Esercizi per il C++

1. Bubble sorter

Scrivi una classe BubbleSorter che ordina un array di interi.

BubbleSorter ha un **costruttore** che prende un array di interi tramite puntatore (int*) - e se serve la dimensione dell'array.

Ha un metodo **sort** che quando chiamato ordina l'array (**nota**: se non è stato già ordinato) senza però modificare l'array originale (deve quindi fare una copia).

Ha un metodo **getArrayOrdinato** che restituisce l'array ordinato.

Separa il .h e il .cpp di BubbleSorter e scrivi un file di prova con qualche array.

2. Scrivi una classe Complex Number – overload di operatori

Scrivi una classe complex che rappresenta numeri complessi. Complex ha tre costruttori (senza nulla, costruisce lo 0, con un argomento per numeri reali, e con due argomenti, parte reale e immaginaria). Prova con tre costruttori separati o con uno solo e argomenti di default.

Scrivi anche un metodo che restituisce la parte reale e immaginaria di un numero complesso.

Metodi

Scrivi dei metodi che fanno la somma, prodotto, differenza tra numeri complessi. Scrivi i metodi in modo che restituiscano un puntatore ad un nuovo numero complesso uguale al risultato dell'operazione tra l'oggetto che esegue il metodo e il numero complesso che è passato (per riferimento). Del tipo:

```
complex* somma(complex &)
```

Overload di operatori:

In C++ si posso ridefinire anche operatori non solo metodi. Ad esempio, posso ridefinire gli operatori +, - , ... in questo modo:

```
complex operator+(const complex & )const;
complex operator-(const complex & )const;
complex &operator=(const complex & );
complex operator*(const complex &)const;
bool operator==(const complex &)const;
bool operator!=(const complex &)const;
```

Prova ad implementare questi operatori.

Posso anche ridefinire gli operatori << e >> (che però appartengono alla classe ostream e istream) in questo modo:

```
//overloading di estrazione dello streaming
  friend ostream &operator<<( ostream & , const complex  &);
//overloading di immissione dello streaming
  friend istream &operator>>( istream &, complex & );
```

prova ad implementare << e >>.

Scrivi un main di prova in cui domandi (>>) 3 numeri complessi fai tutte le operazioni e stampi il risultato.

parte 2 ZooAnimal

1. The ZooAnimal class definition below is missing a prototype for the Create function. It should have parameters so that a character string and three integer values (in that order) can be provided when it is called for a ZooAnimal object. Like the Destroy function, it should have return type void. Write an appropriate prototype for the ZooAnimal Create function.

```
class ZooAnimal
{
 private:
   char *name;
   int cageNumber;
   int weightDate;
   int weight;
 public:
   void Destroy (); // destroy function
   char* reptName ();
   int daysSinceLastWeighed (int today);
};
```

2. Write a function header for the ZooAnimal class member function daysSinceLastWeighed. This function has a single integer parameter today and returns an integer number of days since the animal was last weighed.

```
void ZooAnimal::Destroy ()
{
 delete [] name;
}
// -------- member function to return the animal's name
char* ZooAnimal::reptName ()
{
 return name;
}
// -------- member function to return the number of days
// -------- since the animal was last weighed

{
 int startday, thisday;
 thisday = today/100*30 + today - today/100*100;
 startday = weightDate/100*30 + weightDate -
weightDate/100*100;
 if (thisday < startday)
     thisday += 360;
```

```
 return (thisday-startday);
}
```

3. In the main function there is a cout statement that attempts to print the animal's name. However, it is not allowable because it attempts to access the private data member called name. Modify that statement so that it uses a public member function that returns the ZooAnimal's name.

```
class ZooAnimal
{
 private:
   char *name;
   int cageNumber;
   int weightDate;
   int weight;
 public:
   void Create (char*, int, int, int); // create function
   void Destroy (); // destroy function
   char* reptName ();
   int daysSinceLastWeighed (int today);
};

// -------- member function to return the animal's name
char* ZooAnimal::reptName ()
{
 return name;
}
// ========== an application to use the ZooAnimal class
void main ()
{
 ZooAnimal bozo;
 bozo.Create ("Bozo", 408, 1027, 400);

 cout << "This animal's name is " << bozo.name << endl;

 bozo.Destroy ();
}
```

