Arrays e Memoria Dinamica

Programmazione Avanzata e Parallela
2022/2023

Alberto Casagrande

Arrays (Come in C)

Sono strutture dati indicizzate per memorizzare più valori dello stesso tipo

Non possono cambiare dimensione

```
int I[10];      // dichiariamo un array di interi di dimensione 10
double D[4];    // dichiariamo un array di float di dimensione 4

D[0] = 9;       // il primo elemento dell'array
D[3] = 3.3;     // l'ultimo elemento dell'array

I[7] = D[7];    // C++ non ha un meccanismo automatico di controllo dei limiti
                // 7 non è un indice valido, ma il compilatore non emette
                // alcun avviso e il programma potrebbe non produrre un errore
```

Arrays e Puntatori (Come in C)

Il valore di una variabile "array" è il puntatore al primo byte dell'area
in cui è memorizzato l'array stesso

```
int A[]={3,0,-4};  // dichiaro un array di 3 valori interi

int* p{A};         // inizializzo 'p' con il puntatore all'area di
                   // memoria di 'A'

*p = 8;            // modifico l'area puntata da 'p', i.e., A[0]
```

Sintassi dei Puntatori e degli Array

La sintassi dei puntatori può essere usata con gli array

```
int A[]={3,0,-4}; int n{2};

*(A+n) = 2+*(A+n);  // cambia l'(n+1)-esimo valore di A
```

L'indicizzazione può essere usata con i puntatori

```
int A[]={3,0,-4}; int *p{A}; int n{2};

p[n] = 2+p[n] // cambia l'(n+1)-esimo valore di A come prima
```

Arrays e Funzioni (Come in C)

Possono essere usati come parametro in una funzione

```
void dummy(double A[]) {
  A[0] = 7;
}
...

double A[] = {2.1, 7.2, 0.0};
dummy(A);
```

Viene passato il **valore** del puntatore all'area di memoria

Stack e Heap

Le variabili e gli array vengono allocati nello *stack*

Esse vengono de-allocate alla chiusura del blocco

Come facciamo a riservare uno spazio di memoria "persistente"?

Con la **gestione dinamica della memoria**

La memoria allocata dinamicamente si trova nella *heap*

La Gestione Dinamica delle Memoria in C

Allocare uno spazio di memoria con `malloc` e deallocarlo `free`

```c
int *p = (int *)malloc(sizeof(int));    // alloco lo spazio di memoria
                                        // necessario a rappresentare un int
                                        // e ne assegno il puntatore a `p`

*p = *p-2;                              // uso lo spazio di memoria
                                        // tramite `p`

free(p);                                // de-alloco lo spazio di memoria
```

Senza `free` la memoria rimane occupata fino alla fine dell'esecuzione

Array "Dinamici"

```
int *p = (int *)calloc(5,sizeof(int)); // alloco lo spazio di memoria
                                       // necessario a rappresentare 5 int
                                       // e ne assegno il puntatore a `p`
...

p[3] = p[3]-2;                         // uso lo spazio di memoria come  se
                                       // `p` fosse un array nello stack
...

free(p);                               // de-alloco lo spazio di memoria
```

Gestione dinamica della memoria in C++

È gestita usando `new` e `delete`

```
int *p = new int(5);    // alloco lo spazio per un `int` e lo inizializzo a 5
auto q = new int();     // alloco lo spazio per un `int`

*q = *p * 5;

delete q;               // de-alloco lo spazio puntato da 'q'
delete p;               // de-alloco lo spazio puntato da 'p'
```

Supporta l'inizializzazione

Array "dinamici" in C++

Si usano `new tipo[]` e `delete[]`

```
int *A = new int[5];     // alloco un array di 5 int
auto F = new (double (*[7])(double));

F[2] = sqrt;



delete[] A;
```

Memory Leak e Segmentation Fault

L'utilizzo di puntatori e memoria dinamica può portare a:

- **segmentation fault**: cerchiamo di accedere a un'area di memoria che non ci è stata assegnata o non esiste

```
int *a{nullptr};

*a = 5;
```

- **memory leak**: perdiamo riferimenti ad aree di memoria che ci sono state attribuite, ma non vengono de-allocate

`Valgrind`

È un framework per l'analisi della gestione della memoria dinamica

Include molti strumenti tra cui *MemCheck* che identifica:

- accessi ad aree di memoria non disponibili
- memory leak
- `delete` doppi
- errori sugli indici degli array

Utilizzo di `valgrind`

1. Compilare il programma con i simboli di *debug* (`-g`)

```
g++ esempio.cpp -g -o esempio
```

2. Invocare `valgrind` con il programma da testare e i suoi parametri

```
valgrind ./esempio
```

Smart pointer

Il C++ offre anche un meccanismo "automatico": gli **smart pointer**

Possimo associare uno spazio di memoria a uno *smart pointer*

Quando nessun smart pointer farà riferimento a questo spazio esso verrà de-allocato

- `unique_ptr` l'area di memoria è puntata da un solo puntatore
- `shared_ptr` l'area di memoria è puntata da più puntatori

Gli `unique_ptr`

```
#include <memory>      // definisce gli smart pointer
...
{
  std::unique_ptr<int> i1{new int(3)};    // alloco un'area di memoria per un 'int',
                                          // la inizializzo a 3 e inizializzo uno
                                          // smart pointer che punta a questa area

  // std::unique_ptr<int> i2{i1};         // ERRORE: gli unique_ptr non si possono
                                          // copiare

  *i1 = *i1 + 3;             // uso i1 come fosse un puntatore "tradizionale"

}          // finito il blocco 'i1' non esiste più e l'area viene de-allocata
```

Gli `shared_ptr`

```
#include <memory>
...

std::shared_ptr<int> i1{new int(4)};

{
  std::shared_ptr<int> i3{new int(3)};  // alloco un'area di memoria per un 'int',
                                        // la inizializzo a 3 e inizializzo uno
                                        // smart pointer che punta a questa area

  i2 = i1;                  // OK: gli shared_ptr si possono copiare. L'area puntata
                            // da i2 non è più raggiungibile e viene de-allocata

  *i1 = *i1 + 3;            // 'i1' e 'i2' puntano alla stessa area di memoria

}         // finito il blocco 'i2' esiste ancora e l'area non viene de-allocata
```