
UNIVERSIDAD TECNICA
FEDERICO SANTA MARIA

# Plantillas (Templates)

Agustín J. González

ELO-329

# Definición

- Una plantilla (template) es un patrón para crear funciones y clases usando tipos de datos como parámetros. Hay dos tipos de templates:
  - Funciones Templates (tipos parametrizados)
  - Clases Templates (clases parametrizadas)

# Propósito

- Liberar al programador de la molestia de tener que escribir múltiples versiones de la misma función sólo para llevar a cabo la misma operación sobre datos de distinto tipo.
- Consideremos la función swap(). ¿Cuántos tipos diferentes de datos pueden ser intercambiados?
- int double
  string Student
  Motor bool
  etc...

# Función Templates

- Una función plantilla tiene uno o más parámetros.

```
template<class T>
return-type function-name( T param )
```

- T es un parámetro de la plantilla (o template). Este indica el tipo de dato que será suministrado cuando la función sea llamada.

# Ejemplo: Display

- La función Display puede mostrar cualquier tipo de dato con el operador de salida (<<) definido.

```
template <class T>
void Display( const T & val )
{
  cout << val;
}

Display( 10 );
Display( "Hello" );
Display( true );
```

# Función swap()

- Notar las diferencias entre estas dos funciones sobrecargadas (overloaded). Éstas son funciones distintas para el compilador dado que la firma que caracteriza unívocamente a una función está compuesta por el nombre de la función y su lista de parámetros. Este ejemplo es un buen candidato para una plantilla de función.

```
void swap(int & X, int & Y)
{  int temp = X;
  X = Y;
  Y = temp;
}

void swap(string & X, string & Y)
{  string temp = X;
  X = Y;
  Y = temp;
}
```

# Plantilla para la Función swap()

- Esta plantilla trabaja con cualquier tipo de dato que soporte el operador de asignación.

```
template <class TParam>
void Swap( TParam & X, TParam & Y )
{
  TParam temp = X;
  X = Y;
  Y = temp;
}
```

# Función swap() específica

- La siguiente instancia específica de la función swap() puede coexistir en el mismo programa con la plantilla para la función swap().

```
void swap( string * pX, string * pY )
{
  string temp = *pX;
  *pX = *pY;
  *pY = temp;
}
```

# Llamando a swap()

- Cuando llamamos a swap(), el único requerimiento es que ambos parámetros sean del mismo tipo y que sean modificables vía el operador de asignación =.

```
int A = 5;
int B = 6;
swap( A, B );

string P("Al");
string Q("George");
swap( P, Q );

// calls the specific instance of swap()
string * pA = new string("Fred");
string * pB = new string("Sam");
swap( pA, pB );
```

# Llamado a swap()

- Por supuesto, algunas combinaciones no funcionan. No se puede pasar constantes, y no podemos pasar tipos incompatibles.

```
/* casos que no funcionan */
int B = 6;
swap( 10, B );

swap( "Harry", "Sally" );

bool M = true;
swap( B, M );
```

# Parámetros Adicionales

- Una plantilla de función puede tener parámetros adicionales que no son parámetros genéricos.

```
template <class T>
void Display( const T & val, ostream & os )
{
  os << val;
}

Display( "Hello", cout );
Display( 22, outfile );
```

# Parámetros adicionales

- Pueden haber parámetros plantilla adicionales, mientras cada uno sea usado al menos una vez en la lista de parámetros formales de la función. Aquí,  T1 es cualquier contenedor que soporte la operación begin(). T2 es el tipo de datos almacenado en el contenedor.

```
template <class T1, class T2>
void GetFirst( const T1 & container, T2 & value )
{
  value = *container.begin();
}

// more...
```

# Llamado a la función GetFirst ()

- A la función previa le podemos pasar un vector de enteros...

```
vector<int> testVec;
testVec.push_back(32);

int n;
GetFirst(testVec, n );
```

# Llamados a la Función GetFirst()

- O le podemos pasar una lista de datos doubles.
- Pronto veremos la biblioteca estándar de contenedores y veremos vector y list entre otros.

```
list<double> testList;
testList.push_back( 64.253 );

double x;
GetFirst( testList, x );
```

# Plantillas para Clases (Class Templates)

- Las plantillas de clases nos permiten crear nuevas clases durante la compilación.
- Usamos plantillas de clases cuando de otra manera estaríamos forzados a crear múltiples clases con los mismos atributos y operaciones, excepto su tipo.
- En Java también podemos crear plantillas para clases. Ver web, aquí veremos sólo caso C++.
- Las clases estándares de contenedores C++ (Standard C++ Container classes -vector, list, set) son buenos ejemplos.

# Declaración de una Plantilla de Clase

- Este es el formato general para declarar una plantilla de clases. El parámetro T representa un tipo de datos.

```
template <class T>
class className {
public:

private:

};
```

# Ejemplo: Clase Array

- Una clase simple de arreglo podría almacenar entradas de cualquier tipo de datos. Por ejemplo, ésta podría almacenar un arreglo asignado dinámicamente de enteros long:

```
class Array {
public:
  Array( int initialSize );
  ~Array();
  long & operator[]( int i );

private:
  long * m_pData;
  int m_nSize;
};
```

# Ejemplo: clase Array

- Aquí, la clase almacena strings. Difícilmente hay variaciones respecto a la versión previa. Esto la hace un buen candidato para una plantilla de clase.

```
class Array {
public:
  Array( int initialSize );
  ~Array();
  string & operator[]( int i );

private:
  string * m_pData;
  int m_nSize;
};
```

# Plantilla de clase Array

- Esta podría ser la plantilla de la clase Array. Notamos el uso de T como parámetro donde antes teníamos long o string.

```
template <class T>
class Array {
public:
  Array( int initialSize );
  ~Array();
  T & operator[]( int i );

private:
  T * m_pData;
  int m_nSize;
};
```

# Implementación de Array

- Implementación del constructor. Notamos la declaración de la plantilla, la cual es requerida antes de cualquier función en una plantilla de clase.

```
template<class T>
Array<T>::Array( int initialSize )
{
  m_nSize = initialSize;
  m_pData = new T[m_nSize];
}
```

- Nota: Al usar plantillas, la implementación de las funciones debe estar en el archivo de encabezado.

# Implementación de Clase Array

- Implementación del Destructor.

```
template<class T>
Array<T>::~Array()
{
  delete [] m_pData;
}
```

# Implementación de clase Array

- Operador subíndice. No chequearemos el rango todavía. Retorna una referencia, por ello puede ser usada tanto para escribir o leer miembros del arreglo.

```
template<class T>
T & Array<T>::operator[]( int i )
{
  return m_pData[i];
}
```

# Prueba de la Plantilla Array

- Un programa cliente puede crear cualquier tipo de arreglo usando nuestra plantilla mientras la clase admita la asignación (operador =).

```
Array<int> myArray(20);
myArray[2] = 50;

Array<string> nameArray(10);
nameArray[5] = string("Fred");
cout << nameArray[5] << endl;

Array<Student> students(100);
students[4] = Student("123456789");
```

# Clase Student

- Aún si la clase no sobrecarga el operador asignación, C++ crea uno por defecto. Éste hace la copia binaria de cada uno de los miembros dato de la clase; esto es copia baja.
- Ojo: Poner atención con miembros puntero !!

```
class Student {
public:
  Student() { }

  Student(const string & id)
  { m_sID = id; }

private:
  string m_sID;
};
```