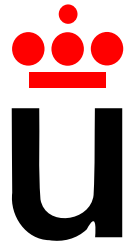
7. Plantillas. Librería STL

Julio Vega

julio.vega@urjc.es



Universidad
Rey Juan Carlos

Contenidos

- Las plantillas persiguen el objetivo de siempre: reutilizar código.
- Existen **plantillas de funciones** y **plantillas de clases**.
- Permiten especificar, con un código, varias func./clases relacionadas.
  - A esta técnica se le conoce como **programación genérica**.
  - Las variedades generadas son las **especializaciones de las plantillas**.
- Son como las plantillas que usábamos de pequeño para trazar figuras.
  - Una especialización sería dibujar la figura en azul; otra, en verde, etc.
- E.g. plantilla de clase tipo pila que valga para cualquier tipo de elem.
  - C++ genera las especializaciones de plantillas de clases separadas.
    - Tamaño código = implementación func. sobrecargadas por separado.
    - E.g. *clase pila de* `int`, otra de `float`, otra de `string`, etc.
- Normal/ las plantillas se definen en el `.h` y se incluyen en los fuentes.

- Las funciones sobrecargadas realizan operaciones similares.
- Las plantillas de funciones realizan operaciones idénticas.
  - Lo único que cambian son los tipos de datos sobre los que operan.
- La definición de plantilla de función comienza por `template`.
  - Seguida por los parámetros de plantilla entre `<` y `>`.
  - A cada parámetro se le antepone la palabra `class` o `typename`.
    - Significan *cualquier tipo integrado o definido por el usuario*.

```
template <typename T>

template <class Mamifero>

template <typename Alumno, typename Profesor>
```

- La función se define después y su sintaxis es como siempre.

```
template <typename T>
void printArray (const T * const array, int numElems) {
  // Si T es un tipo definido por el usuario, el operador <<
      debe estar sobrecargado para ese tipo; si no, no compila
  for (int i = 0; i < numElems; i++) cout << array [i] << " ";
}

int main() {
  const int sizeA = 6, sizeB = 7, sizeC = 8; // arrays size
  int arrayA [sizeA] = {1, 2, 3, 4, 5, 6};
  double arrayB [sizeB] = {1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7};
  char arrayC [sizeC] = "WELCOME"; // 8th position for null
  cout << "Array A: " << endl;
  printArray (arrayA, sizeA); // receives an int-array
  cout << "Array B: " << endl;
  printArray (arrayB, sizeB); // receives a double-array
  cout << "Array C: " << endl;
  printArray (arrayC, sizeC); // receives a char-array
}
```

- Pensemos en una funcionalidad independiente de los datos a manejar.
  - E.g. una pila, cuyo comportamiento es independiente de los elementos.
  - Podemos crear esa *plantilla* de comportamiento = gran reutilización sw.
- Se escribe solo una definición de plantilla de clase.
  - Y por cada especialización de instancia, se indica el tipo a usar.

```
// Class-template definition:
template <typename T>
class Stack { ... };
// [...]


// Use of Stack class:
Stack <double> miDoubleStack (5);
Stack <int> miIntStack (12);
// [...]
```

```
template <typename T>
class Stack {
public:
  Stack (int = 20); // (default Stack size = 20)
  ~Stack() { delete [] stackPtr; }
  bool push (const T &); // push an element onto the Stack
  bool pop (T &); // pop an element off the Stack
  bool isEmpty() const { return top == -1; }
  bool isFull() const { return top == size - 1; }
private:
  int size; // number of elements in the stack
  int top; // location of the top element (-1 means empty)
  T *stackPtr; // pointer to internal representation of the Stack
}; // end class template Stack

template <typename T>
Stack <T>::Stack (int s): size (s > 0 ? s : 20), // validation
  top (-1), // initially is empty
  stackPtr (new T [size]) { } // allocate memory for elements
```

```
template <typename T>
bool Stack <T>::push (const T &elem) {
  if (!isFull()) {
    stackPtr [++top] = elem; // place elem on Stack
    return true; // push successful
  } // otherwise:
  return false; // push unsuccessful
}

template <typename T>
bool Stack <T>::pop (T &elem) {
  if (!isEmpty()) {
    elem = stackPtr [top--]; // remove elem from Stack
    return true; // pop successful
  } // otherwise:
  return false; // pop unsuccessful
}
```

```
int main() {
  Stack <double> dStack (5);
  double dValue = 1.0;
  while (dStack.push (dValue)) { // pushing 5 doubles
    cout << dValue << ' ';
    dValue += 1.0;
  }
  while (dStack.pop (dValue)) // poping elements
    cout << dValue << ' ';

  Stack <int> iStack; // default size 20
  int iValue = 1;
  while (iStack.push (iValue)) { // pushing 20 integers
    cout << intValue++ << ' ';
    iValue += 1.1;
  }
  while (intStack.pop (intValue)) // poping elements
    cout << intValue << ' ';
} // end main
```

