Arrays y Templates

Prof. Ing. José María Sola.

Introducción a Arrays y Templates Por Ing. José María Sola

1. Array

Array

2. Array of Array

Array of Array

3. Matrix Template

Matrix Template

4. Matrix Template Module

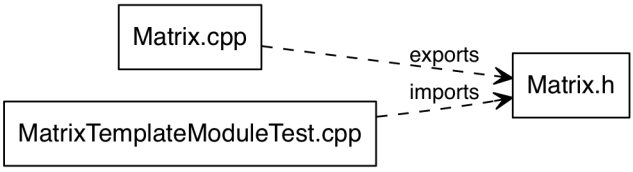

Figura 1. Conceptual

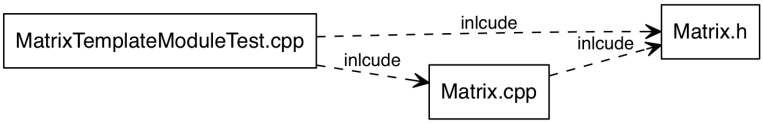

Figura 2. Implementación

5. Listados Completos

5.1. Array

Array.cpp.

```
/* Arrays, template functions,
```

```
 * range-for, size_t, sizeof, auto, intializazation, const, parameters
 (in-out-inout), type inference.
 * 201304-201608
 * José María Sola
 */

int main(){
 void TestArrayVariables();
 TestArrayVariables();

 void TestNonGenericArrayFunctions();
 TestNonGenericArrayFunctions();

 void TestGenericLengthArrayFunctions();
 TestGenericLengthArrayFunctions();

 void TestGenericTypeAndLengthArrayFunctions();
 TestGenericTypeAndLengthArrayFunctions();
}

#include <cstddef>
using std::size_t;

#include <array>
using std::array;

#include <iostream>
using std::cout;

#include <string>

void TestArrayVariables(){
 cout << __func__ << '\n'; // Imprime el nombre de la función.

 array<int,7> x;          // Declaración, sin incialización.
 x.at(1) = 42;            // Set/Write.
 cout << x.at(1) << '\n'; // Get/Read.

 array<int,3> a = {{10, 20, 30}}; // Declaración e inicialización.
 // Un array es un struct con un data member, por eso las dobles llaves.

 cout
  << a.at(0)          << '\t'  // Muestra el primer elemento,
  << a.at(2)          << '\n'  // y el último de forma particular
  << a.at(a.size()-1) << '\n'; // y genéricamente.
```

```
// El tamaño de un arreglo es igual al tamaño del tipo de sus elementos
por la cantidad de elementos del arreglo.
cout
 << "sizeof a                          : " << sizeof a
         << '\n'
 << "sizeof(array<int,3>)              : " << sizeof(array<int,3>)
         << '\n'
 << "sizeof a.at(0)                    : " << sizeof a.at(0)
         << '\n'
 << "sizeof(int)                       : " << sizeof(int)
         << '\n'
 << "a.size()                          : " << a.size()
         << '\n'
 << "sizeof a / sizeof a.at(0)         : " << sizeof a / sizeof a.at(0)
         << '\n'
 << "sizeof(array<int,3>) / sizeof(int): "
<< sizeof(array<int,3>) / sizeof(int) << '\n';

// Iterar arreglo para get con for (for general).
for(size_t i=0; i < 3; ++i) // size_t.
 cout << a.at(i) << '\t';
cout << '\n';

// Iterar arreglo para set con for.
for(size_t i=0, n=a.size(); i < n; ++i) // n.
 a.at(i) *= 10;

// Iterar arreglo para get con range-for (for-auto).
for(auto e : a) // auto.
 cout << e << '\t';
cout << '\n';
/* También
for(const auto& e : x)
 ...
pero para tipos simples no es necesario, hasta es menos performante.
*/

// Iterar arreglo para set con range-for.
for(auto& e : a) // auto&.
 e /= 10;

for(auto e : a)
 cout << e << '\t';
cout << '\n';

// Array de otros tipos
```

```
 // Array de doubles
 array<double,7> ad = {{0.1, 2.3, 4.5, 6.7}}; // 4:0.0, 5:0.0, 6:0.0
 for(auto e : ad)
  cout << e << '\t';
 cout << '\n';

 // Array de strings
 array<std::string,3> as = {{"C++", "Moderno"}}; // 2:""
 for(auto e : as)
  cout << e << '\t';
 cout << '\n';
}

// Non-Generic functions declarations.

void PrintArrayInt5(const array<int,5>& x); // in array.

void MultiplyArrayInt5(array<int, 5>& x, int k); // inout array.

array<int, 5> AddArrayInt5(const array<int, 5>& x, const array<int, 5>&
