Conceptes bàsics per treballar amb la STL

Estructura de Dades

Jordi Gonzàlez i Robert Benavente - Febrer 1999

Introducció als template

Classes template

Els templates són una eina del C++ que ens permet modelitzar les entitats que pugi tenir un programa d'una manera molt més genèrica que no pas ho fa una classe. Imaginem que s'ha de realitzar un programa que necessita treballar amb un conjunt d'enters i un conjunt de flotants. La primera aproximació possible seria construir una classe *ConjuntInt* i una classe *ConjuntFloat* que realitzessin les funcions associades al concepte de conjunt pels respectius tipus de dades. Al codificar les dues classes es veuria que ambdues són gairebé idèntiques (mateixes funcions d'interficie, mateixa implementació interna, …), amb la diferència de que es treballa amb tipus de dades diferents. Els templates de C++ ens permeten poder definir una classe genèrica *Conjunt*, a la que el tipus dels objectes que conté se li passen com a paràmetres. D'aquesta manera, codificant només una vegada el comportament que ha de tenir una classe *Conjunt*, podem utilitzar-la per gestionar conjunts de diferents tipus.

Així doncs, la necessitat d'incloure el concepte de classe parametritzada en un llenguatge de programació va sorgir al percebre que determinades estructures de dades, amb un comportament perfectament definit, es traduien en un entorn orientat a objectes com a classes que mantenien internament objectes d'algun tipus determinat. A aquestes classes se les anomena classes contenidor (Container Class), i és on es poden aprofitar ampliament les caracterísitiques dels templates. Exemples de classes contenidor serien la classe Vector, la classe Conjunt, la classe Llista, etcètera.

Declaració d'una classe template en C++

La nomenclatura que segueix C++ per declarar una classe template és la següent:

```
template <class T>
class _NomClasse{

        // Objectes T continguts a la classe

        T _VarMembre1;
        T *_VarMembre2;
        ….
public:
        _NomClasse();
        ~_NomClasse();

        T _Funcio(int a, T b);
};
```

Com es pot veure, és el mateix que declarar una classe normal, amb dues diferències:

- Hi ha el prefixe `template <class T>`
- Dins la classe es treballa amb objectes del tipus `T`.

La definició d'una funció membre es faria de la següent manera;

```
template<class T>
 Valor de retorn
_NomClasse<T>::_NomFuncio(int a, Tb)
{
….
}
```

La nomenclatura és la que s'utilitza en les funcions membre habituals, excepte que

- Hi ha el prefixe template <class T>
- El nom de la classe s'especifica com _NomClasse<T>.
- En els paràmetres podrem tenir objectes del tipus T.

A l'hora de declarar un objecte d'una classe template, la notació que seguiríem seria la que es mostra a continuació. Noteu que el nostre template haurà de tenir definit un constructor per defecte.

```
_NomClasse<TipusDesitjat> a;
```

El mecanisme és el mateix que s'utilitza per les classes normals, amb la diferència de que cal passar un tipus com a paràmetre a la classe template.

Funcions Template:

De la mateixa manera que hi ha classes templates, C++ permet tenir funcions template. És a dir, es poden definir funcions que duguin a terme un determinat algorisme per a qualsevol tipus de dades. Per exemple, en un programa es podria necessitar ordenar una llista d'enters i una llista de flotants. En lloc de definir una funció d'ordenació per a cada tipus de llista, es podria definir una funció template que ordenés llistes de qualsevol tipus. Per exemple, en C++ podríem definir:

```
template<class T> void sort(List<T>&){ … }
```

Es té una funció sort que ordena llistes de qualsevol tipus. Notar que l'especificació del paràmetre que es rep es fa mitjançant una classe template List. La crida a aquesta funció es faria de la següent manera:

```
List<int> LlistaEnters;

sort(LlistaEnters);
```

Com es pot veure, que sort sigui una funció template és transparent a l'usuari. L'ús dels templates deixa de ser transparent quan:

- Declarem/Definim una classe template.
- Declarem/Definim una funció template.
- Declarem un objecte d'una classe template.

En la instanciació d'objectes d'una classe template o crides a funcions template es treballarà de la mateixa manera que amb els objectes o funcions normals.

Declaració/Definició d'una funció template en C++

La nomenclatura que segueix C++ per declarar i definir una funció template és la que segueix:

```
template<class T>
Valor de Retorn
_NomFuncio ( int a, T *b, TemplateClass<T>& c)
{
…
};
```

La notació és la mateixa que la de qualsevol funció exceptuant:

- Hi ha el prefixe template <class T>
- En els paràmetres es tindran objectes relacionats amb el tipus T (doncs sinò no tindria sentit la funció template).

