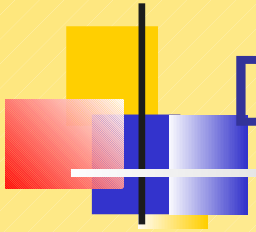
UNIVERSIDAD TECNICA
FEDERICO SANTA MARIA

Plantillas (Templates)

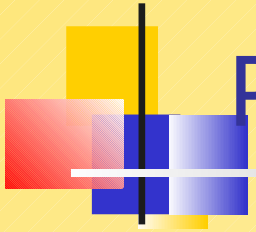
Agustín J. González

ELO-329



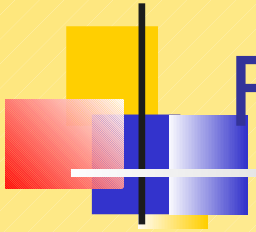
Definición

- 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 distintas funciones para el compilador dado que la firma que caracteriza unívocamente a una función está compuesta por el nombre de la función y sus 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 );
```

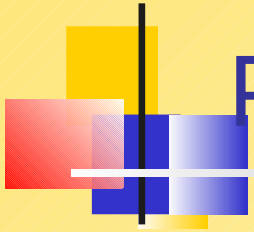
Llamado a swap()

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()

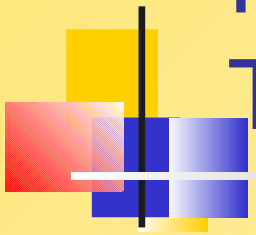
O le podemos pasar una lista de datos doubles.

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.
- Las clases estándares de contenedores C++ (Standard C++ Container classes -vector, list, set) son un 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 a 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: En el caso de plantillas de función, 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

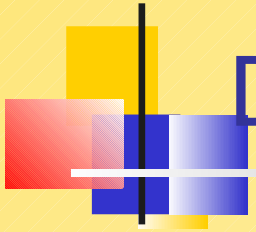
Aún si la clase no sobrecarga el operador asignación, C++ crea uno por defecto. Éste hace la copia binaria de cada uno de los miembros dato de la clase.

Ojo: Poner atención con miembros puntero !!

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

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

private:
  string m_sID;
};
```



Diccionario

Sea *diccionario* una estructura de datos que asocia un valor clave con información relacionada. Por ejemplo, podemos querer asociar número de partes de automóviles (claves) con los nombres de las partes correspondientes:

100000 ---> tire
100001 ---> wheel
100002 ---> distributor
100003 ---> air filter

Para más detalles, ver el código del programa de ejemplo en la página web.

Plantilla para la Clase Dictionary

La plantilla para la clase Dictionary es declarada con dos parámetros de plantilla TKey y TVal. Sus datos miembros incluyen un arreglo de claves y un arreglo de valores asociados.

```
template<class TKey, class TVal>
class Dictionary {
public:
  Dictionary( int initialSize );

  bool Add( const TKey & tk, const TVal & tv );
  // Add a new key and value to the dictionary.

  int Find( const TKey & tk );
  // Search for a key; if found, return its
    // index position; otherwise, return -1
```

Dictionary Class Template

```
  TVal At(int index) const;
  // Return the value at position index

  int size() const;
  // Return the number of items

  friend ostream & operator <<( ostream & os,
     const Dictionary<TKey, TVal> & D );

private:
  vector<TKey> m_vKeys;
  vector<TVal> m_vValues;
};
```

Dictionary Class Template

El constructor toma un valor de tamaño inicial.

```
template<class TKey,class TVal>
Dictionary<TKey,TVal>::Dictionary(int initialSize)
{
  m_vKeys.reserve( initialSize );
  m_vValues.reserve( initialSize );
}
```

Dictionary::Add

```
template<class TKey,class TVal>
bool Dictionary<TKey,TVal>::Add( const TKey & tk,
                                 const TVal & tv )
{
  if( Find(tk) == -1 )        // not in dictionary?
  {
    m_vKeys.push_back(tk);    // add new key
    m_vValues.push_back(tv);  // add associated value
    return true;              // success!
  }
  // must have been a duplicate key
  throw DuplicateKey<TKey>(tk);
  return false;
}
```