- Programadores usan normal/ muchas estruct. de datos y algoritmos.
  - Por ello, el comité del estándar de C++ agregó la STL a este.
- La STL está formada por contenedores, iteradores y algoritmos.
  - Permite escribir códigos que no dependen del contenedor subyacente.
    - Este estilo de programación se conoce como **programación genérica**.
- Los **containers** son estr. datos que almacenan todo tipo de datos.
  - Cada contenedor tiene asociadas varias funciones miembro.
- Los **iter.** ($\sim$ punteros) son para manipular los elem. de los *containers*.
  - Se pueden usar punteros pero iteradores quedan mucho más *elegantes*.
- Los **algoritmos** son funciones que hacen manipulaciones comunes.
  - E.g. búsqueda, ordenación y comparación de elementos.
  - STL proporciona unos 70 algoritmos aprox., la mayoría con iteradores.

- Pueden ser de secuencia, asociativos o adaptadores de contenedores.
  - Los adapt. de cont. tb se llaman de 2.ª clase; y, los otros, de 1.ª.
- **De secuencia**: representan estruc. datos lineales (vector, lista enl.).
  - `vector`: con acceso directo e inserción/eliminación rápida al final.
  - `deque`: ídem al anterior, pero ins./elim. rápida en inicio y final.
  - `list`: lista con enlace doble, ins./elim. rápida en cualquier parte.
- **Asociativos**: son no lineales, y pueden encontrar elem. rápidamente.
  - `set`: búsqueda rápida, no se permiten duplicados.
  - `multiset`: ídem al anterior, pero sí se permiten duplicados.
  - `map`: asignación uno-uno, $\nexists$ duplicados, búsqueda rápida por claves.
  - `multimap`: ídem al anterior, pero asignación uno-varios.
- **Adapt. de cont.**: versiones restringidas de los cont. anteriores.
  - `stack`: último en entrar, primero en salir (LIFO).
  - `queue`: primero en entrar, primero en salir (FIFO).
  - `priority_queue`: elem. de más prioridad es el primero en salir.

- Constructor predeterminado, const. de copia y destructor.
- `empty`: devuelve `true` si no hay elem. en el cont.
- `insert`: inserta un elemento en el contenedor.
- `size`: devuelve el n.^{o} de elementos del contenedor.
- `capacity`: n.^{o} elem. que caben antes de cambiar tamaño dinámica/.
- `operator=`: asigna un contenedor a otro.
- `operator<`: devuelve `true` si el 1.^{er} cont. es menor que el 2.^{o}.
  - Y siguiendo la misma dinámica, los comparadores `<=`, `>`, `>=`, `==`, `!=`.
- `swap`: intercambia los elementos de los contenedores.

