Programmazione generica

Programmazione di Sistema

A.A. 2017-18

© C. Barberis, G. Malnati 2004-18

Argomenti

- Programmazione generica
- Smart pointer

© C. Barberis, G. Malnati 2004-18

Programmazione generica

- Un sistema di tipi stringente facilita la creazione di codice robusto
  - Identificando, in fase di compilazione, quei frammenti di codice che violano il sistema dei tipi
- In certe situazioni, per non violare il sistema dei tipi occorre replicare una grande quantità di codice generando problemi in fase di manutenzione
  - Occorre garantire che eventuali modifiche ad una versione del codice vengano propagate a tutte le altre
- Il linguaggio C++ offre un supporto alla generalizzazione
- dei comportamenti di un blocco di codice attraverso il concetto di “template”

© C. Barberis, G. Malnati 2004-18

Template

- Frammenti (funzioni, classi) parametrici
  - Che vengono espansi in fase di compilazione, in base al loro uso effettivo
- Non richiedono controlli in fase di esecuzione
- Permettono di implementare una varietà di funzionalità diverse adattandole ai singoli tipi di dato
  - Ad esempio algoritmi per il valore minimo, massimo, medio, …

© C. Barberis, G. Malnati 2004-18

Usi dei template

- Algoritmi generici
- Collezioni di dati coerenti per il tipo incapsulato e relativi algoritmi
- Funzioni generalizzate
- La parte principale della libreria standard è basata su template

© C. Barberis, G. Malnati 2004-18

Funzioni generiche

- Si può definire una funzione in modo che operi su un tipo di dato non ancora precisato
  - ```
    template <class T>
    const T& max(const T& t1, const  T& t2) {
      return (t1< t2 ? t2 : t1);
    }
    ```
- Opera sul tipo generico T, a patto che supporti
  - L'operatore "< " tra argomenti omogenei
  - Il costruttore di copia, per derivare una variabile temporanea a partire da un dato costante
- Ogni volta che si usa la funzione, il compilatore determina quale tipo effettivo assegnare a "T", in base ai parametri passati

© C. Barberis, G. Malnati 2004-18


    ```

Funzioni generiche

```
int i=max(10,20);  //  T → int

std::string  s, s1 = …, s2= …;
s = max( s1, s2);  // T →
std::string

max<double>(2, 3.1415928);
      //forza la scelta di “T” a
double
```

© C. Barberis, G. Malnati 2004-18

Classi generiche

- Lo stesso principio può essere adottato nella definizione delle classi
  - Quando queste vengono usate, si correda il tipo generico con l'indicazione della specializzazione richiesta

```
template <class T>
class Accum {
  T total;
public:
  Accum(T start): total(start) {}
  T operator+=( const T& t) {
    return total= total+t;
  }
  T value() { return total;}
};

Accum<std::string> sa("");
Accum<int> ia(0);
```

© C. Barberis, G. Malnati 2004-18

Template C++

```
template <class T, int size>
   class Stack {
   public:
      Stack() { }
      ~Stack() { }

      void push(T val);
      T pop();

   private:
      T data[size];
      int index;
   };
```

```
Stack<int,100> s;
Stack<double,30> s1;
Stack<CProva,20> s2;
```

© C. Barberis, G. Malnati 2004-18

Template C++

- I parametri del template possono essere identificatori di tipo di dato o valori costante
  - Evitano di ridefinire classi per tipi di dato o dimensioni differenti
- Ogni volta che si istanzia un template, indicando il valore dei parametri, il compilatore genera una nuova classe/funzione
  - Soluzione altrettanto versatile rispetto all'uso di gerarchie di ereditarietà e meno soggetta ad errori
  - Richiede pattern di programmazione appositi

© C. Barberis, G. Malnati 2004-18

Specializzare un template

- Alcuni tipi potrebbero non essere utilizzabili all'interno di un dato template
  - Ad es., potrebbero non avere l'implementazione di un operatore usato nella definizione del template stesso
- Occorre modificare la classe
  - Fornendo le implementazioni e la semantica richieste
- Altrimenti, si può specializzare il template
  - Ovvero fornirne una definizione specifica per la classe non altrimenti usabile

© C. Barberis, G. Malnati 2004-18

Specializzare un template

