Programmazione

Andrea Malvezzi

Settembre 2024 - ..

Contents

1 HEAP e STACK **3**
1.1 L'HEAP 3
1.2 Lo STACK 3

2 Puntatori **3**
2.1 Cosa sono e come si definiscono 3
2.2 Operazioni con i puntatori 4
2.2.1 NULL 4
2.2.2 Operatore & 4
2.2.3 Dereferenziazione (*) 4
2.2.4 Istruzione new 5
2.2.5 Istruzione delete 5
2.3 L'istruzione typedef 5

3 Strutture dati dinamiche **6**
3.1 Le linked lists 6
3.1.1 Introduzione all'implementazione delle linked lists . . . 6
3.1.2 Implementare le linked lists 6
3.1.3 Operazioni sulle linked lists 8

1 HEAP e STACK

L'HEAP e lo STACK sono due divisioni della memoria in cui vengono salvati i dati durante l'esecuzione di un programma. I due svolgono due ruoli diversi ma fondamentali:

1.1 L'HEAP

L'HEAP è una porzione di memoria dedicata all'allocazione dinamica delle variabili, quindi mediante istruzioni come new e delete. Grazie a questa dinamicità, qui si possono sfruttare strutture dati di lunghezza indefinita, come liste o alberi, ma con una rapidità limitata.

1.2 Lo STACK

Lo STACK è una porzione di memoria dedicata alla gestione automatica (quindi a cura del compilatore) di variabili locali e simili. Accedere ai dati al suo interno risulta veloce grazie alla filosofia LIFO (Last-In-First-Out).

2 Puntatori

2.1 Cosa sono e come si definiscono

Un puntatore è uno speciale tipo di dato capace di immagazzinare un indirizzo di memoria corrispondente ad un altro dato.
Quando si dichiara un puntatore bisogna specificare il tipo di dato a cui lo si vuole far puntare. Vediamo un esempio pratico:

```
1     int *int_p;          // Questo punta a un intero
2     char *char_p;        // Questo punta a un carattere
3     // E via dicendo ...
```

Inoltre in cpp la dichiarazione di un puntatore ha una sintassi variabile: il * può difatti essere posto dopo il tipo di puntatore o prima del nome di questo. A causa di questa caratteristica bisogna tuttavia prestare attenzione a quando si vogliono dichiarare due puntatori nella stessa riga:

```
1     // Qui r corrisponde a un intero, non a un puntatore!
2     int* p, r;
3
```

```
    // Per due puntatori, occorre usare la seguente sintassi:
    int *p, *r;
```

2.2 Operazioni con i puntatori

Si possono effettuare operazioni diversi con i puntatori, tra cui le più importanti sono:

2.2.1 NULL

Quando si assegna il valore NULL a un puntatore si indica che questo non punta a nessuna cella di memoria. Si usa spesso in seguito all'istruzione delete.

```
    int *p = NULL;       // p non punta a una cella di memoria
```

2.2.2 Operatore &

L'operatore ampersand '&' si usa in due contesti: per ottenere l'indirizzo di una variabile e per creare un riferimento a questa. Il primo caso risulta utile quando si vuole creare un puntatore ad una certa variabile:

```
    int n = 10;
    int *n_p = &n;       // ora n_p punta all'indirizzo di n!
```

Mentre il secondo caso risulta più utile quando si vuole creare un alias di una variabile per modificare quest'ultima mediante reference.

```
    int n = 10;
    int &alias_n = n;    // reference ad n
    alias_n = 20;        // n = 20;
```

La differenza sostanziale tra i due approci sta nella flessibilità: i puntatori possono essere riutilizzati e possono essere annullati, mentre le reference lavorano su una sola variabile definita esattamente alla definizione della reference stessa e non possono essere NULL.

2.2.3 Dereferenziazione (*)

La dereferenziazione si usa in due casi: per dichiarare un puntatore e per accedere all'indirizzo a cui questo sta puntando.

```
1     int value = 42;
2     int *p = &value;     // Dichiarazione puntatore
3     int dereferencedValue = *p;  // Accede al valore 42
```

2.2.4 Istruzione new

L'istruzione new alloca dinamicamente memoria (nell'HEAP), restituendo un puntatore all'area di memoria allocata.

```
1     int *p = new int;
2     *p = 42;
```

Qui fare p = 42 darebbe errore: staremmo assegnando un int a un int*. Invece in questa maniera assegnamo al contenuto della cella di memoria a cui punta p il valore 42.

