

Algoritmos y Estructuras de Datos II

Trabajo Práctico N° 4.1	Unidad 4.1
Modalidad: Semi -Presencial	Estratégica Didáctica: Trabajo individual
Metodología de Desarrollo: Det. docente	Metodología de Corrección: Vía Classroom.
Carácter de Trabajo: Obligatorio – Con Nota	Fecha Entrega: A confirmar por el Docente.

MARCO TEÓRICO

TEMPLATES – FUNCIONES LIBRES – STL – PROGRAMACIÓN GENÉRICA

Responder el siguiente cuestionario en función de la bibliografía Obligatoria.

1. Qué entiende por Plantillas de Funciones

Las plantillas de funciones son funciones especiales que pueden operar con tipos genéricos. Permiten definir una única función que puede adaptarse a más de un tipo o clase sin necesidad de reescribir el código para cada tipo específico.

En lugar de definir una función específica para un tipo de dato, se utiliza una plantilla que acepta un tipo como parámetro, el cual se instala en tiempo de compilación con el tipo de dato específico que se usará.

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

Las plantillas de funciones son un mecanismo que permite la creación de funciones genéricas parametrizadas por tipo. Su principal ventaja es la reutilización de código, evitando la redundancia al escribir una función genérica una sola vez para usarla con diferentes tipos de datos

Ejemplo: Una función para obtener el máximo de dos valores.

```
template <class T>
T GetMax (T a, T b) {
    return (a > b ? a : b);
}
```

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

Las plantillas de clases permiten definir clases que pueden tener miembros que usan parámetros de plantilla como tipos. Al igual que con las funciones, permiten crear estructuras de datos o algoritmos que funcionan con una variedad de tipos de datos sin duplicar código.

Ejemplo: Una clase para almacenar un par de valores de cualquier tipo.

```
template <class T>
class mypair {
```

```
T values[2];  
public:  
    mypair(T first, T second) {  
        values[0] = first;  
        values[1] = second;  
    }
```

4. Que función Cumple la Especialización de Plantillas

La especialización de plantillas permite definir una implementación diferente y específica para una plantilla cuando se le pasa un tipo concreto como parámetro. Se utiliza cuando se necesita un comportamiento distinto para un tipo de dato particular, manteniendo la implementación genérica para el resto de los tipos. Por ejemplo, una clase contenedora podría tener una implementación especial optimizada o con funcionalidades diferentes si el tipo almacenado es char en lugar de int.

Marco Práctico

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.
 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.

```
4 > a.cpp > main()  
1  #include <iostream>  
2  using namespace std;  
3  
4  template <typename R, typename T>  
5  R calcularCosto(T precioUnitario, int cantidad) {  
6      return precioUnitario * cantidad;  
7  }  
8  
9  int main() {  
10     cout << "Costo total: $" << calcularCosto<double>(2500.99, 2) << endl;  
11 }
```

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

```
4 > 4 b.cpp > main()
1   #include <iostream>
2   using namespace std;
3
4   template <typename T>
5   class Producto {
6       T modelo;
7       double precio;
8   public:
9       Producto(T m, double p) : modelo(m), precio(p) {}
10      void info() { cout << modelo << " - $" << precio << endl; }
11  };
12
13 int main() {
14     Producto<string> laptop("XPS-13", 120000);
15     laptop.info();
16 }
```

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.

```
4 > 4 c.cpp > main()
1   #include <iostream>
2   using namespace std;
3
4   template <typename Arg, typename Ret>
5   class Especificacion {
6       Arg dato;
7   public:
8       Especificacion(Arg d) : dato(d) {}
9       Ret get() { return dato; }
10  };
11
12 int main() {
13     Especificacion<string, string> ram("16GB DDR5");
14     cout << "RAM: " << ram.get() << endl;
15 }
```

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).

```
4 > 2a.cpp > main()
1 #include <iostream>
2 using namespace std;
3
4 template <typename T>
5 T menor(T a, T b) { return a < b ? a : b; }
6
7 int main() {
8     cout << menor(8, 16) << " GB\n";      // 8
9     cout << menor(3.4, 2.8) << " GHz\n"; // 2.8
10    cout << menor(512.0f, 256.0f) << " GB SSD\n"; // 256
11 }
```

- b) 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).

```
4 > ← 2b.cpp > ↴ main()
1   #include <iostream>
2   using namespace std;
3
4   template <typename T, typename U>
5   T menor(T a, U b) { return a < T(b) ? a : T(b); }
6
7   int main() {
8       cout << menor(8, 12.5) << " GB\n";      // 8
9       cout << menor(3.7, 4.2f) << " GHz\n";    // 3
10 }
```

Lic. Oemig José Luis.