Programación con la STL: algoritmos genéricos

Dr. J.B. Hayet

CENTRO DE INVESTIGACIÓN EN MATEMÁTICAS

Octubre 2009

Outline

1. Algoritmos genéricos
2. Objetos funciones

Outline

1 Algoritmos genéricos

2 Objetos funciones

Previously en la clase

La librería estándar provee herramientas muy convenientes para facilitar la programación. Es el caso por ejemplo de los string, que permiten evitar los problemas mas clásicos con las cadenas de caracteres “a la C”

```
string s1("Primera");
string s2 = "Segunda";
string s3 = s1+" y "+s2;
s3.append("y aun mas");
```

Algoritmos en la libstdc++

La librería estándar también provee, fuera de toda clase/patrón, código genérico para algoritmos, implementado como funciones patrones. Son abstraciones de funciones recurrentes en código C++.

```
#include <algorithm>
...
double src[] = {10.0,30.0,4.0,7.0,35.0};
const size_t s = sizeof(src)/sizeof(src[0]);
double dst[s];
copy(src,src+s,dst);
```

Algoritmos en la libstdc++

Esta primera función copia todos los elementos dentro de un arreglo, dentro de unos limites (dados en terminos de apuntadores), hacia otro arreglo. La ventaja es obvia: se ahorra escribir ciclos!

Como esta implementado como template, se puede usar cualquier tipo de apuntador.

```
int src2[] = {10,30,4,7,35};
const size_t s2 = sizeof(src2)/sizeof(src2[0]);
int dst2[s2];
copy(src2,src2+s2,dst2);
```

Algoritmos en la libstdc++

Podemos añadir:

```
if (equal(src2,src2+s2,dst2))
  cout << "perfect copy" << endl;
```

que nos imprime:

```
perfect copy
```

La función patrón equal recorre todos los elementos designados a través de los dos primeros valores (un valor de inicio del "apuntador" y un valor que no alcanzar) y checa si son iguales a los del segundo "contenedor".

Algoritmos en la libstdc++

Se podría pensar que el código de copy se parece a:

```
template<typename T> void copy(T* begin,
                               T* end,
                               T* dest) {
  while(begin != end)
    *dest++ = *begin++;
}
```

Pero es aun mejor porque se puede usar también iteradores!

Algoritmos en la libstdc++

Ejemplo:

```
double src[] = {10.0,30.0,4.0,7.0,35.0};
const size_t s = sizeof(src)/sizeof(src[0]);
list<double> l(src,src+s); // Notar el constructor
list<double> l2(s); // Aqui tambien
copy(l.begin(), l.end(), l2.begin());
```

Aquí el contenedor $l2$ tiene el tamaño adecuado desde la construcción.

Algoritmos en la libstdc++

Notar que lo siguiente falla:

```
double src[] = {10.0,30.0,4.0,7.0,35.0};
const size_t s = sizeof(src)/sizeof(src[0]);
list<double> l(src,src+s);
list<double> l2;
copy(l.begin(), l.end(), l2.begin());
```

Porque copy es un patrón para actuar sobre todo lo que parece a apuntadores o iteradores pero no usando especificicamente métodos de listas (por ejemplo para añadir nuevos elementos...)

Algoritmos en la libstdc++

El verdadero copy se parece a

```
template<typename Iterator>
void copy(Iterator begin,
          Iterator end,
          Iterator dest) {
  while(begin != end)
    *dest++ = *begin++;
}
```

Algoritmos en la libstdc++

Pero el código arriba podría no fallar al usar un tipo especial de iterador, definido en el caso de listas, en particular, los iteradores de inserción:

```
double src[] = {10.0,30.0,4.0,7.0,35.0};
const size_t s = sizeof(src)/sizeof(src[0]);
list<double> l(src,src+s);
list<double> l2;
copy(l.begin(), l.end(), back_inserter(l2));
```

Algoritmos en la libstdc++

Estos algoritmos pre-implementados de la librería estándar implementan típicamente operaciones en ciclos.

