Vectores (Vector)

Agustín J. González

ELO320

Plantillas (Templates)

- Como podemos definir una colección de datos sin comprometer el tipo de datos siendo agrupados?
- La idea de una template es poder parametrizar el tipo de datos que una clase puede contener
- Se puede pensar un template como una función con parámetro, en donde el parámetro es un tipo de dato.

Ejemplos: Plantillas (Templates)

- Ejemplo de Template de una función

```
template <class T>
 T max(T a, T b)
   // return the maximum of a and b
{
   if (a < b)
            return b;
   return a;
}
template <class T>
 void swap (T & a, T & b)
   // swap the values held by a and b
{
   T temp = a;
   a = b;
   b = temp;
}
```

Declaración de tipos basados en templates

- Para declarar un valor (variable, o mejor objeto) con un tipo template (plantilla), el tipo se indica en paréntesis.

```
vector<int>       a(10);
vector<double>  b(30);
vector<string>   c(15);
```

Operaciones sobre vectores

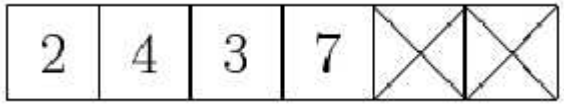
Table 8.1 Operations for the `vector` data type

Constructors		
`vector<T> v;`	Default constructor	$O(1)$
`vector<T> v (int);`	Initialized with explicit size	$O(n)$
`vector<T> v (int, T);`	Size and initial value	$O(n)$
`vector<T> v (aVector);`	Copy constructor	$O(n)$
Element Access		
`v[i]`	Subscript access, can be assignment target	$O(1)$
`v.front ()`	First value in collection	$O(1)$
`v.back ()`	Last value in collection	$O(1)$
Insertion		
`v.push_back (T)`	Push element on to back of vector	$O(1)$ [a]
`v.insert(iterator, T)`	Insert new element after iterator	$O(n)$
`v.swap(vector<T>)`	Swap values with another vector	$O(n)$
Removal		
`v.pop_back ()`	Pop element from back of vector	$O(1)$
`v.erase(iterator)`	Remove single element	$O(n)$
`v.erase(iterator, iterator)`	Remove range of values	$O(n)$
Size		
`v.capacity ()`	Maximum elements buffer can hold	$O(1)$
`v.size ()`	Number of elements currently held	$O(1)$
`v.resize (unsigned, T)`	Change to size, padding with value	$O(n)$
`v.reserve (unsigned)`	Set physical buffer size	$O(n)$
`v.empty ()`	True if vector is empty	$O(1)$
Iterators		
`vector<T>::iterator itr`	Declare a new iterator	$O(1)$
`v.begin ()`	Starting iterator	$O(1)$
`v.end ()`	Ending iterator	$O(1)$

a. push_back can be $O(n)$ if reallocation of the internal buffer is necessary, otherwise it is $O(1)$.

Tamaño de un vector

- El vector mantiene un buffer interno. El tamaño de este buffer permite almacenar al menos tantos elementos como los contenidos en el vector.
- Los dos tamaños pueden ser accedidos y cambiados a través de llamados a funciones.



Ejemplo: generador de sentencias

- Cada sentencia puede ser vista como la combinación de un sujeto un verbo y un complemento.
- Asignaremos tres vectores inicialmente vacíos para cada uno de estas categorías de datos.

vector <string> sujeto, verbo, complemento.

- Luego ponemos valores. Los vectores se adecuan al tamaño automáticamente.

```
sujeto.push_back("Paula");
sujeto.push_back("Gato");
sujeto.push_back("gente");
sujeto.push_back("Profe");
verbo.push_back("come");
verbo.push_back("pinta");
verbo.push_back("ayuda");
```

Ejemplo: generador de sentencias

```
complemento.push_back(“ratón”);
complemento.push_back(“estudiantes”);
complemento.push_back(“cuadros”);
```

- Luego podemos formar sentencias con

```
for (int i=0; i < 10;  i++)
    cout << sujeto[rand()%sujeto.size()]
         << “ “ verbo[rand()%verbo.size()]
         << “ “ complemento[rand()%complemento.size()];
```

- Una posible salida es:

```
Gato come ratón
Paula pinta cuadros
Gente come cuadros
    ...
```

Algoritmos genéricos útiles con vectores

Table 8.2 Generic algorithms useful with vectors

`fill (iterator start, iterator stop, value)`
Fill vector with a given initial value

`copy (iterator start, iterator stop, iterator destination)`
Copy one sequence into another

`max_element(iterator start, iterator stop)`
Find largest value in collection

`min_element(iterator start, iterator stop)`
Find smallest value in collection

`reverse (iterator start, iterator stop)`
Reverse elements in the collection

`count (iterator start, iterator stop, target value, counter)`
Count elements that match target value, incrementing counter

`count_if (iterator start, iterator stop, unary fun, counter)`
Count elements that satisfy function, incrementing counter

`transform (iterator start, iterator stop, iterator destination, unary)`
Transform elements using unary function from source, placing into destination

