPtr`
- L'asterisco in una espressione funge da **operatore di dereferenziazione**: `b=*pi`
  Gli operatori di **dereferenziazione** (*) e di **indirizzo** (&) sono uno l'inverso dell'altro.
  • data la dichiarazione int a → *&a equivale a scrivere a (Il valore memorizzato nell'indirizzo di memoria della variabile a == a)
  • data la dichiarazione int *pi → &*pi ha valore uguale a pi (L'indirizzo di memoria del contenuto puntato dal puntatore pi == pi)

Le variabili puntatore devono essere **inizializzate**, di norma il compilatore assegna un indirizzo random alle variabili puntatore, il programmatore tuttavia non saprà cosa è memorizzato all'interno di quella locazione di memoria. Se vengono eseguite operazioni con puntatori **non inizializzati**, si rischia di rendere **corrotta la memoria** e causare problemi di esecuzione.

FORMA SBAGLIATA:

- **Esempio**

```
int a; int* pi; // puntatore non inizializzato
int b = 3; // variabile inizializzata
a = *pi;
*pi = 500;
```



FORMA CORRETTA:

```c
#include <stdio.h>

int main() {
    int a;
    int b = 3;      // Variabile inizializzata
    int* pi = &b;   // Puntatore inizializzato all'indirizzo di b

    a = *pi;        // a assume il valore di b (quindi a = 3)
    *pi = 500;      // Modifica il valore di b tramite il puntatore

    printf("a = %d\n", a);
    printf("b = %d\n", b);

    return 0;
}
```

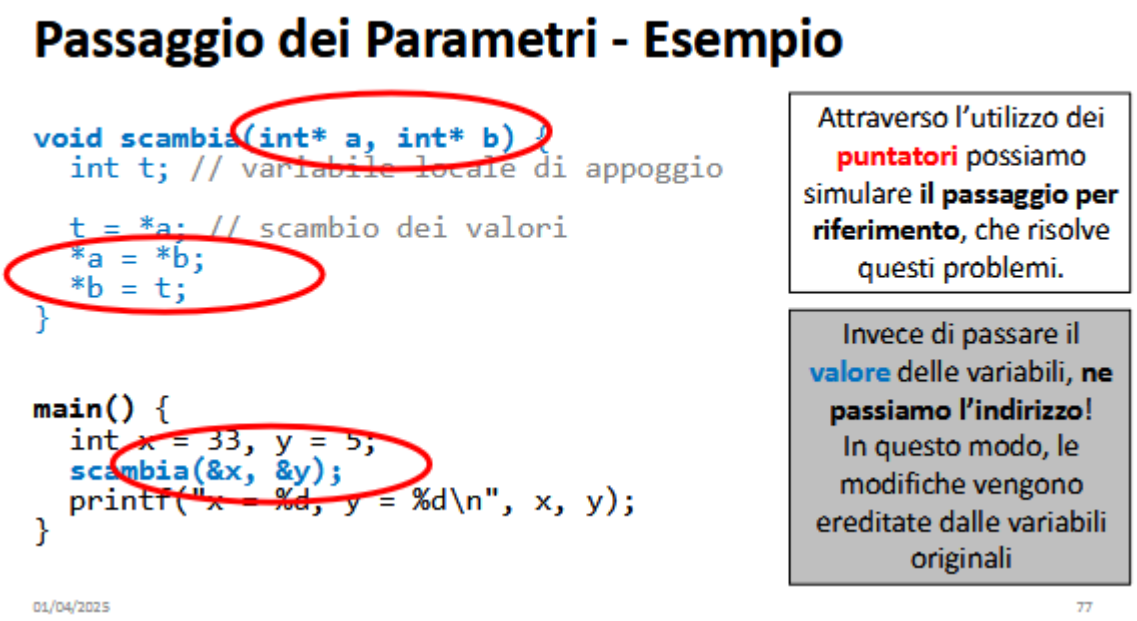
Output atteso:

```
ini

a = 3
b = 500
```

Oppure si può impostare la variabile puntatore direttamente a **NULL**.