- `max_size`: devuelve el n.º máx. de elementos para un cont.
- `begin`: devuelve iterador que apunta al primer elemento del cont.
  - $\exists$ dos versiones según devuelva un `iterator` o un `const_iterator`.
- `end`: devuelve iterador a la siguiente posición del último elem.
  - $\exists$ dos versiones según devuelva un `iterator` o un `const_iterator`.
- `rbegin` y `rend`: consideran la estructura a la inversa (*reverse*).
  - `rbegin`: devuelve iterador que apunta al último elemento del cont.
  - `rend`: devuelve iterador a la posición de antes del primer elem.
  - $\exists$ dos versiones, con `reverse_iterator` o `const_reverse_iterator`.
- `erase`: elimina uno o varios elementos del contenedor.
- `clear`: elimina todos los elementos del contenedor.

- `<vector>`
- `<list>`
- `<deque>`
- `<queue>` Contiene tanto a `queue` como a `priority_queue`.
- `<stack>`
- `<map>` Contiene tanto a `map` como a `multimap`.
- `<set>` Contiene tanto a `set` como a `multiset`.
- `<valarray>`
- `<bitset>`

Todo el contenido de estos archivos cabecera está en el `namespace std`.

- `allocator_type`: tipo de obj. usado para asignar memoria del cont.
- `value_type`: tipo de elem. almacenado en el contenedor.
- `reference`: ref. al tipo de elem. almacenado en el contenedor.
- `const_reference`: ref. cte. al tipo de elem. almacenado en el cont.
  - Esta solo se puede usar para leer elem. e insertarlos en el cont.
  - También para realizar operaciones tipo `const`.
- `pointer`: puntero al tipo de elem. almacenado en el contenedor.
- `const_pointer`: puntero cte. al tipo elem. almacenado en el cont.
- `iterator`: iterador que apunta al tipo elem. almacenado en el cont.
- `const_iterator`: iterador similar al `const_reference`.
- `reverse_iterator`: iterador para iterar en sentido inverso.
- `const_reverse_iterator`: como `const_iterator` en sent. inverso.
- `difference_type`: tipo resultado al restar dos iter. del mismo cont.
- `size_type`: tipo usado para contar elem. en un contenedor.

- Los iteradores son muy similares a los punteros, pero más sofisticados.
- Si `i` apunta a elem., `++i` apunta al sig. y `*i` es el contenido de `i`.
- El it. que vierte `end` se usa normal/ en comparación para saber si fin.
- Tipo `iterator` se usa para referenciar elem. que puede modificarse.
  - Y el `const_iterator` para ref. elemento que no puede modificarse.
- Los contenedores `vector` y `deque` admiten it. de acceso aleatorio.
  - `list`, `set`, `multiset`, `map` y `multimap`, solo bidireccional.
  - `stack`, `queue` y `priority_queue` no soportan iteradores.
- Para operar con flujos: `istream_iterator` y `ostream_iterator`.
  - Estos flujos pueden estar en contenedores o pueden ser flujos de I/O.

```
#include <iostream>
#include <iterator>
using namespace std;

int main() {
  cout << "Enter a pair of integers: ";
  istream_iterator <int> inInt (cin); // read integers << cin

  int n1 = *inInt; // first value from standard input
  ++inInt; // shift iterator to next value
  int n2 = *inInt; // second value from standard input

  ostream_iterator <int> outInt (cout); // write integers >> cout

  cout << "Adding numbers: ";
  *outInt = n1 + n2;
  cout << endl;
}
```

- Se pueden utilizar genéricamente mediante diferentes contenedores.
  - Tb. pueden ser extendidos sin modificar los contenedores de la STL.
- Los alg. operan sobre los elementos indirectamente, con iteradores.
  - Por contra de un diseño puro de clases donde métodos = algoritmos.
  - Para facilitar tener alg. genéricos aplicables a muchos contenedores.
- $\exists$ diferentes tipos de algoritmos, según actúen sobre los contenedores.
  - Algoritmos de secuencia cambiantes: modifican los contenedores.
  - Algoritmos de secuencia no cambiantes: no producen modificaciones.
    - Contenedores de secuencia (cambiante o no): `vector`, `list` y `deque`.
  - Algoritmos numéricos (librería `<numeric>`): op. numéricas.

