Modellazioni

Esercizio 1

Definire un template di classe `C<T,size>` con parametro di tipo `T` e parametro valore `size` di tipo intero che soddisfi le seguenti specifiche:

1. `MultiInfo<T>` è un template di classe associato ed annidato nel template `C<T,size>`. Un oggetto di `MultiInfo<T>` rappresenta un oggetto di tipo `T`, detto *informazione*, con una certa *molteplicità* `m` ≥ 0.
2. Un oggetto di `C<T,size>` rappresenta un array allocato dinamicamente di dimensione `size` di oggetti di `MultiInfo<T>`.
3. `C<T,size>` rende disponibile un costruttore `C(const T&, int)` con il seguente comportamento: una invocazione `C(t,k)` costruisce un oggetto di `C<T,size>` il cui array contiene in ogni posizione un oggetto di `MultiInfo<T>` con informazione `t` e quando `k` è ≥ 1 con molteplicità `k`, altrimenti (cioè quando `k` è < 1) con molteplicità 0.
4. Nel template `C<T,size>` il costruttore di copia, l'assegnazione e il distruttore devono essere "profondi", cioè la costruzione di copia e l'assegnazione di copia non devono provocare alcuna condivisione di memoria mentre la distruzione deve provocare anche la deallocazione di tutta la memoria dinamica.
5. `C<T,size>` rende disponibile l'overloading dell'operatore di indicizzazione `T* operator[](int)` con il seguente comportamento: se $0 \leq$ `k` $<$ `size` allora una invocazione `c[k]` ritorna un puntatore all'informazione di tipo `T` memorizzata nell'array di `c` in posizione `k`, altrimenti ritorna il puntatore nullo.
6. `C<T,size>` rende disponibile un metodo di istanza `int occorrenze(const T&)` con il seguente comportamento: una invocazione `c.occorrenze(t)` ritorna la somma delle molteplicità di tutte le occorrenze dell'informazione `t` nell'array memorizzato in `c`.
7. Deve essere disponibile l'overloading dell'operatore di output per oggetti di `C<T,size>` che permette di stampare tutte le informazioni di tipo `T` con relativa molteplicità memorizzate nell'array di un oggetto di `C<T,size>`.

Esercizio 2

Definire un template di classe `Array<T>` i cui oggetti rappresentano una struttura dati "array ridimensionabile" di elementi di uno stesso tipo `T`. Si ricorda che in un array ridimensionabile la sua dimensione (cioè il numero di elementi correntemente contenuti) è sempre $\leq$ alla sua capacità (cioè il numero di elementi che può contenere senza dover ridimensionarsi), e quando si inserisce un nuovo elemento in un array con dimensione uguale alla capacità, l'array viene ridimensionato raddoppiandone la capacità. Il template `Array<T>` deve soddisfare i seguenti vincoli:

1. Ovviamente, `Array<T>` non può usare i contenitori STL o Qt come campi dati (inclusi puntatori e riferimenti a contenitori STL o Qt).
2. Deve essere disponibile un costruttore `Array(int k = 0 , const T& t = T())` che costruisce un array contenente `k` copie di `t` quando `k` > 0, mentre se `k` ≤ 0 costruisce un array vuoto.
3. Gestione della memoria senza condivisione.
4. Deve essere disponibile un metodo `void pushBack(const T&)` con il seguente comportamento: `a.pushBack(t)` inserisce l'elemento `t` alla fine dell'array `a` dopo il suo ultimo elemento corrente; ciò provoca quindi il ridimensionamento di `a` se e solo se la dimensione di `a` è uguale alla sua capacità.
5. Deve essere disponibile un metodo `T popBack()` con il seguente comportamento: se l'array `a` non è vuoto, `a.popBack()` rimuove l'ultimo elemento corrente di `a` e quindi lo ritorna; se invece `a` è vuoto allora solleva una eccezione di tipo `Empty` (una opportuna classe di eccezioni di cui è richiesta la definizione).
6. Opportuno overloading dell'operatore di uguaglianza.
7. Opportuno overloading dell'operatore di output.

Cosa stampa

Esercizio 2.10.2. Definire un costruttore di default legale per la classe C.

```
class E {
private:
  int x;
public:
  E(int z=0): x(z) {}
};

class C {
private:
  int z;
  E x;
  const E e;
  E& r;
  int* const p;
  public:
  C();
};
```

Esercizio 2.10.3. Il seguente programma compila correttamente (si intende la compilazione `g++` con l'opzione `-fno-elide-constructors`). Quali stampe provoca la sua esecuzione?

```
class D {
private:
  int z;
public:
  D(int x=0): z(x) {cout << "D01 ";}
  D(const D& a): z(a.z) {cout << "Dc ";}
};

class C {
private:
  D d;
public:
  C(): d(D(5)) {cout << "C0 ";}
  C(int a): d(5) {cout << "C1 ";}
  C(const C& c): d(c.d) {cout << "Cc ";}
```

```
int main() {
  C c1; cout << "UNO" << endl;
  C c2(3); cout << "DUE" << endl;
  C c3(c2); cout << "TRE" << endl;
}
```

Esercizio 3.6.1. I seguenti programmi compilano correttamente e sono compilati con l'opzione `-fno-elide-constructors`. Quali stampe produce la loro esecuzione? Attenzione: senza l'opzione `-fno-elide-constructors` potrebbero produrre un output diverso.

```
// PROGRAMMA UNO

class C {
public:
  string s;
  C(string x="1"): s(x) {}
  ~C() {cout << s << "Cd ";}
};

// funzione esterna, passaggio del parametro per valore
C F(C p) { return p; }

C w("3"); // variabile globale

class D {
public:
  static C c; // campo dati statico
};
C D::c("4");

int main() {
  cout << "PROGRAMMA UNO\n";
  C x("5"), y("6"); D d;
  y=F(x); cout << "uno\n";
  C z=F(x); cout << "due\n";
}
```

```
// PROGRAMMA DUE

class C {
public:
  string s;
  C(string x="1"): s(x) {}
  ~C() {cout << s << "Cd ";}
};

// passaggio del parametro per riferimento
C F(C& p) { return p; }

C w("3");

class D {
public:
  static C c;
};
C D::c("4");

int main() {
  cout << "PROGRAMMA DUE\n";
  C x("5"), y("6"); D d;
  y=F(x); cout << "uno\n";
  C z=F(x); cout << "due\n";
}
```

```
// PROGRAMMA TRE

class C {
public:
  string s;
  C(string x="1"): s(x) {}
  ~C() {cout << s << "Cd ";}
};

// passaggio del parametro e valore ritornato per riferimento
C& F(C& p) { return p; }

C w("3");

class D {
public:
  static C c;
};
C D::c("4");

int main() {
  cout << "PROGRAMMA TRE\n";
  C x("5"), y("6"); D d;
  y=F(x); cout << "uno\n";
  C z=F(x); cout << "due\n";
}
```