Class

March 31, 2022

1 Class templates

1.1 How to write class with template parameters?

```
template<ListOfParameters>
class ClassName
{
    ...
};
```

1.1.1 Example of implementation

[1]:
```
#include <iostream>

template<typename FirstType, typename SecondType>
struct Pair
{
    Pair(const FirstType& f, const SecondType& s)
        : first(f)
        , second(s)
    {
        std::cout << "Generic constructor" << std::endl;
    }

    FirstType first;
    SecondType second;
};
```

[2]:
```
Pair<int, char> pair(3, 'c');
```

```
Generic constructor
```

1.2 Specialization

1.2.1 Total

Syntax

```
template<>
class ClassName<ListOfParameterTypes>
```

```
{
    ...
};
```

Example

[3]:
```
template<>
struct Pair<double, double>
{
public:
    Pair(const double& a, const double& b)
        : first(a)
        , second(b)
    {
        std::cout << "double are everywhere!" << std::endl;
    }

    double first;
    double second;
};
```

[4]:
```
Pair<double, double> doublePair(3.0, 5.4);
```

```
double are everywhere!
```

1.2.2 Partial

Syntax

```
template<ListOfParameterTypesToCallIt>
class ClassName<ParameterTypesUsedToCallTemplate>
{
};
```

Examples

With same types

[5]:
```
template<typename Type>
struct Pair<Type, Type>
{
    Pair(const Type& f, const Type& s)
        : first(f)
        , second(s)
    {
        std::cout << "Same Types used" << std::endl;
    }

    Type first;
    Type second;
};
```

With one specific type

[6]:
```
template<typename Type>
struct Pair<int, Type>
{
    Pair(const int& f, const Type& s)
        : first(f)
        , second(s)
    {
        std::cout << "FirstType is int" << std::endl;
    }

    int first;
    Type second;
};
```

With pointers

[7]:
```
template<typename FirstType, typename SecondType>
struct Pair<FirstType*, SecondType*>
{
    Pair(FirstType* f, SecondType* s)
        : first(f)
        , second(s)
    {
        std::cout << "Pointers are used" << std::endl;
    }

    FirstType* first;
    SecondType* second;
};
```

Execution

[8]:
```
Pair<float, double> pairFloatDouble(3.f, 4.5);
Pair<int, float> pairIntFloat(3, 5);
Pair<float, float> pairFloat(3.f, 4.f);
Pair<int*, float*> pairPointer(new int(3), new float(4.f));
```

```
Generic constructor
FirstType is int
Same Types used
Pointers are used
```

[9]:
```
Pair<int, int> pairInt(4, 3);
```

```
input_line_16:2:17: error: ambiguous partial
specializations of 'Pair<int, int>'
 Pair<int, int> pairInt(4, 3);
                ^

input_line_12:2:8: note: partial specialization matches
[with Type = int]
struct Pair<Type, Type>
       ^

input_line_13:2:8: note: partial specialization matches
[with Type = int]
struct Pair<int, Type>
       ^
```

```
Interpreter Error:
```

[10]:
```
Pair<float*, float*> pairPointerFloat(new float(5.f), new float(6.f));
```

```
input_line_17:2:23: error: ambiguous partial
specializations of 'Pair<float *, float *>'
 Pair<float*, float*> pairPointerFloat(new float(5.f), new float(6.f));
                      ^

input_line_12:2:8: note: partial specialization matches
[with Type = float *]
struct Pair<Type, Type>
       ^

input_line_14:2:8: note: partial specialization matches
[with FirstType = float, SecondType = float]
struct Pair<FirstType*, SecondType*>
       ^
```

```
Interpreter Error:
```

1.3 Default parameters

It's like with functions.

1.3.1 Example with a single parameter filled

[11]:
```cpp
template<typename FirstType, typename SecondType = FirstType>
struct SimplePair
{
    SimplePair(const FirstType& f, const SecondType& s)
        : first(f)
        , second(s)
    {
        std::cout << "Simple pair constructed" << std::endl;
    }

    FirstType first;
    SecondType second;
};
```

[12]:
```cpp
SimplePair<int> simplePair(3, 5);
```

```
Simple pair constructed
```

1.4 Alias

1.4.1 Non-templated alias

[13]:
```cpp
using IntPair = Pair<int, int>;
```

1.4.2 Templated alias

[14]:
```cpp
template<typename Type>
using PlainPair = Pair<Type, Type>;
```

[15]:
```cpp
PlainPair<double> doublePair(4.5, 9.6);
```

```
double are everywhere!
```

1.4.3 Alias in templated class

Non-templated alias

[16]:
```cpp
template<typename TypeIn>
struct A
{
    using Type = TypeIn;
};
```

[17]:
```cpp
std::cout << typeid(typename A<int>::Type).name() << std::endl;
```

```
i
```

Templated alias

[18]:
```
template<typename TypeIn>
struct SomeCompute
{
    using Type = TypeIn;
};

template<template<typename> class ComputeFunctor>
struct A
{
    template<typename T>
    using Compute = typename ComputeFunctor<T>::Type;
};
```

[19]:
```
std::cout << typeid(A<SomeCompute>:: template Compute<double>).name() << std::
  ↪endl;
```

```
d
```