Scopo dei puntatori

La principale funzione dei puntatori viene applicata nel **passaggio dei parametri**, quando non è possibile farlo per valore.
Il valore del parametro attuale viene copiato nel parametro formale (che è una variabile locale della funzione). Questo è il metodo più **sicuro** per evitare modifiche accidentali ai parametri; se all'interno di una funzione effettuiamo modifiche ai valori dei parametri, queste modifiche vengono perse.



Attraverso l'uso dei puntatori possiamo simulare il passaggio per **riferimento**, passando alla **funzione l'indirizzo delle variabili** al posto dei loro valori. Questo serve necessariamente quando si ha il bisogno di **modificare** i valori dei parametri passati.

Un ulteriore vantaggio è la possibilità di far **restituire** alle funzioni **più di un valore**.
Esempio:

```
int scambio-somma(int* a, int* b){
    int t, somma;
    t=*a; //scambio valori
    *a=*b;
    *b=t;

    somma = *a+*b;
```

```
    return somma;
}
```

In questo esempio è vero che si restituisce un singolo valore intero, ma nello specifico stiamo restituendo tre valori, poiché scambia anche il valore di *a* e *b*.

Aritmetica dei puntatori

L'aritmetica dei puntatori è il meccanismo utilizzato per accedere tramite i puntatori agli elementi di un array. Infatti come già sappiamo, il nome dell'array è il puntatore al suo primo elemento.

Cosa stampa?

```c
#include <stdio.h>

int main() {
  int array[10]; // dichiaro un array di interi
  int* p_array; // dichiaro un puntatore a un intero

  p_array = &array[0];

  printf("Nome dell'array: \t\t%X \n", array);
  printf("Indirizzo del primo elemento: \t%X \n", &array[0]);
  printf("Puntatore all'array: \t\t%X \n", p_array);
}

```

```
gcc version 4.6.3

Nome dell'array:                A9FBE990
Indirizzo del primo elemento:   A9FBE990
Puntatore all'array:            A9FBE990
```

Tutti e 3 gli output sono equivalenti, come si può notare.
Attraverso l'aritmetica dei puntatori possiamo utilizzare i puntatori in espressioni aritmetiche, con una forte relazione tra puntatori ed array.
Ex:

```
int* arrayPtr=&array[0]; //puntare al primo elemento
arrayPtr=arrayPtr+4
```

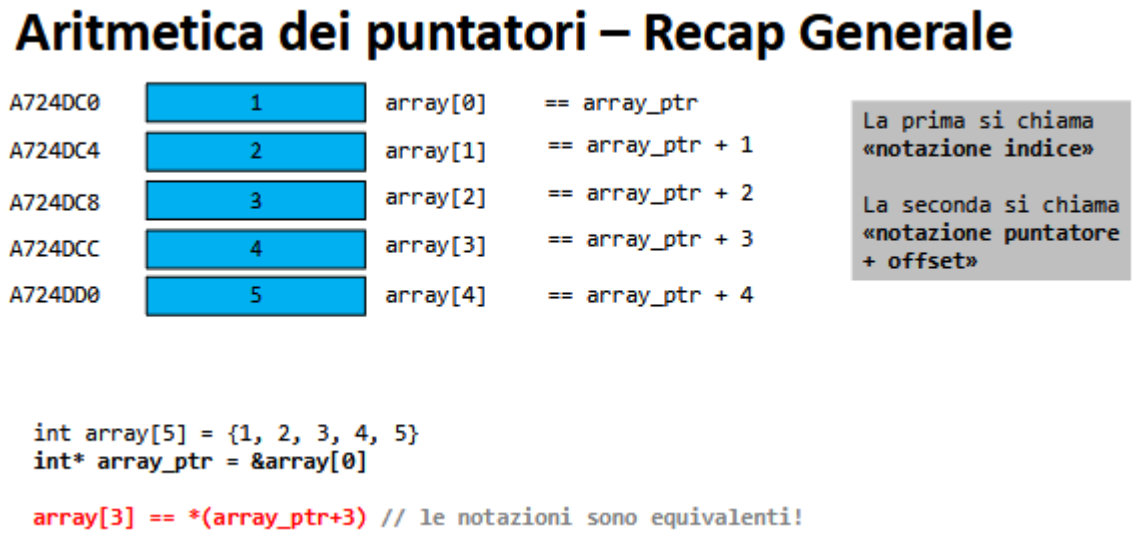
Con questa espressione *arrayPtr* punterà al quinto elemento dell'array, non stiamo incrementando il puntatore di 4 byte ma si va a **puntare all'elemento di posizione 4 dell'array** e arrayPtr conterrà il nuovo indirizzo di memoria di quella cella.
Con l'aritmetica dei puntatori quindi possiamo spostarci all'interno dell'array in qualsiasi situazione.

