Functions

March 31, 2022

1 Write a function

1.1 Example of a basic function

[1]:
```
template<typename Type>
Type sum(Type lhs, Type rhs)
{
    return lhs + rhs;
}
```

[2]:
```
#include <iostream>

std::cout << sum(4, 5) << std::endl;
std::cout << sum(3.8, 6.4) << std::endl;
std::cout << sum(std::string("hello"), std::string("world"));
```

```
9
10.2
helloworld
```

[3]:
```
std::cout << sum(3.8, 6) << std::endl;
```

```
input_line_10:2:15: error: no matching function for
call to 'sum'
 std::cout << sum(3.8, 6) << std::endl;
              ^~~

input_line_7:2:6: note: candidate template ignored:
deduced conflicting types for parameter 'Type' ('double' vs. 'int')
Type sum(Type lhs, Type rhs)
     ^
```

```
Interpreter Error:
```

1.2 How to write function with template parameters?

```
template<ListOfParameters>
ReturnType FunctionName(Inputs);
```

We have used *typename* keyword before our parameter but we could use other 'types'. The condition for *Type* inside our example is that it must support + operator and to be copyable to return it.

1.2.1 void could be a type

[4]:
```
#include <typeinfo>
#include <iostream>

template<typename Type>
void testVoid(Type* pType)
{
    std::cout << typeid(Type).name() << std::endl;
}
```

[5]:
```
?std::type_info
```

https://en.cppreference.com/w/cpp/types/type_info

[6]:
```
void* p = nullptr;
testVoid(p);
```

v

1.3 Two-phase translation

1.3.1 Compile-time error

[7]:
```
struct A{};
struct B{};

A a;
B b;
auto t = sum(a, b);
```

```
input_line_14:6:10: error: no matching function for
call to 'sum'
auto t = sum(a, b);
         ^~~
input_line_7:2:6: note: candidate template ignored:
deduced conflicting types for parameter 'Type' ('__cling_N57::A' vs. 'B')
Type sum(Type lhs, Type rhs)
```

```
    ^
```

```
Interpreter Error:
```

We have a compile time error because the types A and B are different.

Definition time During this phase there is no instantiation, so the code is checked ignoring the template parameters.

Elements checked during this phase: 1. syntax errors 2. names: unknown and independant of template parameters 3. static assertions independant of template parameters

Instantiation time During this phase, there are instantiations, so the code is checked to valid it and all parts that contain template parameters are checked too.

Example

[8]:
```
template<typename Type>
void example(Type type)
{
    Unknown(); // compile-time error during definition time
    Unknown(type); // compile-time error during instantiation time
}
```

```
input_line_15:4:5: error: use of undeclared identifier

'Unknown'
    Unknown(); // compile-time error during definition time
    ^
```

```
Interpreter Error:
```

> This double check is called **two-phase lookup** and the performance during the definition time is compiler dependant.

1.4 Type Conversions

During type deduction, all automatic type conversions are limited.

1.4.1 Declaring by reference

If we declare parameters by reference, even cv conversions are not applied. So inside our `Sum` function, parameters named Type must be equal.

1.4.2 Declaring by value

If we declare parameters by value, only simple conversions are applied. * *const* and *volatile* are ignored * references are converted to the referenced type * arrays and functions are converted to the pointer type associated.

Example

[9]:
```
template<typename Type>
void testValueType(Type lhs, Type rhs)
{
    std::cout << typeid(lhs).name() << " " << typeid(rhs).name() << std::endl;
}
```

[10]:
```
const int toto = 2;
testValueType(toto, toto);
```

```
i i
```

[11]:
```
int tata = 10;
int& refTata = tata;
testValueType(tata, refTata);
```

```
i i
```

[12]:
```
int array[23];
testValueType(&tata, array);
```

```
Pi Pi
```

1.4.3 Some errors

Non deduction errors

[13]:
```
std::string str = "world";
testValueType("hello", str);
```

```
input_line_20:3:1: error: no matching function for call

to 'testValueType'
testValueType("hello", str);
^~~~~~~~~~~~~

input_line_16:2:6: note: candidate template ignored:
deduced conflicting types for parameter 'Type' ('const char *' vs.
'std::__cxx11::basic_string<char>')
void testValueType(Type lhs, Type rhs)
     ^
```

```
Interpreter Error:
```

[14]:
```
testValueType(3, 4.5);
```

```
input_line_21:2:2: error: no matching function for call
to 'testValueType'
 testValueType(3, 4.5);
 ^~~~~~~~~~~~
input_line_16:2:6: note: candidate template ignored:
deduced conflicting types for parameter 'Type' ('int' vs. 'double')
void testValueType(Type lhs, Type rhs)
     ^
```

```
Interpreter Error:
```

