

Introducción al Cómputo

Standard Template Library
Contenedores y secuencias
Algoritmos

Templates - STL

- ❑ No vamos a entrar en detalles sobre cómo programar clases o funciones template.
- Si vamos a aprender a utilizar la biblioteca standard de C++.
- Standard Template Library. Conocida como STL.
- Abstracciones implementadas con templates.
- ❖ ¡No hay que reinventar la rueda!

STL – Overview

- String library: `string`, `regex`, etc.
- I/O streams: `cout`, `cin`, `fstream`, `stringstream`, etc.
- Containers: `array`, `vector`, `list`, `queue`, `stack`, `deque`, `priority_queue`, `map`, `set`, `multimap`, `unordered_map`, etc.
- Iteradores (secuencias): `iterator`, etc.
- Algoritmos: `find`, `copy`, `transform`, `sort`, `replace`, `unique`, `merge`, etc.
- Utilities: `cmath`, `complex`, `exception`, `pair`, `valarray`, `chrono`, `random`, `numeric`, etc.
- Multithreading y concurrencia. Memoria y recursos. C Standard Library.
- ❖ Ref: cplusplus.com, en.cppreference.com, es.cppreference.com

STL – string

- **string** es un alias al template **basic_string<char>**.
- Contenedor de caracteres con operaciones convenientes:
 - Constructores y operaciones con string nativos literals ("hola").
 - Operadores de comparación y concatenación: **==**, **!=**, **<**, **<=**, **>=**, **+**, **+=**.
 - Operaciones sobre tamaño: **size()**, **length()**, **resize()**, **reserve()**, **capacity()**, **clear()**, **empty()**.
 - Operaciones de acceso: **[]**, **at()**, **front()**, **back()**, **push_back()**, **c_str()**, **copy()**.
 - Secuencias: **begin()**, **end()**, **rbegin()**, **rend()**.
 - Búsqueda con la familia de **finds**.
 - Manejo de substrings: **substr()**.
 - I/O: **<<**, **>>**
 - Otras operaciones: **assign()**, **append()**, **insert()**, **erase()**, **replace()**.
 - Etc...
- ❖ Ref: cplusplus.com, en.cppreference.com, es.cppreference.com

STL – Contenedores

- Los contenedores mantienen una colección de objetos.
- Gestionan el espacio de almacenamiento de sus elementos y proporciona funciones miembro para acceder a ellos, ya sea directamente o mediante iteradores (objetos con propiedades similares a los punteros).
- Categorías por tipo de acceso:
 - De secuencias: proveen acceso a la secuencia de elementos contenidos.
 - Asociativos: proveen acceso a los elementos a través de una búsqueda asociativa según una clave (key).

❖ Ref: cplusplus.com, en.cppreference.com, es.cppreference.com

Contenedores de secuencias

- `array<class T, size_t N>`: Arreglo contiguo de tamaño `N` estático de `Ts`.
- `vector<class T>`: Arreglo contiguo dinámico de `Ts`. Elección de contenedor por defecto.
- `list<class T>`: Lista doblemente ligada de `Ts`. Se utiliza cuando es necesario insertar y eliminar elementos sin mover otros ya existentes.
- `forward_list<class T>`: Lista simplemente ligada de `Ts`. Ideal para secuencias cortas o vacías.
- `deque<class T>`: Cola de doble extremo de `Ts`. Intermedio entre `vector` y `list`.

Contenedores Asociativos Ordenados

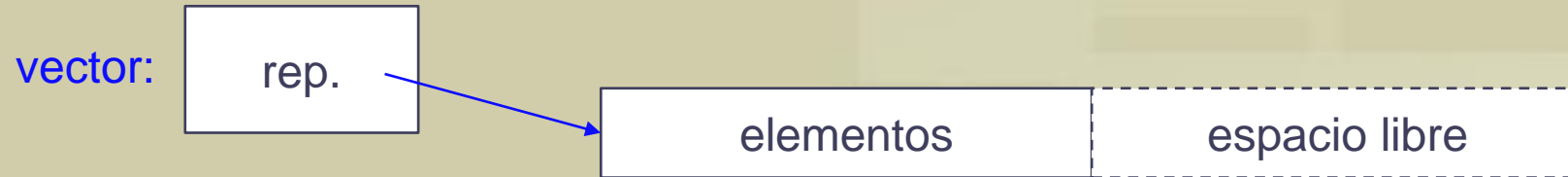
- **map<K, V, C>**: Colección de pares clave-valor (**K**, **V**) ordenados con comparación **C** por clave única.
- **multimap<K, V, C>**: Colección de pares (**K**, **V**) ordenados con comparación **C** por clave. Las claves pueden estar repetidas.
- **set<K, C>**: Conjunto de claves **Ks** únicas ordenado con comparación **C** por clave.
- **multiset<K, C>**: Conjunto de claves **Ks** ordenado con comparación **C** por clave. Las claves pueden estar repetidas.
 - Estos contenedores usualmente están implementados como árboles binarios balanceados (Red-Black Trees). Búsqueda rápida con complejidad $O(\log n)$.
 - El criterio por defecto para comparar las claves **K** es **less<K>**.

