Metaprogramming

Raul P. Pelaez

October 31, 2024

Contents

1 Generic programming: Templates

Someone using a vector is unlikely to always want a vector of `int`. They might want a vector of `double`, `std::string`, or even a vector of their own custom type.
On the other hand, it would be really inconvenient to define many classes, virtually identical, that only differ in the type of the data they store, like:

Source code

```
// A really inconvenient way of defining vector of different types
class vector_int{
  int* data;
  // ...
};
class vector_double{
  double* data;
  // ...
};
//...
```

Luckily, C++ allows us to parametrize types by making our classes and functions templates.

Info

Template
Basically, a template is a mechanism that allows to use types as parameters for a class or a function. The container `std::vector` is an example of a template class, which we can instantiate with different types:

Source code

```
std::vector<int> v1;
std::vector<double> v2;
std::vector<std::string> v3;
```

While arguments to functions are passed in between parentheses, arguments to templates (type parameters) are passed in between angle brackets, after the name of the type or function to be called.

We have been using templates from day one, whenever we specialized a type with angle brackets (like in `std::vector<int>`). But thus far we have not had to define our own templates. Let's see how to do it.
The notation to define a type parameter T is the `template <typename T>` prefix, meaning "for all types T" (just like in math).

2 Function templates

A templated function is a function that can operate on many different types. Templated functions are also referred to as "metafunctions". Technically, they are not functions, but rather a "template" for a function. In fact, the compiler will only produce actual code for the function when it is called with a specific type (when it is "instantiated"). Up until seeing the actual type, the compiler will not have enough information to generate the final function.
For instance, lets define a function that finds the max of two values of the same type:

Source code

```
template <typename T>
T max(T a, T b){
  return a>b?a:b;
}
```

This is a function that can be called with any type that supports the > operator. For instance, we can call it with ints, doubles, or even strings:

Source code

```
auto max_int = max(1,2); // 2
auto max_double = max(1.1,2.2); // 2.2
auto max_string = max("hello","world"); // "world"
```

Advice

The ternary operator `a>b?a:b` is a shorthand for an if-else statement. It returns the first value if the condition is true, and the second value otherwise.

Note that the number of parameter types is unconstrained, we can have as many as we want:

Source code

```
template <typename T, typename U>
T max(T a, U b){
  return a>b?a:b;
}
// Compiles as long as types are comparable
auto max_int_double = max(1,2.2);
```

2.1 Template deduction

Note that in the previous example we did not have to explicitly specify the template parameter. We could have written `max<int>(1,2)` but the compiler is able to deduce the type from the arguments. This is called "template argument deduction".

Advice

Whenever possible, the compiler will deduce the type of the template parameter. This is similar to how the `auto` keyword works.

You can see deduction in action by giving the compiler an ambiguous situation:

Source code

```
template <typename T>
T max(T a, T b){
  return a>b?a:b;
}
auto max_int = max(1,2.2); // Error, cannot deduce type
```

Which can be fixed by explicitly specifying the type:

Source code

```
auto max_int = max<double>(1,2.2); // 2.2
```

This version works because we are explicitly asking for the function to be instantiated with the type `double`, and there is an implicit conversion from `int` to `double`.

2.2 Template specialization

Sometimes we want to provide a different implementation for a specific type. For instance, we might want to provide a different implementation for the `max` function when the type is a string. We can do this by specializing the template:

Source code

```
template <>
std::string max(std::string a, std::string b){
  return a.size()>b.size()?a:b;
}
```

3 Non type template parameters

In addition to types, we can also pass values as template parameters. For instance, we can define a function that returns an array filled with a given value:

Source code

```
template <typename T, size_t N>
auto create_filled_array(T value){
  std::array<T,N> res;
  for(auto &v: res){
    v = value;
  }
  return res;
}
auto arr = create_filled_array<int,5>(42); // {42,42,42,42,42}
```

The key here is the second template parameter, `size_t N`. This is a non-type template parameter, which means that it is a value that is known at compile time.

4 Aliases

Sometimes we want to rename or somewhat abstract away a particular type. For instance, we might need to use a container at the start of our development, but predicting that we might need to change it later. We can use an alias to make this change easier:

Source code

```
template <typename T>
using Container = std::vector<T>;
Container<int> vint;
Container<double> vdouble;
```

Aliases can be templated as well, and they can be used to define functions, classes, and even namespaces.

Advice

Remember the `DynamicArray` exercise, there is a trivial solution for it now that you know aliases:

Source code

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

5 Variadic templates

We saw about the type `std::tuple` long ago, which could contain an arbitrary number of types:

Source code

```
std::tuple<int,double,std::string> t{1,2.2,"hello"};
```

Tuple can do this because it is a variadic template. A variadic template is a template that can take an arbitrary number of arguments. We can define our own variadic templates by using the `...` syntax:

Source code

```
template <typename... Args>
void print_all(Args... args){
  (std::cout << ... << args) << std::endl;
}
print_all(1,2.2,"hello"); // 1 2.2 hello
print_all("The time is: ", std::chrono::system_clock::now());
```

Variadic templates are used in many places in the standard library. For instance, the `std::make_tuple` function is a variadic template that creates a tuple with the arguments passed to it.
The syntax for variadic templates is a bit weird, but it is very powerful. Inside variadic templates, we can use a couple mechanisms to manipulate the arguments:

