Variabili e puntatori

- **int a;** <--- Dichiaro la variabile a
- **&a;** <--- Indica l'indirizzo della variabile a.
- ***a;** <--- Deferenzio **a, quindi (*a**) sta ad indicare l'oggetto puntato da quel preciso indirizzo

---

Passo la variabile a per valore

Eventuali modificihe alla variabile, prendendo in questo caso **a** come esempio, ho le modifiche si ripercuotarenno solamente sulla variabile interna del programma.

Cioè verrà effettuata una *copia* della variabile **a**, sempre in questo caso, e tutte le operazioni come modifiche di ogni tipo avranno solamente ripercussioni "interne" al programma. **nome_funzione(a);**

Passo la variabile per riferimento

In questo caso, invece di passare una copia della variabile, passo alla funzione l'indirizzo della variabile stessa. **nome_funzione(&a);**

In questo modo tutte le operazioni che verranno effettuate all'interno della funzione su quella specifica variabile, si ripercuoteranno anche sul valore originario della variabile, visto che non abbiamo passato alla funzione una copia del suo valore, ma il suo indirizzo fisico di memoria.

In questo caso deve essere differente anche la sintassi per quanto riguarda la dichiarazione della funzione. Cioè, quando chiamo la funzione gli passo l'indirizzo della variabile, ma la funzione cosa ci deve fare con quell'indirizzo? Lo deferenzia, dunque la struttura della funzione sarà la seguente:

int function (int *a){}

In quanto nel nostro caso avevamo dichiarato la variabile a come intero, dunque deferenziando il suo indirizzo, quello che devo ottenere è ancora un intero.

Dichiarazione puntatore

Per dichiare un puntatore, posso utilizzare la seguente sintassi

- **int a;** <--- Dichiaro la variabile a
- **int b;** <--- Dichiaro il puntatore b
- **b = &a**; <--- Associo al puntatore b l'indirizzo di memoria di a. In questo modo b punterà allo stesso indirizzo di a, e potrò evitare di dover scrivere **function(&a)**, in quanto ora posso utilizzare la variabile b, e dunque **function(b)**.

Utilizzare **function(&b)** in questo caso non è corretto, in quanto equivale ad *indirizzo indirizzo*. Dunque quando voglio passare ad una variabile un puntatore, devo omettere il segno dell'indirizzo. Perché?

- **int *b;** <--- Dichiaro il puntatore **b**
- **b;** <--- *b* rappresenta l'indirizzo di memoria del puntatore
- ***b;** <--- **b* mi permette di deferenziare l'indirizzo, e dunque accedere all'elemento puntato

---

Stringhe

Le stringhe in C sono dei **vettori di caratteri** terminati da **\0**. E' possibile dichiarare una stringa in due differenti modi:

- **char *s = "stringa";** | Il **tipo di dato s** è un puntatore a char, e **sizeof** dipende dalla dimensione degli indirizzi. Se si prova a modificare la stringa **s**, cosa succede? Il comportamento **dipende dal compilatore**, in quanto se la stringa verrà allocata dal compilatore bella sezione **.rodata** ed il sistema operativo marcherà questa regione di memoria come di sola lettura, allo il programma verrà terminato
- **char s[]= "stringa";** | Il **tipo di dato di s** è **char[8]**, sizeof => 8 bytes (7 caratteri + terminatore '\0')

In entrambe le dichiarazioni, **s** è una variabile che contiene l'indirizzo di base del vettore. Quindi per passare come argomento una stringa ad una funzione, quest'ultima deve essere dichiarata in modo da **deferenziare** l'indirizzo (es. *def nome_funzione*(char ***s**))

Accesso valori stringhe

Per accedere al valore di una stringa, bisogna utilizzare la sequente sintassi (e fare il seguente ragionamento). Se ho una stringa s, dichiarata in uno dei due modi visti sopra, **s** conterrà l'indirizzo di base del vettore, cioè l'indirizzo del primo elemento della stringa. Certamente nello sviluppo dei programmi saremo interessati a lavorare con l'intera stringa, cioè con tutti gli elementi della stringa, magari per contare il numero di elementi della stringa e così via. Come procedo allora?