Dictionary::Find

La función Find usa busqueda lineal para retornar el indice de la posición que corresponde al elemento.

```
template<class TKey,class TVal>
int Dictionary<TKey,TVal>::Find(const TKey & tk)
{
  for(int i = 0; i < m_vKeys.size(); i++)
  {
    if( tk == m_vKeys[i] )
      return i;
  }
  return -1;                    // not found
}
```

Dictionary::At

La función At retorna el valor a una posición de índice particular.

```
template<class TKey,class TVal>
TVal Dictionary<TKey,TVal>::At(int index) const
{
  return m_vValues[index];
}
```

Operador de salida de datos

El operador de salida de datos es sobrecargado:

```
template<class TKey, class TVal>
ostream & operator <<( ostream & os,
        const Dictionary<TKey, TVal> & D )
{
  for(unsigned i = 0; i < D.size(); i++)
    os << D.m_vKeys[i] << " -> "
         << D.m_vValues[i] << '\n';

  return os;
}
```

Con esto conseguimos que es posible enviar el diccionario completo por la salida.

Ejemplo: Caja de Depósito

El siguiente Dictionary contiene un único número de caja de depósito y su correspondiente número de cliente.

```
Dictionary<int,int> accounts( 100 );

try {
  accounts.Add( 101, 2287 );
  accounts.Add( 102, 2368 );
  accounts.Add( 103, 2401 );
  accounts.Add( 104, 2368 );
  accounts.Add( 103, 2399 );    // duplicate key
}
catch( const DuplicateKey<int> & dk ) {
  cout << dk << endl;
}
```

Ejemplo: Búsqueda

El siguiente código le permite al usuario buscar un número de caja específico. Éste usa las funciones Find y At de la clase Dictionary.

```
cout << "Safety deposit box number to find? ";
int boxNum;
cin >> boxNum;
int pos = accounts.Find( boxNum );
if( pos != -1 )
{
  cout << "Box number " << boxNum
       << " is registered to account number "
       << accounts.At(pos) << '.' << endl;
}
else
  cout << "Box not found." << endl;
```

Ejemplo: Lista de partes o productos

Podemos usar la clase Dictionary para crear la lista de partes de un automóvil en la cual los número de partes son la clave y la descripción de las partes son los valores asociados.

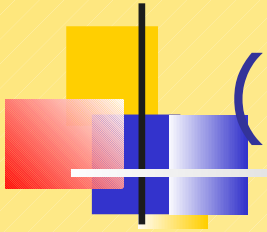
```
cout << "Automobile parts list:\n";

Dictionary<int,string> pList( 200 );
```

(Lista de Partes)

Este código lee la lista de partes desde un archivo de entrada.
Notar el chequeo de valores de clave duplicados.

```
try {
  infile >> partNum;
  while( !infile.eof())
  {
    infile.get();
    getline(infile, description);
    pList.Add( partNum, description );
    infile >> partNum;
  }
}
catch( const DuplicateKey<int> & dk ) {
  cout << dk << endl;
}
```



(Lista de Partes)

Muestra de datos de entrada:

10101 tire
12000 wheel
20001 distributor
22222 air filter
30000 carburetor
30012 fuel pump
33333 flywheel
40000 clutch plate

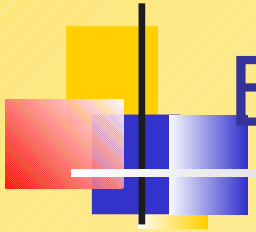
Ejercicio #1

- Crear un operador subíndice sobrecargado que reciba el valor de parámetro de entrada y retorne una referencia a un valor del diccionario. Por ejemplo:

```
string descrip = pList[20001];
```

- El sub-indice podría ser también un string:

```
Dictionary<string,int> aList(20);
aList.Add("Sam",1234);
int n = aList["Sam"];
```



Ejercicio #2

- Crear una función llamada FindVal que nos permita buscar un valor en el diccionario y retorne un vector conteniendo todos las claves asociadas a el valor. Esta es su declaración:

```
vector<TKey> FindVal( const TVal & searchFor );
```