Templates

09 June 2021 07:07

1. Template parameters can be types, non-types, and templates themselves.

Non-types can be a

- Pointer, nullptr, lvalue references - to a class object, function, class member function… how about r-value refs-wont work
  Manual-type deduction by specifying type at caller or Automatic deduction.. References decompose though
- integral values, enumerator of a enum
- floating-point values (C++20)
- Other template class/func/var?

```
template <typename T, template <typename, typename> class Container >  ---- nested templates
class Matrix
{
    Container<T, std::allocator<T>> data;
}

Matrix<int, std::vector>
Matrix<double, std::vector>

template<class T, class Allocator = std::allocator<T>> class vector;
```

2. Template arguments can have default parameters too.

3. Template instantiation

when u define a template - a generic version + specialized version is created during compilation
SPECIALIZED VERSION IS CALLED TEMPLATE INSTANTIATION

Template instantiation is done only when used -> class or function

4. Type aliases using 'using' can also use templates -> called alias templates… flexible

```
template<typename T, size_t size>
class typeClass
{
  public:
      T value{};
};

template<typename T>
using word = typeClass<T, 4>;

template<typename T>
using nibble = typeClass<T, 2>;

word<int> i{0};
```

5. Non-dependent name and dependent name -> two phase look-up

When compiling a template, the compiler distinguishes between names that

1. depend on the template parameters (called dependent names) and
2. those that do not (non-dependent names).

1. Non-dependent names are looked up normally when the template is **declared**.
2. Dependent names, on the other hand, must wait until the template is **instantiated**, when the template arguments are bound to the parameters.

Only then can the compiler know what those names truly mean. This is sometimes known as two-phase lookup .

6. Issue due to two phase look up

```
template <typename T>
class Base{
public:
    void func(){
        std::cout << "func\n";
    }
};
template <typename T>
class Derived: public Base<T>{
public:
    void callBase(){
        func(); // func is non-dependent -- lookup happens during derived<T> declaration and not during definition/inst. So lookup
issue?
        this->func(); // Make func dependent
        Base::func(); // Make func dependent -----> Base<T>:: or Base:: .. ALWAYS EXPLICIT!!
    }
};

Derived<int> derived;
derived.callBase();
```

7. Function template infer type from arguments…Class templates too -> C++17 offers this based on constructor args.. Eg pair, tuples

8. Non-templated function vs templated function

   If there is a templated and non-templated function, compiler chooses non-template function if it's a match

   FORCING compiler to choose template function using <> .. However if you specify that compiler should only consider the function template

```
auto res2 = max<>(5.5, 6.0);
```

9. Return type of templated functions (from C++1_)

   Return type of templated functions can be auto. Compiler will infer the type from return statement in function and does promotions if required.

10. Template class and member functions - replace ClassName with ClassName<T>

```
template <typename T, std::size_t N>
class Array{
public:
    std::size_t getSize() const;
private:
    T elem[N];
};
template <typename T, std::size_t N>   // (1)
std::size_t Array<T, N>::getSize() const {
    return N;
}
```

11. Template class declaration, definition (where type is specified) in file1.cpp, member method definition in file2.cpp

    In order for the compiler to use a template, it must see both the template definition (not just declaration) and the template type used to instantiate the template i.e. don't define member methods in different cpp and class in different cpp.

12. Template specialization

    The template <> tells the compiler that this is a template function, but that there are no template parameters.. Compiler will use specified definition for type specified - precedence over generic version

    Can specialize one or many template parameters for class

13. partial template specialization - works for class NOT FUNCTION

    Note that as of C++14, partial template specialization can only be used with classes, not template functions (functions must be fully specialized)

```
template<typename T, std::size_t n>
class AClass
{
public:
    AClass() { std::cout << "AClass Orig" << std::endl; }
    T avar[n]{};
};

template<>
class AClass<int, 5>
{
public:
    AClass() { std::cout << "AClass Spl1" << std::endl; }
    int avar[5]{};
};

template<typename T> ---> remove non-generic args from template.. Order/Sequence doesn't matter
class AClass<T, 4>
{
public:
    AClass() { std::cout << "AClass Spl2" << std::endl; }
    T avar[4]{};
};


template<std::size_t n> ---> remove non-generic args from template
class AClass<int, n>
{
public:
    AClass() { std::cout << "AClass Spl2" << std::endl; }
    int avar[n]{};
};
```

14. Partial template specialization for pointers

```
template<class T>
Class A{}
template<class T>
Class A<T*>{}

template <typename T>
class AClass
{
```

```
public:
    T avar;
    AClass()
    {
        std::cout << "AClass" << std::endl;
    }
};

template <typename T>
class AClass<T*>
{
public:
    T* aptr;
    AClass(T* ptr) : aptr(ptr)
    {
        std::cout << "AClassptr" << std::endl;
    }
};

int main()
{
    int a = 0;
    AClass <int*>aptr(&a);
}
```

15. Partial template function specialization

To specialize types just enter typename in <> after function-name before (...)

