UNIVERSITAT POLITÈCNICA DE VALÈNCIA

DSIC DEPARTAMENTO DE SISTEMAS INFORMÁTICOS Y COMPUTACIÓN

Máster Universitario en Computación en la Nube y de Altas Prestciones

Tecnología de la Programación Paralela (TPP).

Ejercicios C++ Sesión 2

Introducción

Esta serie de ejercicios es continuación de la Sesión l y consta de ejercicios avanzados de C++.

Ejercicio 1: Plantillas (`templates`), clases derivadas y más cosas.

1. Tomad el código del ejercicio anterior (Ejercicio 3 de la Sesión 1) y convertir la clase `NumeroR2` en una "plantilla" (`template`) que permita su utilización, por ejemplo, con números reales en simple o doble precisión. Probad la nueva clase con el programa de la actividad anterior.

2. Implementad la clase `Complejo`. Esta clase debe representar a un número complejo. Se implementará como subclase de la clase `NumeroR2<T>` y se añadirá un atributo más que representará el módulo.

   El módulo de un número complejo se define como

   $$\sqrt{x^2 + y^2}\,,$$

   donde x e y son las dos componentes del número. Implementad un método privado que calcule el módulo del número complejo según la fórmula anterior y actualice el atributo creado para albergar el módulo.

3. Implementad los tres constructores que se han implementado para la clase `NumeroR2`. Para su implementación se reutilizará el código de los constructores de la clase `NumeroR2` y dentro de cada constructor se actualizará el atributo que almacena el módulo llamando al método privado construido. Para reutilizar un constructor se emplea la sintaxis siguiente:

   ```
   Complejo() : NumeroR2<T>() { ... }
   ```

4. Implementad el operador `<<` para que muestre los números complejos así: `x + yi`.

5. Implementad todos los operadores de la clase superior (`NumeroR2`) reutilizando código y actualizando el atributo módulo. Por ejemplo, el operador `-=` sería:

   ```
   template<class T>
   Complejo<T>& Complejo<T>::operator-=( const Complejo<T>& c ) {
     NumeroR2<T>::operator-=( c );
     modulo();
     return *this;
   }
   ```

   NOTA 1: Para añadir funcionalidad a un método, se puede reutilizar código de la clase superior utilizando la siguiente sintaxis para llamar al método de la clase superior:

```
ClaseSuperior<T>::metodo(argumentos);
```

También se puede reutilizar un operador. Por ejemplo, para reutilizar el operador `-=` se haría

```
ClaseSuperior<T>::operator-=(argumento);
```

NOTA 2: Si se quiere utilizar un operador de la clase superior tal cual está diseñado allí se puede hacer. Por ejemplo, para reutilizar el operador `=` de la clase superior, es suficiente con añadir lo siguiente en la interfaz de la clase `Complejo`:

```
using NumeroR2<T>::operator=;
```

Sin embargo, si se desea añadir funcionalidad entonces habrá que implementarlo de nuevo. Probad las dos alternativas. Hay que saber también que, justo en este caso, el operador `=` "superficial" (*shallow*) es suficiente dado que todos los atributos son variables simples (no hay punteros). (No se aconseja reutilizar otros operadores de la clase superior con este método.)

Probad el funcionamiento correcto de algunos de los operadores en el programa principal.

6. Implementad los operadores de comparación siguientes:

   `<` `>` `<=` `>=` `==` `!=`

   teniendo en cuenta que los cuatro primeros afectan al módulo del número complejo y para los dos últimos se han de comparar las dos componentes. Sugerencia: implementad los métodos en la interfaz de la clase.

   Probad el funcionamiento correcto de algunos de los operadores en el programa principal.

7. Implementad el operador de llamada a función (`operator()`) de manera que permita actualizar valores a los atributos del número complejo. Por ejemplo, el siguiente código debería funcionar:

   ```
   Complejo<double> e, f, g;
   e(1.0,3.0);
   f(-11.0);
   g = e + f;
   cout << "g = " << g << endl;
   ```

   Para mostrar la salida:

   ```
   g = -10 + 3i
   ```

8. Utilizad el "contenedor" de C++ de tipo `vector` para crear un vector de números complejos de tipo `double`. Existen dos maneras alternativas de realizar esto:

   ```
   vector< Complejo<double> > v;
   ```

   o

   ```
   typedef Complejo<double> cmplx;
   vector< cmplx > v;
   ```

   Ahora, añadid unos cuantos números complejos mediante el método `push_back`.

   Después, utilizad un "iterador" para mostrar todos los números complejos contenidos en el vector. Se trata de obtener el iterador del tipo de dato `vector` (`vector<Complejo<double>>::iterator it`). Este iterador (`it`) se inicializa llamando al método `begin()` sobre el objeto `v`. El iterador se incrementa de uno en uno (`it++`) mientras no lleguemos al final de la colección de objetos del vector `v` (`it!=v.end()`);

   Ahora, haced el mismo recorrido del vector de números complejos mediante la "forma comprimida" del bucle `for`, o también llamada forma de tipo *for-each*.

```
for ( type variable : vector ) {
  // Bloque de código a ejecutar en cada iteración.
}
```

9. Utilizad el algoritmo de C++: `for_each` para mostrar el módulo de todos los números complejos contenidos en el vector utilizado en el ejercicio anterior. Lo primero es incluir el fichero cabecera `algorithm` que contiene los algoritmos de C++ de la STL. Este algoritmo recibe tres argumentos: el primer elemento de la lista, el último y un *functor*. Los dos primeros argumentos son iteradores. El tercero es una clase (o estructura) que implementa el operador de acceso a función u objeto función o una función. El objeto función debe recibir como argumento un elemento de la lista (http://www.cplusplus.com/reference/algorithm/for_each). Este tercer argumento podría ser la propia clase `Complejo` que hemos implementado.

   Realizad la implementación de tres formas distintas:

   a) Utilizando una función. Declarad la siguiente función:

   ```
   template <typename T>
   void modulo ( Complejo<T> c ) {
     cout << c.mod() << endl;
   }
   ```

   Ahora llamad al algoritmo `for_each` utilizando la función `modulo` como tercer argumento.

   b) Utilizando un *functor*. Declarad la clase siguiente:

   ```
   template <typename T>
   class functor {
     public:
       void operator() (Complejo<T> &c) { cout << c.mod() << endl; }
   };
   ```

   Utilizad esta clase como tercer argumento del algoritmo `for_each`.

   c) Utilizando una expresión *lambda*. Declarad la clase siguiente:

   Probad también con la función que permite ordenar los elementos en el vector.

10. Por último, utillizad el algoritmo `sort` para ordenar los números complejos del vector según su módulo. En este caso utilizad únicamente expresiones *lambda*.