TEMPLATES

Los **templates** o plantillas de función permiten definir una única vez funciones que ejecutan las **mismas operaciones** sobre **distintos tipos de datos**. Todas las plantillas comienzan con la palabra reservada template seguida por una lista de parámetros formales, precedido cada uno por la palabra class, encerrados entre < >. A continuación se coloca la definición de la función. Si bien los templates no forman parte del lenguaje C++ original, son soportados por la mayoría de los compiladores modernos.

Funciones genéricas

Definen un **conjunto de operaciones** que se aplicarán a **varios tipos de datos**. El tipo específico sobre el que se operará se pasa como parámetro.

```
template <class|typename T [, ...]> <tipo_retorno> <identificador> (<parámetros>)
{
    // cuerpo de la función
};
```

T es un *sustituto* del tipo utilizado (*placeholder*). El compilador lo reemplazará por el tipo de dato adecuado cuando se cree una versión específica (*especialización*) de la función. La acción de generar una función se conoce como su *instanciación*. Es tradicional usar la palabra clave **class** en la declaración del template (framework) aunque también se puede utilizar la palabra clave **typename** (menos usual).

Puede existir más de un tipo parametrizado, cada uno precedido por class y separados por comas. La lista de parámetros también puede incluir tipos estándares (int, double, string, etc.).

Template de funciones – Ejemplo

```
#include <iostream>
#include "complejos.h"

using namespace std;

// Intercambia los valores de los parámetros
template <class Tsw>
        void swapargs(Tsw &a, Tsw &b)
{
   Tsw temp;
   temp = a;
   a = b;
   b = temp;
};

// Retorna el máximo de los parámetros
template <class T> T maximo(T x, T y)
{
  return (x > y) ? x : y;
};
```

```
int main()
{
   int i = 10,  j = 20;
   double x = 30.1,  y = 23.3;
   char a = 'x',  b = 'z';
   complejo  c1(2, 3),  c2(4, 5);

   cout << "Original i, j: " << i << ' ' << j << '\n';
   cout << "Original x, y: " << x << ' ' << y << '\n';
   cout << "Original a, b: " << a << ' ' << b << '\n';
   cout << "Original c1, c2: " << c1 << ' ' << c2 << '\n';
   swapargs(i, j);          // swap integers
   swapargs(x, y);          // swap floats
   swapargs(a, b);          // swap chars
   swapargs(c1, c2);        // swap complejos
   cout << "\nSwapped i, j: " << i << ' ' << j << '\n';
   cout << "Swapped x, y: " << x << ' ' << y << '\n';
   cout << "Swapped a, b: " << a << ' ' << b << '\n';
   cout << "Swapped c1, c2: " << c1 << ' ' << c2 << '\n';

   cout<<"\n\nMaximo entre "<<i<<" y "<<j<<" = "<<maximo(i, j);
   cout<<"\n\nMaximo entre "<<x<<" y "<<y<<" = "<<maximo(x, y);
   cout<<"\n\nMaximo entre "<<a<<" y "<<b<<" = "<<maximo(a, b);

   return 0;
}
```

Ej. 01: Swap max

Template de funciones – Ejemplo múltiples tipos

```
#include <iostream>
#include ”complejos.h”

using namespace std;

template <class T1, class T2> void imprime(T1 &obj1, T2 &obj2, string msg)
{
   cout << msg <<  obj1 << ' ' << obj2 << "\n\n";
};

int main()
{
   string msg = "Mensaje original";
   double dou = 30.17428;
   char ch = 'x';
   complejo cpx(3.14, 6.28);

   imprime(msg, dou, "string + double: ");
   imprime(ch, cpx, "caracter + complejo: ");
   imprime(cpx, msg, "complejo + string: ");
   imprime(dou, ch, "double + char: ");

   return 0;
}
```

