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).

Actividades para el aprendizaje de paralelismo con C++

Instrucciones

Este boletín está formado por una serie de actividades. Para cada actividad hay que crear una carpeta con nombre "Actividad_X" donde X indica el número de actividad. Dentro de la carpeta figurarán los códigos implementados y los documentos solicitados. Después se creará un fichero comprimido que incluya todas las carpetas y se subirá a la tarea de PoliformaT correspondiente. Borrad todos los ejecutables antes de crear el archivo comprimido y todos aquellos archivos que no sean el fichero fuente o el documento con resultados solicitado. Solo se solicitan los códigos fuente desarrollados.

Introducción

Esta parte contiene una serie de ejercicios propuestos de C++. El objetivo consiste en familiarizarse con la sintaxis de este lenguaje para que resulte más sencillo el trabajar con Threading Building Blocks (TBB). Se sugiere utilizar siempre el compilador de C++ de GNU (`g++`) con la opción de compilación `-std=c++11`.

Actividad 1: C++, punteros, referencias y sobrecarga.

Implementad un programa en C++ que realice lo siguiente.

1. El programa debe mostrar un mensaje por pantalla utilizando la sentencia:

   ```
   cout << "Hola mundo." << endl;
   ```

2. Realizar lo siguiente:

   a) Declarar un entero (`e`), un puntero que apunte a dicho entero (`p`) y una referencia que apunte al mismo entero también (`r`).

   b) Asignar un valor al entero utilizando la variable `e` (p.e. `e = 2;`) y mostrar su contenido de tres formas diferentes, es decir, utilizando `e`, `p` y `r`.

   c) Asignar un valor al entero utilizando el puntero `p` y mostrar su contenido de tres formas diferentes, es decir, utilizando `e`, `p` y `r`.

   d) Asignar un valor al entero utilizando la referencia o alias `r` y mostrar su contenido de tres formas diferentes, es decir, utilizando `e`, `p` y `r`.

3. Ahora realizad lo siguiente:

a) Implementad una función (`funcion1`) que reciba un puntero a entero y lo modifique asignándole, por ejemplo, un valor constante. Realizad una llamada a esta función para ver su funcionamiento.

b) Implementad otra función (`funcion2`) que realice lo mismo recibiendo una referencia en este caso. Realizad una llamada a esta función para ver su funcionamiento. Observad que esta función puede ser llamada pasándole como argumento, tanto `e` como `r`.

c) Utilizad ahora el mismo nombre para las dos funciones (*sobrecarga*).

Entrega: De esta actividad se entrega solo el código. (**0.15 puntos**)

Actividad 2: Clases en C++.

1. Implementad la clase `Tabla` de manera que contenga dos atributos **privados**, un puntero a vector de enteros y un entero que contiene el número de elementos almacenados. Utilizad el código facilitado (`actividad2.cpp`) en la carpeta de `material` para validar el código.

2. Implementad los constructores:

```
Tabla( );          // Crea una tabla con espacio para 10 elementos
Tabla( int n );    // Crea una tabla con espacio para  n elementos
```

3. Implementad un método (`getN`) que devuelva la cantidad de elementos que hay en la tabla y que tenga la característica de ser `inline`.

4. Implementad el operador de acceso (`[i]`) que devuelva el entero que hay en la posición `i`, de manera que, si `t` es de tipo `Tabla`, se pueda obtener el valor del entero de la posición 3 mediante `t[3]`.

5. Implementad el constructor de copia para que no sea "superficial" (*shallow*):

```
Tabla( Tabla t );  // Crea una tabla copia de otra
```

   Cambiad el último constructor para que reciba una referencia constante.

6. Implementad el destructor:

```
~Tabla( );
```

   de manera que elimine el espacio de memoria reservado para el vector.

7. Implementad también el operador de asignación `=`.

8. Implementad la función "amiga" para mostrar los elementos de una `Tabla` con el prototipo:

```
friend ostream& operator<<( ostream&, const Tabla& );
```

   Declarad la función con a signatura anterior dentro de la clase e implementarla fuera.

