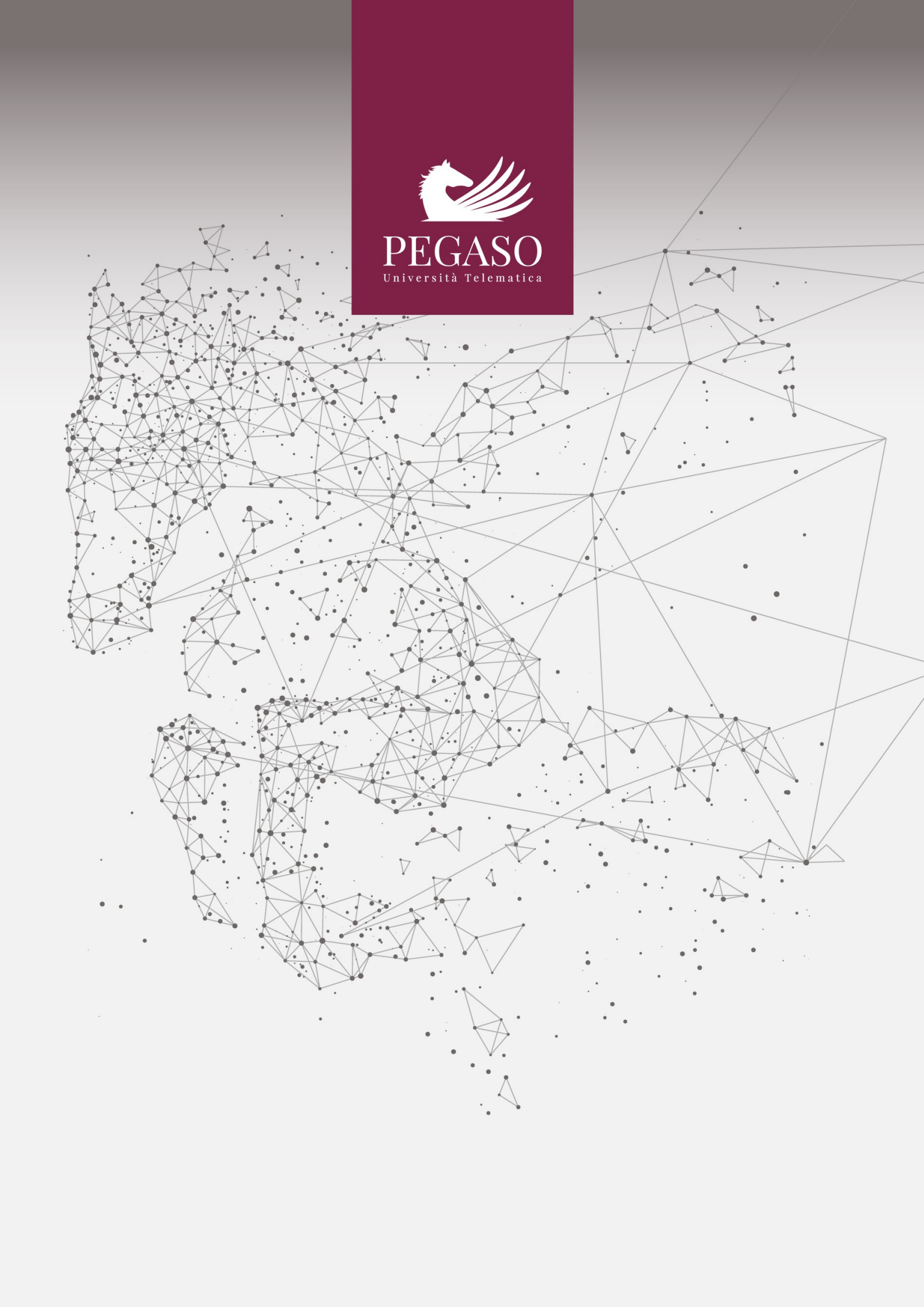
PEGASO
Università Telematica

Indice

1. Pools di memoria

In C/C++ vi sono a disposizione del programmatore, 3 pools di memoria:

- **static**: storage di variabili globali (mantenute in vita per tutta l'esecuzione del programma);
- **stack**: storage di variabili locali (memoria continua, limitata e gestita in modalità automatica);
- **heap**: storage dinamico (memoria non allocata in modalità continua, in grande quantità e gestita in maniera manuale).

