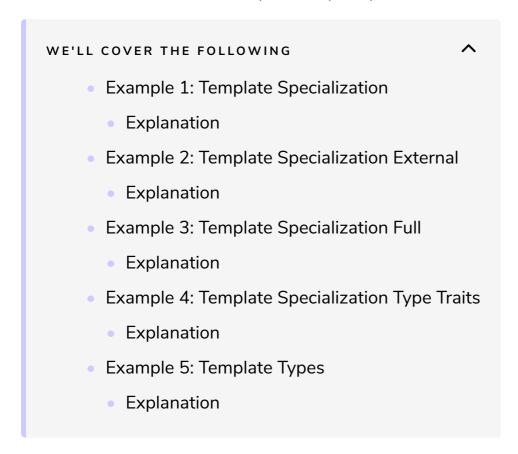
- Examples

In this lesson, we'll look at a few examples of template specialization.



Example 1: Template Specialization

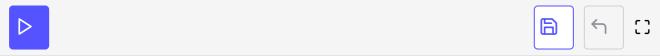
```
// TemplateSpecialization.cpp

#include <iostream>

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

template <typename T, int Line, int Column>
class Matrix{
    std::string getName() const { return "Primary Template"; }
};
```

```
tempiate (typename i)
class Matrix<T,3,3>{
  std::string name{"Partial Specialization"};
template <>
class Matrix<int,3,3>{};
template<typename T>
bool isSmaller(T fir, T sec){
  return fir < sec;
template <>
bool isSmaller<Account>(Account fir, Account sec){
  return fir.getBalance() < sec.getBalance();</pre>
int main(){
  std::cout << std::boolalpha << std::endl;</pre>
  Matrix<double,3,4> primaryM;
  Matrix<double,3,3> partialM;
  Matrix<int,3,3> fullM;
  std::cout << "isSmaller(3,4): " << isSmaller(3,4) << std::endl;</pre>
  std::cout << "isSmaller(Account(100.0),Account(200.0)): "<< isSmaller(Account(100.0),Account</pre>
  std::cout << std::endl;</pre>
```



Explanation

In the above example, we're modifying the codes that we have used in the previous lesson.

- **Primary** template is called when we use values other than Matrix<data_type, 3, 3> (line 43).
- Partial specialization is called when we instantiate Matrix<data_type, 3,
 3> where data_type is not int (line 44).
- Full specialization is called when we explicitly use int as a data type: Matrix<int, 3, 3> (line 45)
- **Full** specialization of the function template <code>isSmaller</code> is only applicable for <code>Account</code> objects. This allows it to compare two <code>Account</code> objects based on their balance (line 48).

Example 2: Template Specialization External

```
//TemplateSpecializationExternal.cpp
                                                                                           0
#include <iostream>
template <typename T=std::string, int Line=10, int Column=Line>
class Matrix{
public:
 int numberOfElements() const;
};
template <typename T, int Line, int Column>
int Matrix<T,Line,Column>::numberOfElements() const {
  return Line * Column;
template <typename T>
class Matrix<T,3,3>{
public:
 int numberOfElements() const;
};
template <typename T>
int Matrix<T,3,3>::numberOfElements() const {
  return 3*3;
template <>
class Matrix<int,3,3>{
public:
 int numberOfElements() const;
int Matrix<int,3,3>::numberOfElements() const {
  return 3*3;
int main(){
  std::cout << std::endl;</pre>
  Matrix<double,10,5> matBigDouble;
  std::cout << "matBigDouble.numberOfElements(): " << matBigDouble.numberOfElements() << std:</pre>
                           // ERROR
  // Matrix matString;
  Matrix<> matString;
  std::cout << "matString.numberOfElements(): " << matString.numberOfElements() << std::endl;</pre>
  Matrix<float> matFloat;
  std::cout << "matFloat.numberOfElements(): " << matFloat.numberOfElements() << std::endl;</pre>
  Matrix<bool,20> matBool;
  std::cout << "matBool.numberOfElements(): " << matBool.numberOfElements() << std::endl;</pre>
  Matrix <double,3,3> matSmallDouble;
  std::cout << "matSmallDouble.numberOfElements(): " << matSmallDouble.numberOfElements() <<</pre>
```

```
Matrix <int,3,3> matInt;
std::cout << "matInt.numberOfElements(): " << matInt.numberOfElements() << std::endl;
std::cout << std::endl;
}</pre>
```







[]

Explanation

In the above example, we have set the default value of line to 10 (line 6) and used the value for line as the default for column. The method numberOfElements returns the product of both numbers as a result. If we call the Matrix with arguments, then these passed arguments override the default. For float and string, it returns the 100 as no arguments are passed and the default arguments are used (lines 48 and 51).

