Template instantiation and specialization

```
/*
    . Topics
        .#1: Implicit instantiation through use

        .#2: Explicit template instantiation definition

        .#3: Explicit template instantiation declaration

        .#4: Template full specialization

        .#5: Template partial specialization

        .#6: Type traits
*/
```

Implicit instantiations

```
// Implicit instantiation: Instances are created when you create concrete objects.
export template <typename T>
class MyClass {
public:
    // A function that has no error
    void safe_function() {
        fmt::println("{}", "Safe function called");
    }

    // A function that has an error
    void error_function() {
        // Intentional error: assuming T is a class with a member function 'non_existent_function'
        T instance;
        instance.non_existent_function();  // This will cause an error if instantiated
    }
};
```

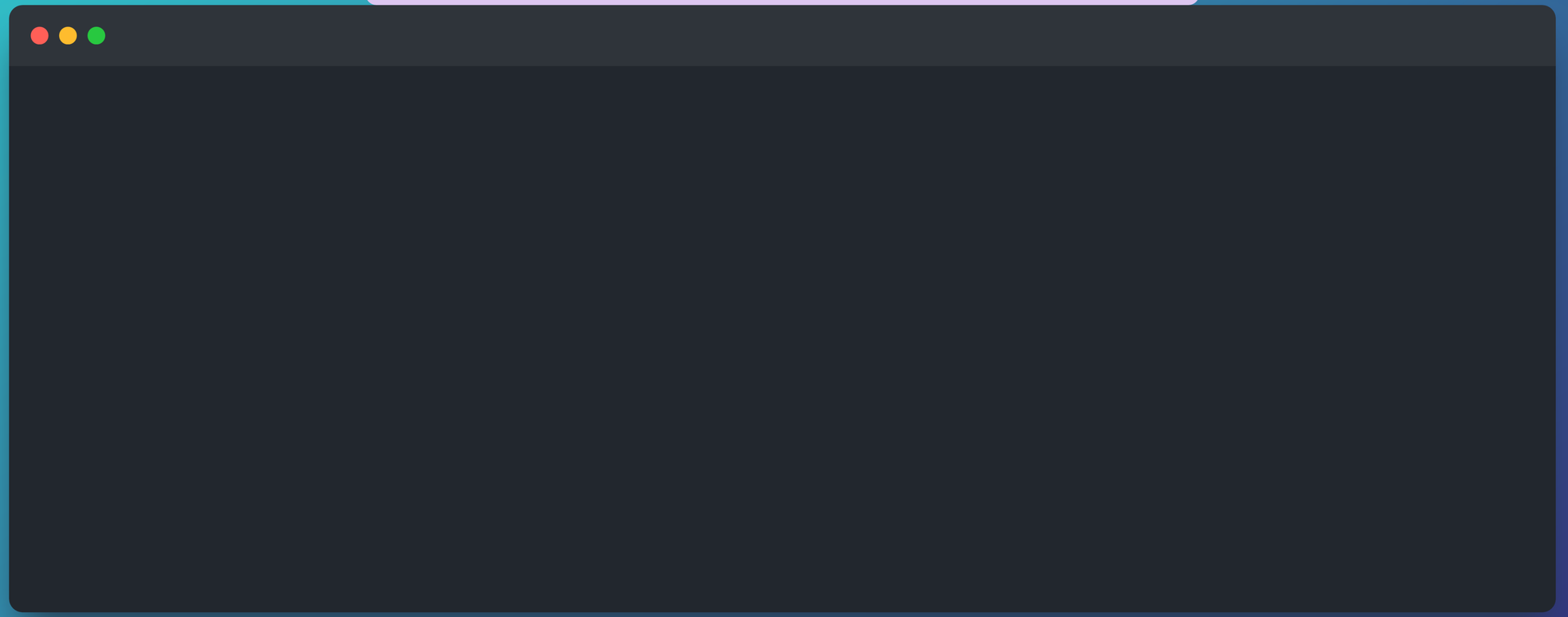
Implicit instantiations

```
//Implicit instantiation: Instances are created when you create concrete objects.
MyClass<int> my_int_instance;
my_int_instance.safe_function();  // This works

MyClass<double> my_double_instance;
my_double_instance.safe_function();  // This works

//my_int_instance.error_function(); // Notice that the code compiled, even if the call to error_function was in there.
                                    // So no checks are done before instantiating the template.
//my_double_instance.error_function();
```

