Instituto Superior de Formación Técnica Nº 151

Carrera: Analista de Sistemas
2do Año. Algoritmos y Estructuras de Datos II

Trabajo Práctico Nº 4.1	**Unidad 4.1**
Modalidad: Semi-Presencial	**Estratégica Didáctica:** Trabajo Grupal.
Metodología de Desarrollo: acordar	**Metodología de Corrección:** acordar docente
Carácter de Trabajo: Obligatorio – con Nota	**Fecha Entrega:** A confirmar por el Docente.

TEMPLATES **– FUNCIONES LIBRES – STL – PROGRAMACION GENERICA**

Marco Teórico:

1. Que entiende por Plantillas de Funciones
2. Que son las Plantillas de Funciones, dar un ejemplo
3. Que son las Plantillas de clases, dar un Ejemplo.
4. Que función Cumple la Especialización de Plantillas

Marco Practico

1. Este ejercicio servirá para practicar la declaración de plantillas. A continuación se da una lista de descripciones de funciones y clases de plantillas, se pide escribir una declaración apropiada para las mismas y comprobar que éstas compilan.

    a. 1. Declarar una función que toma dos parámetros de plantilla distintos de los cuales uno es el tipo de retorno y el otro es argumento.
    b. 2. Declarar una clase que toma un parámetro de plantilla, el cual es una variable miembro (atributo) de la misma.
    c. 3. Declarar una clase que toma dos parámetros de plantilla, uno como argumento al constructor y otro como tipo de retorno de una función miembro (método) sin argumentos.
2. El objetivo de este ejercicio es declarar una función de plantilla sencilla y ver algunas de sus posibilidades.
   Se pide escribir una función menor que tome dos argumentos genéricos y use el operador < para devolver el menor de ellos como valor de retorno. La función debe ser capaz de dar este tipo de resultados:

menor(2, 3) == 2
menor(6.0, 4.0) == 4.0
A continuación:

a. Comprobar su funcionamiento para parejas de argumentos numéricos del mismo tipo (int,double, float).
b. 2. Comprobar que pasa si los argumentos son de distinto tipo (la respuesta dependerá de si se usó un parámetro de plantilla o dos para la declaración de la función, probar ambas posibilidades).

Lic. Oemig José Luis.

Marco Teórico:

1.Que entiende por Plantillas de Funciones

Las plantillas de funciones en C++ son una característica del lenguaje que permite la programación genérica. Esto significa que puedo construir funciones y clases que pueden trabajar con diferentes tipos de datos, eliminando la necesidad de sobrecarga. Esto hace que el código sea más reutilizable y flexible.

Las plantillas pueden ser utilizadas en clases, estructuras y funciones. Se pueden describir como planos o fórmulas para crear una clase, estructura o función genérica. También son apropiadas para definir contenedores, es decir, estructuras que sirven para almacenar colecciones de objetos (como listas, vectores, grafos, etc.) y para definir algoritmos genéricos que se aplican a una familia de clases.

Los templates en C++ son una especie de “plantilla” que guía al compilador para generar código. No se compilan por sí mismas, sino que el compilador genera código a partir de ellas cuando se utilizan en el código fuente (por ejemplo, cuando se crea un objeto de una clase plantilla). Este código generado es específico para los tipos de datos con los que se utiliza la plantilla.

Las ventajas de las plantillas son la generalización y la simplicidad. La generalización se logra al permitir pasar cualquier tipo de parámetros y la simplicidad se logra al codificar una sola función o clase sin importar el tipo pasado como parámetro. Sin embargo, es importante tener en cuenta que el uso de plantillas puede aumentar el tiempo de compilación.