- Escritos genericamente como patrones, en general con iteradores.
- Ahorras lineas de código al usarles.
- Son cajas negras, entonces tienes que confiar en los diseñadores.
- Normalmente están implementados de manera eficiente: implementan funcionalidad dada y garantizan una complejidad dada ($\mathrm{count\_if}$ toma N operaciones) o el orden de magnitud de la complejidad (sort en promedio en $N \log N$).

Algoritmos en la libstdc++: predicados

Muchos algoritmos vienen con condiciones, expresadas a partir de funciones de evaluación

```
// Evaluacion del predicado
bool test(double x) { return x>10.0; }
...
double src[] = {10.0,30.0,4.0,7.0,35.0};
const size_t s = sizeof(src)/sizeof(src[0]);
list<double> l(src,src+s);
list<double> l2;
remove_copy_if(l.begin(),l.end(),
                back_inserter(l2),test);
list<double>::iterator begin2 = l2.begin();
while(begin2 != l2.end())
  cout << *begin2++ << " ";
```

Algoritmos en la libstdc++

...que nos da:

```
10
4
7
```

Los nombres de las funciones con predicados son generalmente doThis_orDoThatOtherwise_if o simplemente doThis_if.

Algoritmos en la libstdc++

Otro ejemplo con string:

```
// Checa si la palabra tiene un y
bool tieneUnY(const string& s) {
  return s.find('y') != string::npos;
}
...
string src[] = {"que", "onda", "guey"};
const size_t s = sizeof src / sizeof src[0];
string dst[s];
string* enddst = replace_copy_if(src,src+s,dst,
                 tieneUnY, string("amigo"));
string* begindst = dst;
while(begindst != enddst)
   cout << *begindst++ << endl;
```

Un ciclo cabe en la sola llamada de replace_copy_if.

Algoritmos en la libstdc++

Otra posibilidad es actuar directamente sobre el contenedor/apuntador que está pasado como argumento (en sitio):

```
string src[] = {"que", "onda", "guey"};
const size_t s = sizeof src / sizeof src[0];
replace_if(src,src+s,
           tieneUnY, string("amigo"));
string* beginsrc = src;
while(beginsrc != src+s)
    cout << *beginsrc++ << endl;
```

El replace_if es de tipo void.

Algoritmos en la libstdc++

- Notar que se puede combinar los tipos de contenedores en las operaciones de copia, listas con apuntadores por ejemplo.

```
list<double> l(src,src+s);
double dst3[s];
copy(l.begin(), l.end(), dst3);
```

- Como se puede ver, el único ciclo que nos queda es el de la impresión sobre la salida estándar. Si hay para los otros tipos de operaciones como copia, reemplazo deberá de haber algo para la impresión en flujos de *output*...

Algoritmos en la libstdc++: IO

BINGO !

```
#include <iterator>
...
string src[] = {"que", "onda", "guey"};
const size_t s = sizeof src / sizeof src[0];
string dst[s];
replace_copy_if(src,src+s,
                ostream_iterator<string>(cout,"***\n"
                tieneUnY, string("amigo"));
```

que imprime

```
que***
onda***
amigo***
```

Algoritmos en la libstdc++: IO

- Estamos restringidos por el patrón de las funciones-algoritmos de usar iteradores.
- Este iterador que usamos acá es un tipo de iterador especial, formado a partir de un patrón, de un flujo (objeto $\mathrm{ostream}$) y de un carácter adicional.
- La diferencia con un iterador normal es que el operador de asignación por copia ("=") es muy diferente del de un iterador normal: los datos no están copiados en un contenedor sino enviados hacia el flujo pasado a la construcción, con el carácter adicional.