Implicit instantiations

```
//Implicit instantiation: Instances are created when you create concrete objects.
MyClass<int> my_int_instance;
my_int_instance.safe_function();  // This works

MyClass<double> my_double_instance;
my_double_instance.safe_function();  // This works

//my_int_instance.error_function(); // Notice that the code compiled, even if the call to error_function was in there.
                                    // So no checks are done before instantiating the template.
//my_double_instance.error_function();
```

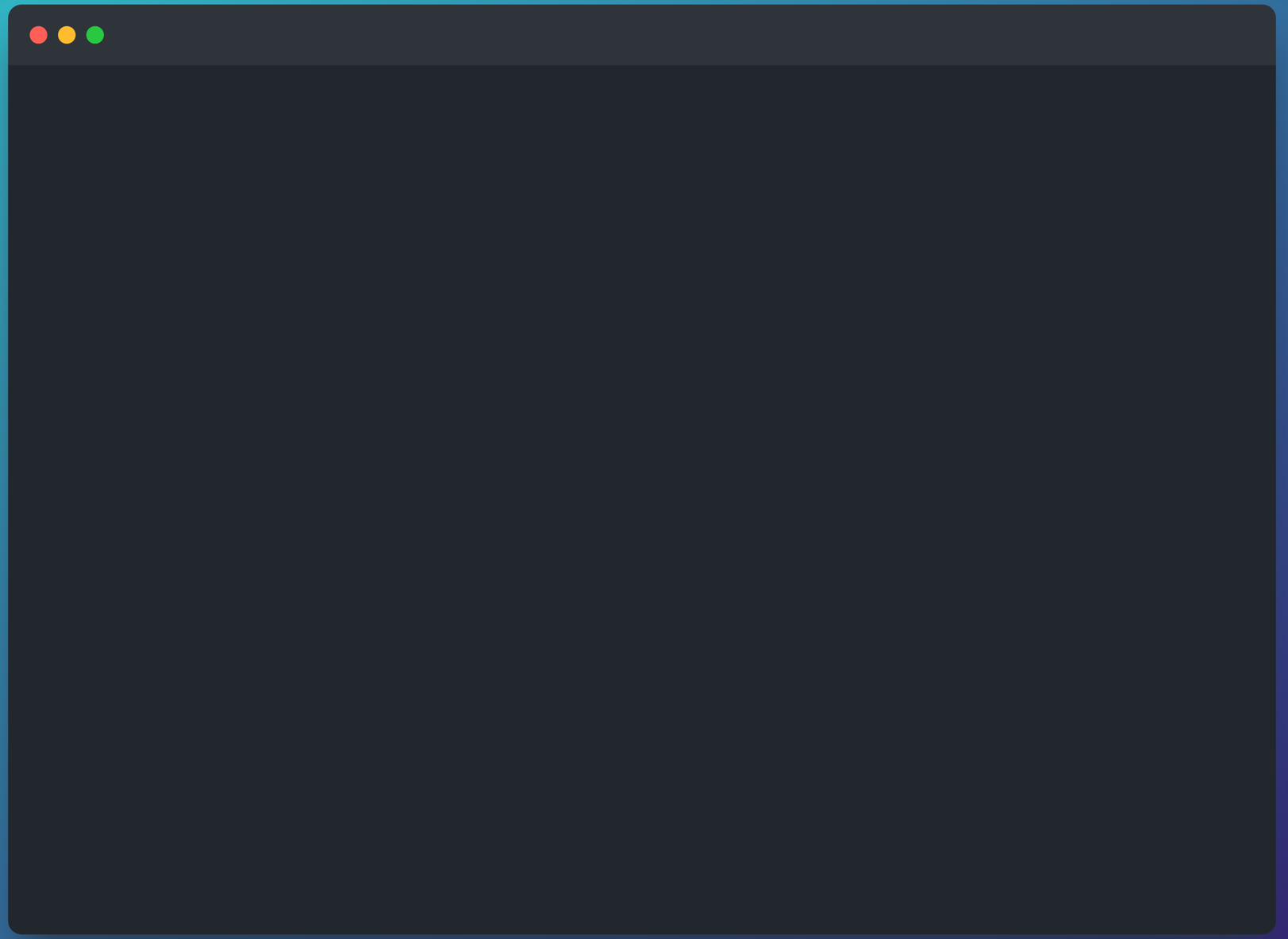
Demo time!



