Grado en Ingeniería Información

PROGRAMACIÓN II - Sesión 7

Tema 4.

Plantillas -Templates

Curso 2022-2023

Marta N. Gómez

UNIVERSIDAD NEBRIJA

UNIVERSIDAD NEBRIJA

T4. Plantillas o Templates

2.1. Funciones templatizadas

2.2. Clases templatizadas

ÍNDICE

UNIVERSIDAD NEBRIJA

T4. Plantillas o Templates

2.1. Funciones templatizadas

2.2. Clases templatizadas

ÍNDICE

La **sobrecarga de funciones** permite que varias funciones con el mismo nombre y distintos parámetros realizan diferentes tareas.

```
#include <iostream>

void escribirDato(int a){
  cout << "El numero es: " << a << "\n";
}

void escribirDato(std::string a){
  cout << "La cadena de caracteres es: " << a << "\n";
}

int main() {
  escribirDato(222);
  escribirDato("Hola mundo");

  cout << "\n\n\t";
  return 0;
}
```

Las plantillas son una característica del lenguaje C++ que podemos utilizar para generalizar nuestro código, para que las **funciones** y las **clases** trabajen con **tipos de datos genéricos**.

Plantillas- Funciones templatizadas

```
#ifndef TEMPLATES_H
#define TEMPLATES_H

#include <iostream>

using namespace std;

template<typename T>
void escribirDato(T const &a){
    cout << "\n\n\tEl dato es: " << a << "\n\n";
}

#endif // TEMPLATES_H
```

```
#include <iostream>
#include <Templates.h>

using namespace std;

int main() {
    escribirDato<int>(222);
    escribirDato<string>("Hola mundo");

    cout << "\n\n\t";
    return 0;
}
```

La declaración y definición de las **funciones templatizadas SIEMPRE** se hace en **archivos .h.**

template<typename T>

Indica que se va a declarar una función **templatizada** con **un tipo T** sin específicar **cuál será**, por **eso es una plantilla o *template.***

void escribirDato(T const &a)

Función que no devuelve nada (***void***) y recibe un parámetro de ***tipo T*** (es, es una **plantilla**). El paso se hace por ***referencia constante*** porque no se conoce qué tipo será ***T.*** Por tanto, nunca se hará una copia de su valor, sea un **tipo simple** o un **tipo complejo**.

cout << "\n\n\tEl dato es: " << a << "\n\n";

Se muestra por pantalla el texto y la variable ***a***.

escribirDato<int>(222);

Desde el ***main*** se llama a la **función templatizada** pasando un parámetro ***int.***

El compilador, automáticamente, crea internamente la función:

```
// Función Interna
void escribirDato(int const &a){
    cout << "\n\n\tEl dato es: " << a << "\n\n";
}
```

escribirDato<string>("Hola mundo");

Desde el ***main*** se llama a la **función templatizada** pasando un parámetro ***string.***

El compilador, automáticamente, crea internamente la función:

```
// Función Interna
void escribirDato(string const &a){
    cout << "\n\n\tEl dato es: " << a << "\n\n";
}
```

¿Qué pasa con otro **tipo de datos**?

```
#include <iostream>
#include <Templates.h>

using namespace std;

struct TipoDato{
  string texto;
  int valor;
};

int main() {
    TipoDato dato{"Ejercicio tamplate", 10};

    escribirDato<TipoDato>(dato);

    cout << "\n\n\t";
    return 0;
}
```

// ERROR

// Función Interna

```
void escribirDato(TipoDato const &a){
    cout << "\n\n\tEl dato es: " << a << "\n\n";
}
```

// ERROR

Hay **dos opciones**:

1. **Sobrecargar** el operador <<
2. **Particularizar** la plantilla.

Sobrecargar el operador <<

```
struct TipoDato{
  string texto;
  int valor;
};
```

```
ostream & operator <<(ostream & os, TipoDato const &p){
      os << "\n\t\t\ttexto.- " << p.texto
         << "\n\t\t\tvalor.- " << p.valor << "\n";

      return os;
}
```

```
int main() {
    TipoDato dato{"Ejercicio tamplate", 10};

    escribirDato<TipoDato>(dato);

    cout << "\n\n\t";
    return 0;
}
```

// Función Interna

```
void escribirDato(TipoDato const &a){
    cout << "\n\n\tEl dato es: " << a << "\n\n";
}
```

