- Examples

In this lesson, we'll learn about some examples of template parameters.

WE'LL COVER THE FOLLOWING

Example 1: Type Parameter

```
// templateTypeParameter.cpp

#include <iostream>
#include <typeinfo>

class Account{
public:
  explicit Account(double amt): balance(amt){}
private:
  double balance;

};

union WithString{
  std::string s;
  int i;
  WithString():s("hello"){}
  ~WithString(){}
};

template <typename T>
class ClassTemplate{
public:
  ClassTemplate(){
    std::cout << "typeid(T).name(): "  << typeid(T).name() << std::endl;
  }
};

int main(){

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

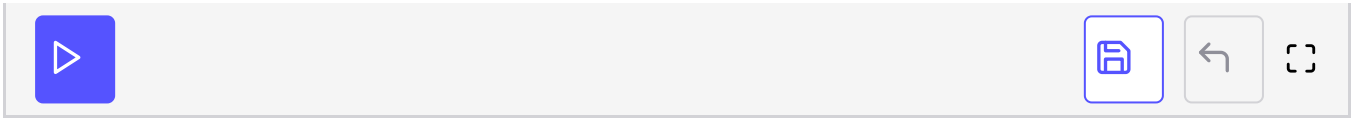
```
  ClassTemplate<int> clTempInt;
  ClassTemplate<double> clTempDouble;

  ClassTemplate<std::string> clTempString;

  ClassTemplate<Account> clTempAccount;
  ClassTemplate<WithString> clTempWithString;

  std::cout << std::endl;

}
```



Explanation

In the code above, we are identifying the type of different data types that we have passed in the parameter list. We can identify the type of variable passed to the function by using the keyword `typeid` in line 25. If we pass `string` or `class` type objects in the parameter list, it will display the type of parameter passed along with the size of the object.

Example 2: Template-Template Parameter

```
// templateTemplateTemplatesParameter.cpp

#include <initializer_list>
#include <iostream>
#include <list>
#include <vector>

template <typename T, template <typename, typename> class Cont >
class Matrix{
public:
  explicit Matrix(std::initializer_list<T> inList): data(inList){
    for (auto d: data) std::cout << d << " ";
  }
  int getSize() const{
    return data.size();
  }

private:
  Cont<T, std::allocator<T>> data;

};

int main(){

  std::cout << std::endl;

  Matrix<int,std::vector> myIntVec{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
  std::cout << std::endl;
  std::cout << "myIntVec.getSize(): " << myIntVec.getSize() << std::endl;
```

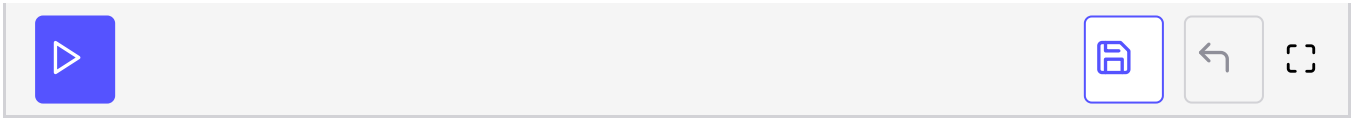
```
  std::cout << std::endl;

  Matrix<double,std::vector> myDoubleVec{1.1, 2.2, 3.3, 4.4, 5.5};
  std::cout << std::endl;
  std::cout << "myDoubleVec.getSize(): "  << myDoubleVec.getSize() << std::endl;

  std::cout << std::endl;

  Matrix<std::string,std::list> myStringList{"one", "two", "three", "four"};
  std::cout << std::endl;
  std::cout << "myStringList.getSize(): " << myStringList.getSize() << std::endl;

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



Explanation

We have declared a `Matrix` class which contains a function, `getSize`, and an explicit constructor that prints all entries of the passed parameter. `Cont` in line 8 is a template, which takes two arguments. There's no need for us to name the template parameters in the template declaration. We have to specify them in the instantiation of the template (line 19). The template used in the template parameter has exactly the signature of the sequence containers. The result is that we can instantiate a matrix with an `std::vector` or an `std::list`. Of course `std::deque` and `std::forward_list` would also be possible. In the end, we have a `Matrix`, which stores its elements in a vector or list.

For more examples of template-template parameters, check out container adaptors.

We'll be solving a small exercise on template parameters in the next lesson.