Ej. 02: Doble template

Sobrecarga de un template

Como sucede con cualquier función, se puede sobrecargar el template para que se utilice la versión de función que concuerde con la cantidad de parámetros pasados.

```
template <class T> <tipo_retorno> <identificador> (T obj1)
{
    // cuerpo de la función para 1 sólo parámetro
};

template <class T1, class T2> <tipo_retorno> <identificador> (T1 obj1, T2 obj2)
{
    // cuerpo de la función para 2 parámetros
}
```

Sobrecarga explícita

Se puede adaptar una función genérica a un tipo particular de dato. El proceso se conoce como *especialización explícita*. La versión específica oculta la versión genérica. Por ejemplo:

```
template <class T>  void swapargs(T &obj1, T &obj2)      // versión genérica

void swapargs(int &obj1, int &obj2)                      // especialización para int
```

Clases genéricas

Se definen todos los métodos utilizados por la clase sin especificar los tipos de dato sobre los que operan, los que serán asignados al ser creados los objetos.

```
template <class T [, ...]> class  <nombre_clase>
{
    // atributos y métodos miembros de la clase
};
```

Como en el caso de las funciones, se pueden utilizar dos o más tipos genéricos en el template, cada uno separado por comas. Además se pueden utilizar argumentos de tipos no genéricos y asignarles valores por defecto. Por ejemplo:

```
template <class T, int valor = 10> class <nombre_clase>
```

La clase se adapta al tipo genérico T y se puede utilizar otro parámetro de un tipo base (int) y asignarle un valor por defecto (10). El argumento *valor* debe ser una constante, nunca una variable, su valor debe conocerse durante la compilación (**las plantillas no son definiciones**).

Clases genéricas – valores por defecto

Se puede establecer que el tipo T se adapte a un tipo por defecto, de la forma:

template <class T = int**> class** <nombre_clase>

En este caso si no se especifica T por defecto será de tipo int.

En la forma más versátil: el tipo genérico y el básico pueden adoptar valores por defecto:

template <class T = **int**, **int** *valor* = 10**> class** <nombre_clase>

Posibilita crear objetos de la clase <nombre_clase> de tres maneras:

- ✓ especificando explícitamente el tipo genérico y el valor entero, <double, 20>obj,
- ✓ especificando el tipo genérico y adoptando el segundo valor por defecto, <double>obj,
- ✓ adoptando el tipo int y el valor del segundo parámetro ambos por defecto, <>obj.

Template de clases – Ejemplo

```
#include <iostream>
#include "persona.h"

using namespace std;

template <class, int> class pila;
template <class ST, int cnt> ostream&
  operator <<
  (ostream&, pila<ST, cnt>&);

template <class ST=char, int cnt=10>
  class pila
{
   private:
      ST *st;
      int tos;
      int tam;
   public:
      pila();
      ~pila();
      void push(ST);
      ST pop(); friend ostream&
        operator<< <> (ostream & co,
                       pila<ST, cnt>&);
      ST operator[] (int);
};
```

```
t<> ST pila<>::operator[](int indice)
   if(indice && indice<tam)
         return st[indice];
   else
      // mostrar error!!

 t<> ostream& operator<< ()
   for(int i=0; i<obj.tos; i++)
      cout << obj.st[i] << endl;
    return co;

t<> pila<>::pila()
   tos = 0; tam = cnt;
   st = new ST[tam];

t<> pila<>::~pila()
   delete [] st;

t<> void pila<>::push(ST obj)
   // verificar llenado
   st[tos] = obj;  tos++;

t<> ST pila<>::pop()
   // verificar no vacía
   tos--;
   return st[tos];
```

```
int main()
{
   char car;  pila<> st1; // char, 10
   st1.push('a');    st1.push('b');
   cout << st1;     car = st1.pop();

   pila<double> ds1;     // tamaño = 10
   ds1.push(1.1);    ds1.push(3.3);
   ds1.push(5.5);    cout << ds1;

   string texto;    pila<string, 5> pstr;
   pstr.push("hola");
   pstr.push("mundo");
   pstr.push("y chau demasiado largo");
   cout << pstr;
   texto = pstr[6];   // excede indice

   persona ped,  pab("nn", 1, fecha());
   pila<persona, 2> ppers;
   ppers.push(ped);
   ppers.push(pab);
   cout << ppers;
   ppers.push(ped);       // pila llena!!!

   return 0;
}
```