Explicit instantiation definition.

```cpp
//Explicit instantiation definition
    export template<typename T>
    T add(T a, T b) {
        //T instance;
        //instance.hello();
        return a + b;
    }

    export template<typename T>
    class Point {
    public:
        Point(T x, T y) : x(x), y(y) {}
        T getX() const { return x; }
        T getY() const { return y; }

        //A compiler error will be thrown if the explicit instantiation definition is in place.
        //Because that forces the compiler to generate an complete int instance in this case.
        void error_function(){
            //T instance;
            //instance.some_function();
        }
    private:
        T x, y;
    };

    //Explicit template instantiation definitions
    template int add<int>(int, int);
    template class Point<int>;
```

Demo time!



Explicit instantiation declaration

temlates_3.ixx

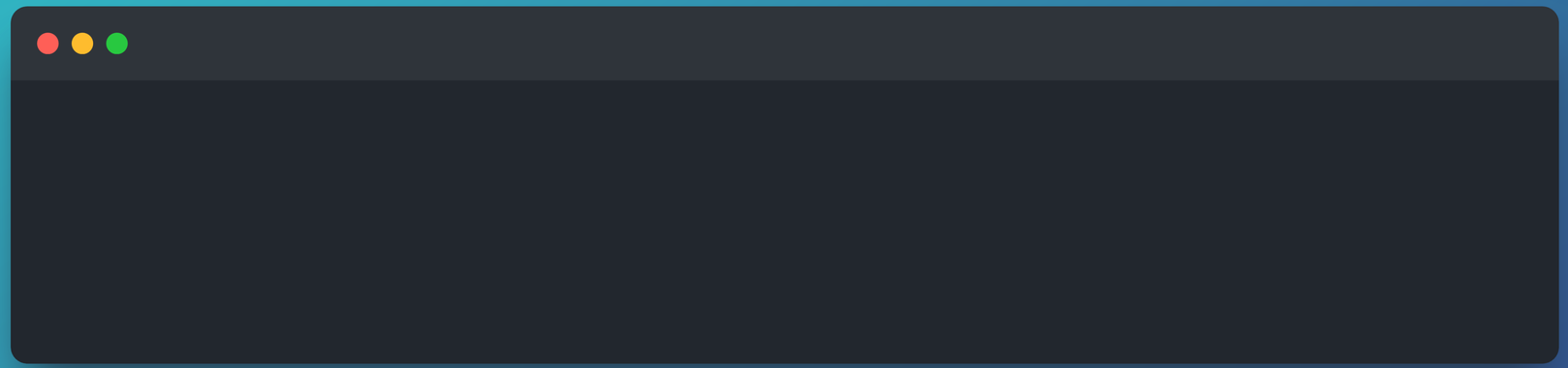
```
//Explicit instantiation declaration
    export template<typename T>
    T add(T a, T b) {
        //T instance;
        //instance.hello();
        return a + b;
    }

    export template<typename T>
    class Point {
    public:
        Point(T x, T y) : x(x), y(y) {}
        T getX() const { return x; }
        T getY() const { return y; }

        //A compiler error will be thrown if the explicit instantiation definition is in place.
        //Because that forces the compiler to generate an complete int instance in this case.
        void error_function(){
            //T instance;
            //instance.some_function();
        }
    private:
        T x, y;
    };

    //Explicit template instantiation declarations. These are declarations, only
    //hinting to the fact that the definitions will be provided elsewhere.
    //the extern keyword makes all the difference here.
    extern template int add<int>(int, int);
    extern template class Point<int>;
```

Explicit instantiation declaration

templates_3.cpp

```cpp
//Explicit instantiation declaration: the definitions in another file.
template int add<int>(int, int);
template class Point<int>;
```

Explicit instantiation declaration: Demo time!



