- Examples

In this lesson, we will look at a few examples of template arguments.

WE'LL COVER THE FOLLOWING

Example 1

```cpp
// templateArgumentDeduction.cpp

#include <iostream>

template <typename T>
bool isSmaller(T fir, T sec){
  return fir < sec;
}

template <typename T, typename U>
bool isSmaller2(T fir, U sec){
  return fir < sec;
}

template <typename R, typename T, typename U>
R add(T fir, U sec){
  return fir + sec;
}

int main(){

  std::cout << std::boolalpha << std::endl;

  std::cout << "isSmaller(1,2): " << isSmaller(1,2) << std::endl;
  // std::cout << "isSmaller(1,5LL): "  << isSmaller(1,5LL) << std::endl; // ERROR

  std::cout << "isSmaller<int>(1,5LL): " << isSmaller<int>(1,5LL) << std::endl;
  std::cout << "isSmaller<double>(1,5LL): " << isSmaller<double>(1,5LL) << std::endl;

  std::cout << std::endl;

  std::cout << "isSmaller2(1,5LL): "  << isSmaller2(1,5LL) << std::endl;

  std::cout << std::endl;

  std::cout << "add<long long int>(1000000,1000000): " << add<long long int>(1000000, 1000000
  std::cout << "add<double,double>(1000000,1000000): " << add<double,double>(1000000, 1000000
  std::cout << "add<double,double,float>(1000000,1000000): " << add<double,double,float>(1000

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

Explanation

In the above example, we defined 3 function templates.

- `isSmaller` takes two arguments, which must have the same type. The template returns true if the first element is less than the second element (line 6). Invoking the function with arguments of a different type would give a compile-time error (line 25).
- `isSmaller2` takes two arguments, which can have different types. The template returns true if the first element is less than the second element(line 11).
- `add` takes two arguments, which have different types (line 16). The return type must be specified because it cannot be deduced from the function arguments.

Example 2

```
// templateAutomaticReturnType.cpp

#include <iostream>
#include <typeinfo>

template<typename T1, typename T2>
auto add(T1 first, T2 second) -> decltype(first + second){
    return first + second;
}

int main(){
  std::cout << std::endl;

  std::cout << "add(1, 1)= " << add(1, 1) << std::endl;
  std::cout << "typeid(add(1, 1)).name()= " << typeid(add(1, 1)).name() << std::endl;

  std::cout << std::endl;

  std::cout << "add(1, 2.1)= " << add(1, 2.1) << std::endl;
  std::cout << "typeid(add(1, 2.1)).name()= " << typeid(add(1, 2.1)).name() << std::endl;

  std::cout << std::endl;

  std::cout << "add(1000LL, 5)= " << add(1000LL, 5) << std::endl;
  std::cout << "typeid(add(1000LL, 5)).name()= " << typeid(add(1000LL, 5)).name() << std::enc

  std::cout << std::endl;

}
```

Explanation

- The function template `add` takes two type parameters and deduces the return type. The lines 16, 21, and 26 shows a string representation n of the types, and the compiler deduces with the help of `decltype`.
- Line 16 is an `int`, line 21 is a `double`. Line 26 is a `long long int`.
- The GCC displaces a `long long int` as `x`.

Example 3

```
// templateDefaultArgument.cpp

#include <functional>
#include <iostream>
#include <string>

class Account{
public:
  explicit Account(double b): balance(b){}
  double getBalance() const {
    return balance;
  }
private:
  double balance;
};

template <typename T, typename Pred= std::less<T> >
bool isSmaller(T fir, T sec, Pred pred= Pred() ){
  return pred(fir,sec);
```

```
}

int main(){


  std::cout << std::boolalpha << std::endl;

  std::cout << "isSmaller(3,4): " << isSmaller(3,4) << std::endl;
  std::cout << "isSmaller(2.14,3.14): "  << isSmaller(2.14,3.14) << std::endl;
  std::cout << "isSmaller(std::string(abc),std::string(def)): " << isSmaller(std::string("abc

  bool resAcc= isSmaller(Account(100.0),Account(200.0),[](const Account& fir, const Account&
  std::cout << "isSmaller(Account(100.0),Account(200.0)): " << resAcc << std::endl;

  bool acc= isSmaller(std::string("3.14"),std::string("2.14"),[](const std::string& fir, cons
  std::cout << "isSmaller(std::string(3.14),std::string(2.14)): " << acc << std::endl;

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

Explanation

- In Example 1, we passed only the built-in data types. In this example, we have used the built-in types `int` and `double`, the `std::string`, and an `Account` class in lines (26 - 28).

- The function template `isSmaller` is parametrized by a second template parameter, which defines the comparison criterion. The default for the comparison is the predefined function object `std::less`.

- A function object is a class for which the call operator (`operator ()`) is overloaded. This means that instances of function objects behave similarly as a function.

- The `Account` class does not support the `<` operator.

- Due to the second template parameter, a lambda expression, such as in lines 30 and 33, can be used. This means `Account` can be compared by their balance, and their strings can be compared by their number. `stod` converts a string to a double.

In the next lesson, we will learn about template specialization.