Contenedores Asociativos Desordenados

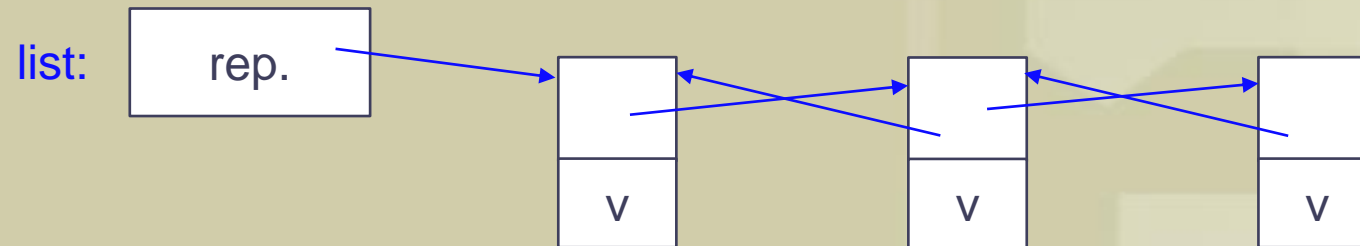
- `unordered_map<K,V,H,E>`: Colección de pares clave-valor (K,V) desordenados con clave única.
- `unordered_multimap<K,V,H,E>`: Colección de pares (K,V) desordenados. Las claves pueden estar repetidas.
- `unordered_set<K,C>`: Conjunto de claves K s únicas desordenado.
- `unordered_multiset<K,C>`: Conjunto de claves K s desordenado. Las claves pueden estar repetidas.
- Estos contenedores están implementados como hashes. El tipo de función de hash H para el tipo K por defecto es `hash<K>`. Búsqueda muy rápida con complejidad $O(1)$ amortizado y $O(n)$ para el peor caso.
- El criterio por defecto de igualdad E para claves K es `equal_to<K>`.

Representación

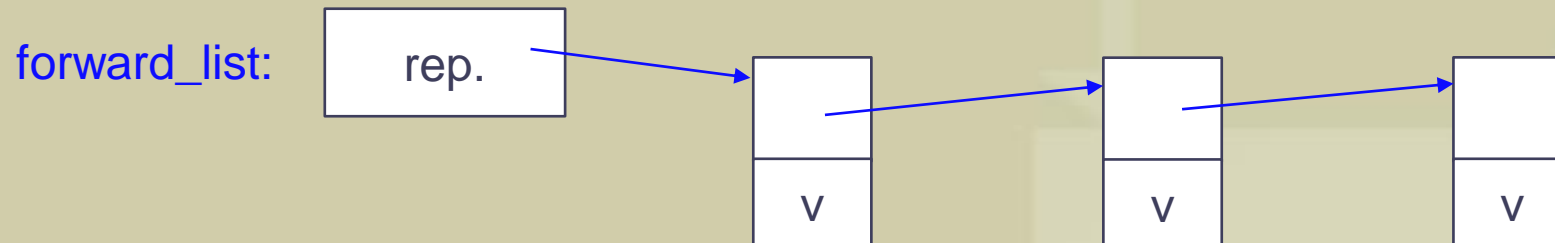
- **vector**: muy parecido a nuestro **Vector**. Va a contener un puntero al arreglo de elementos, la cantidad de elementos y su capacidad.



- **list**: secuencia de nodos doblemente ligados, número de elementos.

