Metaprogramming II

Raul P. Pelaez

October 31, 2024

Contents

1 Class templates

Similarly to function templates, we can also give our user-defined types template parameters (type arguments). The syntax for it is very similar to function templates.
Example

Source code

```
template<typename T>
struct MyType{
  std::vector<T> data;
  T value;
  T get_value(){return value;}
};
int main(){
  MyType<int> mt{{1,2,3}, 10};
  MyType<double> mt2{{1.1,2.2,3.3}, 10.1};
  mt.get_value();
}
```

We created a class template called "MyType" that has a vector of type T, a value of type T, and a member function that returns the value. In this context, T has the mathematical meaning of "for all types T, there exists a class called MyType". Similarly as with function templates, the compiler will generate code for each "template instance" of MyType whenever it encounters a new one (like `MyType<int>` and `MyType<double>` int the example).

Advanced

Constrained template arguments
C++20 introduced the "concepts" library, with it we can constrain the types that can be used as template arguments.
For instance, we can have a templated function that only accepts floating point numbers (like `float`, `double`) but will not compile for integers.

Source code

```
#include <iostream>
#include <concepts>
template<std::floating_point E>
void foo(E e){
  std::cout<<e;
}
int main(){
  foo(1); // Error: constraints not satisfied
  foo(1.0); // OK
  foo(1.0f); // OK
}
```

We can also write our own concepts, for instance, we could write a concept that checks if a type has a member function called "size", or whether the type is a container (like `std::vector`, `std::list`, `std::array`).

1.1 Template specialization

We can specialize a template for a specific type, providing a different implementation for that type.
Example

Source code

```
template<typename T>
struct MyType{
  T value;
  void print(){std::cout<<value;}
};

template<>
struct MyType<int>{
  int value;
  void print(){std::cout<<"This is an integer:"<<value;}
};

int main(){
  MyType<int> mt{10};
  MyType<double> mt2{10.1};
  std::cout<<mt.get_value()<<std::endl;
  std::cout<<mt2.get_value()<<std::endl;
}
```

1.1.1 Partial specialization

We can also partially specialize a template, providing a different implementation for a subset of types.
Example

Source code

```
template<typename T, typename U>
struct ValueCast{
  T value;
  U get_value(){return static_cast<U>(value);}
};

template<typename T>
struct ValueCast<T, T>{
  T value;
  T get_value(){return value;}
};

int main(){
  ValueCast<int, double> mt{10}; // Uses the first template
  ValueCast<double, double> mt2{10.1}; // Uses the second template
  std::cout<<mt.get_value()<<std::endl; // Returns a double
  std::cout<<mt2.get_value()<<std::endl; // Returns a double
}
```

Its important to understand the subtlety here of why we need to specify `<double, double>` instead of just `<double>`. The partial specialization `template<typename T> struct ValueCast<T, T>` doesn't create a new template with one parameter. Instead, it creates a pattern for specializing the primary two-parameter template when both parameters are the same.
When we try to instantiate a template, like `ValueCast<double, double>`, the compiler will first look for the a primary template that matches the arguments. If we write `ValueCast<double>`, the compiler will not find one because it is defined with two parameters. If we write `ValueCast<double, double>`, the compiler will find the primary template and then look for a specialization that matches the arguments. In this case, it will find the partial specialization and use it.

1.2 Separated declaration/definition

We can still separate the declaration and definition of class templates, but we need to include the "template" keyword in the definition.
Example

Source code

```cpp
template<typename T>
class Vector{
  std::unique_ptr<T[]> data;
  size_t size;
public:

  Vector(size_t size);

  T& operator[](size_t i){
    return data[i];
  }

  size_t get_size() const;
};

template<typename T>
Vector<T>::Vector(size_t size): size(size), data(new T[size]){}

template<typename T>
size_t Vector<T>::get_size() const{
  return size;
}

int main(){
  Vector<int> v(10);
  v[0] = 10;
  std::cout<<v[0];
}
```

Advice

We are starting to get the tools to write incredibly complex code with an increasingly "leaner" syntax. We can express more and more things in less and less code. The subtleties of whats going on under the hood become occluded by higher and higher levels of abstraction.
When we started we needed to write all the essential operations (copy, move, destruction,...) manually, and without templates our types where not generic. Now we can write a class that can hold any type, and the compiler will generate most of the "annoying" code for us. Do not let this automation fool you, though, we still need to understand what is going on under the hood in order to write efficient and correct code, as well as being able to read and understand other people's code.