Observaciones

1) En la declaración de la función **friend** se han incluído los símbolos **<>:**

```
friend ostream& operator<< <> (ostream& co, pila<ST, cnt>&);
```

Uno de los parámetros de la función es un objeto de la clase genérica. Si se eliminan:

warning: *friend declaration 'std::ostream& operator<<(std::ostream&, pila<ST, cnt>&)' declares a non-template function [-Wnon-template-friend]*
note: (if this is not what you intended, make sure the ***function template has already been declared*** *and* ***add <> after the function name here****)* |
error: *undefined reference to `operator<<(std::ostream&, pila<char, 10>&)'*

2) Debe realizarse una **declaración anticipada** de la función **friend**, y de la clase genérica:

```
t <> class pila;        t <> ostream& operator<< (ostream&, pila<ST, cnt>&);
```

Si no se incluyen estas declaraciones el compilador produce el mensaje de error:

template-id 'operator<< <>' for 'std::ostream& operator<<(std::ostream&, pila<>&)' does not match any template declaration|

3) La **definición** de la función friend **no incluye** los símbolos **<>**:

```
template <class ST, int t> ostream& operator<< (ostream& co, pila<ST, t>& obj) { ; }
```

Separar declaración de implementación

Modularizar: separar el archivo principal (main.cpp) de la declaración (clase_gen.h) y definición (clase_gen.cpp). Cuando se declara un objeto de la clase genérica para un tipo específico el compilador lanza un error, cuando se instancia un template se crea una nueva clase con el argumento que se le pasa a la plantilla (**template no es declaración**).

1) Incluir en el archivo de cabecera (clase_gen.h) la implementación inline de todos los métodos utilizados. En este caso no se necesitan las declaraciones anticipadas cuando se trabaja con funciones friend ni la utilización de los símbolos <> . También se puede incluir la implementación en el archivo de cabecera.

   Ej. 05 a: Stack_h_inline / b:_h_full

2) Escribir la declaración del template en un archivo header y la implementación en otro archivo con cualquier extensión (por ejemplo clase_gen.cxx). Al final del archivo de cabecera incluirlo mediante la directiva #include “clase_gen.cxx”. Se tiene el mismo caso que el mostrado en el Ejemplo completo. Con Code::Blocks NO DEBE INCLUIRSE el archivo con la implementación en el Project (error de compilación).

   Ej. 06: Stack_clase.cxx

3) Mantener la implementación separada e instanciar explícitamente las opciones específicas que se necesiten en el mismo archivo (clase_gen.cpp). Si se utilizan funciones ***friend*** las mismas deben estar definidas en el main.

   Ej. 07: Stack_opciones

Declaración explícita de clases

Como en el caso de las plantillas de funciones, se pueden crear especializaciones explícitas de una clase genérica. Para hacerlo se utiliza el constructor **template <>**. Por ejemplo :

template <> class <nombre_clase><int>

le indica al compilador que una especialización del tipo entero se crea para la clase <nombre_clase>. Esto permite manejar fácilmente algunos pocos casos particulares, quedando a cargo del compilador el resto de las variantes.

```
#include <iostream>

using namespace std;

/*** Clase genérica ***/

template <class T> class myclass
{
   private:
      T x;
  public:
      myclass(T a)  { x = a; }
      T getx() { return x; }
};
```

```
/*** Especialización explícita para
     el tipo int ***/
template <> class myclass<int>
{
   private:
      int x;
   public:
      myclass(int a)  { x = a * a; }
      int getx() { return x; }
};
```

```
/*** Programa principal ***/

int main()
{
   myclass<double> d(10.1);
   cout << "\tdouble: " << d.getx() ;

   myclass<int> i(5);
   cout << "\tint: " << i.getx() << "\n\n";
   return 0;
}
```

Ej. 08: Explicita