Requeriments d'un template

En les funcions membre d'una classe template o en una funció template es treballarà amb objectes del tipus T. Evidentment els operadors o funcions que s'apliquin sobre objectes del tipus T, hauran d'estar definits pel tipus que es passarà com a paràmetre a la classe template. Per exemple posem la funció template min

```
template<class T>
T
min(const T& a, const T&b)
{
        return (a<b? a: b);
}
```

Si utilitzem aquesta funció en variables de tipus enter, no tindrem cap mena de problema. En canvi, si s'utilitza per un tipus propi, si aquest tipus no te sobrecarregat l'operador < tindrem un error en temps de compilació.

Es habitual, a l'hora de treballar amb un template, que s'informi dels requeriments que ha de tenir el tipus que se li passa per paràmetre.

Què passa quan es té un tipus de dades que no compleix els requeriments d'un template, o bé que compleix els requeriments però amb un significat diferent a l'esperat. En el primer cas es detectarà un error en temps de compilació, mentre el segon cas es detectarà a l'observar un comportament no esperat a l'executar el programa. El següent exemple es força aclaridor.

```
char    *NomA="Saladich";
char    *NomB="Marfà";

cout << min(NomA,NomB);
```

A primer cop d'ull s'entèn que es vol mostrar per pantalla el nom amb ordre alfabètic menor. En realitat la funció template min el que farà és comparar char* < char*, és a dir, adreces de memòria. Per aconseguir el comportament desitjat de la funció min en objectes de tipus char*, C++ permet definir una funció min especial per aquest tipus concret.

```
char * min(char* a, char* b){
        return (strcmp(a,b)<0?a:b);
}
```

Això és possible perquè C++ no *expandeix* una funció template X per un tipus determinat si en els fitxers del programa ja hi ha una funció X específica per el tipus determinat.

Aquest mateix problema que s'ha vist per la funció min també es pot donar a nivell de classe. És a dir, pot haver-hi un determinat tipus d'objectes que a l'aplicar-se a una classe template provoquin problemes. La solució a aquest casos és anàloga a la proposada per les funcions template. El que es fa és declarar/definir una classe amb el mateix nom que la classe template però éspecífica per el tipus de dades problemàtic.

```
template <class T>
class X
{
        ….
};
```

Si el tipus que dona problemes és el char*, aleshores es descriurà una classe específica per ell.

```
class X<char*>
{

};
```

La Standard Template Library

Un cop s'han vist les possibilitats del templates, ara podríeu plantejar-vos implementar les classes que més habitualment es fan servir com a classes template, i així aprofitar-vos-en en propers programes que pugueu haver de fer. A l'hora de fer la tria de quines classes template construir s'arriba a la conclusió que és en les classes contenidor on té més sentit i utilitat aplicar el concepte de template. De la mateixa manera, decidir quines funcions implementar com a template us portarà, entre altres, a implementar de forma genèrica (i.e. per qualsevol tipus) els algorismes d'ordenació, recerca, … que molt sovint heu d'utilitzar.

No obstant us podeu estalviar de fer-vos la vostra pròpia llibreria, i utilitzar la Standard Template Library, dissenyada per Alexander Stepanov i Meng Lee. Utilitzant la STL passareu a formar part del creixent número d'usuaris que té, i us podreu aprofitar de la documentació que es pot trobar a Internet (exemples, manuals, FAQ, …). Actualment, les darreres versions de compiladors de C++ i Visual C++ per a PC ja porten la STL incorporada.

Estructuració de la Standard Template Library

La llibreria STL es pot estructurar en els següents components principals:

Contenidors: Classes template de les classes contenidor més utilitzades.
Algorismes: Funcions template dels algorismes més típics.

Amb aquest parell de components tindríem els algorismes + estructures de dades genèrics. No obstant l'anàlisi realitzat per Stepanov-Lee del problema els va portar a dissenyar alguns components més que són bàsics.

Iteradors: Mecanisme per recorrer els objectes que es troben dins d'un contenidor. El concepte d'iterador és una de les parts fonamentals que fa que la STL sigui com és.

Objectes Funció: Mecanisme per poder proveir una funció a un algorisme a través d'un objecte.

Adaptador: Classes per adaptar un component STL proveint-li un interface diferent.

Contenidors

La idea de contenidors és la consequència lògica de l'aplicació de templates en les estructures de dades clàssiques (llistes, arbres, …). Cadascun dels contenidors tindrà la seva implementació interna característica, i oferirà una interfície determinada. Un dels detalls clau del disseny fet per Stepanov-Lee és que tots els contenidors tenen les funcions begin() i end() que retornen un iterador al primer element i a l'element següent del darrer que contenen, i és mitjançant els iteradors com es passen els contenidors als algorismes. Cal doncs aclarir realment què és un iterador, doncs això donarà una visió global de com funciona la llibreria.