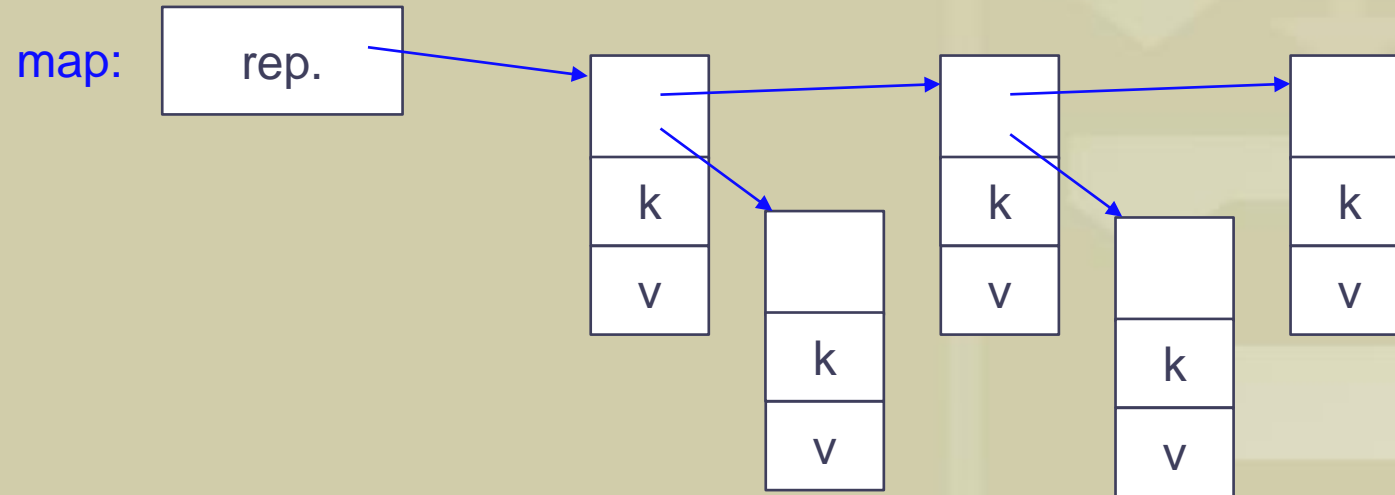


- **forward_list**: secuencia de nodos simplemente ligados.