 y); // return array.

void TestNonGenericArrayFunctions(){
 cout << __func__ << '\n';

 array<int, 5> a = {{1,2,3,4,5}};
 cout << "a    :\t"; PrintArrayInt5(a);

 auto b = a;
 cout << "b    :\t"; PrintArrayInt5(b);

 MultiplyArrayInt5(a,10);
 cout << "a*10 :\t"; PrintArrayInt5(a);

 auto c = AddArrayInt5(a,b); // Variable auxiliar.
 cout << "c=a+b:\t"; PrintArrayInt5(c);

 cout << "a+b  :\t"; PrintArrayInt5(AddArrayInt5(a,b)); // Composición.
}

// Non-Generic functions definitions.

void PrintArrayInt5(const array<int,5>& x){
 for(auto e : x)
  cout << e << '\t';
```

```
 cout << '\n';
}

void MultiplyArrayInt5(array<int, 5>& x, int k){
 for(auto& e : x)
  e *= k;
}

array<int, 5> AddArrayInt5(const array<int, 5>& x, const array<int, 5>&
 y){
 array<int, 5> z;
 for(size_t i=0; i < 5; ++i)
  z.at(i) = x.at(i) + y.at(i);
 return z;

 /* Otra forma:
 auto z = x;
 for(auto i=0; i < 5; ++i)
  z.at(i) += y.at(i);
 return z;
 */
}

// Length-generic functions declarations.

template <size_t n>
void PrintArrayInt(const array<int, n>& x); // in array.

template <size_t n>
void MultiplyArrayInt(array<int, n>& x, int k); // inout array.

template <size_t n>
array<int, n> AddArrayInt(const array<int, n>& x, const array<int, n>&
 y); // return array.

void TestGenericLengthArrayFunctions(){
 cout << __func__ << '\n';

 array<int, 5> a = {{1,2,3,4,5}};
 cout << "a    :\t"; PrintArrayInt(a);

 auto b = a;
 cout << "b    :\t"; PrintArrayInt(b);

 MultiplyArrayInt(a,10);
 cout << "a*10 :\t"; PrintArrayInt(a);
```

```
 auto c = AddArrayInt(a,b); // Variable auxiliar.
 cout << "c=a+b:\t"; PrintArrayInt(c);

 cout << "a+b  :\t"; PrintArrayInt(AddArrayInt(a,b)); // Composición.
}

// Length-generic functions definitions.

template <size_t n>
void PrintArrayInt(const array<int, n>& x){
 for(auto e : x)
  cout << e << '\t';
 cout << '\n';
}

template <size_t n>
void MultiplyArrayInt(array<int, n>& x, int k){
 for(auto& e : x)
  e *= k;
}

template <size_t n>
array<int, n> AddArrayInt(const array<int, n>& x, const array<int, n>&
 y){
 auto z = x;
 for(size_t i=0; i < n; ++i)
  z.at(i) += y.at(i);
 return z;
}

// Type-and-length generic functions declarations.

template <typename T, size_t n>
void PrintArray(const array<T, n>& x); // in array.

template <typename T, size_t n>
void MultiplyArray(array<T, n>& x, int k); // inout array.

template <typename T, size_t n>
array<T, n> AddArray(const array<T, n>& x, const array<T, n>& y); //
 return array.

void TestGenericTypeAndLengthArrayFunctions(){
 cout << __func__ << '\n';
```

```
 array<int, 5> a = {{1,2,3,4,5}};
 cout << "a    :\t"; PrintArray(a);

 auto b = a;
 cout << "b    :\t"; PrintArray(b);

 MultiplyArray(a,10);
 cout << "a*10 :\t"; PrintArray(a);

 auto c = AddArray(a,b); // Variable auxiliar.
 cout << "c=a+b:\t"; PrintArray(c);

 cout << "a+b  :\t"; PrintArray(AddArray(a,b)); // Composición.
}

// Type-and-length generic functions definitions.

template <typename T, size_t n>
void PrintArray(const array<T, n>& x){
 for(auto e : x)
  cout << e << '\t';
 cout << '\n';
}

template <typename T, size_t n>
void MultiplyArray(array<T, n>& x, int k){
 for(auto& e : x)
  e *= k;
}

template <typename T, size_t n>
array<T, n> AddArray(const array<T, n>& x, const array<T, n>& y){
 auto z = x;
 for(size_t i=0; i < n; ++i)
  z.at(i) += y.at(i);
 return z;
}
```

