Programmazione a oggetti

Programmazione generica

A.A. 2020/2021
Francesco Fontanella

Programmazione generica

- Consente la definizione di una classe (o funzione) senza che venga specificato il tipo di dato di uno o più dei suoi membri (o dei suoi parametri)
- Utile quando il codice non dipende dal tipo di dato da elaborare
- Esempi:
  - ADT (pila, coda, ecc)
  - Algoritmi di ordinamento

Templates

- Classi (o funzioni) per i quali uno o + tipi di dati non sono specificati sono dette
- I templates definiscono delle classi (o funzioni) cosiddette *generiche,* o anche *parametriche,*
- Questo codice deve poi essere “istanziato” dall'utente per produrre del codice che lavori con specifici tipi di dato

```
template <class T>
T funz(T par)
{
    T var1, var2, … varN;
        .
        .
        .
  return var1;
}

int main ()
{
  int  i;
  float f;
  char  c;

  funz(i);
  funz(f);
  funz(c);

  return 0;
}
```

```
int funz(int par)
{
  int var1, var2, … varN;
        .
        .
        .
  return var1;
}
```

```
float funz(float par)
{
  float var1, var2, … varN;
        .
        .
        .
  return var1;
}
```

```
char funz(char par)
{
  char var1, var2, … varN;
        .
        .
        .
  return var1;
}
```

<class T> indica che T è un Tipo (ma non solo, vedi dopo) da istanziare

```
template<class T>
T find_max(T v[], int n)
  {
    int i, m;

    m = 0;

    for (i=1; i < n; ++i)
       if (v[i] > v[m])
        m = i;

    return v[m];
  }
```

Per il tipo T deve essere stato definito `operator>`

```
int main()
{
  int i, vi[], n;
  float f, vf[];
  char c, str[];
      .
      .

  i = find_max(vi, n);
  f = find_max(vf, n);
  c = find_max(str, n);

return 0;
}
```

Istanziare un template

- Una volta definito, un template può essere usato specificando i tipi di dato da istanziare

  Esempio

```
template <class T>
void swap(T &x, T &y)
{ T t = x;
  x = y;
  y = t;
}
```

- Può essere istanziata con tipi differenti:

```
swap<int>
swap<float>
```

```
int i = 3, j = 4;
swap<int>(i, j);
    .
    .
float f = 4.0, g = 5.0;
swap<float>(f, g);
```

Conoscendo i tipi degli argomenti, il compilatore è in grado d'inferire l'istanziazione corretta

```
int i = 3, j = 4;
swap(i, j); // uses swap<int>
   .
   .
float f = 4.0, g = 5.0;
swap(f, g); // uses swap<float>
```

Templates di classi

```
template <class T>
class Myclass {
   T var1, var2, varN;
      .
      .
   T funz1()
   void funz2(T par1, …);
      .
      .
};
```

- Per istanziare una classe generica è necessaro specificare il tipo di T:

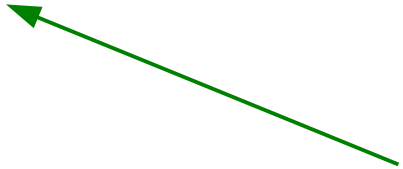
```
Myclass<int> mi;
Myclass<float> mf;
```

Parametri di un template

```
template <class T, int SIZE = 100>
class myVector {
    T v[SIZE];
  public:
    T& operator[](int i);
};

template <class T, int SIZE = 100>
T& myVector<T, size>:: operator[](int i) {
  if (i >= 0 && i < SIZE);
    return v[i];
  else cerr<<endl<<"ERRORE!";
}

myVector<int, 50> ivec;
myVector<double>  dvec;
```

Non tutti i parametri devono essere tipi generici

```
template <class T, class U>
class Pair {
    T x1;
    U x2;
  public:
    Pair(T a1, U a2): x1(a1), x2(a2) {}
};


Pair<string, int> p1("deep ", 6);
cout << p1.x1 <<" "<< p1.x2 << endl;
```



Un template può avere uno o più parametri

NOTA

Le funzioni membro di una classe template sono istanziate solo se usate

Esempio: stack di interi

```
class Stack {
  public:
    Stack() {top = 0;}
    ~Stack();
    void push (int val); {top = new Node(val, top);}
    int pop ()
private:
  Node* top;
};
```

```
class Node {
  public:
    Node (int v, Node* n): value(v), next(n) {}
    Node*  getNext() {return next;}
    int getValue() {return value;}
  private:
    Node* next;
    int value;
};
```

