Lab Class Scientific Computing 2022, WISM454

Adriaan Graas, Week 11

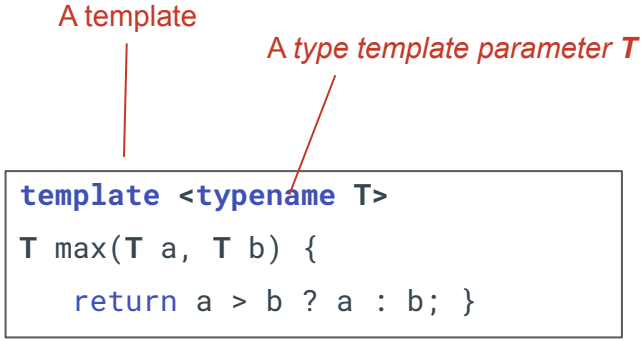
C++ programming

Templates

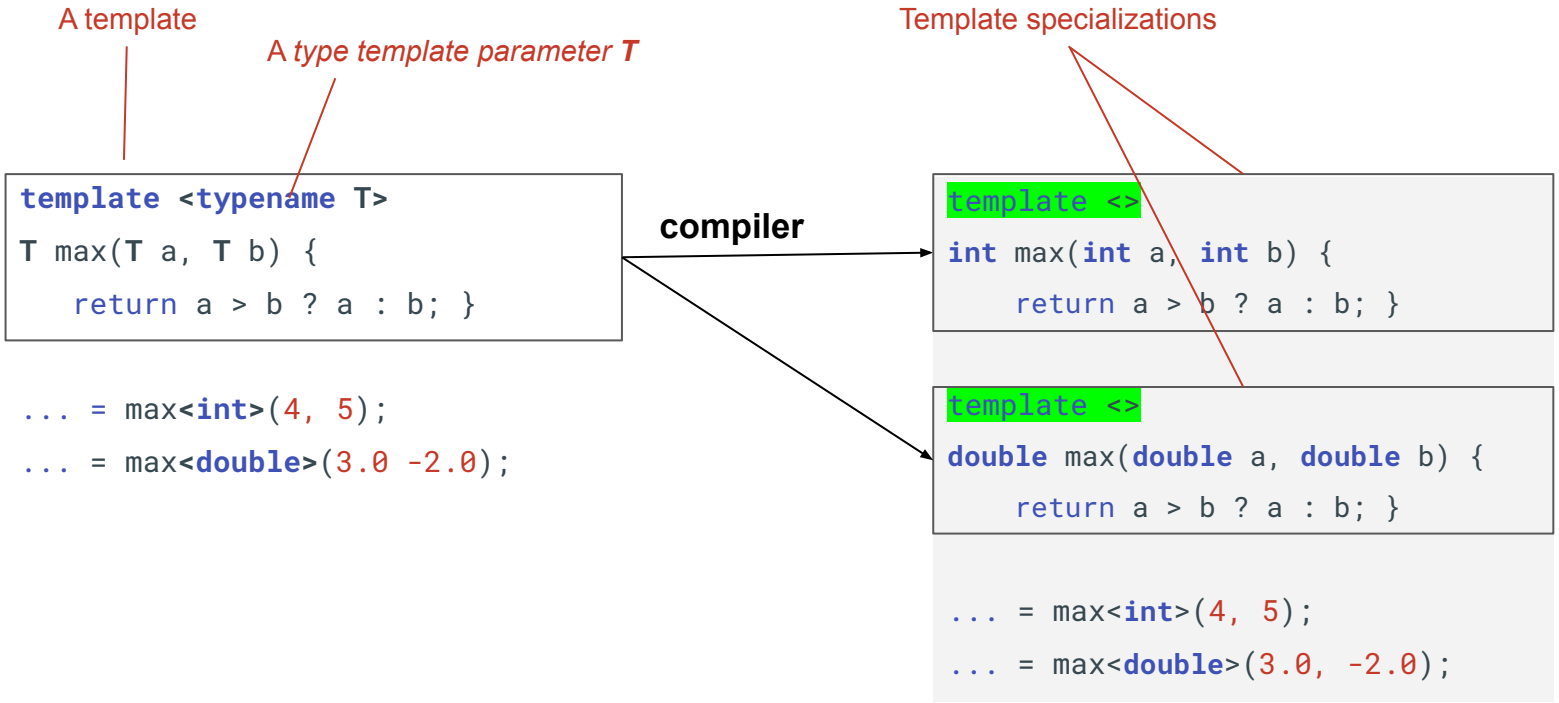
What is template metaprogramming?