Algoritmos en la libstdc++: IO

Igualmente se puede redigirir de esta manera la copia del contenido de un contenedor a un archivo de texto:

```
#include <iterator>
#include <fstream>
...
string src[] = {"que", "onda", "guey"};
const size_t s = sizeof src / sizeof src[0];
string dst[s];
ofstream outf("salida.txt");
replace_copy_if(src,src+s,
                ostream_iterator<string>(outf,"***\n"
                tieneUnY, string("amigo"));
```

Algoritmos en la libstdc++: IO

Igual para los inputs: cada operación de dereferenciación del iterador provoca la lectura del dato desde un flujo input con un iterador especial, istream_iterator

```
#include <iterator>
#include <fstream>
...
ifstream inf("entrada.txt");
replace_copy_if(istream_iterator<string>(inf),
                istream_iterator<string>(),
                ostream_iterator<string>(cout,"***\n"
                tieneUnY, string("amigo")));
```

Algoritmos en la libstdc++: IO

- Notar que el constructor por default del iterador sobre flujos de entrada equivale a un "end()" y se usa para detectar el final del "contenedor" simulado a partir del archivo.
- El uso de todo eso puede parecer artificial pero a largo termino se hace muy útil.

Outline

1 Algoritmos genéricos

2 Objetos funciones

Objetos funciones

En muchas ocasiones se necesita parametrizar una función.

- Pasar el parámetro como argumento es la manera mas clásica de hacerlo pero a veces el prototipo de la función está impuesto:

```
bool test(double x) { return x>5.0; }
```

- Hasta ahora, vimos con los patrones posibilidad de pasarle a una función un parámetro, pero la función está generada al momento de la compilación.
- ¿Como hacer para tener una función parametrizada según tus necesidades, a la ejecución?

Objetos funciones

Los objetos funciones son instancias de clases "clásicas" que tienen la particularidad de implementar un operador "()"

```
class testParam {
  double umbral;
public:
  testParam(double val) : umbral(val) {}
  bool operator()(double d) { return d>umbral; }
};
...
testParam exTest(1.0);
if (exTest(2.0))  // Pensar exTest.operador()(2.0)
    cout << "OK" << endl;
if (exTest(0.0))
    cout << "AIE" << endl;
```

Objetos funciones

- Esos objetos permiten hacer código de patrones manipulando "pseudo-funciones" que estan prototipeadas como requerido, y tomando eventualmente unos parámetros.
- Pueden estar pasados a las funciones de manipulación de contenedores.
- Esas clases de objetos funciones merecen estar implementados a partir de patrones, y la librería estándar provee las herramientas para usarlos así.

Objetos funciones: creación automática

La librería estándar tiene

- una colección de objetos funciones patrones:

```
template <class T> class greater<T>;
template <class T> class equal_to<T>;
```

- funciones para crear automáticamente otros objetos funciones a partir de (1) objetos funciones o funciones y (2) parámetros que van a estar usados como argumentos.

Objetos funciones estándar

Existe una jerarquía de objetos funciones implementados, que se usan con:

```
#include <functional>
```

Estan dividios entre objetos heredando de unary_function y binary_function

```
namespace std {
  template <class Arg, class Result>
  struct unary_function{
    typedef Arg argument_type;
    typedef Result result_type;
  };
}
```

Objetos funciones estándar

Nombre	Operacion
plus	$x + y$
minus	$x - y$
multiplies	$x * y$
divides	x/y
modulus	$x\%y$
negate	$-x$
equal_to	$x == y$
not_equal_to	$x! = y$
greater	$x > y$
less	$x < y$
greater_equal	$x >= y$
less_equal	$x <= y$
logical_and	$x\&\&y$
logical_or	$x\|\|y$
logical_not	$!x$

Encapsulan operadores...