2. Que son las Plantillas de Funciones, dar un ejemplo

Las plantillas de funciones son una característica poderosa en C++ que permite operar con tipos genéricos. Esto les otorga una versatilidad inmensa, ya que puedo crear una función que se comporta de la misma manera con

diferentes tipos de datos. En C++, las plantillas de función se declaran utilizando la palabra clave template, y las variables de tipo genérico se declaran usando los operadores de menor y mayor (en ese orden), que encierran a la variable de tipo template dentro de ellos. Esta flexibilidad hace que mi código sea más reutilizable y fácil de mantener.

Las templates de funciones usan parametros de plantilla que son un tipo especial de parametro que se puede usar para pasar un tipo como argumento. Permiten el paso de valores a una función, de la misma manera que las funciones “normales”. La sintaxis para declarar plantillas de función es:

```
//Para funciones
template<typename identifier> declaracion_funcion;//Uso la palabra clave
//typename

//Para clases
template<typename identifier> declaracion_clases;//Uso la palabra clave
//class
```

```
#include <iostream>

using namespace std;

template <typename T>
//Prototipo de función del tipo void con un tipo de dato genérico
void mostrarAbs (T numero);

int main()
{
	//Declaración de valor que va a ser pasado a la función
	int num = 10;
	//Invocación de función
	mostrarAbs(num);

	//Declaro número de otro tipo para mostrar la capacidad de crear
	//funciones genéricas de los templates
	double num1 = 13.007594;

	//Invocación de función
	mostrarAbs(num1);
	return 0;
}

//Declaro una plantilla de función
template <typename T>
void mostrarAbs (T numero)
{
	if (numero < 0)
```

```
{
        numero = numero * -1;
}
        cout << "El valor absoluto del número es " << numero << endl;
}
```

3. Que son las Plantillas de clases, dar un Ejemplo.

Las plantillas de clases son al igual que las plantillas de función son una nueva característica de las versiones de C++más modernas, que me permite crear clases que contengan tipos de datos genéricos(no todos los valores miembre de una clase “genérica” deben serlo).

```
include <iostream>

using namespace std;

//EjemploTemplate es una plantilla de clase que acepta un tipo de dato T.
template<typename T>
class EjemploTemplate
{
   private:
   //Variable miembro privada de tipo "genérico"
      T valor;
   public:
   //Constructor de la clase EjemploTemplate
   //EjemploTemplate(T x), con esto digo, la variable x es del "tipo genérico T "
   {
      valor = x;
   }
   //Función miembro que retorna cuando sea invocada el dato que contenga la
//variable valor
   T getValor()
   {
      return valor;
   }
};



int main(int argc,char **argv)
{
   cout << "Valores impresos por pantalla que muestran el polimorfismo de los
templates " << endl;

   //Creo un objeto de tipo int, que es una instancia de EjemploTemplate
```

```
    EjemploTemplate<int>objEntero(10);//Le paso al constructor el valor
//entero10

    //Saco por pantalla el valor pasado al constructor, que recibe la función
//miembro del tipo getter
    cout << objEntero.getValor() << endl;

    //Aca se muestran el polimorfismo de los templates(estático), ya que con la
//misma función miembro, puedo enviar un tipo mensaje diferente.
    EjemploTemplate<double> objFloat(100.84);
    cout << objFloat.getValor() << endl;

    //Ahora para terminar de probar el valor genérico de los templates, paso un
        //string
    EjemploTemplate<string> objString("Mi perro me llama");
    cout << objString.getValor() << endl;


    return 0;
}
```

4. Que función Cumple la Especialización de Plantillas

Una especialización de plantillas es una versión personalizada de una plantilla que se comporta de manera única cuando interactúa con un tipo de datos específico. Esta adaptación permite un rendimiento más eficiente o satisface las necesidades particulares del tipo de datos en cuestión. En resumen, proporciona una variación en el comportamiento de la plantilla para un tipo de datos determinado, diferenciándolo del manejo genérico de otros tipos de datos.

```
//Plantilla "común"
template <typename T>
void imprimir(T valor)
{
    std::cout << valor << std::endl;
}
//Especialización de plantilla que adopta un nuevo comportamiento cuando
//encuentra el tipo de dato al que deseo agregarle una variación
template <>
void imprimir(int valor)
{
    cout << "El valor es un número entero: " << valor << endl;
}
```

```
int main(int argc,char **argv)
{
   //Imprime "El valor es un número entero: 123"
   imprimir(123);
   //Imprime "Hola, mundo"
   imprimir("Hola, mundo");
   return 0;
}
```

