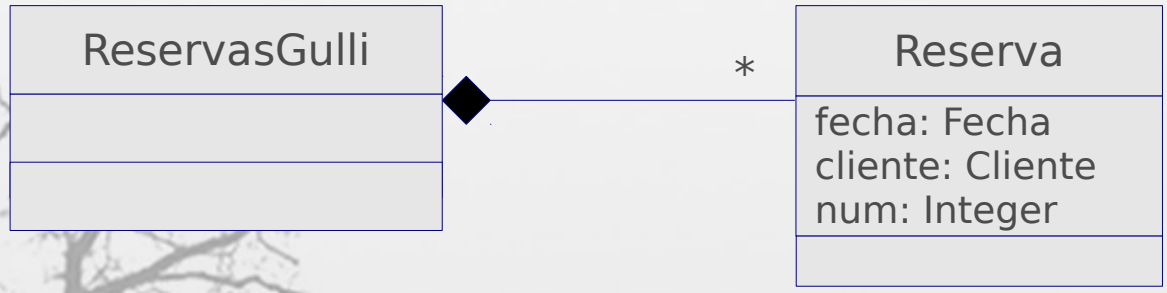
Lección 13:
Conjuntos y Mapas de STL

- Motivación
- Características los contenedores asociativos
- Conjuntos y mapas
- Funciones de comparación
- Conjuntos y multiconjuntos
- Mapas y multimapas
- Clases de comparación
- De ahora en adelante....

Motivación

- El restaurante Gulli está creando un sistema de reservas por internet para sus 20 mesas
- La reserva tiene datos del cliente, el número de comensales, etc.
- Se quieren gestionar las reservas por fechas



- Para hacer búsquedas pueden usarse árboles AVL
- Pero el campo fecha no es clave única, habría que modificar la implementación para soportar claves repetidas

Características de STL

- La ventaja de STL es la facilidad de manejo, la robustez y la implementación eficiente
- Mantienen coherencia en la sintaxis de las operaciones (insert, delete, etc.), al igual que en la sobrecarga de operadores
- STL permite trabajar con contenedores lineales, pero también asociativos con acceso por clave en tiempo logarítmico
- Los contenedores asociativos basados en árboles son los mapas (**map**, **multimap**) y conjuntos (**set**, **multiset**)

Conjuntos y mapas

- Los datos de este tipo de contenedores deben mantener una relación de orden, que se especifica mediante una **función de comparación**
- Los contenedores asociativos son versátiles al permitir **repetir claves** o **diferenciar entre clave y dato**
- De hecho están implementados sobre un **árbol rojo-negro**, un tipo de árbol binario de búsqueda equilibrado que garantiza las operaciones de búsqueda, inserción y borrado en tiempo **O(*log*n)**
- Añaden el puntero al padre, lo que permite realizar recorridos secuenciales en Inorden

Conjuntos y mapas

- Un conjunto (**set**) almacena datos que son directamente valores clave
- El mapa (**map**) sin embargo almacena en cada nodo una pareja (pair) de valores: <clave, dato>
- El prefijo **multi** (multiset y multimap) establece que la clave puede repetirse

	clave == dato	clave != dato
claves repetidas: no	set	map
claves repetidas: si	multiset	multimap

Funciones de comparación

- Para instanciar mapas y conjuntos se necesita especificar la relación de orden entre cada par de elementos, normalmente **menor** o **mayor**
- En STL estos comparadores son:
  - **less<T>**: operator<
  - **greater<T>**: operator>
- El tipo T debe tener implementados el operador correspondiente, si no da un error de compilación
- El operador por defecto es less<T>, quedando los datos ordenados de menor a mayor

Conjuntos (set)

- Utilizamos conjuntos cuando coincidan dato y clave y no se permitan datos repetidos
- Definición:

```
#include <set>
```

  - **set<*tipo_dato*,[*clase_comparación*]>*nombre_set*;**
- Ejemplo de conjunto de enteros de menor a mayor (se ha obviado el operador less<T>)

```
set<int> setA;
```

- Tienen funciones comunes con el resto de contenedores de STL:
  - front(), back(), clear(), empty(), etc.