Regola generale

Dato un array di qualsiasi tipo e dato un puntatore assegnato al primo elemento dell'array, utilizzando l'aritmetica dei puntatori si può accedere in modo alternativo all'i-esimo elemento dell'array, attraverso:

- L'operatore **sizeof** sul tipo di dato: `sizeof(int)=4`
- La moltiplicazione della dimensione del dato per il numero dell'elemento che si vuole accedere: `i=3, address=12, address=i*size`
- L'aritmetica per accedere all'elemento. Il risultato ottenuto sarà equivalente: `array[i]==&arrayPtr+address;`

Se volessimo accedere al valore dei singoli elementi dell'array senza usare una variabile indice ma con l'aritmetica dei puntatori, bisogna usare **l'operatore di indirezione**, in modo tale da accedere al contenuto dei puntatori.




```
int array[5] = {1, 2, 3, 4, 5}
int* array_ptr = &array[0]

array[3] == *(array_ptr+3) // le notazioni sono equivalenti!
```

Passaggio degli array

Quando sono utilizzati come parametri nelle funzioni, gli array vengono passati automaticamente per riferimento.
Ora riusciamo a capire meglio il perché, dato che il nome dell'array è un **puntatore al primo elemento**, quindi passandolo ad una funzione stiamo **implicitamente** passando l'indirizzo.

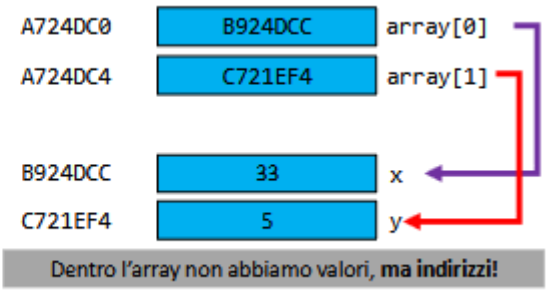
Es:
Gli array possono essere passati in due modi:
`int sum(int v[], int n)`
`int sum(int* v, int n)`
L'array può essere passato alla funzione in entrambi i modi. La scelta è personale. La prima è probabilmente più **leggibile** e più comprensibile. La seconda è più **utile** se si utilizza l'aritmetica dei puntatori dentro la funzione.

Vettori dei puntatori

Dato che i puntatori sono delle variabili a tutti gli effetti, contengono solo indirizzi, possono essere inseriti in **array o strutture**.

• **Esempio**

```
int x = 33; // dichiaro i valori int
y = 5;
// array di puntatori ad interi
int* array[2] = {&x, &y};
```



Quando si utilizzano i vettori di puntatori è necessario utilizzare la notazione con il doppio operatore di indirezione $(**)$.
Il primo operatore di indirezione serve a **risalire all'indirizzo di memoria** del primo elemento. Il secondo operatore di indirezione serve a **risalire al suo valore**, che è quello che ci serve per eseguire le operazioni che ci interessano `*(*a)`.

Anche le stringhe sono array di puntatori, poiché una stringa è un **array di char**, una stringa quindi è un puntatore al primo carattere, in base alle affermazioni precedenti.

Restituzione di un puntatore

Dato che i puntatori sono variabili, di conseguenza possono essere restituiti; per far ciò però bisogna seguire degli **accorgimenti** particolari.
Il motivo di questi accorgimenti è l'**allocazione della memoria**. Questo perché in *C* la memoria viene gestita **staticamente**. Ciò significa che dobbiamo conoscere a priori quanto saranno grandi le variabili utilizzate; per questo stesso motivo tutte le variabili devono essere dichiarate ed avere un tipo, prima dell'esecuzione.
Con i puntatori questo crea un limite, dato che non possiamo sapere quanto esso sarà grande. La soluzione a questo problema è l'uso di una allocazione **dinamica della memoria**, attraverso la funzione `malloc()/calloc()$.
Queste due funzioni servono a riservare una quantità di memoria, **non definita a priori**, al nostro programma.