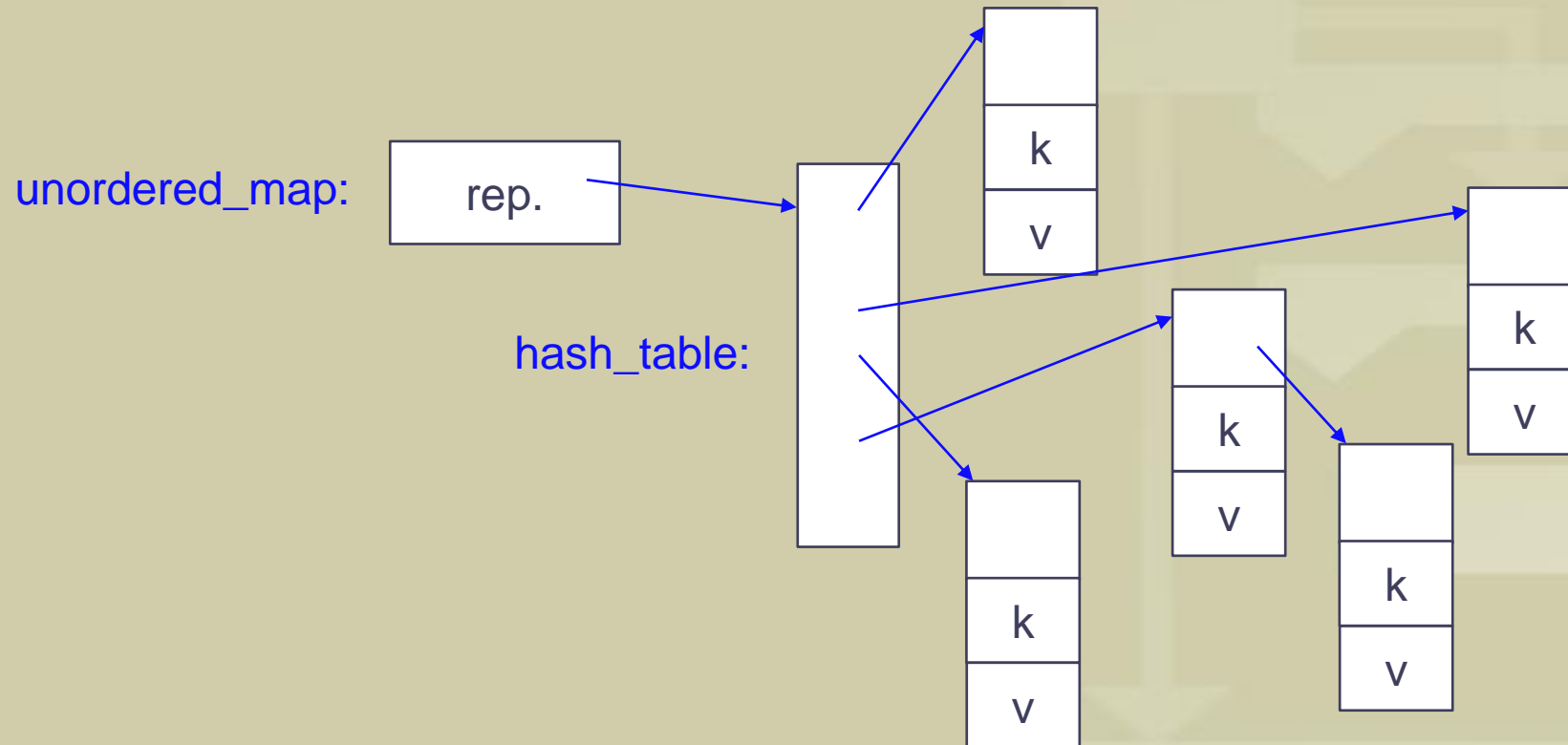
Representación

- **map**: árbol binario balanceado de nodos con pares (key, value).



Representación

- **unordered_map**: implementado como una table de hash.



unordered_map<K, V, H, E>

Operaciones sobre Contenedores

- Operaciones comunes a todos los contenedores:
 - Constructores y destructores
 - Asignación y comparación: `=`, `==`, `!=`
 - Acceso a la secuencia: `begin()`, `end()`.
 - `size()`, `swap()`.
- Otras operaciones:
 - De acceso indexado a elementos con `[]` y `at()`.
 - De lista: `remove()`, `unique()`, `merge()`, `sort()`.
 - En el comienzo: `front()`, `push_front()`, `pop_front()`.
 - Al final: `back()`, `push_back()`, `pop_back()`.
- ❖ Tabla de todas las funciones miembro: en.cppreference.com

Ejemplos vector

```
void f(vector<int> &v, int i1, int i2)
{
    int sum = 0;
    for( int i = 0; i < v.size(); i++ )
        sum += v[i];        // indice chequeado, uso v[i] no chequeado

    v.at(i1) = v.at(i2); // hay que chequear el rango, puede disparar
    // ...                // excepcion out_of_range
}

void g(vector<double> &v)
{
    double d = v[v.size()];    // undefined, indices de 0 a size()-1
}
```

Ejemplos vector

```
struct Record { string s; int i; double d; };

vector<Record> vr(10000);           // cada elemento es
                                   // inicializado con Record()

void f(int s1, int s2) {
    vector<int> vi(s1);             // cada elemento es
                                   // inicializado con 0

    vector<double> *p = new vector<double>(s2);
}

class Num {
public:
    Num(long);    // sin constructor default
    // ...
};

vector<Num> v1(1000);               // error, no hay un constructor default Num()
vector<Num> v2(1000, Num(0));      // ok
```

Ejemplos vector

```
vector<int> v1(10);           // ok: vector de 10 ints
vector<int> v2 = vector<int>(10) // ok: vector de 10 ints
vector<int> v3 = v2           // ok: v3 es una copia de v2
vector<int> v4 = 10;          // error: intenta utilizar una conversión
                              // implícita de 10 a vector<int>

void f1(vector<int> &);
void f2(const vector<int> &);
void f3(vector<int>);

void h() {
    vector<int> v(10000);

    f1(v);           // pasa por referencia
    f2(v);           // pasa por referencia
    f3(v);           // copia los 10000 elementos a un vector temporario!
}
```

Ejemplos vector

```
class Book {
    // ...
};

void f(vector<Num> &vn, vector<char> &vc, vector<Book> &vb, list<Book> &lb)
{
    vn.assign(10, Num(0));           // asigna 10 copias de Num(0) a vn

    char s[] = "hola";
    vc.assign(s, &s[sizeof(s)-1]);  // asigna "hola" a vc

    vb.assign(lb.begin(), lb.end());
}

void g()
{
    vector<char> v(10, 'x');          // v.size() == 10, cada elemento es 'x'
    v.assign(5, 'a');                // v.size() == 5, cada elemento es 'a'
}
```


Ejemplos vector

```
vector<Point> cities;

void add_points(Point sentinel)
{
    Point buf;

    while( cin >> buf ) {
        if( buff == sentinel )
            return;
        cities.push_back(buf);
    }
}

void f()
{
    vector<int> v;
    v.pop_back();           // undefined, v esta vacío
    v.push_back(7);        // undefined, v en estado indefinido
}
```

Ejemplos list

```
struct Entry {
    string name;
    int number;
};

list<Entry> phone_book;

void print_entry(const string &s)
{
    for( auto &e : phone_book ) {
        if( s == e.name )
            cout << e.name << ' ' << e.number << endl;
    }
}

void add_entries(Entry &e, list<Entry>::iterator iter) {
    phone_book.push_front(e);    // agrega al comienzo
    phone_book.push_back(e);     // agrega al final
    phone_book.insert(iter, e);  // agrega antes de 'iter'
}
```