Advice

Fold expressions
The `(std::cout << ... << args)` is a fold expression, which applies the operator to all the arguments in the pack. For the particular usage of `std::cout << ... << args`, it is equivalent to `std::cout << args1 << args2 << args3`.
Example

Source code

```
// A function that sums all the arguments
template <typename... Args>
auto sum_all(Args... args){
  return (... + args); // Fold expression
}
auto sum = sum_all(1,2,3,4); // 10
```

Advice

Parameter pack expansion
The `Args... args` is a parameter pack, which is a way of capturing an arbitrary number of arguments. We can expand the pack by using the `...` syntax after the pack name. For instance, `sum_all(args...)` would expand the pack and call the function with all the arguments separated by commas.
Example

Source code

```
template <typename... Args>
auto call_sum_all(Args... args){
   return sum_all(args...); // Expand the pack
   // Equivalent to print_all(args1,args2,args3);
}
call_sum_all(1,2,3);
```

6 Exercises

I provide you with Gtest tests and some skeleton for a header file. Add these files to your CMake project and implement the function in the header file. Make sure the tests pass.

6.1 The ultimate summing

Goal

Write a single function, called `sum_elements` that sums the contents of a vector, regardless of the type of the elements. The function should work with any type that supports the + operator.

Your function should pass the following tests

Source code

```
//tests/test_sum.cpp
#include "gtest/gtest.h"
#include  "sum.h"
#include <vector>
#include <numeric>
using namespace homework;
TEST(SumVector, Int){
  std::vector<int> vint = {1,2,3,4,5};
  auto sum_ints = sum_elements(vint);
  auto sum_ints_ref = std::accumulate(vint.begin(),vint.end(),0);
  ASSERT_EQ(sum_ints,sum_ints_ref);
}
TEST(SumVector, Double){
  std::vector<double> vdouble = {1.1,2.2,3.3,4.4,5.5};
  auto sum_doubles = sum_elements(vdouble, 0.0);
  auto sum_doubles_ref =
  ↪  std::accumulate(vdouble.begin(),vdouble.end(),0.0);
  ASSERT_NEAR(sum_doubles,sum_doubles_ref,1e-10);
}
TEST(SumVector, String){
  std::vector<std::string> vstring = {"hello","world"};
  auto sum_string = sum_elements(vstring, std::string());
  ASSERT_EQ(sum_strings,"helloworld");
}
```

Learning goal

Learn about basic template syntax and how to define a function template.

Milestone

Write a new header called sum.h and put there your implementation. For templates, it is much more convenient to not have separate definition and declarations, so just define your function in the header file.

Source code

```
// include/sum.h
#pragma once
#include <vector>
namespace homework{
  /**
   * @brief Metafunction for summing the elements of a vector holding
   ↪  elements of any type
   * @param s The vector to sum
   * @param v The initial value of the sum
   * @tparam T The type of the elements in the vector
   * @return The sum of the elements in the vector, plus the initial value
   */
   // Define here your function
} // namespace homework
```

Make sure your function passes the tests above.

hint

- The `@tparam` doxygen tag is used to document template parameters.

Milestone

Specialize your template so that it prints the elements of the vector if the type is `std::string`.

Advanced milestone

Add the following tests, modify your function to pass them:

Source code

```
#include <list>
TEST(SumSequence, List){
  std::list<int> lint = {1,2,3,4,5};
  auto sum_ints = sum_elements(lint, 0);
  auto sum_ints_ref = std::accumulate(lint.begin(),lint.end(),0);
  ASSERT_EQ(sum_ints,sum_ints_ref);
}
#include <array>
TEST(SumSequence, Array){
  std::array<int,5> aint = {1,2,3,4,5};
  auto sum_ints = sum_elements(aint, 0);
  auto sum_ints_ref = std::accumulate(aint.begin(),aint.end(),0);
  ASSERT_EQ(sum_ints,sum_ints_ref);
}
```

hint

- The `std::list` and `std::array` are all containers with the same interface as `std::vector`. Your current code should work as is, except for its signature.
- You no longer need to include vector in the header file.

Advanced milestone

Add the following tests, modify it (just the test code) so that it compiles:

Source code

```
TEST(SumVector, CustomType){
  struct MyType{
    int i;
    double d;
  };
  std::vector<MyType> vmt = {{1,1.1},{2,2.2},{3,3.3}};
  auto sum_mts = sum_elements(vmt, MyType{0,0.0});
  ASSERT_EQ(sum_mts,(MyType{6,6.6}));
}
```

Advanced milestone

Why do we have to explicitly say `std::string()` in the last test instead of just `""`?

hint

What is the type of `""`?

Advanced milestone

Implement the function `my_make_tuple` that takes an arbitrary number of arguments and returns a tuple with those arguments. Your function should pass this test:

Source code

```
#include <tuple>
TEST(Tuple, make){
  auto t = my_make_tuple(1,2.2,"hello");
  ASSERT_EQ(std::get<0>(t),1);
  ASSERT_NEAR(std::get<1>(t),2.2,1e-10);
  ASSERT_EQ(std::get<2>(t),"hello");
}
TEST(Tuple, make2){
  auto t = my_make_tuple("word", std::vector<int>{1,2,3});
  ASSERT_EQ(std::get<0>(t), "word");
  ASSERT_EQ(std::get<1>(t), (std::vector<int>{1,2,3}));
}
```

You actually need to write the vector inside a parentheses in the ASSERT statement, can you figure out why?