Assegnazione di valore ai membri della classe

```
Stack::~Stack()
{
    while(top)
      pop();
}

int Stack::pop () {
   int v;
   Node* tmp;

   if (top) {
     v = top->getValue();
     tmp = top;
     top = top->getNext();
     delete tmp;

     return v;
    } else cerr<<endl<<"ERRORE!";
  }
```

Stack generico

```
template<class K> class Node {
public:
  Node (K v, Node* n): value(v), next(n) {}
  K getValue (){return value; }
  Node*  getNext() {return next;}
private:
  Node* next;
  K value;
};
```

```
template<class T> class Stack {
public:
  Stack() {top = 0;}
  ~Stack();
  void push(T val) { top = new Node<T>(val, top);}
  T  pop();
private:
  Node<T>* top;
};
```

Node **va istanziato, perchè è generico!**

```
template<class U>
Stack<U>::~Stack() {
    while(top)
      pop();
    }

template<class Z>
Z Stack<Z>::pop () {
    Z v;
    Node<Z>* tmp;
    if (top) {
      v = top->getValue();
      tmp = top;
      top = top->getNext();
      delete tmp;

      return v;
    } else cerr<<endl<<"ERRORE!";
  }
```

NOTA

Per le diverse funzioni, anche di una stessa classe, posso usare lettere diverse per i template e classi

```
#include <stack.h>

int main() {
   Stack<int> si;
   Stack<float> sf;

   si.push(1);
   sf.push(3.0);

   cout<<endl<<si.pop();
   cout<<endl<<sf.pop();

   return 0;
}
```

OUTPUT

```
1
3.0
```

Nodo Generico

```
#ifndef NODE_H
#define NODE_H

template<class T>
  class Node {
    public:
      // Costruttore
      Node(T x) : next(0), value(x) {}
      // Funzioni GET
      Node<T>* getNext()  const {return next;}
      T getValue() const {return value;}
    private:
      T value;
      Node* next;
    friend class ListIterator<T>; // Vedi dopo...
    friend class List<T>;
};
#endif
```

Lista Generica

```
#ifndef LIST_H
#define LIST_H


template<class T>
class List {
  public:
    List() : n(0), l(0){ }
    ~List();
    void append(T x);
    const T& operator[](int pos) const;
    int size() const {return n;}
  protected:
    Node<T>* l;
    int n;
  friend class ListIterator<T>;
};
```

```
template<class Z>
void List<Z>::append (Z v) {
    Node<Z>* tmp = l;

    if (l == 0)  // Lista vuota
      l = new Node<Z>(v,l);
    else {
      tmp = l
      while (tmp->next)
        tmp = tmp→next;

      tmp->next = new Node<Z> (v,0);
    }

    return v;
}
```

```
template<class U>
const U& List<U>::operator[](int pos) {
    Node<U> *tmp

    if (pos < 0 || pos >= n ) {
      cerr<<"ERRORE!!: indice pos ERRATO"
      exit(EXIT_FAILURE);
    }

    i = 0
    while (i < pos)
      tmp = tmp→next;

    return tmp->value;
}
```

```
template<class U>
List<U>::~List() {
    Node *tmp

    while (l)
     tmp = l;
     l = l→next;
     delete tmp;
}
```

Iteratore sulla lista


```
#ifndef LISTITER_H
#define LISTITER_H
  #include "list.h"

  template<class T>
  class ListIterator
  {
    public:
      ListIterator(const List<T> &l) {head = cur = l.head;}
      T& operator*() {return cur->value;}
      // Incremento prefisso
      Node<T>* operator++() { return cur  == 0 ? 0 : cur = cur->next;}
      // Incremento postfisso
      Node<T>* operator++(int i){ return cur  == 0 ? 0 : cur = cur->next;}

      void rewind() {cur = head;}
      operator bool() const { return cur;}
    private:
       Node<T> *cur, *head;
  };
#endif
```

Osservazioni

- L'iterarore su lista è una sorta di puntatore “intelligente” che tiene traccia degli accessi precedenti alla lista:
  - `head` punta alla testa della lista “puntata” dall'iteratore;
  - `cur` punta all'ultimo elemento acceduto, è detto l'elemento “corrente”
- Il costruttore dell'iteratore prende come parametro la lista da puntare
- Le funzioni `operator++` (prefisso e postfisso) consentono di passare al prossimo elemento della lista, a partire dall'ultimo elemento puntato, memorizzato nel puntatore `cur`