Malloc e Calloc

La funzione **calloc**, inizializza il buffer di memoria, essa accetta **due parametri**:

- *n* $\rightarrow$ il numero di blocchi
- *size* $\rightarrow$ la dimensione di ogni blocco
  Alloca la memoria per n blocchi, ogni blocco ha una grandezza di byte pari al valore di size.
  La funzione calloc è più **lenta** di malloc.
  La funzione **malloc**, lascia la memoria non inizializzata ed è più **veloce** della calloc.
  Essa accetta un **solo parametro**:
- La dimensione in byte dello spazio di memoria che ci interessa riservare

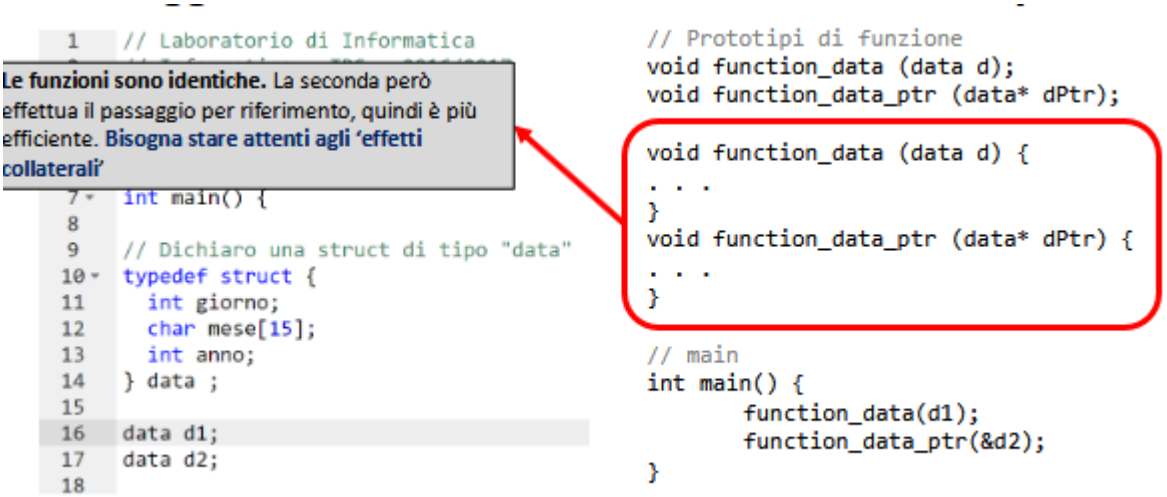
```
char* generatePassword (char* nome, char* cognome){
    //alloco un numero sufficiente di caratteri
    char* s= (char*) calloc(10, sizeof(char)); //chiede di allocare
dinamicamente la memoria per 10 caratteri

    return s; //restituire la stringa
}
```

Puntatori costanti

Si preferisce sempre il passaggio di valore per non modificare dati che potrebbero in futuro servirci, nel caso dell'array però questo non è possibile o non conveniente.
Per non modificare il contenuto di strutture che vengono passate per riferimento usiamo il modificatore **const**, questo modificatore indica al compilatore di **controllare possibili accessi in scrittura** al contenuto puntato.
Il puntatore sarà usato quindi solo in lettura, senza andare a modificare una struttura che deve rimanere intatta.

Passaggio di parametri nelle strutture

Le struct, come le variabili, vengono passato per valore di default, ovvero viene effettuata la copia dell'intera struct.
La copia di un intera struttura però risulta molto **dispendiosa**, per questo in strutture di grande dimensione si preferisce il passaggio per riferimento, si passa l'indirizzo (tramite puntatore) della struttura alla funzione. Anche in questo caso la struttura sarà soggetta a possibili **cambiamenti**, volendo evitare questo inconveniente si usa sempre il modificatore $const$.