Marco Practico

1 a) Declarar una función que toma dos parámetros de plantilla distintos de los cuales uno es el tipo de retorno y el otro es argumento.

```
/*
Declarar una función que toma dos parámetros de plantilla distintos
de los  cuales uno es el tipo de retorno y el otro es argumento.
*/

#include <iostream>

using namespace std;

template <typename T>
T imprimirPorPantalla(T retorno)
{
   cout << retorno << endl;
   //Retorno de la función
  return retorno;
}

int main(int argc,char **argv)
{
   int a = 10;
   string b = "Hola mundo!";
   float c =35.78;

   imprimirPorPantalla(a);
   imprimirPorPantalla(b);
   imprimirPorPantalla(c);


   return 0;
```

}

1 b) Declarar una clase que toma un parámetro de plantilla, el cual es una variable miembro (atributo) de la misma.

```
#include<iostream>
#include<vector>

using namespace std;

//Declaración del template
template<typename T, typename X>
class Persona
{
   private:
      string nombre;
      int edad;
      vector<T> mascotas;
      X licenciaConducir;
   public:
   //Constructor
      Persona(string _nombre,int _edad, vector<T> _mascotas,X
_licenciaConducir)
      {
         nombre = _nombre;
         edad = _edad;
         mascotas = _mascotas;
         licenciaConducir = _licenciaConducir;
      }
   //Destructor
      ~Persona(){};
   //Funciones miembro, en este caso un getter
   X getLicenciaConducir()
   {
      return licenciaConducir;
   }
   //Demas funciones miembro
   void agregarMascota(T nuevamascota)
   {
      mascotas.push_back(nuevamascota);
   }
   void presentarse()
   {
      cout << "Me llamo " << nombre << ", mi edad es " << edad << " mis
mascotas se llaman: ";
      for (const auto& mascota : mascotas) {
         cout << mascota << " ";
```

```
        }
        cout << "y por ultimo tengo mi licencia de conducir numero " <<
licenciaConducir << endl;
    }

};



int main(int argc,char **argv)
{
    //Creo un vector para poder pasarle datos al constructor
    vector<string> mascotas = {"Boby","Garfield","MIMI","Chanta"};

    //Tengo "castear" los datos genericos y volverlos concretos antes de seguirle
//pasando datos al constructor, es decir llenar con elementos "concretos" el
          //"molde template"
    Persona<string,int> persona1 ("Cacho",38,mascotas,456978);

    persona1.agregarMascota("pepito");
    persona1.presentarse();


    return 0;
}
```

1.c) Declarar una clase que toma dos parámetros de plantilla, uno como argumento al constructor y otro como tipo de retorno de una función miembro (método) sin argumentos.

```
/*
Declarar una clase que toma dos parámetros de plantilla,
uno como argumento al constructor y
otro como tipo de retorno de una función miembro (método) sin argumentos.
*/


#include <iostream>

using namespace std;

//Creo template para poder trabajar con datos genéricos
template<typename T, typename U>
class ClaseEjemplo
{
```

```
    private:
        T parametroConstructor;

    public:
        //Constructor que toma argumento del tipo T
        ClaseEjemplo(T _parametroConstructor)
        {
            parametroConstructor = _parametroConstructor;
        }
    //Función miembro que no toma argumentes y devuelve valor del tipo U
    U funcionMiembroConRetorno()
    {
        return U();
    }
    //Función miembro que muestra por pantalla el argumento de la función con
          //retorno
    void ImprimirPorPantalla()
    {   //Guardo en una variable del tipo U, el valor que devuelva la función
//funcionMiembroConRetorno
        U valor = funcionMiembroConRetorno();
        cout << "El valor de la devolución de función es " << valor << endl;
    }


};


int main(int argc,char **argv)
{
    ClaseEjemplo<string,int> ejemplo1("Hola mundo");
    //Devuelve 0 por pantalla
    ejemplo1.ImprimirPorPantalla();

    return 0;
}
```