Conjuntos (set)

- En un conjunto se insertan datos sin indicar la posición:
  - **pair<iterator,bool> insert (T t)**
  - devuelve dos valores: (1) un iterador apuntando al dato insertado y (2) un bool=true si se ha insertado con éxito (no está repetido)
- El borrado debe localizar el dato y luego eliminarlo
  - **erase(T t)**
- Búsqueda: no se justifican los set sin realizar búsquedas
  - **iterator find(T t)**: busca el dato t devolviendo la posición o end() si no está el dato.
  - **iterator lower_bound(T t)**: busca a t; si no está retorna la posición siguiente
  - **iterator upper_bound(T t)**: busca al dato siguiente a t.

Ejemplo (set)

```
int main(void) {
        set<int, less<int> > c, d;
        c.insert(1); c.insert(5);
        c.insert(7); c.insert(2);

        if (c.insert(2).second == false)
        cout << "Error de inserción" << endl;

        set<int, less<int> >::iterator i = c.begin();
        while (i != c.end())
            cout << *(i++) << endl;

         i = c.find(7);
        if (i != c.end())
            cout << "Dato encontrado" << endl;

        i = c.lower_bound(4);
        cout << "El siguiente dato a 4 es " << *i << endl;
        d = c;
        d.erase(5);
}
```

La inserción de 2 no tiene efecto porque ya está el dato

Se muestra:
“Dato encontrado”

El dato no está, muestra 5:
“El siguiente dato a 4”

Operador copia y borrado de un dato

Operaciones de la clase Conjunto

Sobre un set se puede implementar las operaciones de conjunto: unión (**set_union**), intersección (**set_intersection**) y diferencia (**set_difference**)

```
#include <set>
#include <algorithm>
void main (void){
    int mul3[] = {3,6,9,12,15};
    int mul2[] = {2,4,6,8,10,12};
    set<int> a(mul3,mul3+5);
    set<int> b(mul2,mul2+6);
    set<int> c;
    cout << "Union: ";
    set_union(a.begin(), a.end(), b.begin(), b.end(), inserter(c,c.begin()));
    set<int>::iterator itc = c.begin();
    while (itc != c.end()){
        cout << *itc << " ";
        itc++;
    }
    cout << endl;
}
```

a={3,6,9,12,15} // multiplos de 3
b={2,4,6,8,10,12}//multiplos de 2
c=aUb = {2,3,4,6,8,9,10,12,15}

Multi-conjuntos (multiset)

- La única diferencia de un multiset con un set es que permite almacenar datos repetidos
- Definición:

```
#include <set>
```

  - **multiset<*tipo_dato*,*[clase_comparación]*>*nombre_set*;**

- Ejemplo de conjunto de enteros de menor a mayor (se ha obviado el operador less<T>)

```
multiset<int> setA;
```

- Todas las operaciones son similares a las de un set, excepto la inserción:
  - **Iterator insert (T t)**
  - devuelve un iterador al nodo insertado; no devuelve un boolean porque la inserción siempre tiene éxito.

Ejemplo (multiset)

```
int main(void) {
    multiset<int, less<int> > c, d;
    c.insert(1);
    c.insert(5);
    c.insert(7);
    c.insert(2);
    c.insert(2); // Ok.

    multiset<int, less<int> >::iterator i = c.begin();
    while (i != c.end())
        cout << *(i++) << " ";
    cout << "\nEl tamaño del conjunto c es: " << c.size()<<endl;
    i = c.find(7);
    if (i != c.end())
        cout << "Dato encontrado" << endl;
    d = c;
    d.erase(2);
}
```

La inserción de 2 ahora sí tiene éxito

Se muestra:
1 2 2 5 7

Borra todas las ocurrencias de 2

Mapas (map)

- Los mapas diferencian entre clave y dato de modo que los datos quedan ordenados por clave
- Definición:

```
#include <map>
```