For instance, in the code above I am not writing the destructor of the class, which might look like a memory leak. I can do this because `std::unique_ptr` uses RAII and will take care of freeing the memory when the object goes out of scope.

2 Nested template arguments

Template arguments can appear nested, for instance, we can have a vector of lists of integers.

Source code

```cpp
#include <vector>
#include <list>
int main(){
  std::vector<std::list<int>> v{{1,2,3},{4,5,6}};
}
```

While vector is a contiguous container, list is a node-based container (it is the standard library implementation of a double-linked list).
Note that in the code above we are writing what appears to be the "input" operator, >>. This is not an error and the compiler understands that we are trying to close the template argument list of vector and list. Really old compilers (pre-C++11) used to have a problem with this, not being able to distinguish between the "input" operator and the closing of the template argument list. Back then a space was needed between the two closing angle brackets, like > >.

3 Default template arguments

We can also provide default arguments for template parameters, so that the user does not need to specify them.
Example

Source code

```cpp
template<typename T=int>
class MyType{
  T value;
public:
  void set_value(T value){this->value = value;}
  T get_value(){return value;}
};
int main(){
  // This line is an error without the default template argument
  MyType mt; // MyType<int>
  MyType<double> mt2; // MyType<double>
}
```

3.1 Type deduction

When we instantiate a template, the compiler will try to deduce the type of the template arguments from the arguments passed to the constructor. This is similar to what happens with function templates and makes it so that we can write things like:

Source code

```
#include <vector>
#include <array>
using namespace std;
template<typename T>
struct MyType{
  T value;
  vector<T> data;
};

int main(){
  vector v{"Hello", "World"}; // std::vector<const char*>
  vector v2{1,2,3}; // std::vector<int>
  array a{1,2,3}; // std::array<int, 3>
  MyType mt{10, {1,2,3}}; // MyType<int>
}
```

The last line works because of a combination of type deduction and compiler-defined constructors (the rule of 0). Type deduction uses the constructor arguments to deduce the template types. On the other hand the compiler is generating a constructor for us that takes a T and a vector<T>, which is enough to deduce the type of T.

4 Function objects

Funtion objects, or Functors, are classes that overload the `()` operator, making them callable like functions.
Example

Source code

```
struct MyFunctor{
  int operator()(int x, int y){
    return x+y;
  }
};
int main(){
  MyFunctor f;
  f(1,2); // 3
}
```

Functors can be used to pass functions as arguments to other functions, or to store functions in containers.
Many standard library functions take functors as arguments, like `std::transform`.
Example

Source code

```
#include <vector>
#include <algorithm>
struct LessThan{
  template<typename T>
  bool operator()(const T& a, const T& b){
    return a < b;
  }
};

LessThan lt;
std::vector<int> vi{3,1,2};
std::sort(vi.begin(), vi.end(), lt);
std::vector<double> vd{3.1,1.1,2.1};
std::sort(vd.begin(), vd.end(), lt);
```

The sort algorithm optionally takes a comparator function as a third argument. In this case, we are passing a functor that compares two elements of the same type and returns true if the first is less than the second, this will sort the vector in ascending order.