2.ª

```
#include<iostream>

using namespace std;

template<typename T, typename X>
T devolverMenorValor(T valorA,X valorB)
{
    if (valorA > valorB)
    {
```

```
        return valorB;
    }
    else
    {
        return valorA;
    }
}

int main(int argc,char **argv)
{
    int a = 10;
    int b =20;

    cout << "El menor valor es: " << devolverMenorValor(a,b) << endl;

    float c = 30.856;
    float d = 789.654;

    cout << "El menor valor es: " << devolverMenorValor(c,d) << endl;

    double f = 105789.45;
    double g =78945.987;

    cout << "El menor valor es: " << devolverMenorValor(f,g) << endl;

    string h = "Hola!";
    string i = "Hola mundo!";

    //Puede retornar valor la función, aunque lo que devuelve no tiene sentido
    cout << "El menor valor es: " << devolverMenorValor(h,i) << endl;

    //Puedo usar la misma función sin declarar variables pasandolas como
        //argumentos directamente
    //Aca paso dos datos numéricos de distinto tipo
    cout << "Paso de valor sin declarar antes variables " << endl;
    cout << "El menor valor es: " << devolverMenorValor(50,489.56) << endl;


    return 0;
}
```

2b) Aca exploro la posibilidad de hacer un template con un solo parámetro y lo que compruebo es que si no son del mismo tipo, el compilador no realiza su trabajo y marca error en la salida de la compilación.

```
 /*
```

2. El objetivo de este ejercicio es declarar una función de plantilla sencilla y ver algunas de sus posibilidades.
Se pide escribir una función menor que tome dos argumentos genéricos y use el operador < para devolver el menor de ellos como valor de retorno.
La función debe ser capaz de dar este tipo de resultados:
menor(2, 3) == 2
menor(6.0, 4.0) == 4.0
A continuación:
a. Comprobar su funcionamiento para parejas de argumentos numéricos del mismo tipo (int,double, float).
b. 2. Comprobar que pasa si los argumentos son de distinto tipo (la respuesta dependerá de si se usó un parámetro de plantilla o dos para la declaración de la función, probar ambas posibilidades).
*/

```
#include<iostream>

using namespace std;

template<typename T>
T devolverMenorValor(T valorA,T valorB)
{
   if (valorA > valorB)
   {
      return valorB;
   }
   else
   {
      return valorA;
   }
}

int main(int argc,char **argv)
{
   int a = 10;
   int b =20;

   cout << "El menor valor es: " << devolverMenorValor(a,b) << endl;

   float c = 30.856;
   float d = 789.654;

   cout << "El menor valor es: " << devolverMenorValor(c,d) << endl;

   double f = 105789.45;
   double g =78945.987;
```

```
    cout << "El menor valor es: " << devolverMenorValor(f,g) << endl;

    string h = "Hola!";
    string i = "Hola mundo!";

    //Puede retornar valor la función, aunque lo que devuelve no tiene sentido
    cout << "El menor valor es: " << devolverMenorValor(h,i) << endl;

    //Puedo usar la misma función sin declarar variables pasandolas como
        //argumentos directamente
    //Aca paso dos datos numéricos de distinto tipo
    cout << "Paso de valor sin declarar antes variables " << endl;
    //aca da error de compilación, por que el compilador no acepta dos tipos de
        //datos distintos
    cout << "El menor valor es: " << devolverMenorValor(50,45.78) << endl;


    return 0;
}
```