Programación II

Tema 5

jtorresmat@nebrija.es

Índice

Introducción a las plantillas

- Plantillas (templates)
  - Plantilla (RAE): **4.** f. Tabla o plancha cortada con los mismos ángulos, figuras y tamaños que ha de tener la superficie de una pieza, y que puesta sobre ella, sirve en varios oficios de regla para cortarla y labrarla. (Ejemplo1, Ejemplo2, Ejemplo3, …)
  - En programación es una manera de escribir funciones y clases para que puedan ser utilizadas con cualquier tipo de datos.

Introducción a las plantillas

- Hasta ahora las funciones las definíamos para un tipo concreto de datos.
- Con la sobrecarga podíamos definir funciones con el mismo nombre que recibieran diferentes tipos de datos.
- Con las plantillas podemos hacer una función genérica. Una “familia” de funciones.

Introducción a las plantillas

- Ej función devuelve el menor de 2 valores:

```
int menor(int a, int b)
{
    if(a<b) return a;
    else return b;
}
```

- Pero si... ¿a y b son float? ¿si son objetos de una clase Fecha?
- Con plantillas podremos hacer una única definición que devuelva el menor (siempre y cuando sean elemento ordenables…)

Introducción a las plantillas

- Aplicando el mismo concepto visto en funciones podemos hacer lo mismo con la definición de las clases (teniendo clases genéricas)
- ¿Dónde hemos usado plantillas para clases? En los tipos contenedor:
  - https://es.cppreference.com/w/cpp/container
    - std::vector
    - std::array
    - std::set
    - std::list, etc

Introducción a las plantillas

- Ventajas:
  - Generalización: podemos usar cualquier tipo de dato
  - Simplicidad: sólo codificamos una vez
- Inconvenientes:
  - Hay que tomar algunas precauciones
  - El código tarda algo más en compilar

Índice

Plantillas de funciones

- Con sobrecarga podemos hacer por ejemplo:

```
void print(int a){
  std::cout << "Esto es un numero: " << a << "\n";
}

void print(std::string a){
  std::cout << "Esto es un string: " << a << "\n";
}

int main() {
  print(3); // Esto es un numero: 3
  print("hola"); // Esto es un string: hola
  return 0;
}
```

Plantillas de funciones

- Con plantillas podemos hacer:

```
#include <iostream>

template<typename T>
void print(T const & a){
  std::cout << a << "\n";
}


int main() {
  print<int>(3); //  3
  print<std::string>("hola"); //  hola
  return 0;
}
```

Plantillas de funciones

- Vamos por partes...

```
#include <iostream>

template<typename T>            //Indicamos que vamos a hacer un templete de tipo T
void print(T const & a){        //Declaramos la función con valor de ret. y el tipo T
                                //Como no sabemos qué va a ser T lo pasamos por ref y cte
                                //Así seguro que no lo podemos modificar
  std::cout << a << "\n";       //...Cuerpo de la función...
}


int main() {
  print<int>(3);                //Llamamos a la función templatizada->El compilador crea la
                                //función void print(int const &a)
  print<std::string>("hola"); //Lo mismo que antes pero void print(std::string const &a)
  return 0;}
```

Plantillas de funciones

- Vamos por partes… qué pasa si…

```
struct Persona{
    std::string nombre;
    unsigned int edad;
}
Persona Yo{“Pepe”,40};
print<Persona>(Yo); //¿?
```

- El compilador intenta `void print(persona const &a){std::cout << a << "\n";}`

  ¿Y?... ->No sabe mostrar por pantalla persona...