4.1 Lambda expressions

Lambda expressions are a way to define functions inline, without having to define a separate class or function. They are very similar to Functors, but with a more concise syntax.
The syntax for a lambda is:

Source code

```
[capture](arguments){body}
```

Example

Source code

```
#include <vector>
#include <algorithm>
int main(){
  auto sum = [](int a, int b){return a+b;};
  sum(1,2); // 3
}
```

We can reproduce the previous example with lambdas.
Example

Source code

```cpp
#include <vector>
#include <algorithm>
int main(){
  std::vector<int> vi{3,1,2};
  std::sort(vi.begin(), vi.end(), [](int a, int b){return a < b;});
  std::vector<double> vd{3.1,1.1,2.1};
  std::sort(vd.begin(), vd.end(), [](double a, double b){return a < b;});
}
```

4.1.1 Generic lambdas

We can make lambdas templated, making them able to accept any type, by using the `auto` keyword.
Example

Source code

```cpp
#include <vector>
#include <algorithm>
int main(){
  auto lt = [](auto a, auto b){return a < b;};
  std::vector<int> vi{3,1,2};
  std::sort(vi.begin(), vi.end(), lt);
  std::vector<double> vd{3.1,1.1,2.1};
  std::sort(vd.begin(), vd.end(), lt);
}
```

Here we are again reproducing the sort example with lambdas, but this time we are using a generic lambda that can accept any type.
This is another example of the vast ability we are gaining of expressing incredibly complex functionality in a very concise way.

4.1.2 Capturing

One useful feature of lambdas is that they can capture (access) variables from the enclosing scope. We can customize which variables are captured and how they are captured (either by value or by reference). A lambda can have `[&]` as its capture clause to capture all variables by reference, or `[=]` to capture all variables by value.
In the following example, we are going to fill a vector with random numbers and then count how many are less than a given number.
Example

Source code

```cpp
#include <vector>
#include <algorithm>
#include <random>
int main(){
  int x = 10;
  auto ltx = [=](auto a){return a < x;};
  std::vector<int> v(1000);
  std::default_random_engine rng(1234);
  std::generate(v.begin(), v.end(),
                [&](){return rng()%100;});
  auto count = std::count_if(v.begin(), v.end(),
                             ltx);
  std::cout<<count;
}
```

The first lambda, ltx, is capturing all the variables in the enclosing scope by value (in this case only x). This means that it can access the variable x, but it cannot modify it.
The second lambda (without a given name) is capturing the rng generator variable by reference, so it can modify it. It needs to be able to modify it because generating a random number modifies the state of the generator.

Advanced

`std::function`
The `std::function` class is a wrapper around any callable object, like a function, a functor, or a lambda. It is part of the C++ standard library and is very useful when we need to store functions in containers or pass them as arguments to other functions.
Example

Source code

```cpp
#include <functional>
#include <iostream>
int main(){
  std::function<int(int, int)> f = [](int a, int b){return a+b;};
  std::cout<<f(1,2);
}
```

5 Exercises

5.1 Pair

Goal

Create a class template called Pair that can hold two values of any types. Your class should pass the following tests:

Source code

```cpp
//tests/test_pair.cpp
#include "pair.h"
#include "gtest/gtest.h"
#include <string>
using namespace homework;
TEST(Pair, IntPair) {
  Pair<int, int> p{1, 2};
  ASSERT_EQ(p.first, 1);
  ASSERT_EQ(p.second, 2);
}
TEST(Pair, DoublePair) {
  Pair<double, double> p{1.1, 2.2};
  ASSERT_EQ(p.first, 1.1);
  ASSERT_EQ(p.second, 2.2);
}
TEST(Pair, IntStringPair) {
      Pair<int, std::string> p{1, "Hello"};
      ASSERT_EQ(p.first, 1);
      ASSERT_EQ(p.second, "Hello");
}
```

Learning goal

Learn about class templates

Milestone

Create a new header file called pair.h and implement the Pair class template. Your implementation should include:

- Two template parameters for the types of the first and second values.
- Two public member variables to hold the first and second values.

Here's a skeleton to get you started:

Source code

```cpp
// include/pair.h
#pragma once
namespace homework {
  /**
   * @brief A class template that holds two values of any type.
   * @tparam T1 The type of the first value.
   * @tparam T2 The type of the second value.
   */

  struct Pair {
    // Your public member variables here
  };
} // namespace homework
```

Make sure your implementation passes the tests above.

