Metodologie di Programmazione

Corso di Laurea in "Informatica"

16 giugno 2008

Note

1. Su tutti i fogli contenenti le soluzioni indicare, IN STAMPATELLO, la data dell'appello ed il proprio cognome, nome e numero di matricola
2. Leggere attentamente i quesiti e fornire risposte sintetiche: di solito, una risposta troppo verbosa sta ad indicare che si stanno fornendo particolari non richiesti.
3. Non è consentita la consultazione di alcunché.
4. L'orario di consegna scritto alla lavagna è tassativo.
5. Il testo del compito va consegnato insieme ai fogli con le soluzioni.

1. Mostrare il processo di risoluzione dell'overloading per le seguenti chiamate di funzione. Per ogni chiamata, indicare: l'insieme delle funzioni candidate; l'insieme delle funzioni utilizzabili; la migliore funzione utilizzabile (se esiste); il motivo di eventuali errori di compilazione.

```
#include <string>

namespace N {

  class C {
  private:
    std::string& first();             // funzione #1

  public:
    const std::string& first() const; // funzione #2

    std::string& last();              // funzione #3
    const std::string& last() const;  // funzione #4

    C(const char*);                   // funzione #5
  }; // class C

  void print(const C&);               // funzione #6
  std::string& f(int);                // funzione #7

} // namespace N

class A {
public:
  explicit A(std::string&);           // funzione #8
};

void print(const A&);                 // funzione #9

void f(N::C& c)                       // funzione #10
{
  const std::string& f1 = c.first();  // chiamata #1
  std::string& f2 = c.first();        // chiamata #2
  const std::string& l1 = c.last();   // chiamata #3
  std::string& l2 = c.last();         // chiamata #4
}

void f(const N::C& c)                 // funzione #11
{
  const std::string& f1 = c.first();  // chiamata #5
  std::string& f2 = c.first();        // chiamata #6
  const std::string& l1 = c.last();   // chiamata #7
  std::string& l2 = c.last();         // chiamata #8
}

int main() {
  N::C c("begin");   // chiamata #9
  f(c);              // chiamata #10
  f("middle");       // chiamata #11
  print("end");      // chiamata #12
}
```

2. Indicare l'output prodotto dal seguente programma.

```
#include <iostream>

class Animale {
public:
  Animale() { std::cout << "Costruttore Animale" << std::endl; }
  Animale(const Animale&) { std::cout << "Copia Animale" << std::endl; }

  virtual Animale* clone() const {
    std::cout << "Clonazione non specificata" << std::endl;
    return new Animale(*this);
  }

  virtual void verso() const {
    std::cout << "Verso non specificato" << std::endl;
  }

  virtual ~Animale() { std::cout << "Distruttore Animale" << std::endl; }
};

class Cane : public Animale {
public:
  Cane() { std::cout << "Costruttore Cane" << std::endl; }

  void verso() { std::cout << "bau!" << std::endl; }

  ~Cane() { std::cout << "Distruttore Cane" << std::endl; }

  Cane* clone() const { return new Cane(*this); }
};

class Pesce : public Animale {
public:
  Pesce() { std::cout << "Costruttore Pesce" << std::endl; }

  void verso() const { std::cout << "(glu glu)" << std::endl; }

  ~Pesce() { std::cout << "Distruttore Pesce" << std::endl; }

  Pesce* clone() const { return new Pesce(*this); }
};

class Pescecane : public Pesce {
public:
  Pescecane() { std::cout << "Costruttore Pescecane" << std::endl; }
  void verso() const { std::cout << "(glubau!)" << std::endl; }
  ~Pescecane() { std::cout << "Distruttore Pescecane" << std::endl; }
};

int main() {
  Animale a;
  a.verso();
  Cane c;
  c.verso();
  std::cout << "=== 1 ===" << std::endl;
```

```
  Pescecane p;
  p.verso();
  std::cout << "=== 2 ===" << std::endl;
  Animale* pc = c.clone();
  Animale* pp = p.clone();
  std::cout << "=== 3 ===" << std::endl;
  pc->verso();
  pp->verso();
  std::cout << "=== 4 ===" << std::endl;
  delete pp;
  delete pc;
  std::cout << "=== 5 ===" << std::endl;
}
```

3. Modificare la definizione della classe base dell'esercizio precedente al fine di renderne l'utilizzo più intuitivo e meno soggetto ad errori di programmazione. In particolare, oltre a rimuovere tutte le stampe non necessarie, si richiede di rendere impossibile l'esistenza di oggetti aventi come tipo dinamico la classe `Animale`, consentendo solo l'istanziazione delle classi derivate da essa.

4. Scrivere l'implementazione dell'algoritmo generico della STL `max_element`, che prende come input una sequenza ed un criterio di ordinamento (un predicato binario) e restituisce l'iteratore all'elemento massimo contenuto nella sequenza (comportandosi in modo standard nel caso di sequenza vuota). Elencare in modo esaustivo i requisiti imposti dall'implementazione sui parametri di tipo e sugli argomenti della funzione. In particolare, individuare le categorie di iteratori che *non* possono essere utilizzate per istanziare il template, motivando la risposta.

5. Una porzione di programma si dice *exception safe* (versione base) se fornisce le seguenti garanzie anche in presenza di eccezioni:

   - non causa perdite di risorse;
   - è neutrale rispetto alle eccezioni;
   - lascia gli oggetti utilizzati in uno stato consistente.

   Per ognuna delle garanzie elencate, fornire un semplice esempio di violazione, spiegandone brevemente le conseguenze.

6. Discutere brevemente sotto quali condizioni il codice seguente si può considerare exception safe:

```
void foo(Sched& v, Sched& w, unsigned n) {
  get_mutex_for(v);
  try {
    get_mutex_for(w);
    try {
      transmit(v, w, n);
      release_mutex_for(w);
      release_mutex_for(v);
    }
    catch (...) {
      release mutex_for(w);
      throw;
    }
  }
  catch (...) {
    release_mutex_for(v);
    throw;
  }
}
```

   Fornire una codifica alternativa basata sull'idioma "*l'acquisizione di risorse è una inizializzazione*".