L'idea è la seguente:

- **s** indica l'indirizzo base della stringa, dunque ***s** restituisce il valore del primo elemento della stringa.
- Per accedere all'elemento successivo, mi basta incrementare l'indirizzo di base, **s**, dunque mi basta scrivere **++s** e dunque il successivo ***s** restituirà il valore del secondo elemento e così via
- Per *controllare* di essere arrivati alla fine della stringa, basta sfruttare una delle proprietà di questo "tipo di dato". Dobbiamo ricordare che ogni stringa termina con il carattare **\0**, dunque basterà controllare che, con l'incremento di **s**, il valore di ***s** sia diverso da **\0**

E' esattamente quello che aabbiamo visto a lezione e che è presente nel file **str.c**. Qui riproposto in modo da capire meglio.

```
int str_len(const char *s) {
int len;
for (len = 0; *s != '\0'; ++len){
 ++s;
}
return len;
```

Il codice è molto semplice. Si accede all'indirizzo del carattere successivo e si incrementa il contatore **len**, fin quando l'elemento del carattere al quale si sta accedendo è diverso da **\0**, altrimenti significa che ci troviamo alla fine. Il funzionamento è identico per concatenazione di stringhe, uguaglianza di stringhe e così via. Capire bene questa cosa è cruciale.

Allocazione Memoria

Per l'allocazione dinamica della memoria, posso utilizzare le API fornite dalla funzione **malloc()**. Si tratta di funa funzione che prende in ingresso il numero di bytes da allocare e restituisce un puntatore **void*** che può essere NULL se non si riesce a soddisfare la richiesta. Bisogna sottolineare che la memoria allocata *non*

necessariamente viene inizializzata (dipende da molti fattori).

```
void *p = malloc(8);
if (p == NULL) {
    printf("Unable to allocale memory.\n");
    exit(1)
}
...
```

In questo modo vado ad allocare 8 bytes, e tramite il puntatore ***p** vado a controllare se effettivamente la memoria sia stata allocata, cioè se il "valore" (sarebbe l'indirizzo) sia uguale a NULL.

Oltre alla funzione **malloc()** abbiamo a disposizione la funzione **free()** che ci permette di liberare la memoria precedentemente allocata. Dopo l'utilizzo della funzione **free()** è consigliato reimpostare a NULL il puntatore.

```
...
free(p);
p = NULL;
```

Strutture

Le strutture del C sostanzialmente permettono l'aggregazione di più variabili, in modo simile a quella degli array, ma a differenza di questi non ordinata e non omogenea (una struttura può contenere variabili di tipo diverso). Per denotare una struttura si usa la parola chiave **struct** seguita dal nome identificativo della struttura, che è opzionale.

```
// dichiarazione della struct
struct libro
{
  char titolo[100];
  char autore[50];
  int anno_pubblicazione;
  float prezzo;
};
//dichiarazione dell'istanza biblio struct libro biblio;
```

È anche possibile **inizializzare i valori alla dichiarazione**, mettendo i valori (giusti nel tipo) compresi tra parentesi graffe:

```
struct libro biblio = {"Guida al C", "Fabrizio Ciacchi", 2003, 45.2};
```

Per **accedere alle variabili** interne della struttura si usa l'operatore punto '.'. Una variabile interna può essere poi manipolata come qualsiasi altra variabile:

```
// assegna un valore al prezzo del libro
biblio.prezzo = 67.32;

// assegna ad una variabile int l'anno di pubblicazione del libro
int anno = biblio.anno_pubblicazione;

// stampa il titolo del libro
printf ("%s n", biblio.titolo);
```

L'operatore **p->f** equivale a **"(*p).f"**, quindi vale solamente se si opera con i puntatori e si passa la struttura ad una funzione per riferimento e non per valore. Esempio preso dal file **struct.c** della **Lezione 5**.

```
void coords_print_ref(struct coords *c)
{
    printf("x=%d, y=%d\n", c->x, c->y);
    c->y = -5
}
...
coords_print_ref(&indirizzo_struttura);
```

Matrici

Per dichiarare una matrice si utilizza la seguente sintassi

```
short m[3][5]; /* Matrice 3x5 di interi a 16 bit */
```

Dichiarata in questo modo, la memoria della matrice p disposta in maniera continua per righe, ed **m** rappresenta l'indirizzo di base di un vettore di 3x15 elementi. Accedente all'elemeto in posizione i, j utilizzando la sintassi con le doppie parentesi quadre **m[i][j]**, il compilatore calcola l'offset corrispondente dell'elemento nella seguente maniera:

```
m[i][j] => *(m + 5i + j) /* il 5 indica il numero delle colonne in questo caso */
```

Ciò significa che, sfruttando l'aritmetica dei puntatori, l'indirizzo è calcolato correttamente qualsiasi sia il tipo di dato degli elementi della matrice.