5.2 Transform

Goal

Create a function called transform that takes a container and a function as arguments and applies the function to each element in the container. The function should modify the container in place.
Your function should pass the following tests:

Source code

```cpp
  //tests/test_transform.cpp
#include "transform.h"
#include "gtest/gtest.h"
#include <algorithm>
#include <list>
#include <string>
#include <vector>
using namespace homework;
TEST(Transform, IntVector) {
  std::vector v{1, 2, 3};
  auto doubled = v;
  transform(doubled, [](int x) { return x * 2; });
  ASSERT_EQ(doubled.size(), 3);
  for (auto i : v)
    ASSERT_EQ(doubled[i], v[i] * 2);
}
TEST(Transform, StringVector) {
  std::vector<std::string> v{"Hello", "World"};
  auto reversed = v;
  auto reverser = [](std::string s) {
    std::reverse(s.begin(), s.end());
    return s;
  };
  transform(reversed, reverser);
  ASSERT_EQ(reversed.size(), 2);
  ASSERT_EQ(reversed[0], "olleH");
  ASSERT_EQ(reversed[1], "dlroW");
}
TEST(Transform, SquareList) {
  std::list l{1, 2, 3};
  auto squared = l;
  transform(squared, [](int x) { return x * x; });
  ASSERT_EQ(squared.size(), 3);
  std::list expected{1, 4, 9};
  ASSERT_EQ(squared, expected);
}
```

Please carefully go over these tests to understand what each of them is doing.

Learning goal

Learn about function objects, lambda expressions, and how to apply functions to containers using the standard library algorithms.

Milestone

Create a new header file called transform.h and implement the transform function. You can add your implementation here:

Source code

```
//include/transform.h
#pragma once
namespace homework {

  /**
   * @brief A function that transforms each element of a container in
   ↪  place.
   * @tparam Container The type of the container.
   * @tparam UnaryFunction The type of the function that transforms the
   ↪  elements.
   * @param container The container to transform.
   * @param function The function that transforms the elements.
   */
  // Your code here
} // namespace homework
```

Make sure your implementation passes the tests above. Do not use std::transform in your implementation.

Milestone

Add a new test to the transform function that uses a lambda expression to transform a vector of doubles by adding 1 to each element. The test should start like this:

Source code

```
TEST(Transform, DoubleVectorSum) {
  std::vector v{1.1, 2.2, 3.3};
  auto incremented = v;
  // Your code here
}
```

5.3 Is it a raw pointer?

Goal

Create a class template called IsRawPointer that determines whether a type is a raw pointer or not. Your class should pass the following tests:

Source code

```
#include "is_raw_pointer.h"
#include "gtest/gtest.h"
using namespace homework;
TEST(IsRawPointer, Int) {
  ASSERT_FALSE(IsRawPointer<int>::value);
}
TEST(IsRawPointer, IntPointer) {
  ASSERT_TRUE(IsRawPointer<int*>::value);
}
TEST(IsRawPointer, IntConstPointer) {
  ASSERT_TRUE(IsRawPointer<const int*>::value);
}
TEST(IsRawPointer, IntReference) {
  ASSERT_FALSE(IsRawPointer<int&>::value);
}
TEST(IsRawPointer, String){
  ASSERT_FALSE(IsRawPointer<std::string>::value);
}
```

hint

Your class should hold a `static constexpr bool` value that is true if the type is a raw pointer and false otherwise.
You will need to use template specialization to implement this functionality.

Learning goal

Learn how to use class template specialization.

Advanced milestone

Add this test and implement the functionality for it to pass.

Source code

```
TEST(IsRawPointerv, Doubles){
  ASSERT_FALSE(IsRawPointer_v<double>);
  ASSERT_TRUE(IsRawPointer_v<double*>);
}
```

hint

You will need to use a template variable to implement this functionality.
The variable must be decorated with `constexpr`.