```
class Person {
  std::string name;
public:
  Person(std::string n): name(n) {}
  std::string name() { return name; }
};
template<> class Accum<Person> {
  int total;
public:
  Accum(int start = 0): total(start) {}
  int operator+=(const Person &) {return +
+total;}
  int value() { return total;}
};
```

© C. Barberis, G. Malnati 2004-18

Template: punti di forza

- Aumentano notevolmente le possibilità espressive
  - Senza pregiudicare i tempi di esecuzione
  - Risparmiano tempo di sviluppo
  - Non mettono a rischio il sistema dei tipi
  - Introducono flessibilità in vista di futuri miglioramenti
  - Non solo a livello sintattico (lo fa già il compilatore), ma soprattutto a livello semantico

© C. Barberis, G. Malnati 2004-18

Template: punti di attenzione

- Chi usa un template, deve verificarne le assunzioni sui tipi effettivamente usati
  - Eventuali casi particolari possono essere gestiti mediante la specializzazione
- Le violazioni sul tipo utilizzato conducono ad errori di compilazione difficili da interpretare
  - L'espansione del template conduce ad espressioni anche molto differenti dal codice originale
- Spesso, non occorre scrivere nuovi template
  - Occorre però sapere usare molto bene quelli esistenti

© C. Barberis, G. Malnati 2004-18

Smart Pointer

- La ridefinizione degli operatori permette di dare una semantica alle operazioni ‘*’ e ‘->’ anche a dati che non sono (solo) puntatori
- È possibile costruire oggetti che “sembrano” puntatori ma che hanno ulteriori caratteristiche
  - Garanzia di inizializzazione e rilascio
  - Conteggio dei riferimenti
  - Accesso controllato

© C. Barberis, G. Malnati 2004-18

Smart Pointer

- Per aumentare il parallelo con i puntatori semplici, possono anche essere ridefinite le operazioni
  - Aritmetiche (++, --, ...)
  - Di confronto (==, !=, !, ...)
  - Di assegnazione/copia (=)

© C. Barberis, G. Malnati 2004-18

Esempio

```
class int_ptr
{
   int* ptr;
   int_ptr(const int_ptr&);
   int_ptr& operator=(const int_ptr&);
public:
   explicit int_ptr(int* p) : ptr(p) { }
   ~int_ptr() {delete ptr;}
   int& operator*() {return *ptr;}
};
```

© C. Barberis, G. Malnati 2004-18

Esempio

- Questa classe incapsula un puntatore ad un intero
  - Che si suppone essere allocato sullo heap
- Quando un oggetto di questo tipo viene distrutto la memoria viene rilasciata automaticamente
- Si potrebbero inserire controlli nel costruttore per verificare che il puntatore non sia nullo

© C. Barberis, G. Malnati 2004-18

Generic smart_ptr

```
template <class T>
class smart_ptr  {
  T* ptr;
     smart_ptr(const smart_ptr<T>&);
     smart_ptr<T>& operator=(const
  smart_ptr<T>&);
 public:
     explicit
  smart_ptr(T* p = 0) : ptr(p) {}
  ~smart_ptr() { delete ptr; }
  T& operator*() { return *ptr; }
  T* operator->() { return ptr; }

};
```

© C. Barberis, G. Malnati 2004-18

Codici a confronto

```
void esempio1()
{
   MyClass* p =
    new MyClass();
  p->Esegui();
  delete p;
 }
```

```
void esempio2()
{
 smart_ptr<MyClass>
   p(new MyClass());
  p->Esegui();
}
```

- smart_ptr si occupa di chiamare l'operatore delete sul puntatore dell'oggetto incapsulato quando si esce dal blocco
  - Evitando perdite di memoria, anche in caso di eccezioni!

© C. Barberis, G. Malnati 2004-18

Sicurezza per le eccezioni

- La funzione Esegui() potrebbe generare un'eccezione
  - Le righe di codice successive non verrebbero eseguite

```
void esempio1()  {
   MyClass* p =new
   MyClass();
   p->Esegui();
   delete p;
}
```

© C. Barberis, G. Malnati 2004-18

Sicurezza per le eccezioni

- L'uso di costrutti try/catch può fare esplodere le dimensioni del programma
  - L'uso dello smart pointer garantisce che la risorsa venga rilasciata automaticamente

