- Examples

In this lesson, we will look at some examples of template parameters.

WE'LL COVER THE FOLLOWING

Example 1

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

Example 2

```
// templateNotTypeParameter.cpp

#include <cstddef>
#include <iostream>
#include <typeinfo>

template <char c>
class AcceptChar{
public:
  AcceptChar(){
    std::cout << "AcceptChar: "  << typeid(c).name() << std::endl;
  }
};

template < int(*func)(int) >
class AcceptFunction{
public:
  AcceptFunction(){
    std::cout << "AcceptFunction: "  << typeid(func).name() << std::endl;
  }
};
```

```
template < int(&arr)[5] >
class AcceptReference{

public:
  AcceptReference(){
    std::cout << "AcceptReference: " << typeid(arr).name() << std::endl;
  }
};

template < std::nullptr_t N >
class AcceptNullptr{
public:
  AcceptNullptr(){
    std::cout << "AcceptNullpt: " << typeid(N).name() << std::endl;
  }
};

int myFunc(int){ return 2011; };
int arr[5];

int main(){

  std::cout << std::endl;

  AcceptChar<'c'> acceptChar;
  AcceptFunction< myFunc> acceptFunction;
  AcceptReference< arr > acceptReference;
  AcceptNullptr< nullptr > acceptNullptr;

  std::cout << std::endl;

}
```

Explanation

- We have created four different class templates, including `AcceptChar`, `AcceptFunction`, `AcceptReference`, and `AcceptNull` in lines 8, 16, 24, and 32.
- Each class template accepts a different non-type. To verify all types, we declared a character variable, a reference to an array, a function, and `nullptr` in lines (46 – 49). We identify their type using the keyword `typeid` in lines 11, 19, 27, and 35.

Example 3

```
// templateTemplateTemplatesParameter.cpp

#include <initializer_list>
```

```
#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 declared a `Matrix` class which contains a function, such as, `getSize()`. We get an explicit constructor that prints all entries of the passed parameter.

- `Cont` in line 8 is a template that takes two arguments. There is no need for us to name the template parameters in the template declaration. We must specify them in the instantiation of the template (line 19).

- The template used in the template parameter has the signature of the sequence containers, meaning that we can instantiate a matrix with an `std::vector`, or an `std::list`.
- `std:deque` and `std::forward_list` are also possible. In the end, you have a Matrix that stores all its elements in a vector or a list.

In the next lesson, we will go one step further with template parameters and learn about an important tool in modern C++: variadic templates.