Objetos funciones: creación automática

Funciones de binding:

```
template <class Operation>
 class binder1st;

 template <class Operation, class T>
 binder1st<Operation> bind1st(const Operation&,
                              const T&);

 template <class Operation>
 class binder2nd ;

 template <class Operation, class T>
 binder2nd<Operation> bind2nd(const Operation&,
                              const T&);
```

Objetos funciones: creación automática

Funciones de binding:

```
template <class Operation>
  class binder1st
     : public unary_function<typename
                             Operation::second_argume
                             typename Operation::resu
  {
  public:

     binder1st(const Operation&,
               const typename Operation::first_argumer

     typename Operation::result_type
     operator() (const typename
                 Operation::second_argument_type&) c
  };
```

Objetos funciones: creación automática

- aceptan objetos funciones binarios,
- la salida de esas funciones es un objeto de tipo objeto función: tiene el operador "()" implementado,
- esos objetos solo encapsulan el objeto funcion (o el apuntador a la funcion) y el parametro,
- a la llamada con ese objeto se llama la función binaria, poniendo como primer ($1\mathrm{st}$) o segundo ($2\mathrm{nd}$) argumento el parámetro usado en la función $\mathrm{bind1st}$ o en el constructor.

Objetos funciones: creación automática

```
// Evaluación del predicado
bool test(double x) { return x>10.0; }
...
double src[] = {10.0,30.0,4.0,7.0,35.0};
const size_t s = sizeof(src)/sizeof(src[0]);
list<double> l(src,src+s);
remove_copy_if(l.begin(),l.end(),
               ostream_iterator<double>(cout," "),
               bind2nd(greater<double>(),10.0));
```

```
10 4 7
```

Objetos funciones: creación automática

Otro ejemplo:

```
double src[] = {10.0,30.0,4.0,7.0,35.0};
const size_t s = sizeof(src)/sizeof(src[0]);
cout << count_if(src, src + s,
                 not1(bind1st(less<double>(),10.0)));
```

3

El $\mathrm{not1}$ es un ejemplo de adaptador de objetos funcion que toma en entrada un objeto funcion unario y que sale otro objeto funcion cuya llamada regresa la negacion logica de la llamada al primero objeto funcion. Equivalente a un ciclo con:

```
if (!less<double>(10.0,el)) {
        ++count;
}
```

Objetos funciones: creación automática

Las funciones de este tipo (adaptadores) como $\mathrm{bind2nd}$, $\mathrm{not1}$... están implementadas como patrones y regresan objetos de tipo objetos funciones: esos a su vez implementan el operador "()" que tiene que estar calificado, en particular en cuanto a los tipos de sus argumentos y de sus valores de regreso!

La solución implementada es de arreglarse de tal manera que los objetos funciones que se manipulan contienen la información del tipo de regreso y del o de los argumentos (1 o 2) (esos objetos funciones se llaman adaptables).

Objetos funciones: creación automática

Por eso todos hereden de:

```
template <class _Arg, class _Result>
    struct unary_function {
      typedef _Arg argument_type;
      typedef _Result result_type;
    };
 template <class _Arg1, class _Arg2, class _Result>
    struct binary_function {
      typedef _Arg1 first_argument_type;
      typedef _Arg2 second_argument_type;
      typedef _Result result_type;
    };
```

Objetos funciones estándar

Dentro de la librería estándar:

```
template <class _Predicate>
  class binary_negate
  : public binary_function<typename _Predicate::first_argument_type
                           typename _Predicate::second_argument_typ
                           bool> {
  protected:
    _Predicate _M_pred;
  public:
    explicit
    binary_negate(const _Predicate& __x)
    : _M_pred(__x) { }
    bool
    operator()(const typename _Predicate::first_argument_type& __x,
               const typename _Predicate::second_argument_type& __y
    { return !_M_pred(__x, __y); }
  };

template <class _Predicate>
  inline binary_negate<_Predicate>
  not2(const _Predicate& __pred)
  { return binary_negate<_Predicate>(__pred); }
```

Objetos funciones: creación automática

O al nivel de los bindings:

```
template<class Op, class T>
binder1st<Op> bind1st(const Op& f, const T& val) {
  typedef typename Op::first_argument_type Arg1_t;
  return binder1st<Op>(f, Arg1_t(val));
}
...
typename Op::result_type
operator()(const typename Op::second_argument_type& x
  const;
```

Objetos funciones a partir de apuntadores a funciones

Evidentemente, sería interesante usar esos mecanismos con funciones tuyas, pero la mayoría de las funciones de la librería estándar esperan objetos con (por ejemplo) $\mathrm{second\_argument\_type}$, $\mathrm{result\_type}$...