Template full specialization

```
/*
    . Exploring template full specialization.
        . Full specialization happens when you provide a complete definition for a specific
            type or function, completely overriding the generic (original) template.

        . Full specialization takes precedence over the primary(generic) template and any
            partial specialization.

        . When the compiler a call to a function call or object instantiation, it will first
        look for a full specialization, then a partial specialization, and finally the primary template.

*/
```

Template full specialization

```
// Generic (primary) template
export template <typename T>
class Adder {
public:
    T add(T a, T b) {
        fmt::println("Generic template class");
        return a + b;
    }
};

// Full specialization for int
template <>
class Adder<int> {
public:
    int add(int a, int b) {
        fmt::println("Specialized template class for int");
        return a + b;
    }
};

//Full specialization for const char*
template <>
class Adder<const char*> {
public:
    char* add(char* a,char* b) {
        fmt::println("Specialized template class for const char*");
        return std::strcat(a, b);
    }
};
```

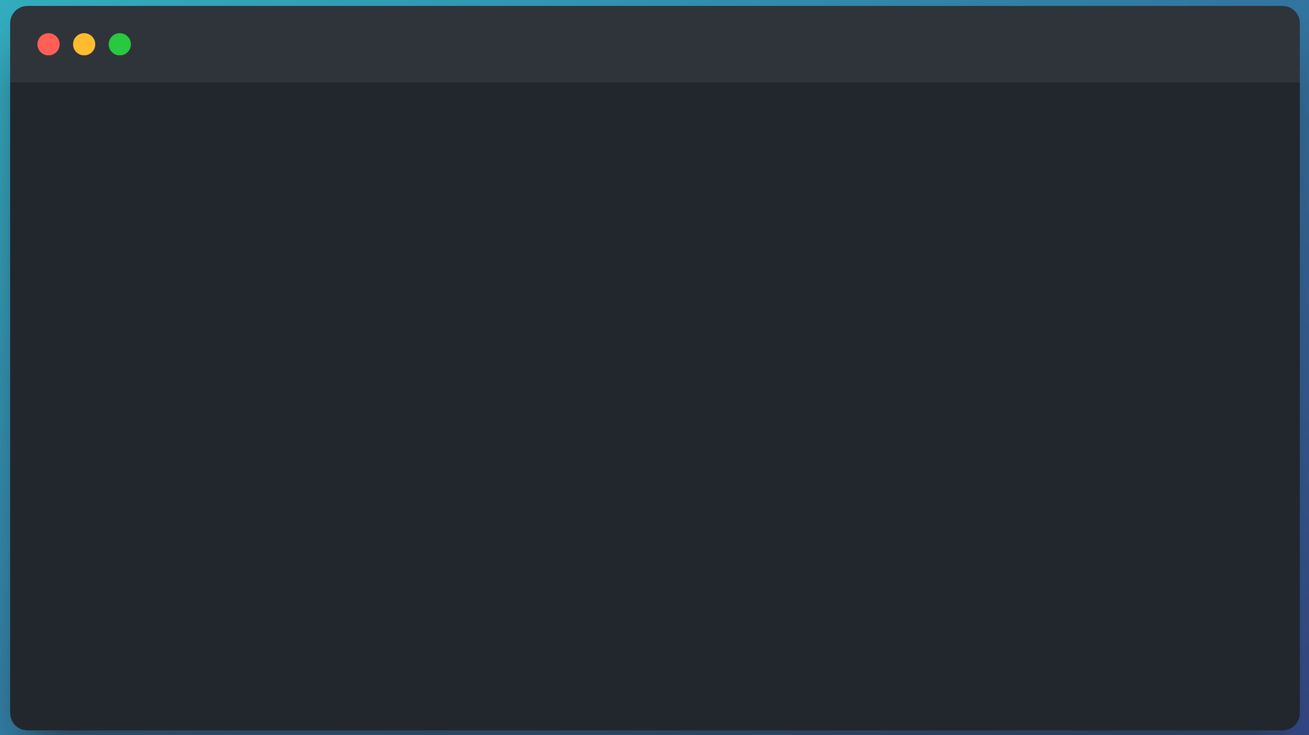
Template full specialization

```
// Primary function template: full specialization
export template <typename T>
T add(T a, T b) {
    fmt::println("Generic template function");
    return a + b;
}

template<>
char* add<char*>(char* a, char* b) {
    fmt::println("Specialized template function for char*");
    return std::strcat(a, b);
}
```

Template full specialization

```cpp
//Class specialization
Adder<int> adder;
fmt::print("Sum: {}\n", adder.add(1, 2));

Adder<const char*> adder2;
char a[20] = "Hello"; //strcat appends to the first parameter.
                      //So, we need to have enough space. Here, 20 bytes.
char b[] = " World";
auto result = adder2.add(a, b);
fmt::print("Concatenated string: {}\n", result);

//Function specialization
fmt::print("Sum: {}\n", add(1, 2));
fmt::println("Concatenated string: {}", add(a, b));
```

Template full specialization: Demo time!