De secuencia cambiantes

`copy` `remove` `reverse_copy` `copy_backward` `remove_copy` `rotate` `fill` `remove_copy_if` `rotate_copy` `fill_n` `remove_if` `stable_partition` `generate` `replace` `swap` `generate_n` `replace_copy` `swap_ranges` `iter_swap` `replace_copy_if` `transform` `partition` `replace_if` `unique` `random_shuffle` `reverse` `unique_copy`

De secuencia no cambiantes

`adjacent_find` `find` `find_if` `count` `find_each` `mismatch` `count_if` `find_endsearch` `equal` `find_first_of` `search_n`

Numéricos (<numeric>)

`accumulate` `partial_sum` `innerproduct` `adjacent_difference`

```
#include <iostream>
#include <vector>
using namespace std;

template <typename T> void printV (const vector <T> &intV);

int main() {
  const int V_SIZE = 8;
  int array [V_SIZE] = {1, 2, 3, 4, 5, 6, 7, 8};
  vector <int> integers; // vector of integers

  cout << "Content before adding elements:"
    << "\n-integers size: " << integers.size()
    << "\n-integers capacity: " << integers.capacity();

  // function push_back adds a new element at the end
  integers.push_back (31);
  integers.push_back (66);
  integers.push_back (11);
```

```
  cout << "\nContent after using push_back:"
    << "\n-integers size: " << integers.size()
    << "\n-integers capacity: " << integers.capacity();

  cout << "\n\nShowing array content using pointers: ";
  for (int *ptr = array; ptr != array + V_SIZE; ptr++)
    cout << *ptr << ' ';

  cout << "\nShowing vector content using iterators: ";
  printV (integers);

  cout << "\nShowing vector content in reverse mode: ";
  vector <int>::const_reverse_iterator rIt;
  vector <int>::const_reverse_iterator tempIt = integers.rend();
  for (rIt = integers.rbegin(); rIt != tempIt; ++rIt)
    cout << *rIt << ' ';

  cout << endl;
} // end main
```

```
// function template for displaying vector elements
template <typename T> void printV (const vector <T> &intV) {
  typename vector <T>::const_iterator constIt;

  // showing array content using a const_iterator
  for (constIt = intV.begin(); constIt != intV.end(); ++constIt)
    cout << *constIt << ' ';
}
```

- Nótese que, tras usar `push_back` tres veces, `capacity = 4`.
  - El comportamiento de `push_back` depende de implementación de STL.
  - En este caso, el vector está aumentando su tamaño al doble.
    - `p_b` $\rightarrow$ `capacity=1`, `p_b` $\rightarrow$ `capacity=2`, `p_b` $\rightarrow$ `capacity=4`.
    - Esta implementación puede provocar gran desperdicio de memoria.
  - Para controlar uso de memoria $\Longrightarrow$ usar funciones `resize` y `reserve`.

```
#include <iostream>
#include <vector>
#include <algorithm> // includes copy algorithm
#include <iterator>
#include <stdexcept>
using namespace std;

int main() {
  const int V_SIZE = 8;
  int array [V_SIZE] = {1, 2, 3, 4, 5, 6, 7, 8};
  vector <int> integers (array, array + V_SIZE);
  ostream_iterator <int> outIt (cout, " ");

  cout << "Showing vector content using output stream: ";
  copy (integers.begin(), integers.end(), outIt);

  cout << "\nThe first element is: " << integers.front()
    << "\nAnd the last one is: " << integers.back();
```

```
integers [0] = 31; // 0 is the first indexed position
integers.at (2) = 23; // another way to set an elem.
integers.insert (integers.begin() + 1, 77); // to 2nd pos.

cout << "\n\nVector content (after changes): ";
copy (integers.begin(), integers.end(), outIt);

try { // trying to access an out-of-range element
  integers.at (100) = 777;
} // an exception will be raised!
catch (out_of_range &outOfRange) { // out_of_range exception
  cout << "\nException: " << outOfRange.what();
} // end catch
```