Existen herramientas para pasar de una representación a otra. En particular, $\mathrm{ptr\_fun}$ es una función que construye un objeto función a partir de una función dada.

Objetos funciones a partir de apuntadores a funciones

```
bool isEven(int x) { return x % 2 == 0; }
...
int a[]={11,7,3,2,8};
const size_t s = sizeof(a)/sizeof(a[0]);
cout << count_if(a,a+s,not1(ptr_fun(isEven)));
```

Existe el equivalente para funciones binarias.

Objetos funciones a partir de apuntadores a funciones

```
template<class Arg, class Result>
class pointer_to_unary_function
: public unary_function<Arg, Result> {
  Result (*fptr)(Arg); // Stores the f-ptr
public:
  pointer_to_unary_function(Result (*x)(Arg)) : fptr(x
  Result operator()(Arg x) const { return fptr(x); }
};
template<class Arg, class Result>
pointer_to_unary_function<Arg, Result>
ptr_fun(Result (*fptr)(Arg)) {
  return pointer_to_unary_function<Arg, Result>(fptr)
}
```

Objetos funciones a partir de apuntadores a funciones

Puede haber problemas con funciones sobrecargadas

```
bind2nd(ptr_fun(pow), 2.0);
```

```
error: no matching function for call
to ptr_fun(<unresolved overloaded function type>)
```

En este caso, hay que explicitar el template:

```
bind2nd(ptr_fun<double, double, double>(pow), 2.0);
```

Objetos funciones a partir de métodos

La cosa se hace mas complicada para invocar métodos desde objetos o apuntadores hacia objetos (los objetos manipulados no son mas argumentos sino los objetos a partir de que llamar los métodos). Existe para eso unas clases adecuadas y las funciones de adaptación ligadas: mem_fun y mem_fun_ref.

Básicamente, esos objetos funciones reimplentan el operador "()" de tal manera a llamar el método cuyo apuntador ha sido pasado desde el objeto "argumento"

Objetos funciones a partir de métodos

Implementación en la librería estándar:

```
template <class _Ret, class _Tp>
class mem_fun_t : public unary_function<_Tp*,_Ret> {
    public:
      explicit mem_fun_t(_Ret (_Tp::*__pf)())
      : _M_f(__pf) {}
      _Ret
      operator()(_Tp* __p) const
      { return (__p->*_M_f)(); }
    private:
      _Ret (_Tp::*_M_f)();
    };
```

Objetos funciones a partir de métodos

- la función mem_fun es para apuntadores a objetos
- la función mem_fun_ref es para referencias (¿cuál es la diferencia en la implementación?)
- ambas estan sobrecargadas para métodos const
- ambas estan sobrecargadas para métodos a uno o cero argumento

Objetos funciones a partir de métodos

Primer ejemplo:

```
vector<Shape*> vs;
vs.push_back(new Circle);
vs.push_back(new Square);
for_each(vs.begin(), vs.end(),
         mem_fun(&Shape::draw));
```

Objetos funciones a partir de métodos

Segundo ejemplo:

```
int a1[] = {11,22,7,33};
int a2[] = {23,-1,3,6};
const size_t s1 = sizeof(a1)/sizeof(a1[0]);
const size_t s2 = sizeof(a2)/sizeof(a2[0]);

list<int> l1(a1,a1+s1);
list<int> l2(a2,a2+s2);
// Lista de listas
list<list<int>* > l;
l.insert(l.begin(),&l1);
l.insert(l.begin(),&l2);
// Clear each list in the list
for_each(l.begin(),l.end(),
         mem_fun(&list<int>::clear));
```