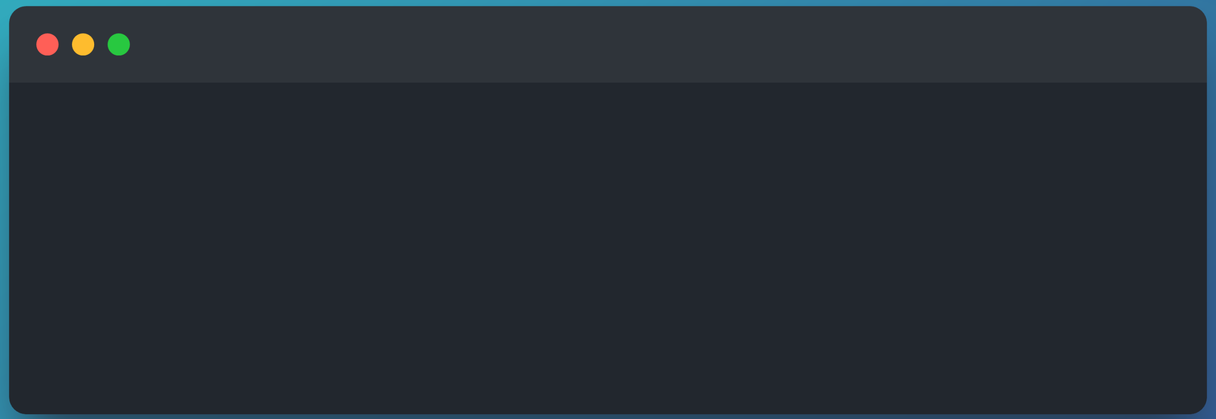
Template full specialization: Build your own type trait

```cpp
// Primary template (default case: type is not int)
export template <typename T>
struct is_int {
    constexpr static  bool value = false;
};

// Specialization for int
template <>
struct is_int<int> {
    constexpr static bool value = true;
};
```

Template full specialization: Build your own type trait

```
fmt::println("Is int: {}", is_int<int>::value);    // true
fmt::println("Is int: {}", is_int<double>::value); // false
fmt::println("Is int: {}", is_int<char>::value);   // false
fmt::println("Is int: {}", is_int<std::string>::value); // false
fmt::println("Is int: {}", is_int<long int>::value); // false
```

Build your own type trait: Demo time!



Partial specialization

```
/*
    . Partial specialization applies only when the primary template has more
      than one   template parameter.
      . You can then partially one template parameter while keeping the other(s) generic.

    . Partial specialization for function templates is not supported in C++.
      . You can use function overloading to achieve the same effect.
*/
```

Partial specialization

```
/*
    . Partial specialization applies only when the primary template has more
      than one   template parameter.
      . You can then partially one template parameter while keeping the other(s) generic.

    . Partial specialization for function templates is not supported in C++.
      . You can use function overloading to achieve the same effect.
*/
```

Partial specialization

```
// Primary template
export template <typename T1, typename T2>
class Adder {
public:
    T1 add(T1 a, T2 b) {
        fmt::println("Generic template class");
        return a + b;
    }
};

// Partial specialization for when the first type is int
export template <typename T2>
class Adder<int, T2> {
public:
    int add(int a, T2 b) {
        fmt::println("Partial specialization with int as the first type");
        return a + b;
    }
};
```

Partial specialization

```cpp
//Partial specialization for function templates is not supported in C++.
export template <typename T1, typename T2>
T1 add(T1 a, T2 b) {
    fmt::println("Generic template function");
    return a + b;
}

//  This will not compile
/*
template <typename T2>
T1 add(int a, T2 b) {
    fmt::println("Partial specialization with int as the first type");
    return a + b;
}
*/

//An overload that achieves the same effect as the partial specialization for
//function template that we can't do above.
export template <typename T>
T add(int a, T b) {
    fmt::println("Function overload for (int, T)");
    return a + b;
}
```

Partial specialization

```
Adder<int,double> adder;
//adder.add(1, 2.0); // Calls the partial specialization version

Adder<double,int> adder2;
//adder2.add(1.0, 2); // Calls the generic version

//add(1, 2.0); //Calls the function overload for (int, T)

add<int,double>(1, 2.0); //Calls the generic function template
```

Partial specialization: Demo time!