Ejemplos list

```
void f()
{
    int myints[]= {17,89,7,14};
    list<int> mylist (myints,myints+4);

    mylist.remove(89);

    cout << "mylist contains: ";
    for( auto v : mylist )
        cout << ' ' << *it;
    cout << '\n';
}
```

Ejemplos list

```
void g()
{
    list<int> mylist;

    for( int i=1; i<10; ++i ) mylist.push_back(i);

    mylist.resize(5);
    mylist.resize(8,100);
    mylist.resize(12);

    cout << "mylist contains:";
    for( auto v : mylist )
        cout << ' ' << v;
    cout << '\n';
}
```

Ejemplos map

```
map<string, int> m1;  
map<string, int, Nocase> m2;  
map<string, int, String_cmp> m3;  
  
void f(map<string, int> &m)  
{  
    auto p = m.find("Gold");  
    if( p != m.end() ) {                                     // si encuentro oro  
        // ...  
    } else if( (p = m.find("Silver")) != m.end() ) {        // si no, busca plata  
        // ...  
    }  
}
```

Ejemplos map

```
void f(map<string, int> &m) {
    pair<string, int> p99("Paul", 99);

    auto p = m.insert(p99);
    if( p.second ) {
        // Paul fue insertado
    } else {
        // Paul ya estaba
    }
}

void g(map<string, int> &m)
{
    m["Dilbert"] = 3;                // bonito, posiblemente ineficiente
}

void h(map<string, int> &m)
{
    int count = phone_book.erase("Ratbert");
    // ...
}
```

Container Operation Complexity

Container	[]	List	Front	Back
vector	Const	$O(n)+$		Const+
list		Const	Const	Const
forward_list		Const	Const	
deque	Const	$O(n)$	Const	Const
map	$O(\log(n))$	$O(\log(n))+$		
set, multimap, multiset		$O(\log(n))+$		
unordered_map	Const+	Const+		
unordered_[set, multimap, multiset]		Const+		
string	Const	$O(n)+$	$O(n)+$	Const+
array	Const			
Built-in array	Const			

Container Complexity

```
static const size_t MAX_FIND_COUNT = 100000;

int main()
{
    vector<string> wordsvec;
    string str;

    ifstream ifs("words");
    while( ifs >> str )
        wordsvec.push_back(str);

    vector<string> quests = wordsvec;

    random_shuffle(wordsvec.begin(), wordsvec.end());

    vector<string> svec;
    set<string> sset;
    unordered_set<string> suset;

    // to be continued
```


Container Complexity

```
vector<string> svec;  
set<string> sset;  
unordered_set<string> suset;  
  
auto t0 = high_resolution_clock::now();  
sset.insert(wordsvec.begin(), wordsvec.end());  
auto t1 = high_resolution_clock::now();  
  
auto dtus = duration_cast<microseconds>(t1-t0).count();  
cout << wordsvec.size() << " inserts en set: "  
      << dtus << " usecs\n";  
  
t0 = high_resolution_clock::now();  
suset.insert(wordsvec.begin(), wordsvec.end());  
t1 = high_resolution_clock::now();  
cout << wordsvec.size() << " inserts en unordered_set: "  
      << duration_cast<microseconds>(t1-t0).count() << " usecs\n";  
  
// to be continued...
```

Container Complexity

```
for( size_t cnt = 10; cnt <= MAX_FIND_COUNT && cnt <= wordsvec.size(); cnt *= 10 ) {  
  
    svec.clear();  
    sset.clear();  
    suset.clear();  
  
    svec.assign(wordsvec.begin(), wordsvec.begin() + cnt);  
    sset.insert(wordsvec.begin(), wordsvec.begin() + cnt);  
    suset.insert(wordsvec.begin(), wordsvec.begin() + cnt);  
  
    t0 = high_resolution_clock::now();  
    for( size_t i = 0; i != cnt; ++i )  
        find(svec.begin(), svec.end(), quests[i]);  
    t1 = high_resolution_clock::now();  
    cout << cnt << " finds en vector: "  
        << duration_cast<microseconds>(t1-t0).count()/(1.*cnt) << " usecs\n";  
  
    t0 = high_resolution_clock::now();  
    for( size_t i = 0; i != cnt; ++i )  
        sset.find(quests[i]);  
    t1 = high_resolution_clock::now();  
    cout << cnt << " finds en set: "  
        << duration_cast<microseconds>(t1-t0).count()/(1.*cnt) << " usecs\n";  
}
```