  - **map<tipo_clave, tipo_dato, [clase_comparación]> nombre_map;**
- Ejemplo de mapa de empleados (se ha obviado el operador less<T>)

```
map<string, Empleado> empleados;
```

- Cada nodo del árbol tiene una pareja (**pair**) de datos:
  - pair<Tclave,Tdato>
  - **first** y **second** permiten acceder a los datos del pair

Mapas (map)

- La inserción de datos repetidos no es posible, por eso devuelve un pair:
  - **pair<iterator,bool> insert (pair<Tclave,Tdato> t)**
  - que devuelve la posición del dato insertado y true si la inserción es correcta al no haber repetidos
- El mapa sobrecarga el **operator[]** para L/E del dato indicando la clave entre los corchetes
  - **Tdato& operator[](Tclave c)**
  - aunque la sintaxis recuerda al vector, hay que recordar que el tiempo de acceso el logarítmico.
- El resto de operaciones son similares a los conjuntos

Ejemplo (map)

```
int main(void) {
    typedef string dni;
    typedef string nombre;
    map<dni, nombre, less<string> > m;
    pair<dni,nombre> d;
    d.first = "7800900";
    d.second = "Molina Ramírez, Juan";
    m.insert(d);

    m.insert(pair<dni,nombre>("24242424","Ruíz Pérez, Jaime"));
    map<dni,nombre,less<dni> >::iterator i;
    for (i = m.begin(); i != m.end(); i ++)
        cout << (*i).first << " " << (*i).second << endl;

    cout << m["24242424"] << endl;
    m["55689002"] = "Ramírez González, Lucía";
    i = m.find("55689002");
    if (i != m.end())
        cout << (*i).second << endl;
}
```

inserción definiendo previamente un pair

inserción directa acceso con first y second

lectura/escritura con []

Multimapas (multimap)

- Los multimapas permiten almacendar datos con la misma clave
- Definición:

```
#include <map>
```

  - **multimap<tipo_clave, tipo_dato, [clase_comparación]> nombre_multimap;**
- La inserción no falla y no devuelve un pair, sólo un iterador apuntando al nodo insertado
  - **iterator insert (pair<Tclave, Tdato> t)**
- El **operator[]** no está disponible porque la clave puede no ser única

Multimapas (multimap)

Para implementar las reservas del restaurante Gulli se puede utilizar un multimapa usando como clave la clase Fecha (muchas reservas en la misma fecha):

multimap<Fecha, Cliente> reservas;

Si el tipo de la clave es una clase, ésta debe tener sobrecargado el operador de comparación que se use

Fecha debe sobrecargar el operator<

```
class Fecha{
   int d, m, a;
public:
    Fecha (int dd, int mm, int aa): d(dd),m(mm),a(aa){}
    bool operator< (const Fecha &f){
        if (a<f.a) return true;
        else if (a==f.a && m<f.m) return true;
        else if (a==f.a && m==f.m && d<f.d) return true;
        return false;
    }
};
```

Clases de comparación

- Si ningún operador resuelve la relación de orden, se puede implementar una clase de comparación
- Esta nueva clase debe sobrecargar el operator()

```
typedef int posicion;
map <Punto, posicion, mayorY> puntos;
```

```
class Punto{
    int x, y;
public:
    Punto (int x1, int y1): x(x1),y(y1){}
    int leeX(void) const {return x;}
    int leeY(void) const {return y;}
};

class mayorY {
public:
  bool operator()(const Punto &p1, const Punto &p2) const
  {
    return (p1.leeY() > p2.leeY() ||
        (p1.leeY()==p2.leeY() && p1.leeX() > p2.leeX()));
  }
};
```

De ahora en adelante

- Los árboles permiten acceso en tiempo logarítmico para realizar búsquedas de forma eficiente
- Si el problema a resolver no aprovecha este proceso de búsqueda eficiente, no se justifica el uso de contenedores asociativos
- Si se desea mejorar el tiempo logarítmico se puede optar por utilizar tablas de dispersión
- La mayoría de los compiladores C++ incluyen una extensión a STL que permite el manejo de tablas de dispersión de manera similar a como se hace con mapas y conjuntos