```
void esempio3() {
   MyClass* p;
   try {
       p = new
   MyClass();
       p->Esegui();
       delete p;
   }
   catch (...)  {
       delete p;
       throw;
   }
 }
```

© C. Barberis, G. Malnati 2004-18

Copia e smart pointer

- A chi appartiene la memoria puntata da uno smart pointer?
  - Il problema si pone nel caso si voglia assegnare uno smart pointer ad un altro
- È possibile implementare strategie diverse
  - Passaggio di proprietà
  - Creazione di una copia
  - Condivisione con conteggio dei riferimenti
  - Condivisione in lettura e duplicazione in scrittura

© C. Barberis, G. Malnati 2004-18

Ownership-Handling

- Se l'oggetto, referenziato dallo smart pointer, viene usato da un unico utilizzatore
  - può essere eliminato al termine delle operazioni dal distruttore dello smart pointer
- Se viene usato da più utilizzatori è necessario definire chi possa eliminare la risorsa
  - L'ultimo a conoscere l'oggetto
  - Un garbage collector che opera per conto del sistema

© C. Barberis, G. Malnati 2004-18

Smart pointer nella libreria standard

- A partire dalla versione C++11, sono disponibili diversi template per la gestione automatica della memoria
  - #include <memory>
- std::shared_ptr<BaseType>
  - Implementa un meccanismo di conteggio dei riferimenti
- std::weak_ptr<BaseType>
  - Permette di osservare il contenuto di uno shared_ptr senza partecipare al conteggio dei riferimenti
- std::unique_ptr<BaseType>
  - Implementa il concetto di proprietà, impedendo la copia ma permettendo il trasferimento

© C. Barberis, G. Malnati 2004-18

std::shared_ptr

- Mantiene la proprietà condivisa ad un blocco di memoria referenziato da un puntatore nativo
  - Molti oggetti possono referenziare lo stesso blocco
  - Quando tutti sono stati distrutti o resettati, il blocco viene rilasciato
- Per default, il blocco referenziato viene rilasciato tramite l'operatore delete
  - In fase di costruzione di shared_ptr, è possibile specificare un meccanismo di rilascio alternativo
- Un oggetto di questo tipo può anche non contenere alcun puntatore valido
  - Se è stato inizializzato o resettato al valore NULL

© C. Barberis, G. Malnati 2004-18

std::shared_ptr

- shared_ptr<T>(T* nat_ptr)
  - Costruisce uno shared_ptr che incapsula il puntatore nat_ptr
- ~shared_ptr<T>()
  - Distrugge uno shared_ptr
  - Libera la memoria, quando il contatore dei riferimenti è O
- operator=(const shared_ptr<T>& other)
  - Assegna il puntatore condiviso, incrementando il contatore
- get(), operator* (), operator-> ()
  - Accesso al puntatore incapsulato

© C. Barberis, G. Malnati 2004-18

std::shared_ptr

- reset()
  - Decrementa il contatore ed elimina il riferimento contenuto nello smart pointer
  - Se il contatore scende a O, libera la memoria
- reset(BaseType* ptr)
  - Come il precedente, ma sostituisce il puntatore contenuto con quello passato come parametro

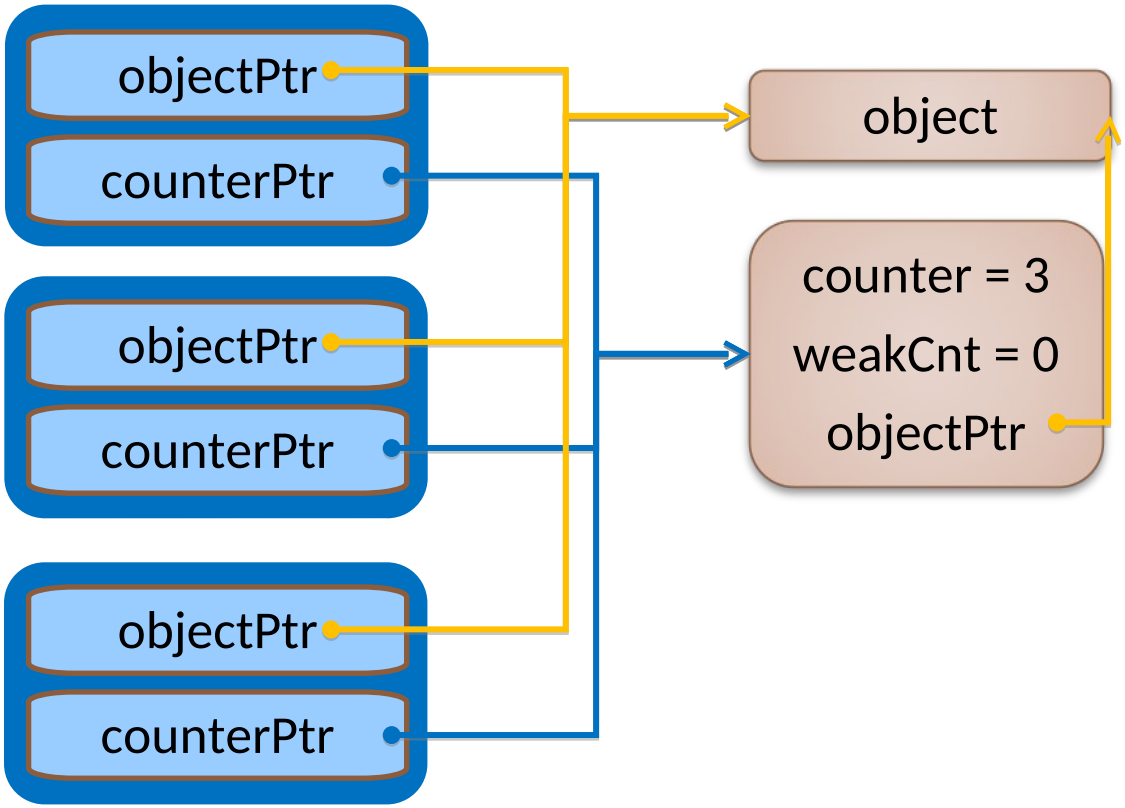
© C. Barberis, G. Malnati 2004-18

std::shared_ptr

- make_shared<BaseType>(params…)
  - Funzione di supporto
  - Crea sullo heap un’istanza della classe BaseType