Container Complexity

```
for( size_t cnt = 10; cnt <= MAX_FIND_COUNT && cnt <= wordsvec.size(); cnt *= 10 ) {  
  
    // ...  
  
    t0 = high_resolution_clock::now();  
    for( size_t i = 0; i != cnt; ++i )  
        suset.find(quests[i]);  
    t1 = high_resolution_clock::now();  
    cout << cnt << " finds en unordered_set: "  
        << duration_cast<microseconds>(t1-t0).count()/(1.*cnt) << " usecs\n";  
  
}  
  
return 0;  
  
} // fin de main
```

Container Complexity

```
[user@c7 ~]$ ./cont_comp
479828 inserts en set: 1151214 usecs
479828 inserts en unordered_set: 275443 usecs

10 finds en vector: 1 usecs
10 finds en set: 0.3 usecs
10 finds en unordered_set: 0.3 usecs

100 finds en vector: 1.32 usecs
100 finds en set: 0.23 usecs
100 finds en unordered_set: 0.12 usecs

1000 finds en vector: 24.853 usecs
1000 finds en set: 0.414 usecs
1000 finds en unordered_set: 0.156 usecs

10000 finds en vector: 329.147 usecs
10000 finds en set: 0.5551 usecs
10000 finds en unordered_set: 0.1593 usecs

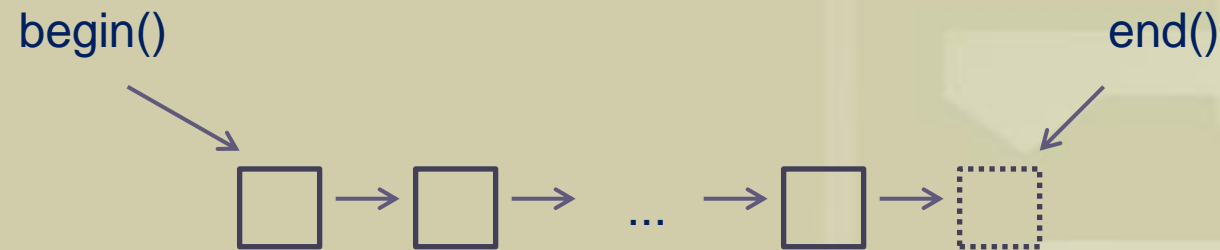
100000 finds too many for vector...
100000 finds en set: 0.76673 usecs
100000 finds en unordered_set: 0.21698 usecs
```

Secuencias – Iteradores

- Un uso muy común de los contenedores es recorrer aplicando un algoritmo la secuencia de sus elementos.
- Para esto cada contenedor define una clase `iterator` apropiada.
- Los iteradores son objetos con operaciones similares a las de un puntero.
 - Permiten acceder a un elemento en un contenedor.
 - Similar a derreferenciar un puntero (`*p`).
 - Permiten acceder al elemento siguiente en la secuencia de un contenedor.
 - Similar a sumarle 1 a un puntero para acceder al siguiente elemento de un arreglo (`++p` o `p++`).