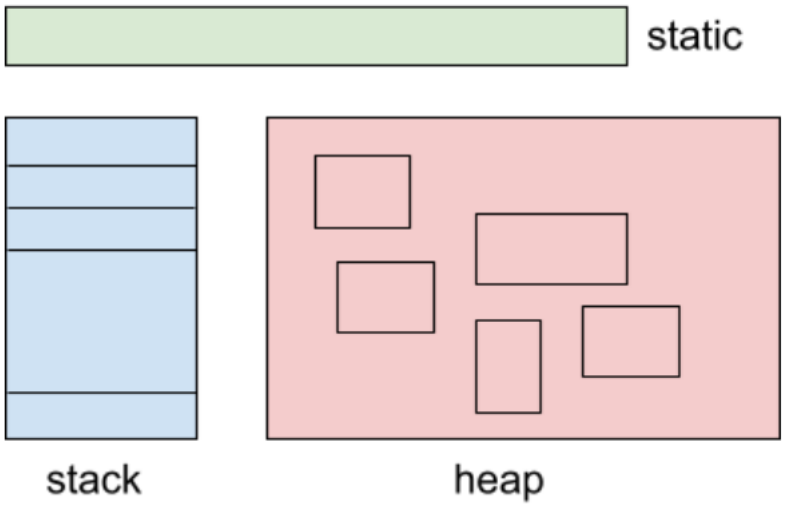

Figura 1 *- Pools di memoria*

Memoria Statica

Persiste per l'intera esecuzione del programma ed è utilizzata tipicamente per memorizzare variabili globali.

Stack

Lo stack è usato per memorizzare variabili all'interno di una funzione (incluso il main); ha una gestione **LIFO** – **Last-In-First-Out**: ogni volta che una funzione dichiara una nuova variabile viene fatto PUSH sullo stack; quando la funzione termina la sua esecuzione, tutte le variabili associate alla funzione sono cancellate dallo stack e la memoria è liberata. Parliamo in questo caso di "scope" locale delle variabili di una funzione.

Lo stack è gestito direttamente dalla CPU in maniera automatica.

Riassumendo:

- lo stack è gestito dalla CPU;

- le variabili sono allocate e de-allocate automaticamente;
- è una risorsa "limitata" (quando si esaurisce si va in **stackoverflow**);
- le variabili nello stack vivono finché vive la funzione che le ha create.

Heap

L'heap è un grande pool di memoria che può essere usato dinamicamente ed è gestito "manualmente": occorre allocare e de-allocare esplicitamente la memoria e se non lo si fa si incorre in un **memory leak** (memoria ancora usata e non disponibile per altri processi). Non vi sono restrizioni sulla dimensione dell'heap (al netto della memoria fisica della macchina).

Riassumendo:

- l'heap è gestito dal programmatore
- l'heap è grande e limitato solo dalla dimensione della memoria fisica
- l'heap richiede un accesso tramite "puntatori" (possono essere acceduti a livello globale da un punto di vista "spaziale" e non di "scope")

Tabella comparativa

Parametero	**Stack**	**Heap**
Tipo di struttura di dato	Struttura di dato lineare	Struttura di dato gerarchica
Velocità di accesso	High-speed	Lenta (rispetto allo stack)
Gestione dello spazio	Lo spazio è gestito in maniera efficiente dal Sistema Operativo: la memoria non è mai frammentata	Lo spazio Heap non è gestito in maniera efficiente: la memoria può essere frammentata
Accesso	Solo per variabili locali	Consente l'accesso alle variabili globali
Limitazioni di spazio	Il limite dipende dal Sistema Operativo	Non vi sono limiti specifici
Resize	Le variabili non ammettono il resize	Le variabili ammettono il resize
Allocazione di memoria	La memoria è allocata in blocchi contigui	La memoria è allocata in modalità randomica
Allocazione	Gestita automaticamente dal compilatore	Gestita manualmente dal programmatore
Deallocazione	Non è richiesta la de-allocazione delle variabili	È richiesta la de-allocazione esplicita da parte del programmatore
Costo	Minore	Maggiore
Problema principale	Shortage of memory	Memory fragmentation

2. Puntatore

Variabile

Essa rappresenta, o astrae, una ben precisa zona della memoria, identificando un tot. di byte in base al tipo della stessa. Per conoscere l'esatto quantitativo di byte, utilizzati per un determinato tipo, è disponibile l'operatore $sizeof()$. Il tipo del valore restituito è $size\_t$. Esso corrisponde a un numero intero senza segno, perciò sempre positivo.