2.2.5 Istruzione delete

L'operatore delete libera la memoria allocata dinamicamente mediante new.

```
1     int *p = new int;
2     delete p;
3     p = NULL;        // come mai p=NULL?
```

Nell'esempio fornito liberiamo la cella di memoria allocata dinamicamente con int *p = new int. Se lasciassimo *p senza una reference, questo diventerebbe un dangling pointer, ovvero un puntatore contenente un valore assegnato dal compilatore (randomico). Per evitarlo, annulliamo *p con p = NULL.

2.3 L'istruzione typedef

L'istruzione typedef permette di definire un nuovo tipo, utilizzabile durante le dichiarazioni di nuovi dati. Questo risulta particolarmente utile quando si lavora con i puntatori in quanto permette di scrivere quanto segue:

```
1     // fuori dal main, idealmente ...
2     typedef int *p_int;      // definisco tipo p_int
3
4     // nel main, ora...
5     p_int p, q;         // equivalente a int *p, *q
6
```

```
7     // ho dichiarato due puntatori in una sola riga!
8     // Questo permette di evitare confusione
```

3 Strutture dati dinamiche

3.1 Le linked lists

Una linked list corrisponde ad una sequenza di nodi di grandezza dinamica. Questo significa che la lista non avrà una grandezza fissa, ma si adatterà alle necessità del programma scritto da noi (se implementata correttamente).

3.1.1 Introduzione all'implementazione delle linked lists

Per implementare una linked list occorrerà utilizzare la direttiva struct (per definire che cos'è un *nodo*) e i puntatori, per puntare al nodo successivo e/o precedente (in questo caso si parlerebbe di doubly-linked-list). Visualizzato, quanto appena spiegato equivale a qualcosa del genere:

Figure 1: "Head" corrisponde ad un puntatore puntante all'indirizzo in memoria del nodo della lista. Ogni elemento (nodo) contiene inoltre un puntatore all'indirizzo del prossimo della struttura.

3.1.2 Implementare le linked lists

Anzitutto, occorre capire come funziona una linked list ad un livello più profondo. Essendo questa una struttura dati dinamica, di cui non si conosce la grandezza a priori, la si dovrà ingrandire mano a mano che si avanza nel programma. Per farlo, occorrerà sfruttare l'Heap, mediante istruzioni come new e delete, olre che ai puntatori.
Partiamo definendo la struttura dati che useremo per immagazzinare il dato da salvare e l'indirizzo del prossimo item: un **Nodo**.

```
    struct Nodo {
        int valore;
        Nodo *next;      // Definizione circolare
    }
```

In seguito, occorrerà definire un head della lista. Questo oggetto non corrisponderà al primo elemento della lista, ma bensì al puntatore puntante all'indirizzo del primo oggetto stesso. Questo significa che per accedere ai campi effettivi del Nodo, occorrerà usare la dereferenziazione (*).

```
    Nodo *head = new Nodo;    // new alloca memoria nell'Heap

    // -> corrisponde a derefernziare e usare il punto
    head -> value = 10;      // come (*head).value = 10
    head -> next = nullptr; // anche NULL andrebbe bene
```

Al termine del codice, occorre ricordarsi di liberare la memoria allocata con *new*, per evitare **Memory Leaks**.

```
    Nodo *head = new Nodo;
    ...codice...

    // al termine del codice libero la memoria allocata
    delete head;
```

Nel caso in cui si avesse una lista con molteplici elementi, occorrerebbe liberare la memoria di ognuno dei nodi di questa. Per farlo si potrebbe usare la seguente funzione:

```
    void empty(Nodo *&head){
        // itero fino a che non ho il Nodo corrente null
        while(head != nullptr)
        {
            // creo una variabile temp per copiare head
            Nodo *temp = head;
            // sposto head al Nodo successivo
            head = head -> next;
            // cancello temp, che punta al nodo precedente
            delete temp;
        }
    }
```

3.1.3 Operazioni sulle linked lists

Si possono inserire elementi all'inizio della lista, in una certa posizione, eliminare certi elementi, controllare che sia vuota ... qualunque cosa sia utile e si sia in grado di implementare.
A questo link è presente il codice delle operazioni più popolari inerenti alle linked lists.