Secuencias – Iteradores

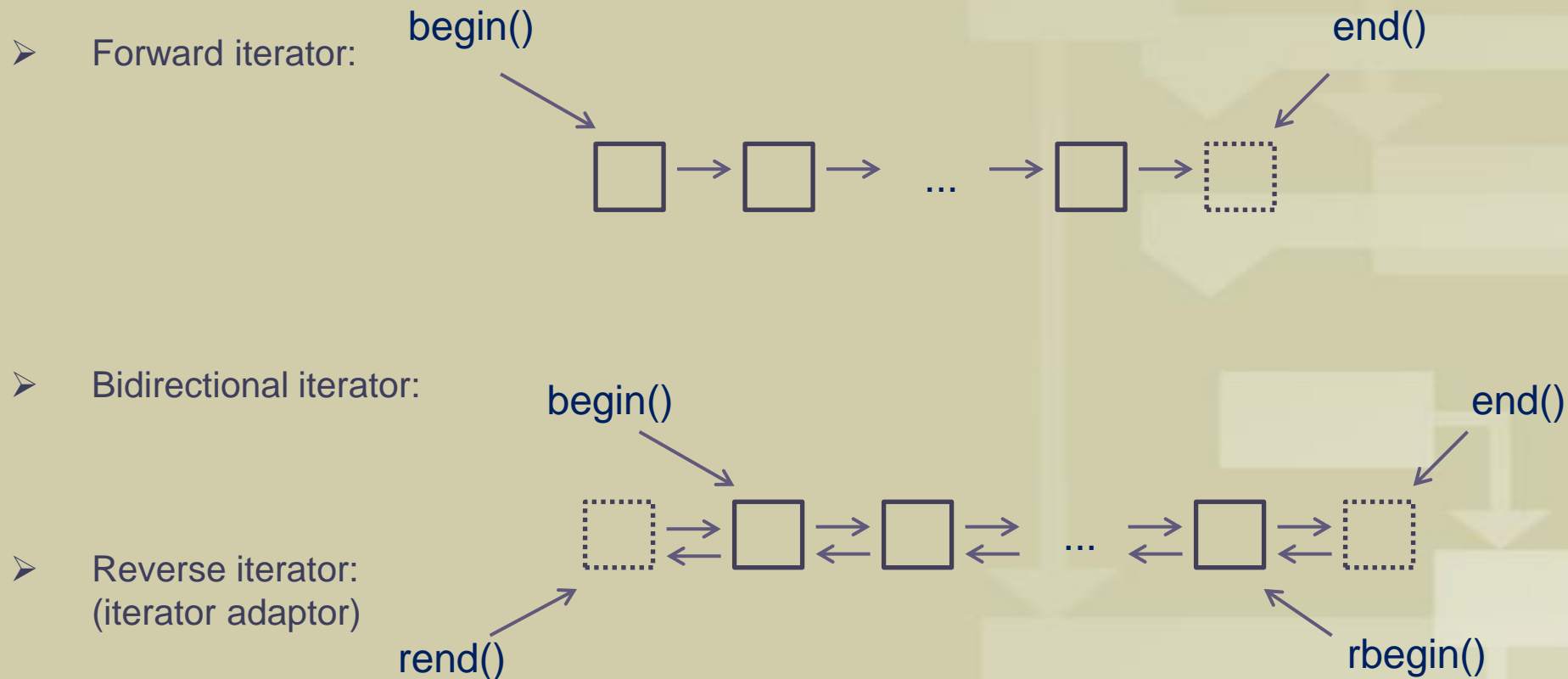
- Una secuencia está definida por un par de iteradores que especifican el primer y “uno más allá del último” elementos en la secuencia.



- Una secuencia define elementos en el rango `[begin, end)`

Secuencias – Iteradores

- Existen distintos tipos de iteradores, varían según el contenedor.



Categorías de Iteradores

- Distintos tipos de iteradores, según el contenedor.
 - Iterator: operaciones mínimas: `*` y `++`.
 - Forward Iterator: `==`, `!=`, `->`, lectura y escritura múltiples.
 - Bidirectional Iterator: `--`.
 - Random-access iterator: `[]`, `+`, `+=`, `-`, `-=`, `<`, `<=`, `>` y `>=`.
- Output Iterator: Iterador escritura única.
- Input Iterator: Forward Iterator con lectura única.

Iteradores de Contenedores

Container	Iterator Category
<code>vector</code>	Random-access
<code>list</code>	Bidirectional
<code>forward_list</code>	Forward
<code>deque</code>	Random-access
<code>map</code> , <code>set</code> , <code>multimap</code> , <code>multiset</code>	Bidirectional
<code>unordered_map</code> , <code>unordered_set</code> <code>unordered_multimap</code> , <code>unordered_multiset</code>	Forward
<code>string</code>	Random-access
<code>array</code>	Random-access
Built-in array	Random-access

Algoritmos

- Un algoritmo es un conjunto finito de reglas que definen una serie de operaciones para resolver un problema específico.
- En C++ los algoritmos están implementados como funciones template que operan sobre secuencias de elementos, esto es un par de iteradores [begin, end) o [first, last).
- Como funcionan sobre secuencias, son independientes del contenedor específico sobre el que operan (**vector**, **list**, **map**, etc.).
- En STL existen cerca de 100 algoritmos ya implementados. Están parametrizados con funtores para hacerlos generales.

❖ Referencia algoritmos: en.cppreference.com

Algoritmos

➤ Algoritmos:

- Que no modifican la secuencia:
 - `for_each()`, `all_of()`, `any_of()`, `none_of()`
 - `count()`, `count_if()`,
 - Familia de `find()`s
 - `equal()`, `mismatch()`
- Que modifican la secuencia:
 - `transform()`
 - Familia de `copy()`s
 - `unique()`, `remove()`, `replace()`
 - `rotate()`, `random_shuffle()`, particiones y permutaciones
- Ordenamiento y búsqueda:
 - `sort()` y sus variantes
 - Familia de `binary_search()`s
 - `merge()`, algoritmos de conjuntos

Algoritmos: Argumentos de Política

- Muchos de los algoritmos de STL vienen en 2 versiones:
 - Sencilla: realiza acciones utilizando operaciones convencionales, como `<` y `==`.
 - Versión que recibe la operación clave como argumento.

```
template<class Iter>
void sort(Iter first, Iter last)
{
    // ... sort using e1 < e2 ...
}

template<class Iter, class Pred>
void sort(Iter first, Iter last, Pred pred)
{
    // ... sort using pred(e1, e2) ...
}
```

- Las operaciones pasadas como argumentos pueden modificar los elementos. La mayoría no lo hacen, solo sirven para comparación. Se llaman predicados.