Per conoscere l'indirizzo di una qualsiasi variabile il C/C++ mette a disposizione un operatore apposito: $\&$. Tale ragionamento è valido a prescindere dal tipo della variabile: qualsiasi sia il numero dei byte che la compongono, solo l'indirizzo del byte iniziale verrà tenuto in considerazione.

Puntatore

Un tipo particolare di variabile adatta a memorizzare l'indirizzo di una locazione di memoria, ovvero adatta a memorizzare l'indirizzo di un'altra variabile, che prenderà il nome di variabile puntata.

La dichiarazione, con conseguente definizione, di una variabile puntatore consiste nell'aggiungere il simbolo di asterisco $*$ tra il tipo della variabile e il suo identificatore: $< tipo > * < identificatore >$

Alla definizione viene subito allocato spazio in memoria anche per questo tipo di variabile, la quale conterrà quindi un valore casuale, ovvero un indirizzo casuale.

Un puntatore è dunque un oggetto il cui valore rappresenta l'indirizzo di un altro oggetto o di una funzione.

Nella seguente dichiarazione a è un puntatore ad interi:

$int * a;$

- per ottenere l'indirizzo di un oggetto si usa l'operatore $\&$
- per accedere all'oggetto riferito da un puntatore si usa l'operatore $*$

Esempio di codice con visualizzazione:

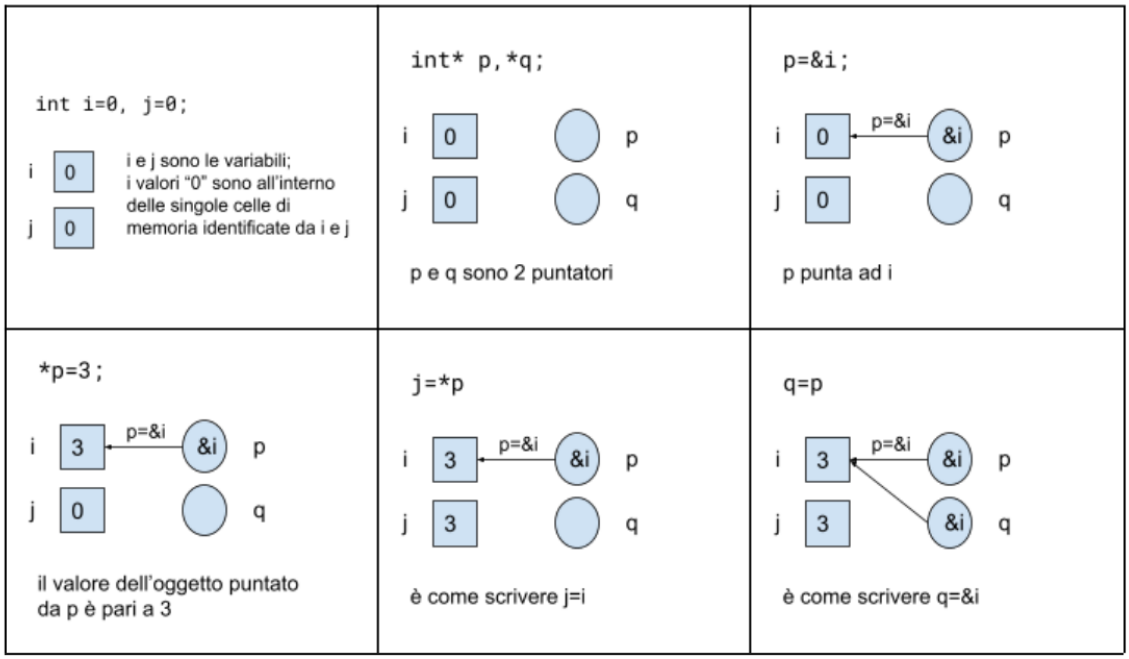

Figura 2 *- Puntatori*

Per evitare problemi, è buona regola annullare (o inizializzare) i puntatori nel momento in cui vengono dichiarati:

```
char *p = NULL;
int x = 25;
int *q = &x;
```

Il C++ supporta l'allocazione dinamica e la deallocazione degli oggetti usando gli operatori new e delete.

Questi operatori allocano memoria per gli oggetti attingendo dall'heap.