Array output.

```
TestArrayVariables
42
10 30
30
sizeof a                          : 12
```

```
sizeof(array<int,3>)              : 12
sizeof a.at(0)                    : 4
sizeof(int)                       : 4
a.size()                          : 3
sizeof a / sizeof a.at(0)         : 3
sizeof(array<int,3>) / sizeof(int): 3
10 20 30
100 200 300
10 20 30
0.1 2.3 4.5 6.7 0 0 0
C++ Moderno
TestNonGenericArrayFunctions
a    : 1 2 3 4 5
b    : 1 2 3 4 5
a*10 : 10 20 30 40 50
c=a+b: 11 22 33 44 55
a+b  : 11 22 33 44 55
TestGenericLengthArrayFunctions
a    : 1 2 3 4 5
b    : 1 2 3 4 5
a*10 : 10 20 30 40 50
c=a+b: 11 22 33 44 55
a+b  : 11 22 33 44 55
TestGenericTypeAndLengthArrayFunctions
a    : 1 2 3 4 5
b    : 1 2 3 4 5
a*10 : 10 20 30 40 50
c=a+b: 11 22 33 44 55
a+b  : 11 22 33 44 55
```

5.2. Array of Array

ArrayOfArray.cpp.

```
/* Arrays-of-Arrays, matrixes, template functions,
 * range-for, size_t, sizeof, auto, intializazation, const, parameters
 (in-out-inout), type inference.
 * 201304-201608
 * José María Sola
 */

int main(){
 void TestArrayOfArrayVariables();
 TestArrayOfArrayVariables();
```

```
 void TestNonGenericArrayOfArrayFunction();
 TestNonGenericArrayOfArrayFunction();

 void TestGenericArrayOfArrayFunctions();
 TestGenericArrayOfArrayFunctions();
}

#include <cstddef>
using std::size_t;

#include <array>
using std::array;

#include <iostream>
using std::cout;

void TestArrayOfArrayVariables(){
 cout << __func__ << '\n';      // Imprime el nombre de la función.

 array<array<int,3>,5> x;       // Declaración, sin incialización.
 x.at(3).at(2) = 42;            // Set/Write.
 cout << x.at(3).at(2) << '\n'; // Get/Read.

 array<array<int,3>,5> m = {{   // Declaración e inicialización.
  {{ 1,  2,  3}},
  {{ 4,  5,  6}},
  {{ 7,  8,  9}},
  {{10, 11, 12}},
  {{13, 14, 15}}
 }};
 // Un array es un struct con un data member, por eso las dobles llaves.

 cout
  << m.at(0).at(0)                                    << '\t'  //
 Muestra el primer elemento,
  << m.at(4).at(2)                                    << '\t'  // y el
 último de forma particular
  << m.at(m.size()-1).at(m.at(m.size()- 1).size()- 1) << '\n'; // y
 genéricamente.

 // El tamaño de un arreglo es igual al tamaño del tipo de sus elementos
 por la cantidad de elementos del arreglo.
 cout
  << "sizeof m                    : " << sizeof m
 << '\n'
```

```
  << "sizeof(array<array<int,3>,5>): " << sizeof(array<array<int,3>,5>)
 << '\n';

 // Iterar arreglo para get con for (for general).
 for(size_t i=0; i < 5; ++i){ // size_t.
  for(size_t j=0; j < 3; ++j)
   cout << m.at(i).at(j) << '\t';
  cout << '\n';
 }

 // Iterar arreglo para set con for.
 for(size_t i=0, n=m.size(); i < n; ++i) // n.
  for(size_t j=0, n=m.at(i).size(); j < n; ++j)
   m.at(i).at(j) = i*10+j;

 // Iterar arreglo para get con range-for (for-auto).
 for(auto row : m){ // auto.
  for(auto e : row)
   cout << e << '\t';
  cout << '\n';
 }