Ejemplos de Algoritmos

```
void f(const string &s) {  
    auto n_space = count(s.begin(), s.end(), ' ');  
    auto n_whitespace = count_if(s.begin(), s.end(), isspace);  
  
    // ...  
}  
  
array<int> x = {1,3,4 };  
array<int> y = {0,2,3,4,5};  
  
void g() {  
    auto p = find_first_of(x.begin(), x.end(), y.begin(), y.end);  
        // p = &x[1]  
    auto q = find_first_of(p+1, x.end(), y.begin(), y.end());  
        // q = &x[2]  
}
```

Ejemplos de Algoritmos: `copy()`

```
void f() {  
    int myints[]={10,20,30,40,50,60,70};  
    vector<int> myvector (7);  
  
    copy ( myints, myints+7, myvector.begin() );  
  
    cout << "myvector contains:";  
    for ( auto i : myvector )  
        cout << ' ' << *it;  
    cout << '\n';  
}
```

```
template<class InIt, class OutIt>  
OutIt copy(InIt first, InIt last, OutIt result)  
{  
    while( first != last ) {  
        *result = *first;  
        ++result; ++first;  
    }  
    return result;  
}
```

Ejemplos de Algoritmos: `transform()`

```
int op_increase (int i) { return ++i; }

void f() {
    std::vector<int> foo;
    std::vector<int> bar;

    for (int i=1; i<6; i++)
        foo.push_back (i*10);      // foo: 10 20 30 40 50

    bar.resize(foo.size());        // allocate space

    transform(foo.begin(), foo.end(), bar.begin(), op_increase);
                                   // bar: 11 21 31 41 51

    // std::plus adds together its two arguments:
    transform(foo.begin(), foo.end(),
               bar.begin(), foo.begin(), std::plus<int>());
                                   // foo: 21 41 61 81 101}
```

Ejemplos de Algoritmos: `transform()`

```
template <class InIt, class OutIt, class UnaryOp>
OutIt transform(InIt first1, InIt last1,
               OutIt result, UnaryOp un_op)
{
    while( first1 != last1 ) {
        *result = un_op(*first1);
        ++result; ++first1;
    }
    return result;
}

template <class InIt, class OutIt, class BinaryOp>
OutIt transform(InIt first1, InIt last1, InIt first2,
               OutIt result, BinaryOp bin_op) {
    while( first1 != last1 ) {
        *result = bin_op(*first1, *first2);
        ++result; ++first1; ++first2;
    }
    return result;
}
```



Ejemplos de Algoritmos

```
vector<string> fruit;  
fruit.push_back("peach");      fruit.push_back("apple");  
fruit.push_back("kiwi");      fruit.push_back("pear");  
fruit.push_back("starfruit");  fruit.push_back("grape");  
  
// no me gustan las frutas que empiezan con 'p'  
sort(fruit.begin(), fruit.end());  
  
auto p1 = find_if(fruit.begin(), fruit.end(), initial('p'));  
auto p2 = find_if(p1, fruit.end(), initial_not('p'));  
fruit.erase(p1, p2);  
  
// chau starfruit  
fruit.erase(find(fruit.begin(), fruit.end(), "starfruit"));
```

Objetos Función

- Muchos algoritmos reciben predicados para fijar la política de operación.
- Los predicados más comunes están definidos en STL
 - Que no modifican la secuencia:
 - `equal_to<T>(x,y) : x==y` para objetos de tipo T.
 - `not_equal_to<T>(x,y) : x!=y` para objetos de tipo T.
 - `greater<T>(x,y) : x>y` para objetos de tipo T.
 - `less<T>(x,y) : x<y` para objetos de tipo T.
 - `greater_equal<T>(x,y), less_equal<T>(x,y)`
 - `logical_and(), logical_or, logical_not ()`

```
vector<int> v;  
// ...  
// ordena v de mayor a menor  
sort(v.begin(), v.end(), greater<int>{});
```



FIN

Administrativa

- ✓ Última clase: lunes 25/10/2021 : repaso, nueva práctica y consultas pre-parcial
- ✓ Fecha del segundo parcial: lunes 01/11/2021 – 14:30 hs.
- ❖ Choripanes?
- ❖ Finales: diciembre (fechas a definir)