UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

4. Smart Pointers in Cpp

2 esercizi risolti + 1 extra

Tutorato di
Programmazione Avanzata

RELATORE
Imberti Federico

SEDE
Dalmine, BG

Key-points della teoria: La memoria

1. La memoria a disposizione del programmatore è suddivisa in 2 macro-aree:
    a. Memoria del codice, contenente il codice sorgente da eseguire
    b. Memoria dei **dati**, la quale si evolve con il programma, divisa in:
        i. **Stack** (LiFo): dimension fissa; contiene variabili, parametri e indirizzi di ritorno;

```
void example() {
    int x = 10;  // 'x' is stored in stack memory
}  // 'x' is deallocated automatically when the function ends
```

```
{ //Imagine you are in the main method...
    int x = 10;  // 'x' is stored in stack memory
}
// 'x' is deallocated automatically when we leave the block
```

Key-points della teoria: La memoria

1. La memoria a disposizione del programmatore è suddivisa in 2 macro-aree:
    a. Memoria del codice, contenente il codice sorgente da eseguire
    b. Memoria dei **dati**, la quale si evolve con il programma, divisa in:
        i. **Heap** (FiFo): usata per l'allocazione dinamica dei dati; gestita dal programmatore; usata per contenere strutture dati di grandi dimensioni (oggetti, array, …)

```
void example() {
    int* ptr = new int(10);  // Memory for the integer is allocated on the heap
    delete ptr;  // The programmer is responsible for freeing the memory
}
```

Key-points della teoria: Memory management

1. Negli esercizi sui tipi opachi abbiamo visto il meccanismo di allocazione della memoria usato da C, basato su “malloc()” e “free()”;

```c
int* prime = malloc(sizeof(int));
*prime = 11;
printf(“%s”, *prime); //prints: 11
free(prime);
```

2. Cpp impiega un meccanismo simile, basato però su “**new**” e “**delete**”;

```cpp
int* prime = new int;
*prime = 11;
cout << *prime <<endl; //prints: 11
delete prime;
```

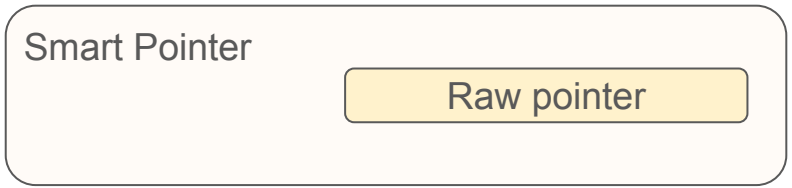
Key-points della teoria: Memory management

- L’utilizzo dei **raw pointer** comporta tuttavia che il programmatore debba gestire in autonomia la memoria, ricordandosi di de-allocare i puntatori, altrimenti si verifica un **memory leak**;
- Cpp mette a disposizione dei modi per gestire automaticamente la memoria quando si entra ed esce da un blocco, rispettivamente allocando gli oggetti in esso e distruggendoli
    - Si tratta degli “Smart Pointers”, di cui ne esistono vari tipi, creati per gestire la memoria lasciando pressoché inalterate le performance;

Key-points della teoria: Memory leak 💦🪠

Key-points della teoria: Smart Pointers (SP)

- Uno smart pointer è un Template Class che viene dichiarato sullo **stack** usando un puntatore raw che punta invece a un'area dell'**heap**;
- Una volta che lo SP viene creato questo assume il **controllo** del puntatore raw passato in input gestendone in autonomia l'allocazione e de-allocazione;
- Per accedere al contenuto dello SP vengono usati i consueti operatori **->** e ***** che tramite overloading restituiscono il puntatore all'interno;
- Un modo intuitivo per vedere gli SP è di immaginarli come dei **wrapper** per un puntatore raw il quale viene incapsulato in essi per essere gestito



Key-points della teoria: Smart Pointers (SP)

Disponiamo di 2 tipi di Smart Pointer:

1. **Unique Pointer**
    a. Ammette l'esistenza di un unico **possessore** del puntatore;
    b. Ownership può essere ceduta ma non sono ammesse copie o condivisioni.
2. **Shared Pointer**
    a. Permette a più possessori di possedere contemporaneamente il puntatore senza crearne **copie**;
    b. Tiene traccia del numero di possessori;
    c. Viene de-allocato solo quando è fuori dallo scope di tutti i possessori.

Key-points della teoria: Smart Pointers (SP)

Disponiamo di 2 tipi di Smart Pointer:

1. **Unique Pointer**

```
unique_ptr <Persona> cristian (new Persona(“Cristian”, 24));
cout << cristian->getNome() << endl; //stampa: Cristian
cout << cristian << endl //Stampa l’indirizzo del raw pointer di Persona
```

2. **Shared Pointer**

```
shared_ptr<Persona> cristian (new Persona(“Cristian”, 24));
cout << cristian->getNome() << endl; //stampa: Cristian

shared_ptr<Persona> copiaDiCristian(cristian);
Cout << cristian.use_count(); //Stampa il numero di possessori: 2
```

Further reading e Approfondimenti

Articolo di Microsoft, semplice e fatto molto bene

Geeks For Geeks, un pò più specifico

UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Domandine?

Esercizio 1/2

Definisci una classe Persona con un attributo privato nome di tipo string. Crea una persona utilizzando un raw pointer, uno smart pointer, e uno shared smart pointer. Illustra e spiega le differenze creando un metodo main e commentando opportunamente il codice.

Hint: per dimostrare come la memoria viene allocata e deallocata va benissimo usare degli inline-block direttamente nel main:

```
Int main(...){
  {
     //Code inside the block
  }
}
```

Esercizio 2/2

Fai un esempio di una funzione **setA** che prende una string S (come puntatore a char) e setta la prima lettera di S a 'A'.
Fai una versione setA con i puntatori raw (char*) e tutte quelle che riesci con i puntatori smart.
Fai poi un main in cui chiami tutte le funzioni con stringhe di prova.

Cioè con unique e shared, weak non mai è richiesto

Esercizio Extra

Considerare la classe Partita dell'esercizio 4. Crea una Partita utilizzando un raw pointer, uno smart pointer, e uno shared smart pointer. Alla distruzione della Partita devono essere distrutte anche la relativa nazionali. Illustra e spiega le differenze relative all'utilizzo dei vari tipi di puntatori creando un metodo main e commentando opportunamente il codice.