Función transform

De mismo tipo que for_each o copy, es una meta función que puede actuar sobre contenedores a traves de iteradores:

```
int a1[] = {11,22,7,33};
const size_t s1 = sizeof(a1)/sizeof(a1[0]);
list<int> l1(a1,a1+s1);
transform(l1.begin(), l1.end(),
   ostream_iterator<int>(cout,"\n"),
   bind2nd(plus<int>(), 10));
```

21
32
17
43

Función transform

Toma en cuarto argumento una función unaria que regresa un objeto de tipo conforme al tercer argumento (contenedor de regreso), y como argumento un objeto conforme a los dos primeros argumentos.

```
bool isEven(int x) { return x % 2 == 0; }
...
int a[]={11,7,3,2,8};
const size_t s = sizeof(a)/sizeof(a[0]);
list<int> l;
transform(a,a+s,back_inserter(l),
          not1(ptr_fun(isEven)));
```

Función transform

- También existe para aplicar una función a todos los elementos de un contenedor con parámetros veniendo de otro contenedor y almacenar los resultados en un tercer contenedor.
- En todos casos, es imperativo que el objeto función pasado a transform regrese un objeto (no void).
- Con for_each no se puede pasar mas de un argumento (o métodos sin argumento).

Función transform

Ejemplo de la segunda forma (con argumentos)

```
vector<Shape> shapes;
vector<Shape> filledShapes;
...
int colors[] = { 133, 222,11, 48, 95 };
transform(shapes.begin(), shapes.end(), colors,
   filledShapes,
   mem_fun_ref(&Shape::fill));
cout << endl;
```

Nuevos adaptadores

Cadenas que convertir en flotantes:

```
vector<string> vs;
...
const char* vcp[s];
transform(vs.begin(), vs.end(), vcp,
  mem_fun_ref(&string::c_str));
vector<double> vd;
transform(vcp, vcp + s, back_inserter(vd),
  std::atof);
```

¿No se podría hacer en una vez?

Nuevos adaptadores

Manera 1:

```
template<typename R, typename E, typename F1, typename F2>
class unary_composer {
  F1 f1;
  F2 f2;
public:
  unary_composer(F1 fone, F2 ftwo) : f1(fone), f2(ftwo) {}
  R operator()(E x) { return f1(f2(x)); }
};

template<typename R, typename E, typename F1, typename F2>
unary_composer<R, E, F1, F2> compose(F1 f1, F2 f2) {
  return unary_composer<R, E, F1, F2>(f1, f2);
}
```

Nuevos adaptadores

Manera 1:

```
int main() {
  double x = compose<double, const string&>(
    atof, mem_fun_ref(&string::c_str))("12.34");
}
```

Nuevos adaptadores

Manera 2: mejor usar objetos funciones adaptables (no se necesitaria especificar parametros de templates)

```
template<typename F1, typename F2> class unary_composer
: public unary_function<typename F2::argument_type,
                        typename F1::result_type> {
  F1 f1;
  F2 f2;
public:
  unary_composer(F1 f1, F2 f2) : f1(f1), f2(f2) {}
  typename F1::result_type
  operator()(typename F2::argument_type x) {
    return f1(f2(x));
  }
};
template<typename F1, typename F2>
unary_composer<F1, F2> compose(F1 f1, F2 f2) {
  return unary_composer<F1, F2>(f1, f2);
}
```

Nuevos adaptadores

Manera 2: ahora tenemos que usar ptr_fun o mem_fun

```
vector<string> vs;
 ...
transform(vs.begin(), vs.end(),
   back_inserter(vd),
   compose(ptr_fun(std::atof),
           mem_fun_ref(&string::c_str)));
```