TRY - PARTIAL

```
#include <iostream>

template <typename T1, typename T2>
void startOneEmTimer(T1 a, T2 b)
{
    std::cout << "0:" << std::endl;
}

template <>
void startOneEmTimer<int, float>(int a, float b)
{
    std::cout << "1:" << std::endl;
}

template <typename T1>
void startOneEmTimer<T1, float>(T1 a, float b)
{
    std::cout << "2:" << std::endl;
}

int main()
{
    startOneEmTimer<float, int>((float)10.0, (int)1);
    startOneEmTimer<>((int)1, (float)2.0);
    startOneEmTimer<char, float>('a', 2.0);

```

input

Compilation failed due to following error(s).

```
main.cpp:24:6: error: non-type partial specialization ‘startOneEmTimer’ is not allowed
   24 | void startOneEmTimer<T1, float>(T1 a, float b)
      |      ^~~~~~~~~~~~~~~~~~~~~~~~~~
```

NON-TYPE FULL

```cpp
#include <iostream>

template<int one, int two, int three>
void startOneEmTimer()
{
    std::cout << one << "," << two << "," << three << std::endl;
}

template<>
void startOneEmTimer<1, 2, 3>()
{
    std::cout << "1:" << std::endl;
}

int main()
{
    startOneEmTimer<0, 2, 3>();
    startOneEmTimer<1, 2, 3>();

    return 0;
}
```

input

```
0,2,3
1:
```

```cpp
#include <iostream>

template<int one, int two, int three>
void startOneEmTimer()
{
    std::cout << one << "," << two << "," << three << std::endl;
}

template<int two, int three>
void startOneEmTimer<1, two, three>()
{
    std::cout << "2:" << std::endl;
}

int main()
{
    startOneEmTimer<0, 2, 3>();
    startOneEmTimer<1, 2, 3>();

    return 0;
}
```

input

```
Compilation failed due to following error(s).

main.cpp:18:6: error: non-type partial specialization 'startOneEmTimer<1, two, three>' is not allowed
   18 | void startOneEmTimer<1, two, three>()
      |      ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

16. Variable templates - templates for variables :)

```cpp
template<typename T>
T variable_name{};
```

Use as

```cpp
variable_name<type>
```

17. Variadic templates

```cpp
template<typename T>
T add(const T& arg) //this function is for last recursive call with single argument
{
    return arg;
}

// this function is used for first and second recursive call.
// First call -> int, uint, uint
// Second call -> uint, uint
template<typename T, typename... ARGSTYPE>
T add(const T& arg, ARGSTYPE&... args)
{
    return arg + add(args...);
```

```
}

add(1, 2u, 3u);
```

18. Function **automatic** type deduction - 3 types

```
template <typename T>
void func(ParameterType param);

func(expression);
```

1. If parameterType is value-type then l-value or references decays to value-type only. Const, volatile qualifiers rejected
2. If parameterType is pointer or l-reference-value-type then reference is taken only when T& is used. Const, volatile qualifiers taken.
   a. Only L-values can be passed if reference is non-const
   b. L-values, r-values can be passed if reference is const
3. If parameterType is r-value-reference, if l-value is passed it is converted to l-value-reference-type. Else r-value-reference-type is used

19. Function explicit type deduction - references

If typename is **explicitly stated as a reference by caller but parameterType is mentioned as value-type**, then arg is rxd as a reference.. Else use std::ref

```
#include <iostream>
#include <functional>

template <typename T>
void func(T a)
{
    a.get() = 2; // reference copy
}

template <typename T>
void func1(T& a)
{
    a = 2; // reference copy
}

int main()
{
    int a = 0;
    auto aref = std::ref(a);
    int& aref2 = a;

    std::cout << aref << "," << a << "," << aref2 << std::endl;

    //func(aref);
    func1(aref2);

    std::cout << aref << "," << a << "," << aref2 << std::endl;

    return 0;
}
```

20. Auto/decltype can be used in template parameters for NON-TYPES .. AWESOME..

```
template <auto N>
class MyClass{
    ....
};

template <auto... ns>
class VariadicTemplate{ .... };

template <auto n1, decltype(n1)... ns>
class TypedVariadicTemplate{ .... };
```

21. Use automatic-type deduction as much as possible to avoid narrowing etc

I

22. F

I

23. F

I

24. D

I

25. F

I

26. F

I

27. F

I

28. F

29. F
    I
30. F
    I
31. F
    I
32. F
    I
33. F
    I
34. F
    I
35. D
    I
36. F
    I
37. F
    I
38. F
    I
39. F
40. F
    I
41. F
    I
42. F
    I
43. F
    I
44. F
    I
45. F
    I
46. D
    I
47. F
    I
48. F
    I
49. F
    I
50. F
51. F
    I
52. F
    I
53. F
    I
54. F
    I
55. F
    I