 // Iterar arreglo para set con range-for.
 for(auto& row : m) // auto&
  for(auto& e : row)
   e *= 10;

 for(auto row : m){
  for(auto e : row)
   cout << e << '\t';
  cout << '\n';
 }
}

void TestNonGenericArrayOfArrayFunction(){
 cout << __func__ << '\n';      // Imprime el nombre de la función.

 array<array<int,2>,3> m = {{
  {{0, 1}},
  {{2, 3}},
  {{4, 5}}
 }};

 void PrintMatrixInt3x2(const array<array<int,2>,3>& x);
 cout << "m:\n"; PrintMatrixInt3x2(m);
}
```

```
void PrintMatrixInt3x2(const array<array<int,2>,3>& x){
 for(auto row : x){
  for(auto e : row)
   cout << e << '\t';
  cout << '\n';
 }
}

// Template functions declarations

template <typename T, size_t rows, size_t columns>
void PrintMatrix(const array<array<T, columns>, rows>& x); // in.

template <typename T, size_t rows, size_t columns>
void MultiplyMatrix(array<array<T, columns>, rows>& x, int k); // inout.

template <typename T, size_t rows, size_t columns>
array<array<T, columns>, rows> // return.
AddMatrix(
 const array<array<T, columns>, rows>& x,
 const array<array<T, columns>, rows>& y
);

void TestGenericArrayOfArrayFunctions(){
 cout << __func__ << '\n';

 array<array<int,3>,5> a = {{
   {{11,12,13}},
   {{21,22,23}},
   {{31,32,33}},
   {{41,42,43}},
   {{51,52,53}}
  }};

 cout << "a    :\n"; PrintMatrix(a);

 auto b = a;
 cout << "b    :\n"; PrintMatrix(b);

 MultiplyMatrix(a,10);
 cout << "a*10 :\n"; PrintMatrix(a);

 auto c = AddMatrix(a,b); // Variable auxiliar.
 cout << "c=a+b:\n"; PrintMatrix(c);
```

```
 cout << "a+b  :\n"; PrintMatrix(AddMatrix(a,b)); // Composición.
}

// Template functions definitions

template <typename T, size_t rows, size_t columns>
void PrintMatrix(const array<array<T, columns>, rows>& x){
 for(auto row : x){
  for(auto e : row)
   cout << e << '\t';
  cout << '\n';
 }
}

template <typename T, size_t rows, size_t columns>
void MultiplyMatrix(array<array<T, columns>, rows>& x, int k){
 for(auto& row : x)
  for(auto& e : row)
   e *= k;
}

template <typename T, size_t rows, size_t columns>
array<array<T, columns>, rows>
AddMatrix(
 const array<array<T, columns>, rows>& x,
 const array<array<T, columns>, rows>& y
){
 auto z = x;
 for(size_t i=0; i < rows; ++i)
  for(size_t j=0; j < columns; ++j)
   z.at(i).at(j) += y.at(i).at(j);
 return z;
}
```

Array output.

```
TestArrayOfArrayVariables
42
1 15 15
sizeof m                      : 60
sizeof(array<array<int,3>,5>): 60
1 2 3
4 5 6
7 8 9
10 11 12
```

```
13 14 15
0 1 2
10 11 12
20 21 22
30 31 32
40 41 42
0 10 20
100 110 120
200 210 220
300 310 320
400 410 420
TestNonGenericArrayOfArrayFunction
m:
0 1
2 3
4 5
TestGenericArrayOfArrayFunctions
a    :
11 12 13
21 22 23
31 32 33
41 42 43
51 52 53
b    :
11 12 13
21 22 23
31 32 33
41 42 43
51 52 53
a*10 :
110 120 130
210 220 230
310 320 330
410 420 430
510 520 530
c=a+b:
121 132 143
231 242 253
341 352 363
451 462 473
561 572 583
a+b  :
121 132 143
231 242 253
341 352 363
451 462 473
```

```
561 572 583
```

