- Examples

Let's check out the examples of template arguments.

WE'LL COVER THE FOLLOWING

Example 1: Deduction of Template Arguments

```
// 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(){
```

```
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 have defined 3 function templates

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

Example 2: Template Default Arguments

```
// 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 the first example, we have passed only the built-in data types. In this example, we have used the built-in types `int`, `double`, `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 doesn't support the `<` operator. Thanks to the second template parameter, a lambda expression like in lines 30 and 33 can be used. This means `Account` can be compared by their balance and strings by their number. `stod` converts a string to a double.

Since C++17, the constructor of a class template can deduce its arguments. Study the first example of Class template argument deduction for a deeper understanding.

Example 3: Function Template Argument Deduction by Reference

```
// functionTemplateArgumentDeductionReference.cpp

template <typename T>
void func(T& param){}

template <typename T>
void constFunc(const T& param){}

int main(){

  int x = 2011;
  const int cx = x;
  const int& rx = x;

  func(x);
  func(cx);
  func(rx);

  constFunc(x);
  constFunc(cx);
  constFunc(rx);
}
```

Explanation

In the above example, we have created two functions `func` and `constFunc` in lines 4 and 7. Both of these functions accept parameters by reference (19 – 21).

For better understanding, click here to analyze the process using **C++ Insight**.

Example 4: Function Template Argument Deduction by Universal Reference

```
// functionTemplateArgumentDeductionUniversalReference.cpp

template <typename T>
void funcUniversal(T&& param){}


int main(){

  int x = 2011;
  const int cx = x;
  const int& rx = x;


  funcUniversal(x);
  funcUniversal(cx);
  funcUniversal(rx);
  funcUniversal(2014);
}
```

Explanation

In the above code, we have defined a function `funcUniversal` in line 4 which accepts its parameters with a universal reference.

For better understanding click here to analyze the process using **C++ Insight**.

Example 5: Function Template Argument Deduction by Value

```
// functionTemplateArgumentDeductionValue.cpp

template <typename T>
void funcValue(T param){}

int main(){

  int x = 2011;
  const int cx = x;
  const int& rx = x;

  funcValue(x);
  funcValue(cx);
  funcValue(rx);
}
```

Explanation

In the above example, we have implemented a function `funcValue` in line 4 which takes its parameter by value.

For better understanding click here to analyze the process using **C++ Insight**.

Let’s solve an exercise in the next lesson.