# Metodologie di programmazione (itinere A, 12 aprile 2022)

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

```
namespace N {
  void gum(int i, double d);          // funzione #1
  void gum(float f, double d);        // funzione #2
  void gum(double d1, double d2);     // funzione #3
} // namespace N

void gum(int i, int j);               // funzione #4
void gum(char c, unsigned long ul);   // funzione #5

int main() {
  gum('k', 2);       // chiamata A
  gum('k', 2UL);     // chiamata B
  gum(3.14, 3.14F);  // chiamata C
  gum(3.14F, 3);     // chiamata D
  using namespace N;
  gum(3.14F, 3);     // chiamata E
  gum(3.14, 3.14F);  // chiamata F
}
```

2. Individuare e correggere gli errori di progettazione (che sono causa di potenziali scorrettezze e/o inefficienze) nell'interfaccia della classe seguente:

```
class Contatto {
  using Info = std::string;
  using Recapiti = std::vector<Info>;
  Info nome;
  Recapiti recapiti;
public:
  Contatto(Info nome);
  Contatto(Contatto c);
  void operator=(Contatto c);

  Info getNome();
  void setNome(Info n);
  int numRecapiti();
  Info& getRecapito(int indice);
  void addRecapito(Info info);
  void delRecapito(int indice);
  bool operator==(Contatto c1, Contatto c2);
  bool operator<(Contatto c1, Contatto c2);
};

std::ostream operator<<(std::ostream out, Contatto c);
```

3. La seguente funzione restituisce il valore `true` se e solo se la stringa `s` è palindroma (ovvero, contiene la stessa sequenza di caratteri sia quando è letta da sinistra verso destra, sia quando è letta da destra verso sinistra).

```
bool palindroma(const std::string& s) {
  unsigned i = 0, j = s.size();
  if (i == j)
    return true;
  --j;
  while (i != j) {
    if (s[i] != s[j])
      return false;
    ++i;
    if (i == j)
      return true;
    --j;
  }
  return true;
}
```

   Fornire una versione generica della funzione (prototipo e implementazione) che sia in grado di lavorare su una sequenza di elementi di tipo qualunque. Indicare i requisiti sugli iteratori usati per rappresentare la sequenza.

4. Le conversioni implicite del C++ sono distinte nelle categorie: corrispondenze esatte (E), promozioni (P), conversioni standard (S), conversioni definite dall'utente (U). Per ognuna delle categorie suddette scrivere una dichiarazione di funzione (con un solo parametro formale) e una corrispondente chiamata di funzione che generi una conversione implicita della categoria considerata.

5. Fornire un semplice esempio di violazione della ODR (regola della definizione unica) che, tipicamente, *non* viene rilevato dal compilatore (in senso stretto, cioè prima della fase di collegamento).

6. Il codice seguente presenta alcuni problemi relativi alla gestione delle risorse. Evidenziare questi problemi, differenziando tra: (a) errori che si verificano in assenza di eccezioni; (b) errori che si verificano in presenza di eccezioni.

```
void foo() {
  A* a1 = new A(1);
  A* a2 = new A(2);
  try {
    job1(a1, a2);
    job2(a1, new A(3));
  } catch (...) {
    delete a2;
    delete a1;
  }
}
```

   Fornire una soluzione, sempre basata sull'utilizzo dei blocchi `try` ... `catch` (ovvero, non è consentito usare smart pointers o adattatori RAII), che si comporti correttamente sia in assenza che in presenza di eccezioni. Spiegare brevemente i motivi per i quali la soluzione proposta si può dire *exception safe*.