- Ya veremos excepciones en profundidad. En STL existen estos tipos:
  - `out_of_range`: indica cuando el subíndice está fuera de rango.
  - `invalid_argument`: si se pasa un argumento inválido a una función.
  - `length_error`: si se intenta crear un contenedor demasiado largo.
  - `bad_alloc`: al intentar asignar memoria con `new` y ∄ memoria.

```
  integers.erase (integers.begin()); // erase first element
  cout << "\n\nVector content after erasing the first elem.: ";
  copy (integers.begin(), integers.end(), outIt);

  integers.erase (integers.begin(), integers.end()); // all
  cout << "\nVector content after erasing remaining elem. "
    << (integers.empty() ? "is" : "is not") << " empty";

  // insert elements from array
  integers.insert (integers.begin(), array, array + V_SIZE);
  cout << "\nVector content before clear(): ";
  copy (integers.begin(), integers.end(), outIt);

  integers.clear(); // it calls erase to empty a collection
  cout << "\nVector content after clear() "
    << (integers.empty() ? "is" : "is not") << " empty" << endl;
} // end main
```

```
#include <iostream>
#include <list>
#include <algorithm>
#include <iterator>
using namespace std;

// prototype for function template printL
template <typename T> void printL (const list< T > &lRefs);

int main() {
  const int V_SIZE = 4;
  int array [V_SIZE] = {1, 2, 3, 4};
  list <int> integers;
  list <int> integers2;

  integers.push_front (2);
  integers.push_front (4);
  integers.push_back (8); // remember: reverse mode!
  integers.push_back (6);
```

```
cout << "integers content: ";
printL (integers);

integers.sort(); // sorting list in ascendent order
cout << "\nintegers content after sorting: ";
printL (integers);

// insert elements of array into integers2
integers2.insert (integers2.begin(), array, array + V_SIZE);
cout << "\nintegers2 content after insert: ";
printL (integers2);

// remove integers2 elements and insert at end of integers
integers.splice (integers.end(), integers2);
cout << "\nintegers content after splice: ";
printL (integers);
cout << "And integers2 content after splice: ";
printL (integers2);
```

```
integers.sort();
cout << "\nintegers content after sorting: ";
printL (integers);

integers2.insert (integers2.begin(), array, array + V_SIZE);
integers2.sort();
cout << "\nintegers2 content after inserting and sorting: ";
printL (integers2);

integers.merge (integers2); // =splice, but insert in order
cout << "\nintegers content after merging integers2: ";
printL (integers);
cout << "And integers2 content: ";
printL (integers2);

integers.pop_front(); // remove front element
integers.pop_back(); // remove back element
cout << "\nintegers content after poping front&back elem.: ";
printL (integers);
```

```
integers.unique(); // remove duplicate elements
cout << "\nintegers content after unique operation: ";
printL (integers);

integers.swap (integers2); // swap elements
cout << "\nintegers content after swapping : ";
printL (integers);
cout << "And integers2 content: ";
printL (integers2);

// integers content is replaced with elements of integers2
integers.assign (integers2.begin(), integers2.end());
cout << "\nintegers content after assign operation: ";
printL (integers);

integers.merge(integers2);
cout << "\nintegers content after merging integers2: ";
printL (integers);
```

```
  integers.remove (2); // remove all 2s
  cout << "\nintegers content after removing all 2s: ";
  printL (integers);
  cout << endl;
} // end main

template <typename T> void printL (const list <T> &lRefs) {
  if (lRefs.empty())
    cout << "List empty!";
  else {
    ostream_iterator <T> output (cout, " ");
    copy (lRefs.begin(), lRefs.end(), output);
  }
}
```

```
#include <iostream>
#include <deque>
#include <algorithm>
#include <iterator>