Example 3: Template Specialization Full

```
// templateSpecializationFull.cpp
#include <iostream>
#include <string>
template <typename T>
T min(T fir, T sec){
  return (fir < sec) ? fir : sec;</pre>
template <>
bool min<bool>(bool fir, bool sec){
  return fir & sec;
int main(){
  std::cout << std::boolalpha << std::endl;</pre>
  std::cout << "min(3.5, 4.5): " << min(3.5, 4.5) << std::endl;
  std::cout << "min<double>(3.5, 4.5): " << min<double>(3.5, 4.5) << std::endl;
  std::cout << "min(true, false): " << min(true, false) << std::endl;</pre>
  std::cout << "min<bool>(true, false): " << min<bool>(true, false) << std::endl;</pre>
  std::cout << std::endl;</pre>
```

Explanation

In the above example, we have defined a full specialization for bool. The primary and the full specialization are implicitly invoked in the lines (20 and 23) and explicitly invoked in the lines (21 and 24).

Example 4: Template Specialization Type Traits

```
// templateSpecializationTypeTraits.cpp
                                                                                              G
#include <iostream>
#include <type_traits>
using namespace std;
template <typename T>
void getPrimaryTypeCategory(){
  cout << boolalpha << endl;</pre>
  cout << "is_void<T>::value: " << is_void<T>::value << endl;</pre>
  cout << "is_integral<T>::value: " << is_integral<T>::value << endl;</pre>
  cout << "is_floating_point<T>::value: " << is_floating_point<T>::value << endl;</pre>
  cout << "is_array<T>::value: " << is_array<T>::value << endl;</pre>
  cout << "is_pointer<T>::value: " << is_pointer<T>::value << endl;</pre>
  cout << "is_reference<T>::value: " << is_reference<T>::value << endl;</pre>
  cout << "is_member_object_pointer<T>::value: " << is_member_object_pointer<T>::value << end
  cout << "is_member_function_pointer<T>::value: " << is_member_function_pointer<T>::value <<</pre>
  cout << "is_enum<T>::value: " << is_enum<T>::value << endl;</pre>
  cout << "is_union<T>::value: " << is_union<T>::value << endl;</pre>
 cout << "is_class<T>::value: " << is_class<T>::value << endl;
  cout << "is_function<T>::value: " << is_function<T>::value << endl;</pre>
 cout << "is_lvalue_reference<T>::value: " << is_lvalue_reference<T>::value << endl;</pre>
 cout << "is_rvalue_reference<T>::value: " << is_rvalue_reference<T>::value << endl;</pre>
  cout << endl;</pre>
}
int main(){
    getPrimaryTypeCategory<void>();
```

Explanation

We have used the type_traits library which detects at compile-time to which primary type category void (line 13) belongs to. The primary type categories

are complete and exclusive. This means each type belongs exactly to one

primary type category. For example, void returns true for the type-trait std::is_void and false for all the other type categories.

Example 5: Template Types

```
// templateTypes.cpp

#include <iostream>
#include <string>

template<typename T>
struct Type{
    std::string getName() const {
        return "unknown";
    }
};

int main(){
    std::cout << std::boolalpha << std::endl;
    Type<float> tFloat;
    std::cout << "tFloat.getName(): " << tFloat.getName() << std::endl;
    std::cout << std::endl;
}</pre>
```

Explanation

In the above example, the method <code>getName</code> returns <code>unknown</code> for any type passed in the argument of <code>type</code> function (line 8). If we specialize the class template for further types, we will implement a type deduction system at compile-time. We'll look at it in the coming exercise.

We'll solve an exercise in the next lesson.