5.3. Matrix Template

MatrixTemplate.cpp.

```
/* Type template, Arrays-of-Arrays, matrixes, template functions,
 * range-for, size_t, sizeof, auto, intializazation, const, parameters
 (in-out-inout), type inference.
 * 201304-201608
 * José María Sola
 */

// Template type declaration

#include <array>    // array
#include <cstddef>  // size_t

template <typename T, std::size_t rows, std::size_t columns>
using Matrix =
 std::array<      // array
        std::array<  //   of arrays
            T,       //     of T
            columns  //     of columns elements
        >,
  rows          //   of rows elements
 >;

int main(){
 void TestMatrixVariables();
 TestMatrixVariables();

 void TestMatrixFunctions();
 TestMatrixFunctions();
}

#include <iostream> // cout
using std::cout;

void TestMatrixVariables(){
 cout << __func__ << '\n';

 Matrix<int, 5, 3> x;           // Declaración, sin incialización.
 x.at(3).at(2) = 42;            // Set/Write.
 cout << x.at(3).at(2) << '\n'; // Get/Read.
```

```
cout
 << x.at(0).at(0)                                        << '\t'  //
Muestra el primer elemento,
 << x.at(4).at(2)                                        << '\t'  // y el
último de forma particular
 << x.at(x.size()-1).at(x.at(x.size()- 1).size()- 1) << '\n'; // y
genéricamente.

Matrix<int, 5, 3>  m = {{      // Declaración e inicialización.
 {{ 1,  2,  3}},
 {{ 4,  5,  6}},
 {{ 7,  8,  9}},
 {{10, 11, 12}},
 {{13, 14, 15}}
}};
// Un array es un struct con un data member, por eso las dobles llaves.

// El tamaño de un arreglo es igual al tamaño del tipo de sus elementos
por la cantidad de elementos del arreglo.
cout
 << "sizeof m                  : " << sizeof m                    << '\n'
 << "sizeof(Matrix<int, 5, 3>): " << sizeof(Matrix<int, 5, 3> )
<< '\n';

// Iterar arreglo para get con for (for general).
for(size_t i=0; i < 5; ++i){ // size_t.
 for(size_t j=0; j < 3; ++j)
  cout << m.at(i).at(j) << '\t';
 cout << '\n';
}

// Iterar arreglo para set con for.
for(size_t i=0, n=m.size(); i < n; ++i) // n.
 for(size_t j=0, n=m.at(i).size(); j < n; ++j)
  m.at(i).at(j) = i*10+j;

// Iterar arreglo para get con range-for (for-auto).
for(auto row : m){ // auto.
 for(auto e : row)
  cout << e << '\t';
 cout << '\n';
}

// Iterar arreglo para set con range-for.
for(auto& row : m) // auto&
```

```
  for(auto& e : row)
   e *= 10;

 for(auto row : m){
  for(auto e : row)
   cout << e << '\t';
  cout << '\n';
 }
}

// Template functions declarations

template <typename T, std::size_t rows, std::size_t columns>
void PrintMatrix(const Matrix<T, rows, columns>& x); // in.

template <typename T, std::size_t rows, std::size_t columns>
void MultiplyMatrix(Matrix<T, rows, columns>& x, int k); // inout.

template <typename T, std::size_t rows, std::size_t columns>
Matrix<T, rows, columns> // return.
AddMatrix(
 const Matrix<T, rows, columns>& x,
 const Matrix<T, rows, columns>& y
);

void TestMatrixFunctions(){
 cout << __func__ << '\n';      // Imprime el nombre de la función.

 Matrix<int, 5, 3> a = {{
   {{11,12,13}},
   {{21,22,23}},
   {{31,32,33}},
   {{41,42,43}},
   {{51,52,53}}
  }};

 cout << "a    :\n"; PrintMatrix(a);

 auto b = a;
 cout << "b    :\n"; PrintMatrix(b);

 MultiplyMatrix(a,10);
 cout << "a*10 :\n"; PrintMatrix(a);

 auto c = AddMatrix(a,b); // Variable auxiliar.
 cout << "c=a+b:\n"; PrintMatrix(c);
```

```
 cout << "a+b  :\n"; PrintMatrix(AddMatrix(a,b)); // Composición.
}

// Template functions definitions

template <typename T, std::size_t rows, std::size_t columns>
void PrintMatrix(const Matrix<T, rows, columns>& x){
 for(auto row : x){
  for(auto e : row)
   std::cout << e << '\t';
  std::cout << '\n';
 }
}

template <typename T, std::size_t rows, std::size_t columns>
void MultiplyMatrix(Matrix<T, rows, columns>& x, int k){
 for(auto& row : x)
  for(auto& e : row)
   e *= k;
}

template <typename T, std::size_t rows, std::size_t columns>
Matrix<T, rows, columns>
AddMatrix(
 const Matrix<T, rows, columns>& x,
 const Matrix<T, rows, columns>& y
){
 auto z = x;
 for(size_t i=0; i < rows; ++i)
  for(size_t j=0; j < columns; ++j)
   z.at(i).at(j) += y.at(i).at(j);
 return z;
}
```