Lista

Una lista è una struttura dati che contiene una sequenza di elementi tutti dello stesso tipo. Ogni elemento è caratterizzato da una posizione all'interno di essa.
La lista è simile ad un array che può **cresce dinamicamente**.
Una lista può essere implementata in diversi modi:

- La più utilizzata è l'implementazione con **puntatori** e l'allocazione dinamica della sua memoria
- Ogni elemento della lista mantiene un riferimento all'elemento successivo
  Possiamo aggiungere, modificare ed eliminare qualsiasi elemento all'interno senza problemi.

Operazioni nella lista

- `isEmpty()` : restituisce true se la lista è vuota altrimenti false
- `add(element)` : aggiunge un elemento in coda alla lista
- `addAt(i, elem)` : aggiunge un elemento in posizione i
- `set(i, elem)` : sostituisce l'elemento in posizione i
- `removeAt(i)` : rimuovi l'elemento in posizione i

- `size()` : restituisce il numero degli elementi in lista
- `get(i)/clear()/find(elem)` :
  - restituisce l'elemento in posizione i
  - rimuove tutti gli elementi
  - restituisce la prima posizione nella quale si trova elem, altrimenti -1

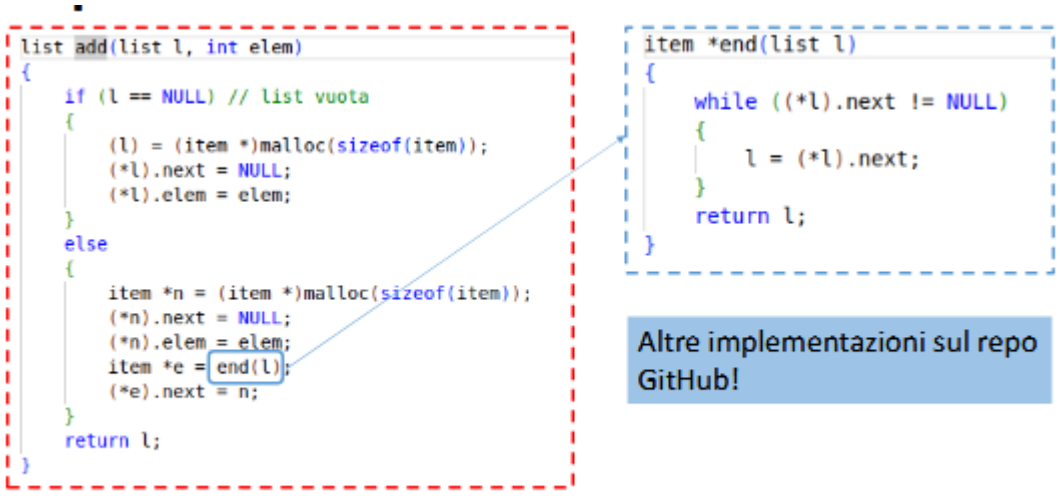
  Il **nodo** della lista può essere rappresentata da una *struct* che contiene un l'elemento ed il puntatore all'elemento successivo.
  Ex:

```
typdef struct list_item{
    int elem;
    struct list_item *next;
}item;
```

Per convenzione si definisce il tipo lista come un puntatore ad **item typdef item *list**.
Alcune di queste funzioni utilizzate nella lista restituiscono il puntatore lista stesso, poiché alcune di queste modificano direttamente *list*, ed essendo list un puntatore, ci si facilita restituendo tutta la lista evitando la previsione per il passaggio di riferimento dell'intera list.

Implementazione

Esempio slide:



Esempio GitHub:

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

#ifndef LISTA_INT_C
#define LISTA_INT_C

typedef struct list_item
{
    int elem;
```

```
    struct list_item *next;
} item;

typedef item *list;

item *at(list l, int i);

item *end(list l);

int size(list l);

item *at(list l, int i)
{
    int pos = -1;
    item *p = l;
    while (pos < i && p != NULL)
    {
        pos++;
        if (pos < i)
        {
            p = (*p).next;
        }
    }
    if (pos == i)
    {
        return p;
    }
    else
    {
        return NULL; //in caso diamo un indice troppo grande della
lunghezza della lista
    }
}

item *end(list l)
{
    while ((*l).next != NULL)
    {
        l = (*l).next;
    }
    return l;
}

int isEmpty(const list l)
{
    if (l == NULL)
        return 1;
    else
        return 0;
}
```