Ways to avoid them

1. By casting parameters

[15]:
```
testValueType(static_cast<double>(3), 4.5);
```

```
d d
```

2. By specifying the template parameters

[16]:
```
testValueType<double>(3, 4.5);
```

```
d d
```

3. By specifying different type parameters

[17]:
```
template<typename TypeA, typename TypeB>
void testValueTypeModified(TypeA lhs, TypeB rhs)
{
    std::cout << typeid(lhs).name() << " " << typeid(rhs).name() << std::endl;
}
```

[18]:
```
std::string myString = "world";
testValueTypeModified(3, 4.5);
testValueTypeModified("hello", myString);
```

```
i d
PKc NSt7__cxx1112basic_stringIcSt11char_traitsIcESaIcEEE
```

Default arguments

[19]:
```
template<typename Type>
void withDefaultArg(Type type = 0)
{
    std::cout << type << std::endl;
}
```

[20]:
```
WithDefaultArg();
```

```
input_line_27:2:2: error: use of undeclared identifier
'WithDefaultArg'; did you mean 'withDefaultArg'?
 WithDefaultArg();
 ^~~~~~~~~~~~~
 withDefaultArg
input_line_26:2:6: note: 'withDefaultArg' declared
here
void withDefaultArg(Type type = 0)
     ^
input_line_27:2:2: error: no matching function for
call to 'withDefaultArg'
 WithDefaultArg();
 ^~~~~~~~~~~~~
input_line_26:2:6: note: candidate template ignored:
couldn't infer template argument 'Type'
void withDefaultArg(Type type = 0)
     ^
```

```
Interpreter Error:
```

[21]:
```
template<typename Type = int>
void withDefaultParameter(Type type = 0)
{
    std::cout << typeid(type).name() << std::endl;
}
```

[22]:
```
withDefaultParameter();
```

```
i
```

When we want to use default values for arguments, we need to define default type for template parameters.

1.5 Template parameters

When we use templated functions we have two types of parameters.

```
template<typename TemplateParameter>
void test(TemplateParameter CallParameter);
```

1.6 Some difficulties with return type

[23]:
```
auto returnType = sum<double>(3, 4.5);
std::cout << typeid(returnType).name() << std::endl;
```

```
d
```

1.6.1 Return type parameter

[24]:
```
template<typename TypeA, typename TypeB, typename ReturnType>
ReturnType sum(TypeA lhs, TypeB rhs)
{
    return lhs + rhs;
}
```

[25]:
```
std::cout << sum(3, 4.5) << std::endl;
```

```
input_line_32:2:15: error: no matching function for
call to 'sum'
 std::cout << sum(3, 4.5) << std::endl;
              ^~~

input_line_31:2:12: note: candidate template ignored:
couldn't infer template argument 'ReturnType'
ReturnType sum(TypeA lhs, TypeB rhs)
           ^

input_line_7:2:6: note: candidate template ignored:
deduced conflicting types for parameter 'Type' ('int' vs. 'double')
Type sum(Type lhs, Type rhs)
     ^
```

```
Interpreter Error:
```

`ReturnType` is not deduced and we need to specify it.

[26]:
```
std::cout << sum<int, double, double>(3, 4.5) << std::endl;
std::cout << sum<int, double, int>(3, 4.5) << std::endl;
```

```
7.5
7
```

If we don't set return type parameter at the first place we need to specify all parameters in order.

[27]:
```
template<typename ReturnType, typename TypeA, typename TypeB>
ReturnType sum(TypeA lhs, TypeB rhs)
{
    return lhs + rhs;
}
```

[28]:
```
std::cout << sum<int>(3, 4.5) << std::endl;
std::cout << sum<double>(3, 4.5) << std::endl;
```

```
7
7.5
```

Using auto

[29]:
```
template<typename TypeA, typename TypeB>
auto sum(TypeA lhs, TypeB rhs)
{
    return lhs + rhs;
}
```

[30]:
```
std::cout << sum(3, 4.5) << std::endl;
```

```
7.5
```

auto means that the return type will be deduced from the return statement in the function body. Available since C++14.

Before we had to use call parameters and determine type in the trailing return type:

To help auto

[31]:
```
template<typename TypeA, typename TypeB>
auto sumTest(TypeA lhs, TypeB rhs)->decltype(lhs + rhs)
{
    return lhs + rhs;
}
```

[32]:
```
std::cout << sumTest(3, 4.5) << std::endl;
```

```
7.5
```

The type is determined by the rules of + operator and a common type is found.

Warning: in some case where template parameters are references, the deduction for the return type could fail.