using namespace std;

int main() {
  deque <double> doubles; // deque of doubles
  ostream_iterator <double> outIt (cout, " ");

  doubles.push_front (3.1);
  doubles.push_front (4.3);
  doubles.push_back (1.5);
```

```
  cout << "doubles content: "; // output: 4.3 3.1 1.5
  for (unsigned int i = 0; i < doubles.size(); i++ )
    cout << doubles [i] << ' '; // subscript op. to get elem.

  doubles.pop_front(); // rm 1st. elem., output: 3.1 1.5
  cout << "\ndoubles content after pop_front: ";
  copy (doubles.begin(), doubles.end(), outIt);

  doubles [1] = 6.7; // subscript op. to set value at pos. 1
  cout << "\ndoubles content after doubles [1] = 6.7: ";
  copy (doubles.begin(), doubles.end(), outIt); // out: 3.1 6.7
  cout << endl;
} // end main
```

```
#include<iostream>
#include<set>
#include<string>
class MyObject {
public:
  MyObject(std::string name_, std::string surname_) :
      name (name_), surname (surname_) {}
  bool operator< (const MyObject & object) const {
      std::string right = object.name + object.surname;
      std::string left = this->name + this->surname;
      return (left < right); }
  friend std::ostream& operator<<(std::ostream& os, const
      MyObject& obj);
private:
  std::string name;
  std::string surname;
}; // end class MyObject
std::ostream& operator<< (std::ostream& os, const MyObject& o) {
    os << "Name: " << o.name << " & " << "Surname " << o.surname
        << std::endl; return os; }
```

```
int main() {
    std::set<MyObject> mySetOfObjects;
    MyObject obj1("name1", "surname1");
    MyObject obj2("name2", "surname2");
    MyObject obj3("name3", "surname3");
    MyObject obj4("name1", "surname1"); // duplicate object
    mySetOfObjects.insert(obj1);
    mySetOfObjects.insert(obj2);
    mySetOfObjects.insert(obj3);
    mySetOfObjects.insert(obj4); // it will not be inserted
        because its duplicate

    // iteration through all the elements in the set
    for (std::set<MyObject>::iterator it=mySetOfObjects.begin();
        it!=mySetOfObjects.end(); ++it)
      std::cout << *it;

    return 0;
}
```

```
#include <iostream>
#include <stack>
#include <vector>
#include <list>
using namespace std;
template <typename T> void pushElems (T &stackRef);
template <typename T> void popElems (T &stackRef);

int main() {
  stack <int> dequeStack; // stack with a deque container
  stack <int, vector <int>> vectorStack; // with a vector cont.
  stack< int, list< int > > listStack; // with a list cont.
  cout << "Pushing elements on the dequeStack: ";
  pushElems (dequeStack);
  cout << "\nPushing elements on the vectorStack: ";
  pushElems (vectorStack);
  cout << "\nPushing elements on the listStack: ";
  pushElems (listStack);
  cout << "\n\n";
```

```
  cout << "Popping elements from the dequeStack: ";
  popElems (dequeStack);
  cout << "\nPopping elements from the vectorStack: ";
  popElems (vectorStack);
  cout << "\nPopping elements from the listStack: ";
  popElems (listStack);
  cout << endl;
} // end main
template <typename T> void pushElems (T &stackRef) {
  for (int i = 1; i <= 10; i++) {
    stackRef.push (i);
    cout << stackRef.top() << ' '; // show the top element
  }
}
template <typename T> void popElems (T &stackRef) {
  while (!stackRef.empty()) {
    cout << stackRef.top() << ' '; // show the top element
    stackRef.pop(); // it removes the top element
  }
}
```

```
#include <iostream>
#include <queue>
using namespace std;

int main() {
  queue <double> doubles;

  doubles.push (3.1);
  doubles.push (4.3);
  doubles.push (1.5);

  cout << "Popping from doubles: ";
  while (!doubles.empty()) {
    cout << doubles.front() << ' '; // show front element
    doubles.pop(); // it removes the front element
  }
  cout << endl;
}
```

7. Plantillas. Librería STL

Julio Vega

julio.vega@urjc.es