- Examples

In this lesson, we'll learn about the 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 above code, 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 object in the parameter list, it will display the type of parameter passed along with the size of the object.

Example 2: Non-Type Template Parameter

```
// array.cpp

#include <algorithm>
#include <array>
#include <iostream>

int main(){

  std::cout << std::endl;

  // output the array
  std::array <int,8> array1{1,2,3,4,5,6,7,8};
  std::for_each( array1.begin(),array1.end(),[](int v){std::cout << v << " ";});

  std::cout << std::endl;

  // calculate the sum of the array by using a global variable
  int sum = 0;
  std::for_each(array1.begin(), array1.end(),[&sum](int v) { sum += v; });
  std::cout << "sum of array{1,2,3,4,5,6,7,8}: " << sum << std::endl;

  // change each array element to the second power
  std::for_each(array1.begin(), array1.end(),[](int& v) { v=v*v; });
  std::for_each( array1.begin(),array1.end(),[](int v){std::cout << v << " ";});
  std::cout << std::endl;
  std::cout << std::endl;
}
```

Explanation

When you define an `std::array` in line 12, you have to specify its size. The size is a non-type template argument, which has to be specified at compile-time.

Therefore, you can output `array1` in line 13 with a lambda-function `[]` and the range-based for-loop. By using the summation variable `sum` in line 19, you can sum up the elements of the `std::array`. The lambda-function in line 23 takes its arguments by reference and can, therefore, map each element to its square. There is nothing really special, but we are dealing with an `std::array`.

With C++11 we have the free function templates `std::begin` and `std::end` returning iterators for a C array. A `C` array is quite comfortable and safe to use with these function templates because we don't have to remember its size.

Example 3: 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, i.e., `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, you have a `Matrix`, which stores its elements in a vector or in a list.

If you want to study more examples for template-template parameter, you can check container adaptors.

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