9. Ahora hay que añadir al final de la función `main` el código siguiente:

```
Tabla t5{5};
for( auto &e : t5 ) {
  e = rand() % 100;
}
cout << "Tabla 5: " << t5;
```

   y hacer que funcione implementando las funciones `begin()` y `end()` correspondientes.

Entrega: De esta actividad se entrega solo el código. (**0.15 puntos**)

Actividad 3: Clases en C++: más operadores.

Para realizar esta actividad hay que utilizar el código facilitado (`actividad3.cpp`) en la carpeta de `material` para validar el código.

1. Implementad la clase `NumeroR2`. Esta clase debe representar números reales en el espacio bidimensional R2 (R^2), por lo tanto, tendrá dos atributos que serán números reales.

   Hay que implementar los constructores habituales:

   - El constructor por defecto (sin argumentos) que inicializa a cero las componentes del número.
   - El constructor que recibe como argumento dos números reales que serán las componentes del número.
   - El constructor de "copia" que recibe como argumento otro número R2.

   No es necesario implementar el destructor.

2. Implementar el operador `<<` en primer lugar para poder ir comprobando en la salida la implementación de cada método u operador.

3. Implementar las operaciones aritméticas habituales de los números sobrecargando los operadores correspondientes:

   `+=  -=  +  -  ++  --  =`

   Suponemos que el autoincremento (autodecremento) afectan a las dos componentes. Además, hay que implementar la versión prefijo y la versión sufijo de ambos. Sugerencia: implementar los operadores en el orden indicado.

4. Probad el funcionamiento correcto de la clase mediante un programa que pruebe todos los operadores.

Entrega: De esta actividad se entrega solo el código. (**0.23 puntos**)

Actividad 4: Plantillas (`templates`), clases derivadas y más cosas.

1. Tomad el código de la actividad anterior 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> a, b, c;
   a(1.0,3.0);
   b(-11.0);
   c = a + b;
   cout << " c = " << c << endl;
   ```

   Para mostrar la salida:

   ```
   c = -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()`);

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.

   Sugerencia: Implementad lo siguiente:

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

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

   Ahora probad con la función `modulo` y la clase `functor` como tercer argumentos alternativamente.

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

10. Una de las formas más elegantes de realizar lo anterior en C++ moderno consiste en utilizar expresiones *lambda*. Recordad que para utilizar una expresión lambda es necesario utilizar la opción `-std=c++11`.

Entrega: De esta actividad se entrega solo el código. (**0.23 puntos**)

Actividad 5: Concurrencia en C++

En esta actividad se va a reimplementar la Actividad 1 del Boletín de OpenMP utilizando threads. Como una primera aproximación se creará un thread por cada iteración del bucle principal, es decir, cada thread creado se encargará de calcular la media y la desviación típica de un vector. Para ello, se van a seguir los siguientes pasos:

1. Copiar el código secuencial de la carpeta `material` del Boletín de OpenMP cambiando su extensión a `.cpp`.

2. Se creará un vector llamado `task` dinámico (`new`) de threads. (Hay que acordarse de eliminarlo al final del programa.)

3. Dentro de la iteración del bucle se asignará la tarea específica al thread:

   ```
   task[v] = thread{ /* tarea */ };
   ```

   Esta tarea estará formada por todo el código de la iteración. La implementación será realizada mediante una expresión *lambda*. Para construir la *lambda* es conveniente tener en cuenta lo siguiente:

- Las variables privadas a cada thread deberían ser declaradas dentro del cuerpo de instrucciones de la *lambda*.
- Los vectores (`media`, `desvt`, `M` y `tam`) son variables compartidas, pero se asume que cada thread accede a "su" posición dentro de dicho vector, por lo tanto, el código será correcto tanto si se "capturan" por valor (`[=]`) como por referencia (`[=]`).
- Existen dos opciones para pasar la variable `v` a la *lambda*: 1) como argumento del operador `operator()`, o 2) capturada por valor. Se sugiere utilizar la primera forma.

4. Antes de terminar, no hay que olvidar sincronizar los threads.

5. Compilación:

```
g++ -o actividad1 actividad1.cpp -std=c++11 ctimer.c -lpthread
```

Entrega: De esta actividad se entrega solo el código. (**0.24 puntos**)
Extensión de la actividad: Realizar una implementación basada en `async`.