Iteradors

El motiu de l'existència del iteradors en la STL és el de donar-li una uniformitat en el tractament dels contenidors. En poques paraules podríem definir que el concepte d'iterador és la generalització del concepte d'apuntador per qualsevol disposició a memòria dels objectes continguts en un contenidor.

Un contenidor molt simple seria un array d'objectes d'un determinat tipus.

```
Int     VectorInt[10]={0,1,2,3,4,5,6,7,8,9};
int     *P = &VectorInt[0];
```

Si tenim un apuntador a un element de l'array, sabem que:

- El contingut de l'apuntador retornarà l'objecte al que està apuntant.
  ```
  Cout << *P;     // Veurem 0 a pantalla;
  ```
- Incrementar en 1 P, farà que aquest apunti al següent element;
  ```
  P++;
  Cout << *P;     // Veurem 1 a pantalla
  ```

Stepanov-Lee van traslladar aquest comportament als diferents contenidors que van definir. Al definir un contenidor (llista, conjunt, …) d'un determinat tipus, interiorment es defineix un tipus iterador per aquest tipus de dades, i dos objectes iterador que 'apuntaran' al primer i darrer element que hi hagi en el contenidor. El contingut d'un iterador ens donarà l'objecte al que està apuntant, i si incrementem un iterador en 1, l'iterador apuntarà al següent element del contenidor, de la mateixa manera que ho fa un apuntador a una adreça de memòria. La diferència està en que amb un apuntador això es pot fer perquè els objectes es troben en posicions contigues a memòria, mentre en els iteradors no hi ha cap restricció respecte la ubicació dels objectes a memòria, doncs el que actua és la sobrecàrrega del operadors ++, --, *, …

Imagineu que es vol treure per pantalla els elements que hi ha en una llista. Amb una llista STL això es podria fer així.

```
#include <list.h>

list<int> a(10);

list<int>::iterator i,j;      // Creem dos objectes del tipus iterador
                              // definit per llistes d'enters.

i=a.begin();                  // Obtenim iteradors al primer i darrer element de la llista.
j=a.end();

while(i!=j){                  // Mostrem per pantalla els elements de la llista a.
        cout << *i;
        i++;
}
```

I el que s'ha fet aquí en llistes, es podria fer per exemple en arbres, donant al operador ++ del iterador el significat de recórrer els seus element en un determinat ordre (preordre, postordre, …).

Una característica important que han de complir els iteradors indicadors dels elements extrems d'un contenidor és la de *rang vàlid*. Els iteradors retornats per `begin()` i `end()` són els llindars d'un determinat rang

```
[ a.begin(), a.end() )
```

`begin` apunta al primer element del contenidor, mentre que `end` apunta al que en anglès anomenen *past-the-end element*; és a dir, l'element després de l'ultim.

El concepte de rang vàlid implica que si `a=begin()` i `b=end()`, després d'aplicar a++ un número finit de vegades s'arribarà a complir la condició `a==b`. És a dir, `b` és accessible a partir de `a`,

Algorismes

La decisió presa per Stepanov-Lee a l'hora de dissenyar les funcions templates encarregades de dur a terme diferents algorismes és consonant amb la filosofia dels iteradors. Les funcions template no rebran com a paràmetre un determinat objecte de tipus contenidor, sinó que rebran *rangs vàlids* mitjançant els iteradors (apuntadors) als elements del contenidor a qui aplicar el determinat algorisme.

Per exemple, la primera aproximació a una funció sort genèrica seria pensar a passar el contenidor com a paràmetre, i que aquest s'ordenés:

```
#include <algo.h>
#include <vector.h>

vector<int>       a;
….
sort(a);
```

L'aproximació feta a la STL és la següent,

```
sort(a.begin(), a.end());
```

Es passen els iteradors dels extrems de la llista (o qualsevol altre contenidor) a ordenar. D'aquesta manera la mateixa funció es podria utilitzar per ordenar només part del contenidor. El que si cal complir és que a partir del primer iterador especificat es pugui arribar al segon.