Part 3

1. slicing

Dichiara una classe base e una sua derivata. Tipo:

```
class A {
   int foo;
};
```

```
class B : public A
{
   int bar;
};
```

Cosa succede se fai (in un main):

```
B b;
A a = b;
```

Aggiungi un metodo M ad A che stampa "A" e prova a ridefinirlo in B in modo che stampi "B". Cosa succede se chiami il metodo M per a? Quale viene eseguito? Prova a metterelo virtual, cambia qualcosa?

La stessa cosa con i mparametri dei etodi. Cosa succede sa fai:

```
void wantAnA(A myA){
   myA.M()
}

B derived;
wantAnA(derived);
```

Quale viene eseguito? Come dovresti cambiare il metodo `wantAnA` per attivare il polimorfismo?

2. ereditarietà multipla

Fai un esempio di eredità multipla.

Ridefinisci qualche metodo nella classe derviata e chiama i metodi nelle classi base in modo che ci sia name clash. Come lo risolvi?

Aggiungi anche una classe in modo di avere una configurazione a damiante (senza derivazione virtual). Fai un modo di avere qualche problema di suplicazione dei campi base. Come la risolvi?

5. STL

STL1: declare a vector of integer values, stores five arbitrary values in the vector and then print the single vector elements to cout.

STL2: declare a vector of string values, asks to the user to insert a sentence of one or more words, store each word in the vector and then print the sentence in reverse order

STL3: come STL2 ma con un iteratore al contrario.

STL4: scrivi un metodo che legge parole da un file, le memorizza in una coda e le ristampa.

STL5: scrivi una piccola applicazione che gestisce una rubrica telefonica in cui usi:

- una map: array associativo per memorizzare nomi e numeri di telefono

- la funzione find (che prende due iteratori) per cercare un certo numero o persona

- la funzione copy per copiare l'intera rubrica a video con una sola istruzione (senza ciclo for)

```
copy(myVector.begin(), myVector.end(), ostream_iterator<int>(cout, " "));
```

Alcuni concetti fondamentali

1. Funzioni Virtual – Che differenza c'è tra dichiarare un metodo virtual o no?

Dichiarate una classe **Studente** con qualche campo (ad esempio nome)

Usate la classe std::string per le stringhe.

Studente ha due sottoclassi **StudenteLS** e **StudenteIL** con alcuni campi propri.

In Studente dichiarate un metodo virtual (**getCorso**) e uno non virtual (**getNome**) che poi ridefinite/override nelle sottoclassi. Entrambi restituiscono il dato con prefisso la classe che esegue il metodo. (Ad esempio "StudenteLS Informatica")

Nel main di prova provate a:

Creare 3 puntatori a tre oggetti delle tre classi Studente, tutti di tipo Studente*.

Chiamate il metodo virtuale e quello non virtuale per i tre oggetti.

Per i metodi virtual c'è binding dinamico

2. Slicing – cosa succede se non uso puntatori e cerco di usare polimorfismo delle variabili

Nell'esercizio precedente provate a dichiarare un oggetto Studente (non come puntatore).

Assegna a studente un oggetto della sottoclasse e chiama i metodi. Quelli virtual cosa stampano?

Se non uso i puntatori ho slicing e non ho binding dinamico

3. Ereditarietà privata – cosa succede se uso l'ereditarietà privata (in termini di visibilità e di sottotipazione)

Se una sottoclasse eredita privatamente ad esempio StudenteLS: private Studente:

- posso avere Studente* s = new StudenteLS?

- posso chiamare un metodo public per Studente anche per StudenteLS?

L'ereditarietà privata non crea sottotipi perché cambia la visibilità