Matrix Template output.

```
TestMatrixVariables
42
0 16 16
sizeof m                 : 60
sizeof(Matrix<int, 5, 3>): 60
1 2 3
4 5 6
7 8 9
```

```
10 11 12
13 14 15
0 1 2
10 11 12
20 21 22
30 31 32
40 41 42
0 10 20
100 110 120
200 210 220
300 310 320
400 410 420
TestMatrixFunctions
a    :
11 12 13
21 22 23
31 32 33
41 42 43
51 52 53
b    :
11 12 13
21 22 23
31 32 33
41 42 43
51 52 53
a*10 :
110 120 130
210 220 230
310 320 330
410 420 430
510 520 530
c=a+b:
121 132 143
231 242 253
341 352 363
451 462 473
561 572 583
a+b  :
121 132 143
231 242 253
341 352 363
451 462 473
561 572 583
```

5.4. Matrix Template Module

MatrixTemplateModule.cpp.

```
/* Modularization, headers, include guards, type template,
 * Arrays-of-Arrays, matrixes, template functions,
 * range-for, size_t, sizeof, auto, intializazation, const, parameters
 (in-out-inout), type inference.
 * 201304-201608
 * José María Sola
 */

int main(){
 void TestMatrixVariables();
 TestMatrixVariables();

 void TestMatrixFunctions();
 TestMatrixFunctions();
}

#include "Matrix.h" // Matrix. Include interface, public part.

#include <iostream> // cout
using std::cout;

void TestMatrixVariables(){
 cout << __func__ << '\n';

 Matrix<int, 5, 3> x;           // Declaración, sin incialización.
 x.at(3).at(2) = 42;            // Set/Write.
 cout << x.at(3).at(2) << '\n'; // Get/Read.

 cout
  << x.at(0).at(0)                                    << '\t'  //
 Muestra el primer elemento,
  << x.at(4).at(2)                                    << '\t'  // y el
 último de forma particular
  << x.at(x.size()-1).at(x.at(x.size()- 1).size()- 1) << '\n'; // y
 genéricamente.

 Matrix<int, 5, 3>  m = {{      // Declaración e inicialización.
  {{ 1,  2,  3}},
  {{ 4,  5,  6}},
  {{ 7,  8,  9}},
  {{10, 11, 12}},
```

```
  {{13, 14, 15}}
 }};
 // Un array es un struct con un data member, por eso las dobles llaves.

 // El tamaño de un arreglo es igual al tamaño del tipo de sus elementos
 por la cantidad de elementos del arreglo.
 cout
  << "sizeof m                  : " << sizeof m                    << '\n'
  << "sizeof(Matrix<int, 5, 3>): " << sizeof(Matrix<int, 5, 3> )
 << '\n';

 // Iterar arreglo para get con for (for general).
 for(size_t i=0; i < 5; ++i){ // size_t.
  for(size_t j=0; j < 3; ++j)
   cout << m.at(i).at(j) << '\t';
  cout << '\n';
 }

 // Iterar arreglo para set con for.
 for(size_t i=0, n=m.size(); i < n; ++i) // n.
  for(size_t j=0, n=m.at(i).size(); j < n; ++j)
   m.at(i).at(j) = i*10+j;

 // Iterar arreglo para get con range-for (for-auto).
 for(auto row : m){ // auto.
  for(auto e : row)
   cout << e << '\t';
  cout << '\n';
 }

 // Iterar arreglo para set con range-for.
 for(auto& row : m) // auto&
  for(auto& e : row)
   e *= 10;

 for(auto row : m){
  for(auto e : row)
   cout << e << '\t';
  cout << '\n';
 }
}

void TestMatrixFunctions(){
 cout << __func__ << '\n';      // Imprime el nombre de la función.

 Matrix<int, 5, 3> a = {{
```

```
   {{11,12,13}},
   {{21,22,23}},
   {{31,32,33}},
   {{41,42,43}},
   {{51,52,53}}
  }};

 cout << "a    :\n"; PrintMatrix(a);

 auto b = a;
 cout << "b    :\n"; PrintMatrix(b);

 MultiplyMatrix(a,10);
 cout << "a*10 :\n"; PrintMatrix(a);

 auto c = AddMatrix(a,b); // Variable auxiliar.
 cout << "c=a+b:\n"; PrintMatrix(c);

 cout << "a+b  :\n"; PrintMatrix(AddMatrix(a,b)); // Composición.
}

#include "Matrix.cpp" // Include definitions, private part of the
 implementation. Because there's no separate compilation for templates,
 this strategy is used, copied from The C++ Programming Language, 4th
 Edition, Chapter 23.
```