Objectes Funció

Hi ha alguns algorismes en els quals és gairebé una necessitat el poder passar-los-hi una funció com a paràmetre. Penseu per exemple que teniu una llista de persones i que voleu tenir-la ordenada segons dos criteris diferents (alfabèticament segons cognom, alfabèticament segons població de residència, ….). En aquests casos podríem pensar en passar la funció de comparació com a paràmetre. Normalment en C el que es fa és construir una funció (Funcio) a la qual se li passa com a paràmetre l'adreça d'una altra funció (FuncParam), la qual rep com a paràmetre uns objectes d'un tipus determinat.

```
void Funcio( int (*FuncParam)(void*,void*));
```

Fixeu-vos que d'entrada ja s'està limitant el tipus de funcions que es passaran com a paràmetre. Es demana una funció amb un tipus de retorn determinat, i amb uns paràmetres determinats. La manera com STL aconsegueix que es pugui passar qualsevol tipus de funció com a paràmetre és el mecanisme dels objectes funció.

Es tracta de que si mitjançant templates podem passar objectes de qualsevol tipus, podem pensar en passar una determinada funció 'amagant-la' dins d'un objecte, fent ús de la idea de 'cavall de Troia'. Aquesta és la interfície de la funció sort de STL:

```
template <class RandomAccessIterator, class Compare>
void sort(RandomAccessIterator first, RandomAccessIterator last,
                                                Compare comp)
```

A aquesta funció se li passa un iterador d'inici i un de final, i un objecte on anirà a buscar la funció amb el criteri d'ordenació. Per conveni la funció que s'anirà a buscar serà la sobrecàrrega del parèntesi. Així doncs, per utilitzar aquesta funció sort primer es crearà una nova classe, que no cal que tingui variables membre (però en podria tenir), on es sobrecarregarà l'operador() amb el criteri d'ordenació propi. Al cridar aquesta funció, seguidament als iteradors requerits es passarà un objecte d'aquesta classe com a tercer paràmetre.

```
#include <vector.h>
#include <algo.h>

Class OrdenaPoblacio: public binary_function<Persona, Persona, bool>
{
public:
	bool operator() (const Persona& personaA, const Persona& personaB)
const
	{
		return(personaA.Poblacio() < personaB.Poblacio());
	}
};

vector<Persona> VectorAmics;

sort(VectorAmics.begin(), VectorAmics.end(),OrdenaPoblacio());
```

Fixeu-vos que la classe que s'utilitzarà per crear objectes funció deriva d'una classe STL binary_function. STL proveeix dues classes base unary_function i binary_function des d'on es deriven les classes pels objectes funció que es passen als algorismes de STL. Mitjançant aquestes classes base, STL pot realitzar una comprovació dels tipus passats com a paràmetre als objectes funció (aconseguint una detecció d'errors en temps de compilació). Els objectes funció que es poden passar als algorismes STL o bé són unaris o binaris.

Adaptadors

STL disposa d'un mecanisme per adaptar la interfície o comportament aparent dels propis objectes cap a una altra. Així doncs es poden adaptar els contenidors perquè es comportin d'una altra manera, modificant la interficie visible que es tingui d'ells. Per exemple, es pot adaptar una llista a una pila. A partir d'aquí les nostres insercions a la llista les fariem mitjançant les funcions push(), i en treuriem elements mitjançant pop(), respectant el comportament normal d'una pila.

```
stack<list<char> > a;
```

STL també permet adaptar els iteradors per modificar-ne el seu funcionament (manera com es recorren els objectes d'un contenidor).

Els adaptadors de funcions són de gran utilitat. En concret hi ha els adaptador negadors, que neguen el resultat booleà que pugui retornar un determinat objecte funció, i els binders, que permeten passar objectes funció de binaris a unaris (fixant a un valor concret un dels seus paràmetres).

Exemple: Recompte del número d'amics que tenim a Manresa.

```
Class ComparaPoblacio: public binary_function<Persona, string, bool>
{
public:
	bool operator() (const Persona& persona, const string& poblacio) const
	{
		return(persona.Poblacio() ==poblacio);
	}
};
```

```
list<Persona> LlistaAmics;
….
int  Num;
count(LlistaAmics.begin(),LlistaAmics.end(),
        bind2nd(ComparaPoblacio,"Manresa"),Num);
```

La funció ComparaPoblacio espera rebre dos paràmetres. Utilitzant bind2nd fixem el segon paràmetre al valor Manresa, de manera que obtenim una nova funció que només espera un paràmetre Persona.

Una aportació interessant de la STL: Els iteradors de streams

STL permet tractar els streams de C++ com iteradors. L' aventatge de poder fer això és que es poden aplicar els algorismes de la STL sobre l'entrada de teclat, la sortida a pantalla, fitxers, … Anem a veure amb un petit exemple com podríem treballar-hi.

Exemple: Còpia a pantalla dels enters d'una llista.

```
#include <list.h>
#include <algo.h>

list<int>Llista(10);
ostream_iterator<int> Pantalla(cout,",");

copy(Llista.begin(), Llista.end(), Pantalla);
```