- **Templates** are patterns, in which the compiler generates code for each required version of the pattern.
  - **We have already used them:**
    - std::vector<float>, std::vector<double>, std::function<int(int)>
  - **Replaces run-time calculations by compile-type calculations**
    - Idea: faster code, and still using abstractions
    - Downside: longer compile-times, "binary bloat"
  - **Leads to complicated mixes between run-time and compile-time concepts**

- They are an example of **metaprogramming**
  - **Code that generates code**
  - **Like preprocessor macros**




```
... = max<int>(4, 5);
... = max<double>(3.0 -2.0);
```


A template
A type template parameter T
Template specializations
template <typename T>
T max(T a, T b) {
return a > b ? a : b; }
... = max<int>(4, 5);
... = max<double>(3.0 -2.0);
compiler
template <>
int max(int a, int b) {
return a > b ? a : b; }
template <>
double max(double a, double b) {
return a > b ? a : b; }
... = max<int>(4, 5);
... = max<double>(3.0, -2.0);

```
template <typename T>
T max(T a, T b) {
   return a > b ? a : b; }
```

compiler

```
template <>
int max(int a, int b) {
    return a > b ? a : b; }
```

```
template <>
double max(double a, double b) {
    return a > b ? a : b; }
```

```
template <> // explicit specialization
bool max(bool a, bool b) {
   return a || b; }

... = max<int>(4, 5);
... = max<double>(3.0 -2.0);
```

```
template <>
bool max(bool a, bool b) {
   return a || b; }

... = max<int>(4, 5);
... = max<double>(3.0, -2.0);
```

Specializations

- Specializations are generated *when the code compiles*
  - **Put all template code in headers**
    - Allows the file *#include*-ing the header to compile the template to the right specialization.
  - **Or: specify which specializations are required at compile-time**
    - This is called *explicit instantiation*.

- If one of the specializations does not compile (incorrect *T*), it is ignored
  - **"Substitution Failure Is Not An Error" (SFINAE)**
    - Can be useful for tricking the compiler into using the right specialization.

Template parameters

```
// - type template parameter
namespace std {
  template <typename T>
  class vector : ... { ... };
}

// - non-type template parameter
template <int N = 1>
class Factorial { ... };

// - template template parameter
template <template<typename> typename T>
class Map {
    T<double> ...;
    T<float> ...;
};
```

Template arguments

```
std::vector<int> quantities = {4, 4, 4};

Factorial<3> f3 {};

// contains a std::vector<double> and std::vector<float>
Map<std::vector> matrix;
```

Template classes

```
template <typename T>
class Foo {
public:
    template <int N>                // a templated function in a templated class
    void bar(T arg, N number) {
        ...
    }
};
```

Template class with inheritance

```
template <typename T>
class Derived : public Base<T> {    // pass template param to base class
public:
    Derived() : Base<T>() {}
    T foo(T arg) {                  // take a type `T` as argument, returns a `T`
        this->member_from_base_;    // use the pointer `this` to access base members
    }
protected:
    T member_;                      // a member of type `T`
};
```

Computing N factorial with templates

```
// https://www.modernescpp.com/index.php/c-core-guidelines-rules-for-template-metaprogramming
template<int N>
class Factorial {
public:
    static const int value = N * Factorial<N-1>::value;
};

template<> // template specialization
class Factorial<1> {
public:
    static int const value = 1;
};

int main() {
    std::cout << Factorial<5>::value << std::endl;  // "120"
    std::cout << Factorial<10>::value << std::endl; // "3628820" (see godbolt.org for assembly)
}
```

This week

- This week/next week:
  - **Try to finish second part of 3.2**
  - **Introduction into template programming**
    - Replace std::vector with a fixed-size std::array
    - Use templates to avoid dynamic dispatch
    - Use templates to speed up vector expressions
- Next week:
  - **No new exercises**