Particularizar la **plantilla/template** programando la función para un determinado **tipo de datos**.

```
#ifndef TEMPLATES_H
#define TEMPLATES_H

#include <iostream>

using namespace std;

struct TipoDato{
  string texto;
  int valor;
};

template<typename T>
void escribirDato(T const & a){
    cout << "\n\n\tEl dato es: " << a << "\n\n";
}

template<>
void escribirDato(TipoDato const &p){
    cout << "\n\n\tEl dato es: ";
    cout << "\n\t\t\ttexto.- " << p.texto;
    cout << "\n\t\t\tvalor.- " << p.valor << "\n";
}

#endif // TEMPLATES_H
```

Plantillas- Funciones templatizadas

```
#include <iostream>
#include <Templates.h>

using namespace std;

int main() {
    TipoDato dato{"Ejercicio tamplate", 10};

    escribirDato<TipoDato>(dato);

    cout << "\n\n\t";
    return 0;
}
```

Las **plantillas** pueden incluir **más** de un **tipo de datos**:

```
#ifndef TEMPLATES_H
#define TEMPLATES_H

#include <iostream>

using namespace std;

struct TipoDato{
  string texto;
  int valor;
};

template<typename T>
void escribirDato(T const & a){
    cout << "\n\n\tEl dato es: " << a << "\n\n";
}

template<typename T1, typename T2>
void mostrarDatos(T1 const & a, T2 const &b){
    cout << "\n\n\tEl primer dato es: ";
    cout << a;
    cout << "\n\n\tEl segundo dato es: " << b;
}

template<>
void escribirDato(TipoDato const &p){
    cout << "\n\n\tEl dato es: ";
    cout << "\n\t\t\ttexto.- " << p.texto;
    cout << "\n\t\t\tvalor.- " << p.valor << "\n";
}

#endif // TEMPLATES_H
```

```
#include <iostream>
#include <Templates.h>

using namespace std;

int main() {
    TipoDato dato{"Ejercicio tamplate", 10};

    mostrarDatos(dato, "Fin del ejemplo.");

    mostrarDatos("Multiplica 15*3 ", 15*3);

    cout << "\n\n\t";
    return 0;
}
```

UNIVERSIDAD NEBRIJA

T4. Plantillas o Templates

2.1. Funciones templatizadas

2.2. Clases templatizadas

ÍNDICE

Las clases también pueden incluir **tipos de datos** definidos a través de **plantillas/templates** en sus **miembros** (**atributos/métodos/funciones**).

```
#ifndef CLASETEMPLATE_H
#define CLASETEMPLATE_H

template <typename T1, typename T2>
class ClaseT {
    private:
        T1 dat1;
        T2 dat2;
    public:
      ClaseT(T1 p1, T2 p2): dat1(p1), dat2(p2){}

      T1 getDat1() const;
      T2 getDat2() const;
};

template <typename T1, typename T2>
T1 ClaseT<T1,T2>::getDat1() const{
  return dat1;
}

template <typename T1, typename T2>
T2 ClaseT<T1,T2>::getDat2() const{
  return dat2;
}
#endif // CLASETEMPLATE_H
```

Plantillas- Clases templatizadas

```
#include <iostream>
#include <ClaseTemplate.h>

using namespace std;

int main() {
    ClaseT<string, int> miClase("Ejercicio tamplate", 10);
    cout << "\n\tTexto.- " << miClase.getDat1() << " valor.- " << miClase.getDat2() << "\n";

    cout << "\n\n\t";
    return 0;
}
```

Realizar una calculadora que, a través de funciones templatizadas, permita realizar la suma, resta, multiplicación y división de diferentes tipos de datos: ***enteros***, ***decimales***, ***complejos***.

NOTA:

Las operaciones de números complejos vienen definidas por:

- Suma: $z1 + z2 = (a + c, b + d)$
- Diferencia: $z1 - z2 = (a - c, b - d)$
- Producto: $z1\ z2 = (ac - bd, ad + bc)$
- División: $\frac{Z1}{Z2} = \left(\frac{ac + bd}{c^2 + b^2}, \frac{bc - ad}{c^2 + b^2}\right)$

Plantillas- Clases templatizadas

- Crear una clase ***Cpares*** que tenga dos atributos, una clave y un valor. Los tipos de ambos atributos deben estar templatizados.
- Crear una clase templatizada ***Vpares*** que contenga un vector de ***Cpares***.
- Sobrecargar el operador >> para ***Cpares*** y ***Vpares***, de modo que se puedan mostrar por pantalla usando ***cout << var***
- Crear un programa ***main*** que realice lo siguiente:
  - Solicitar 5 datos para crear un objeto de **Vpares** donde sus elementos sean de tipo: string, int.
  - Solicitar 5 datos para crear un objeto de **Vpares** donde sus elementos sean de tipo: float, int.
  - Mostrar los correspondientes resultados por pantalla usando el operador <<.