```
list add(list l, int elem)
{
    if (l == NULL) // list vuota
    {
        (l) = (item *)malloc(sizeof(item));
        (*l).next = NULL;
        (*l).elem = elem;
    }
    else
    {
        item *n = (item *)malloc(sizeof(item));
        (*n).next = NULL;
        (*n).elem = elem;
        item *e = end(l);
        (*e).next = n;
    }
    return l;
}

list addAt(list l, int i, int elem)
{
    if (i == 0)
    {
        item *n = (item *)malloc(sizeof(item));
        (*n).next = (*l).next;
        (*n).elem = elem;
        l = n;
    }
    else
    {
        item *p = at(l, i - 1);
        if (p == NULL)
        {
            printf("[WARNING] Out of bound %d.\n", i); //indice non valido
        }
        else
        {
            item *n = (item *)malloc(sizeof(item));
            (*n).next = (*p).next;
            (*n).elem = elem;
            (*p).next = n;
        }
    }
    return l;
}

int size(const list l)
{
    int size = 0;
```

```
    item *p = l;
    while (p != NULL)
    {
        size++;
        p = (*p).next;
    }
    return size;
}

int get(list l, int i)
{
    item *p = at(l, i);
    if (p == NULL)
    {
        printf("[WARNING] Out of bound %d.\n", i);
    }
    else
    {
        return (*p).elem;
    }
}

list removeAt(list l, int i)
{
    if (i == 0)
    {
        item *tmp = (*l).next;
        free(l); //libera l'aria di memoria puntata da un puntatore
        l = tmp;
    }
    else
    {
        item *p = at(l, i - 1);
        if (p == NULL)
        {
            printf("[WARNING] Out of bound %d.\n", i);
        }
        else
        {
            item *tmp = (*p).next;
            (*p).next = (*p).next->next; //arrow point del successivo di
quello succesivo
            free(tmp);
        }
    }
    return l;
}

int find(list l, int elem)
{
```

```
    int s = size(l);
    for (int i = 0; i < s; i++)
    {
        if (get(l, i) == elem)
        {
            return i;
        }
    }
    return -1;
}

void set(list l, int i, int elem)
{
    item *p = at(l, i);
    if (p == NULL)
        printf("[WARNING] Out of bound %d.\n", i);
    else
        (*p).elem = elem;
}

list clear(list l) //la più complessa poiché deve liberare la memoria da
un gran puntatore
{
    /*bisogna salvarsi il puntatore prima di eliminarlo per poter trovare
      la posizione successiva e così via per ognuno se no perderemmo i
collegamenti
      degli elementi nella lista */
    item *p = l;
    while (p != NULL && (*p).next != NULL)
    {
        item *tmp = (*p).next;
        p = (*tmp).next;
        free(tmp);
    }
    free(l);
    l = NULL;
    return l;
}

#endif
```

Esempio applicato nel main per ogni funzione:

```
#include "lista_int.h"
#include <stdio.h>

void printList(list l)
{
    int s = size(l);
```

```
    printf("\n");
    for (int i = 0; i < s; i++)
    {
        printf("%d ", get(l, i));
    }
    printf("\n");
}

int main()
{
    list l = NULL;
    printf("Is empty: %d\n", isEmpty(l));
    l = add(l, 10);
    l = add(l, 19);
    l = add(l, -4);
    l = add(l, 42);
    l = add(l, 0);
    l = addAt(l, 4, 21);
    printf("Is empty: %d\n", isEmpty(l));
    printf("Size: %d\n", size(l));
    printList(l);
    l = removeAt(l, 2);
    printList(l);
    l = removeAt(l, 0);
    printList(l);
    l = removeAt(l, 2);
    printList(l);
    l = add(l, -25);
    l = add(l, 104);
    l = add(l, -93);
    printList(l);
    printf("Find 104: %d\n", find(l, 104));
    set(l, 3, -104);
    printList(l);
    l = addAt(l, 0, -1);
    printList(l);
    l = clear(l);
    printf("Is empty: %d\n", isEmpty(l));
}
```