Matrici con malloc()

Ricorrendo all'allocazione dinamica utilizzando **malloc()**, sorge il problema di comprendere come allocare la matrica e come covertire il puntatore che la funzione **malloc()** restituisce. Per l'allocazione, affinché la memoria sia contigua, si può utilizzare direttamente spazio per **i*j** elementi.

```
void *ptr = malloc(i*j*sizeof(short);
```

malloc() restituisce un puntatore di tipo **void***, dunque per accedere alla memoria è necessario convertire il tipo del puntatore in maniera opportuna.

Convertendolo ad un puntatore di tipo **short***, quello che otteniamo è un modo per accedere alla matrice come se fosse un array.

```
short *ptr = (short *) malloc(i*j*sizeof(short));
```

Date le coordinate **i** e **j**, per accedere all'elemento tuttavia è necessario fare il calcolo dell'offset in modo esplicito

```
short (*m)[5] = (short (*)[5]) ptr;
```

In tal modo è possibile utilizzare la sintassi con le doppie parentesi quadre, quindi accedere all'elemento **m[i][j]** in maniera corretta. Come si può vedere, è necessario specificare il numero di colonne proprio perché questo compare nela formula che viene usata per calcolare l'offset corretto (nel nostro caso 5 moltilica i nella formula).

Liste (struttura dati dinamica)

Come sappiamo, *l'Array* è una struttura dati statica. Possiamo accedere ai suoi elementi, però non ci è possibile modificarli. Al contrario dell'Array, invece, le **liste** sono una struttura dati dinamica. L'implementazione in *C* è diversa da quanto visto in Python. Andremo ad utilizzare le **Strutture** viste già sopra.

Rispetto ad un Array, non abbiamo garanzie sulla continuità della memoria allocata, per questo motivo l'accesso ad una lista sarà generalmen **più lento** rispetto all'accesso ad un Array. Ecco di seguito una semplice struttura per implementare le liste in C:

```
struct n{
    int v;
    struct n *successivo;
};
```

Una **Lista in C** è dichiarata come un insieme di nodi. Ogni nodo rappresenta una struttura con:

- **int v** è il valore del nodo
- **struct n *successivo** è il nodo successivo

Inserimento/rimozione nuovo nodo

Per l'inserimento/rimozione di un nuovo nodo, è cruciale il concetto di *modifica del riferimento*. Se voglio inserire un nuovo elemento in lista tra due valori, cioè se voglio inserire un *nuovo nodo tra due*, devo modificare il riferimento ***successivo** del primo nodo. Ecco il codice per le due operazioni

```
void insert_after_node(struct node *p, struct node *n){
// p indica il nodo precedente, n indica il nodo da inserire

    n->next = p->next;
    p->next = n;
};

void remove_after_node(struct node *p){

    p->next = p->next->next;
};
```

Codici esempio varie funzioni

```
struct node {
    int valore;
    struct node *successivo;
};

struct lista {

// In diversi casi, come le liste ordinate, può essere comodo sapere quale è il primo elemento
// Per poter fare ciò basta utilizzare una struttura di appoggio che memorizza un determinato nodo
// Anche in questo caso, qualora dovessimo voler modificare il primo elemento nella nostra lista,
// bisognerà modificare oltre al riferimento "*successivo" del nodo, anche il riferimento
// "*primo_elemento" della struttura lista

    struct node *primo_elemento
};
```

```
def stampa_lista(struct node *a){

    //Accedo al primo elemento della lista e stampo il suo valore e quello dei nodi successivi
    struct nodo *n;
    for (n=&a; n!=NULL; n=n->successivo) {
    printf("%d\n", n->valore);
    }
}

int main(void){

    struct node d = {6, NULL};
    struct node c = {5, &d};
    struct node b = {4, &c};
    struct node a = {3, &b};

    stampa_lista(&a);
}
```