5.4 The flexible container

Goal

Create a class template called FlexibleContainer that can hold elements of any type and provides basic container functionality. Your class should pass the following tests:

Source code

```
  //tests/test_flexible_container.cpp
#include "flexible_container.h"
#include "gtest/gtest.h"
#include <string>
using namespace homework;
TEST(FlexibleContainer, IntStorage) {
  FlexibleContainer<int> container;
  container.add(5);
  container.add(10);
  ASSERT_EQ(container.size(), 2);
  ASSERT_EQ(container.get(0), 5);
  ASSERT_EQ(container.get(1), 10);
}
TEST(FlexibleContainer, StringStorage) {
  FlexibleContainer<std::string> container;
  container.add("Hello");
  container.add("World");
  ASSERT_EQ(container.size(), 2);
  ASSERT_EQ(container.get(0), "Hello");
  ASSERT_EQ(container.get(1), "World");
}
TEST(FlexibleContainer, DefaultType) {
  FlexibleContainer container;
  container.add(42);
  ASSERT_EQ(container.size(), 1);
  ASSERT_EQ(container.get(0), 42);
}
```

Learning goal

Learn how to create a class template with a default type parameter and implement basic container functionality.

Advice

This is a shallow module
This class is a really bad example of class design. It is a shallow module, with an interface that has more cognitive load than the implementation itself. We should just use `std::vector` instead, or maybe define an alias for it like:

Source code

```
template<typename T>
using FlexibleContainer = std::vector<T>;
```

if we think we might need to change the implementation in the future.
Understand this exercise as a way to learn about class templates and not as a good design practice.

Milestone

Create a new header file called flexible_container.h and implement the FlexibleContainer class template. Your implementation should include:

- A template parameter with a default type of int.
- A method to add elements to the container.
- A method to get elements from the container by index.
- A method to return the current size of the container.

Here's a skeleton to get you started:

Source code

```cpp
// include/flexible_container.h
#pragma once
#include <vector>
namespace homework {
  /**
   * @brief A flexible container that can hold elements of any type.
   * @tparam T The type of the elements in the container.
   */

  class FlexibleContainer {
    // Your implementation

  };
} // namespace homework
```

Make sure your implementation passes the tests above.
Implement some method outside the class definition to get familiar with separate declaration/definition.

hint

Consider using std::vector as your internal storage.

Milestone

Implement a map method for FlexibleContainer that applies a given function to all elements in the container. The map method should modify its elements in place. Add the following test:

Source code

```
TEST(FlexibleContainer, MapFunction) {
  FlexibleContainer<int> container;
  container.add(1);
  container.add(2);
  container.add(3);
  auto doubled = container;
  doubled.map([](int x) { return x * 2; });
  ASSERT_EQ(doubled.size(), 3);
  ASSERT_EQ(doubled.get(0), 2);
  ASSERT_EQ(doubled.get(1), 4);
  ASSERT_EQ(doubled.get(2), 6);
}
```

hint

You'll need to use a function template for the map method to handle different types of functions.
Consider using std::function or a template parameter for the function type.

Advanced milestone

Implement a variadic template constructor for FlexibleContainer that allows initialization with any number of elements. Add the following test:

Source code

```
TEST(FlexibleContainer, VariadicConstructor) {
  FlexibleContainer<int> container(1, 2, 3, 4, 5);
  ASSERT_EQ(container.size(), 5);
  ASSERT_EQ(container.get(0), 1);
  ASSERT_EQ(container.get(4), 5);
  FlexibleContainer<std::string> strContainer("Hello", "World", "C++");
  ASSERT_EQ(strContainer.size(), 3);
  ASSERT_EQ(strContainer.get(0), "Hello");
  ASSERT_EQ(strContainer.get(2), "C++");
}
```

hint

Use parameter pack expansion to implement the variadic constructor.
You may need to use std::initializer_list or recursion to handle the variable number of arguments.