`find (iterator start, iterator stop, value)`
Find value in collection, returning iterator for location

`find_if (iterator start, iterator stop, unary function)`
Find value for which function is true, returning iterator for location

`replace (iterator start, iterator stop, target value, replacement value)`
Replace target element with replacement value

`replace_if (iterator start, iterator stop, unary fun, replacement value)`
Replace elements for which fun is true with replacement value

`sort (iterator start, iterator stop)`
Places elements into ascending order

`for_each (iterator start, iterator stop, function)`
Execute function on each element of vector

`iter_swap (iterator, iterator)`
Swap the values specified by two iterators

Algoritmos genéricos útiles con vectores (cont)

Table 8.3 Generic Algorithms useful with Sorted Vectors

`merge(iterator s1, iterator e1, iterator s2, iterator e2, iterator dest)`
Merge two sorted collections into a third

`inplace_merge(iterator start, iterator center, iterator stop)`
Merge two adjacent sorted sequences into one

`binary_search (iterator start, iterator stop, value)`
Search for element within collection, returns a boolean

`lower_bound (iterator start, iterator stop, value)`
Find first location larger than or equal to value, returns an iterator

`upper_bound (iterator start, iterator stop, value)`
Find first element strictly larger than value, returns an iterator

Ejemplo: Cuenta de elementos

```
vector<int>::iterator start = aVec.begin();
vector<int>::iterator stop = aVec.end();

if (find(start, stop, 17) != stop)
        ...         // element has been found

int counter = 0;
count (start, stop, 17, counter);
if (counter != 0)
        ...         // element is in collection
```

La historia sigue.. Por ejemplo listas

Table 9.1 Summary of list operations

Constructors and Assignment		
`list<T> v;`	Default constructor	$O(1)$
`list<T> v (aList);`	Copy constructor	$O(n)$
`l = aList`	Assignment	$O(n)$
`l.swap (aList)`	Swap values with another list	$O(1)$
Element Access		
`l.front ()`	First element in list	$O(1)$
`l.back ()`	Last element in list	$O(1)$
Insertion and Removal		
`l.push_front (value)`	Add value to front of list	$O(1)$
`l.push_back (value)`	Add value to end of list	$O(1)$
`l.insert (iterator, value)`	Insert value at specified location	$O(1)$
`l.pop_front ()`	Remove value from front of list	$O(1)$
`l.pop_back ()`	Remove value from end of list	$O(1)$
`l.erase (iterator)`	Remove referenced element	$O(1)$
`l.erase (iterator,iterator)`	Remove range of elements	$O(1)$ [a]
`l.remove (value)`	Remove all occurrences of value	$O(n)$
`l.remove_if (predicate)`	Removal all values that match condition	$O(n)$
Size		
`l.empty ()`	True if collection is empty	$O(1)$
`l.size ()`	Return number of elements in collection	$O(n)$ [b]
Iterators		
`list<T>::iterator itr`	Declare a new iterator	$O(1)$
`l.begin ()`	Starting iterator	$O(1)$
`l.end ()`	Ending iterator	$O(1)$
`l.rbegin ()`	Starting backwards moving iterator	$O(1)$
`l.rend ()`	Ending backwards moving iterator	$O(1)$
Miscellaneous		
`l.reverse ()`	Reverse order of elements	$O(n)$
`l.sort ()`	Place elements into ascending order	$O(n \log n)$
`l.sort (comparison)`	Order using comparison function	$O(n \log n)$
`l.merge (list)`	Merge with another ordered list	$O(n)$

a. Freeing the memory used by erased cells will require time proportional to the number of elements deleted.

b. Some implementations keep track of the number of elements in a list, and thus can determine the size in $O(1)$.

Tipos para organizar colecciones (Contenedores)

Structure	Addition of new element	Removal of first element	Removal of middle element	Inclusion test
Vector	$O(1)$ or $O(n)$ †	$O(n)$	$O(1)$ or $O(n)$ †	$O(n)$ or $O(\log n)$ ‡
indexed, random access to elements, bounded size				
List	$O(1)$	$O(1)$	$O(1)$	$O(n)$
sequential access to elements, rapid insertion and removal				
Deque	$O(1)$	$O(1)$	$O(n)$	$O(n)$ or $O(\log n)$ ‡
random access, rapid insertion to front and back				
Stack	$O(1)$	$O(1)$	NA	NA
insertion and removal only from front				
Queue	$O(1)$	$O(1)$	NA	NA
insertion only from front, removal only from back				
Priority Queue	$O(\log n)$	$O(1)$ or $O(\log n)$ §	NA	NA
rapid removal of largest element				
Set	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
ordered collection, unique values, rapid insertion, removal and test				
a `multiset` allows repeated elements				
Map	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
collection of key-value pairs				
a `multimap` allows multiple elements with same key				
Notes	†constant if accessing existing position, linear if inserting/removing ‡logarithmic if ordered, linear if not ordered §constant access time, logarithmic removal			