Se voglio puntare un intero:

```
int * p = NULL;
p = new int;
*p = 123;
delete p;
```

In questo esempio il puntatore a interi p, non è un numero, ma è un indirizzo. Il tipo di p è $int*$ ed il suo indirizzo è assegnato con $p = new\ int$. Il numero intero invece l'abbiamo assegnato quando facciamo $*p = 123$;

Riassumendo il numero intero $*p$ si trova all'indirizzo $p = new\ int$.

Quando il vettore non serve più possiamo restituire la memoria ormai inutile: $delete\ p$

Se ho un vettore formato da 100 numeri interi, l'indirizzo puntato è quello del primo byte della sequenza dei 100 numeri. Quindi:

```
int * p = NULL;
p = new int[100];
for(int i = 0; i < 100; i++) {
    p[i] = rand();
}
delete []p;
```

3. Array vs Puntatori

Di fatto, gli array sono dei puntatori: quando si dichiara un array, in realtà si dichiara un puntatore, con alcune caratteristiche in più:

- la dichiarazione di un puntatore comporta allocazione di memoria per una variabile puntatore, ma non per la variabile puntata;
- la dichiarazione di un array comporta allocazione di memoria non solo per una variabile puntatore (il nome dell'array), ma anche per l'area puntata, di cui viene predefinita la lunghezza; inoltre, il puntatore viene dichiarato $const$ e inizializzato con l'indirizzo dell'area puntata (cioè del primo elemento dell'array).

Esempi:

- $int * lista$; alloca memoria per la variabile puntatore lista ma non per la variabile puntata da lista
- $int\ lista[5]$; alloca memoria per il puntatore costante lista, alloca memoria per 5 valori di tipo int ed inizializza la lista con $\&lista[0]$

Il fatto che il puntatore venga assunto come costante ($const$), comporta che l'indirizzo dell'array non è modificabile e quindi il nome dell'array non può essere usato come $l-value$ (left value: localizzatore dell'area di localizzazione), mentre un normale puntatore sì.

Inoltre, in un array, l'area puntata può essere inizializzata tramite la lista degli elementi dell'array, mentre in un puntatore ciò non è ammesso. A questa regola fa eccezione il caso di un puntatore a char quando l'area puntata è inizializzata mediante una stringa letterale (per compatibilità con vecchie versioni del linguaggio).

Importante:

Una stringa è una successione di caratteri

Per *null-terminated string* si intende una stringa con il terminatore $\backslash 0$ alla fine (che determina la fine della stringa). Quando in c++ si scrive una stringa (delimitata con le ") in automatico sono *null-terminated* dal compilatore; quindi, una stringa "ciao" è la stessa cosa di

$$\{'c','i','a','o','\backslash 0'\}$$

Consideriamo il seguente codice:

```
char a[] = {'C','i','a','o','\0'};
char a2[] = {'C','i','a','o'};
char b[] = "Ciao";
char b2[5] = "Ciao";
```

Eseguendo un $cout$ sulle varie variabili, otteniamo sempre Ciao.

Attenzione: nel caso di $b2$ occorre specificare come lunghezza 5 perchè va incluso il carattere per il terminatore $\0$ che deve essere inserito in automatico dal compilatore; pertanto, se si scrivesse $b2[4]$ si avrebbe un errore a compile-time.

Se volessi provare a scrivere: $char * c = "Ciao";$ occorre fare attenzione alla versione del compilatore (dalla versione 11 non è ammesso ed occorre usare direttamente $string$). In versioni precedenti invece è consentito e il $cout$ restituisce anche in questo caso Ciao.

Notare che il seguente codice:

```
char * a = "Ciao";
cout <<* a << endl;
cout << a[0] << endl;
```

restituisce in output in entrambi i casi il carattere C

Bibliografia

- Alan Bertossi, Alberto Motresor: Algoritmi e strutture di dati, Città Studi Edizioni, terza edizione
- C. Demetrescu, I. Finocchi, G. F. Italiano: Algoritmi e strutture dati, McGraw-Hill, seconda edizione
- Crescenzi, Gambosi, Grossi: Strutture di Dati e Algoritmi, Pearson/Addison-Wesley
- Sedgewick: Algoritmi in C, Pearson, 2015
- Cormen Leiserson Rivest Stein-Introduzione Agli Algoritmi E Strutture Dati-Prima Edizione