Plantillas de funciones

- Posible soluciones:
  - Sobrecargamos el operador << para Persona

```
std::ostream & operator <<(std::ostream & os, Persona const & p){
  os << "nombre: " << p.nombre << "\n";
  os << "edad: " << p.edad << "\n";
  return os;
}
```

  - Particularizamos el template

Plantillas de funciones

- Soluciones:
  - Particularizamos el template:
    - Para un caso particular no dejamos elegir al compilador. Codificamos explícitamente ese caso

```
template<typename T>
void print(T const & a){
  std::cout << a << "\n";
}

template<>
void print(Persona const & a){
  std::cout << "nombre: " << a.nombre << "\n";
  std::cout << "edad: " << a.edad << "\n";
}
```

Plantillas de funciones

- Cómo hemos visto el template se compila cada vez para el tipo de dato con el que se usa (se compila después de pasarle el tipo de parámetro)
- El compilador “sufre”, pero puede con ello :)
- De esto podemos deducir que:
  - Si la plantilla se va a utilizar únicamente en un *.cpp => puede ser codificada en ese cpp.
  - Si va a ser utilizada en varios *.cpp debe ir en una cabecera (*.h) para poderla “incluir” donde la necesitemos

Documentación en GitHub

- https://github.com/Nebrija-Programacion/Programacion-II
  - `Funciones templatizadas:`

(https://github.com/Nebrija-Programacion/Programacion-II/blob/master/temario/funcionestempl.md)

Índice

Ejemplos “funciones templatizadas”

- Ejercicio 5 github: “templatizar” las funciones suma, resta, multiplicación para que pueda recibir:
  - 2 enteros -> entero
  - 2 decimales -> decimal
  - 2 matrices -> matriz

  código

Ejemplos “funciones templatizadas”

- Ejercicio 6 github: realizar una clase persona que tenga nombre, edad y DNI. Hacer una función plantilla find que recorra un vector de punteros a persona y devuelva:
  - El puntero a persona cuyo DNI corresponda con el buscado
  - nullptr si no encontramos a nadie con ese DNI

  código

Índice

Plantillas de clase

- Como pasaba con las funciones, las clases también pueden "templatizarse" a través de sus miembros:
  - Atributos (variables encapsuladas)
  - Métodos (funciones encapsuladas)
- Sintaxis equivalente a la que usábamos en funciones

```
template <typename T>
Coordenada<T>::Coordenada(T x, T y){          //¿Qué estamos templatizando?
        this -> x = x;
        this -> y = y;
};
```

- Ejemplo

Plantillas de clase

- Diferencias plantilla funciones y clases:
  - Con plantilla de funciones podemos hacer llamadas sin especificar el tipo… código
  - Con plantilla de clases… código
    - Con plantillas de clases siempre hay que especificar el tipo
  - template <typename T> es equivalente a template <class T>

Plantillas de clase

- Declaración y definición deben ir en el mismo fichero:
  - Una plantilla no es una función o una clase. Una plantilla es un patrón que el compilador usa para hacer una familia de funciones o clases.
  - El compilador necesita tener la declaración y la definición en el “mismo sitio” para poder crear la nueva instancia.

Plantillas de clase

- Ejemplo:

```
template <typename K, typename V>
class Pair{
public:
    Pair(K v0, V v1);        //argumentos tipo K y tipo V
    K key() const;           //método público tipo K
    V value() const;         //método público tipo V
private:
    K _key;                  //variable privada Tipo K
    V _value;                //variable privada Tipo V

};
```

Plantillas de clase

- Ejemplo:

```
template <typename K, typename V>
Pair<K,V>::Pair(K v0, V v1){
        _key = v0;
        _value = v1;
}

template <typename K, typename V>
V Pair<K,V>::value() const{
     return _value;
}

template <typename K, typename V>
K Pair<K,V>::key() const{
     return _key;
}
```

Plantillas de clase

- Ejemplo:

```
int main() {
	Pair<std::string, int> myPair{"tel", 34434423};
	std::cout << myPair.key() << ":" << myPair.value() << "\n";

	return 0;
}
```

Documentación en GitHub

- https://github.com/Nebrija-Programacion/Programacion-II
  - `Funciones templatizadas:`

    (https://github.com/Nebrija-Programacion/Programacion-II/blob/master/temario/clasescionestempl.md)
  - `Clases templatizadas:`

    (https://github.com/Nebrija-Programacion/Programacion-II/blob/master/temario/clasescionestempl.md)

Ejemplos “para casa”

- Implementar una clase persona, con los atributos edad y nombre. En la función principal definir un tipo de datos std::set que albergue personas de manera que no puedan repetirse personas con la misma edad.
- error
- ok