Matrix Template Moudule output.

```
TestMatrixVariables
42
0 16 16
sizeof m                  : 60
sizeof(Matrix<int, 5, 3>): 60
1 2 3
4 5 6
7 8 9
10 11 12
13 14 15
0 1 2
10 11 12
20 21 22
30 31 32
40 41 42
0 10 20
100 110 120
```

```
200 210 220
300 310 320
400 410 420
TestMatrixFunctions
a    :
11 12 13
21 22 23
31 32 33
41 42 43
51 52 53
b    :
11 12 13
21 22 23
31 32 33
41 42 43
51 52 53
a*10 :
110 120 130
210 220 230
310 320 330
410 420 430
510 520 530
c=a+b:
121 132 143
231 242 253
341 352 363
451 462 473
561 572 583
a+b  :
121 132 143
231 242 253
341 352 363
451 462 473
561 572 583
```

Matrix.h.

```
/* Matrix.h
 * Interface. Module implemantation public part. Template declarations.
 * 201304-201608
 * José María Sola
 */

#ifndef MATRIX_H_INLCUDED
#define MATRIX_H_INLCUDED
```

```
#include <array>    // array
#include <cstddef>  // size_t

template <typename T, std::size_t rows, std::size_t columns>
using Matrix =
 std::array<      // array
        std::array<  //   of arrays
            T,       //     of T
            columns  //     of columns elements
        >,
  rows          //   of rows elements
 >;

template <typename T, std::size_t rows, std::size_t columns>
void PrintMatrix(const Matrix<T, rows, columns>& x); // in.

template <typename T, std::size_t rows, std::size_t columns>
void MultiplyMatrix(Matrix<T, rows, columns>& x, int k); // inout.

template <typename T, std::size_t rows, std::size_t columns>
Matrix<T, rows, columns> // return.
AddMatrix(
 const Matrix<T, rows, columns>& x,
 const Matrix<T, rows, columns>& y
);

#endif
```

Matrix.cpp.

```
/* Matrix.cpp
 * Module implemantation private part. Template function definitios.
 * 201304-201608
 * José María Sola
 */

#include "Matrix.h" // Include to check definitions match declarations.

#include <iostream> // cout

template <typename T, std::size_t rows, std::size_t columns>
void PrintMatrix(const Matrix<T, rows, columns>& x){
 for(auto row : x){
  for(auto e : row)
```

```
   std::cout << e << '\t';
  std::cout << '\n';
 }
}

template <typename T, std::size_t rows, std::size_t columns>
void MultiplyMatrix(Matrix<T, rows, columns>& x, int k){
 for(auto& row : x)
  for(auto& e : row)
   e *= k;
}

template <typename T, std::size_t rows, std::size_t columns>
Matrix<T, rows, columns>
AddMatrix(
 const Matrix<T, rows, columns>& x,
 const Matrix<T, rows, columns>& y
){
 auto z = x;
 for(size_t i=0; i < rows; ++i)
  for(size_t j=0; j < columns; ++j)
   z.at(i).at(j) += y.at(i).at(j);
 return z;
}
```