Using decay

[33]:
```
?std::decay
```

https://en.cppreference.com/w/cpp/types/decay

[34]:
```
#include <type_traits>
template<typename TypeA, typename TypeB>
auto sumTestDecay(TypeA lhs, TypeB rhs)->typename std::decay<decltype(lhs +␣
 ↪rhs)>::type
{
    return lhs + rhs;
}
```

[35]:
```
std::cout << sumTestDecay(3, 4.5) << std::endl;
```

```
7.5
```

We can use default parameter type too:

[36]:
```
template<typename TypeA, typename TypeB, typename ReturnType = typename std::
 ↪decay<decltype(TypeA() + TypeB())>::type>
ReturnType sumTestDecayDefault(TypeA lhs, TypeB rhs)
{
    return lhs + rhs;
}
```

[37]:
```
std::cout << sumTestDecayDefault(3, 6.5) << std::endl;
```

```
9.5
```

Find a common type

[38]:
```
?std::common_type
```

https://en.cppreference.com/w/cpp/types/common_type

[39]:
```
template<typename TypeA, typename TypeB>
typename std::common_type<TypeA, TypeB>::type sumCommon(TypeA lhs, TypeB rhs)
{
    return lhs + rhs;
}
```

[40]:
```
std::cout << sumCommon(3, 8.5) << std::endl;
```

```
11.5
```

We can use default parameter type too.

[41]:
```
template<typename TypeA, typename TypeB, typename ReturnType = typename std::
 ↪common_type<TypeA, TypeB>::type>
ReturnType sumCommonDefault(TypeA lhs, TypeB rhs)
{
    return lhs + rhs;
}
```

[42]:
```
std::cout << sumCommonDefault(3, 8.5) << std::endl;
```

```
11.5
```

1.7 Manage overload

1.7.1 Compare with non-templated functions

[43]:
```
namespace Overload
{
    int sum(int lhs, int rhs)
    {
        std::cout << "Sum not templated" << std::endl;
        return lhs + rhs;
    }

    template<typename Type>
    Type sum(Type lhs, Type rhs)
    {
        std::cout << "Sum templated" << std::endl;
        return lhs + rhs;
    }
}
```

[44]:
```
std::cout << Overload::sum(3, 8) << std::endl;
std::cout << Overload::sum<>(3, 8) << std::endl;
```

```
Sum not templated
11
Sum templated
11
```

The non-templated functions are always privileged versus templated functions during overloading resolution.

Question During overload resolution why do plain vanilla functions preempt templated functions?

Answer This behavior is logical because non templated functions are always compiled.

1.7.2 Resolution with templated functions

[45]:
```
namespace Overload
{
    template<typename TypeA, typename TypeB>
    auto sum2(TypeA lhs, TypeB rhs)
    {
        std::cout << "No return type" << std::endl;
        return lhs + rhs;
    }

    template<typename ReturnType, typename TypeA, typename TypeB>
    ReturnType sum2(TypeA lhs, TypeB rhs)
    {
        std::cout << "With return type" << std::endl;
        return lhs + rhs;
    }
}
```

[46]:
```
std::cout << Overload::sum2(3, 3.5) << std::endl;
```

```
No return type
6.5
```

[47]:
```
std::cout << Overload::sum2<int>(3, 3.5) << std::endl;
```

```
input_line_53:2:15: error: call to 'sum2' is
ambiguous
 std::cout << Overload::sum2<int>(3, 3.5) << std::endl;
              ^~~~~~~~~~~~~~~~~~

input_line_51:4:10: note: candidate function [with
TypeA = int, TypeB = double]
    auto sum2(TypeA lhs, TypeB rhs)
         ^

input_line_51:11:16: note: candidate function [with
ReturnType = int, TypeA = int, TypeB = double]
    ReturnType sum2(TypeA lhs, TypeB rhs)
               ^
```

```
Interpreter Error:
```

[48]:
```
std::cout << Overload::sum2<double>(3, 3.5) << std::endl;
```

```
With return type
```

`6.5`

Question Why does the code compile with double and not with int?

Explanation With the `Overload::sum2<int>(3, 3.5)`, the result of the second template definition could be int as requested in the specification. There is in the first template definition the `auto` return type in *int* (double + int) and we have specified it to int, so it's an admissible candidate too.

With the `Overload::sum2<double>(3, 3.5)`, the result of the second template definition could be double as requested in the specification. In the first template definition the `auto` return type in *double* (double + int) and we have specified it to double with implicit conversion of the integer. So only one candidate is possible.

Solution During resolution - Firstly the instanciation of the first template function definition triggers a silent compile error. - Secondly, the instanciation of the second template function dfefinition is fine. - Consequently, there is no ambiguity on which function should be called:

This behavior is called SFINAE (Substitution Failure Is Not An Error).