- Examples

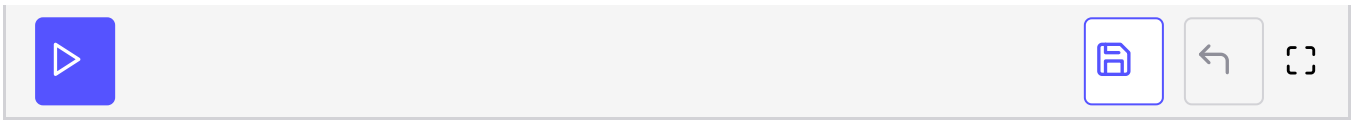
Let's look at the examples of variadic templates.

WE'LL COVER THE FOLLOWING

Example 1: Variadic Template

```
// templateVariadicTemplates.cpp

#include <iostream>

template <typename... Args>
int printSize(Args... args){
  return sizeof ...(args);
}

template<int ...>
struct Mult;

template<>
struct Mult<>{
  static const int value= 1;
};

template<int i, int ... tail>
struct Mult<i, tail ...>{
  static const int value= i * Mult<tail ...>::value;
};

int main(){

  std::cout << std::endl;

  std::cout << "printSize(): " << printSize() << std::endl;
  std::cout << "printSize(template,2011,true): " << printSize("template",2011,true) << std::e
  std::cout << "printSize(1, 2.5, 4, 5, 10): " <<  printSize(1, 2.5, 4, 5, 10) << std::endl;

  std::cout << std::endl;
```

```
  std::cout << "Mult<10>::value: " << Mult<10>::value << std::endl;
  std::cout << "Mult<10,10,10>::value: " << Mult<10,10,10>::value << std::endl;

  std::cout << "Mult<1,2,3,4,5>::value: " << Mult<1,2,3,4,5>::value << std::endl;

  std::cout << std::endl;

}
```



Explanation

In the above example, we have used `printSize` function, which prints the number of elements (of any type) passed as arguments. It detects the number of elements on compile-time using the `sizeof` operator, and in case of an empty argument list, it returns 0.

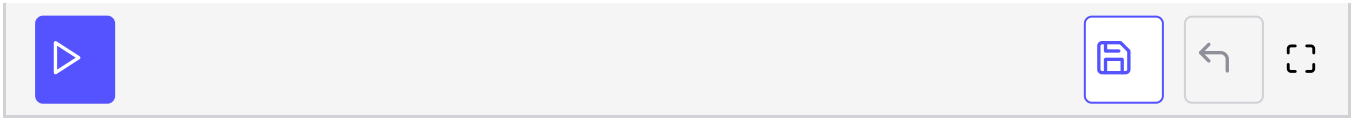
There is a `struct` defined as `Mult` which takes arguments of integer type and return their product. If there is no argument passed, then it returns 1 which is the neutral element for multiplication. The result is stored in the `value` in the fully specialized template in lines 13 – 16. The partial specialization in lines 18 – 21 starts the recursion, which ends with the aforementioned fully specialization for 0. The primary template in line 10 is never used and must, therefore, never be defined.

To visualize the template instantiation for the above-mentioned example click here.

Example 2: Template Perfect Forwarding

```
// templatePerfectForwarding.cpp

#include <iostream>
#include <utility>

template<typename T, typename ... Args>
T createT(Args&& ... args){
  return T(std::forward<Args>(args) ...);
}

struct MyStruct{
  MyStruct(int&, double&, double&&){}
  friend std::ostream& operator<< (std::ostream& out, const MyStruct&){
      out << "MyStruct" << std::endl;
      return out;
  }
```

```
};

int main(){


  std::cout << std::endl;

  double myDouble= createT<double>();
  std::cout << "myDouble: " << myDouble << std::endl;

  int myInt= createT<int>(1);
  std::cout << "myInt: " << myInt << std::endl;

  std::string myString= createT<std::string>("My String");
  std::cout << "myString: "  << myString << std::endl;

  MyStruct myStruct= createT<MyStruct>(myInt, myDouble, 3.14);
  std::cout << "myStruct: "  << myStruct << std::endl;

  std::cout << std::endl;
}
```



Explanation

In the above example, we have created a `createT` function which invokes the constructor `T` with the arguments `args`. If there is no value passed, it invokes the default constructor. The magic of the factory function `createT` is that it can invoke each constructor. Thanks to perfect forwarding, each value can be used such as an lvalue or an rvalue; thanks to parameter packs, any number of arguments can be used. In the case of `MyStruct`, a constructor that requires three arguments is used.

The pattern of the function template `createT` is exactly the pattern, factory functions such as `std::make_unique`, `std::make_shared`, `std::make_pair`, or `std::make_tuple` use.

In the next lesson, we'll solve exercises related to variadic templates.