    - Usando i parametri passati
    - come parametri del
    - costruttore
  - Ne incapsula il puntatore in uno shared_ptr

© C. Barberis, G. Malnati 2004-18

Implementazione tipica



© C. Barberis, G. Malnati 2004-18

Implementazione tipica

- Ogni shared_ptr punta ad un blocco di controllo oltre che all'oggetto stesso
  - La dimensione dello smart pointer raddoppia
- Occorre memorizzare il blocco di controllo

© C. Barberis, G. Malnati 2004-18

std::auto_ptr<BaseType>

- Il puntatore incapsulato negli oggetti di tipo auto_ptr appartiene a tali oggetti
  - La memoria cui fa riferimento deve appartenere allo heap ed essere stato allocato tramite new
- All'atto della distruzione di auto_ptr, viene invocato il metodo delete
  - Cosa capita se si duplica un oggetto di tipo auto_ptr, attraverso un'assegnazione o un costruttore di copia?
- La proprietà del puntatore viene trasferita al destinatario
  - L'oggetto sorgente viene modificato, ponendo a NULL il puntatore incapsulato
  - Altrimenti si rischierebbe di invocare delete due volte!

© C. Barberis, G. Malnati 2004-18

Dipendenze cicliche

- Il conteggio dei riferimenti garantisce il rilascio della memoria in modo deterministico
  - Non appena un oggetto non ha più riferimenti viene liberato
- In alcuni casi, tuttavia, non funziona
  - Se si forma un ciclo di dipendenze (A→B, B→A) il contatore non può mai annullarsi, anche se gli oggetti A e B non sono più conosciuti da nessuno
- Occorre evitare la creazione di cicli ricorrendo a
  - std::weak_ptr<BaseType>

© C. Barberis, G. Malnati 2004-18

std::weak_ptr

- Mantiene al proprio interno un riferimento “debole” ad un blocco custodito in uno shared_ptr
  - Modella il possesso temporaneo ad un blocco di memoria
- Si acquisisce temporaneamente l’accesso al dato tramite il metodo lock() che restituisce uno shared_ptr
  - Si verifica la validità del riferimento con il metodo expired()

© C. Barberis, G. Malnati 2004-18

std::weak_ptr

```
std::weak_ptr<int> gw;
void f(){
    if (auto spt = gw.lock())
         std::cout<<"gw:"<< *spt << "\n";
    else
        std::cout<< "gw e' scaduto\n";
}
int main() {
    { auto sp = std::make_shared<int>(42);
         gw = sp;
         f();
    }  // sp viene distrutto, gw scade
    f();
}
```

© C. Barberis, G. Malnati 2004-18

std::unique_ptr

- Incapsula il puntatore ad un oggetto o ad un array
  - Non può essere copiato né assegnato
  - Il puntatore può essere trasferito esplicitamente ad un altro unique_ptr con la funzione std::move()
- Quando viene distrutto, il blocco viene rilasciato

© C. Barberis, G. Malnati 2004-18

std::unique_ptr – Usi tipici

- Garantire la distruzione di un oggetto
  - Anche in caso di eccezioni
  - Passare la proprietà di un oggetto con ciclo di vita dinamico tra funzioni
- Gestire in modo sicuro oggetti polimorfici

© C. Barberis, G. Malnati 2004-18

Spunti di riflessione

- Si implementi una lista circolare generica incapsulando i puntatori agli elementi in oggetti di tipo shared_ptr<T>

© C. Barberis, G. Malnati 2004-18