- La funzione `operator*` consente di accedere all'elemento corrente usando l'operatore di dereferenziazione (dereference operator)
- La funzione `rewind()`, invece, consente di “riavvolgere” il puntatore `curr`, che punterà di nuovo alla testa della lista
- La funzione **`bool`**, infine, consente di usare la variabile `ListIterator` come se fosse un variabile booleana (vedi slide successive)

Esempio

```
#include <iostream>
#include "List.h"
#include "ListIterator.h”
const int MAX = 1000;

int main()
{
  List<int> lista;
  int i;

  for (i=0; i < MAX; ++i)
     lista.append(i);

  // uso inefficiente!
  for (i=0; i < MAX; i++)
      cout << lista[i];

  // iterazione efficiente
  ListIterator<int> iter(lista);
  while (iter) {
    cout << *iter <<" ";
      iter++;
  }

  return 0;
}
```

si costruisce una lista con MAX elementi,

Accesso inefficiente

- l'accesso alla lista per mezzo della funzione **`operator`**`[](`**`int`**` pos)` è inefficiente in quanto questa funzione parte sempre dal primo elemento e scandisce la lista fino ad arrivare all'elemento `pos`.
- Pertanto per accedere all'elemento in posizione pos sono necessarie pos operazioni.
- Ne consegue che per visualizzare una lista di MAX elementi sono necessarie almeno MAX*(MAX-1)/2 operazioni.
- Si parla in questo caso di *complessità computazionale* **quadratica**.

Accesso efficiente

- L'iteratore `iter` della classe `ListIterator`, invece, consente un accesso efficiente agli elementi alla lista in quanto esso tiene traccia dell'ultimo accesso effettuato.
- Pertanto per visualizzare l'i-esimo elemento della lista è necessaria una singola operazione, quella di incremento dalla posizione corrente
- Ne consegue che per visualizzare una lista di `MAX` elementi sono necessarie `MAX` operazioni
- Si parla in questo caso di complessità computazionale **lineare**

Compilazione di template

- Il compilatore NON PUÒ generare codice oggetto per i template:
  - Il codice per un determinato tipo è generato solo se un template di quel tipo è effettivamente istanziato
- Per generare una qualsiasi istanza il compilatore deve accedere al codice sorgente del template
- Il C++ standard definisce due modelli per la compilazione del codice dei templates:
  - **compilazione per inclusione** (supportata da tutti i compilatori)
  - **compilazione separata** (supportata solo da alcuni)

Compilazione per inclusione

- La compilazione di template richiede che negli header file vengano inclusi anche le definizioni delle funzioni (implementazione)
- A tal fine, bisogna inserire una direttiva `#include` nel header file per inserire in quest'ultimo le definizioni contenute nel file d'implementazione
- **Esempio**

tempClass.h

```
#ifndef DEMO_H
#define DEMO_H
template<class T>
class TempClass {
 public:
  TempClass();
     .
     .
};

#include "demo.cpp"
#endif
```

tempClass.cpp

```
template<class T>
TempClass<T>::TempClass
{
     .
     .
}


     .
     .
     .
```

Osservazioni

- scrivere una funzione generica implica pensare in astratto, evitando le dipendenze da tipi di dato, costanti numeriche, ecc.
- Una definizione di funzione template è solo un modello e non produce effettivamente codice
- Il template è un generatore automatico di funzioni specifiche
- templates tendono a generare un codice eseguibile grande, poiché duplicano codice

Cast Type Overloading

- In C++, oltre che dei normali operatori, è possibile anche l'overloading del casting ,
- **Esempio**

```
class Cents
{
  private:
    int m_nCents;
  public:
    Cents(int nCents=0) { m_nCents = nCents;}

    // Overloaded int cast
    operator int() { return m_nCents; }

    int GetCents() { return m_nCents; }
    void SetCents(int nCents) { m_nCents = nCents; }
};
```

L'overloading è sul casting

```
int main()
{
    Cents c(7);
    int a;

    a = c;

    cout<<a;

    return 0;
}
```


```
a = Cents::int(&c)
```

- Il Cast Type Overloading è il meccanismo che consente:

```
   .
   .
ifstram ins("input.txt");

if (ins) {          diventa          if (ios::bool)
   .
   .
```