Some Advanced C++ Topics

Templates

- What are templates and why do we need them?

Templates

- What are templates and why do we need them?

```
int mySum(const int& a, const int& b){
    return a+b;
}
```

Templates

- What are templates and why do we need them?

```
int mySum(const int& a, const int& b){
        return a+b;
 }
```

```
double mySum(const double& a, const int& b){
        return a+b;
 }
```

```
double mySum(const double& a, const int& b){
        return a+b;
 }
```

```
double mySum(const int& a, const double& b){
        return a+b;
 }
```

```
MyClass mySum(const MyClass& a, const MyClass& b){
        return a+b;
 }
```

Templates

- What are templates and why do we need them?

```
int mySum(const int& a, const int& b){
        return a+b;
 }
```

```
double mySum(const double& a, const int& b){
        return a+b;
 }
```

```
double mySum(const double& a, const int& b){
        return a+b;
 }
```

```
double mySum(const int& a, const double& b){
        return a+b;
 }
```

```
MyClass mySum(const MyClass& a, const MyClass& b){
        return a+b;
 }
```

Templates

- What are templates and why do we need them?

```
template <typename T>
T mySum(const T& a, const T& b){
    return a+b;
}
```

Templates

Now we can use the function like this

```
int a{4}; double b{3.2};
mySum(a,a);
mySum(b,b);
MySum<double>(a,b);
```

Templates

What if we want 2 template parameters?

```
template <typename T1, typename T2>
void myFunction(const T1, T2&&){
 ... something happening ...
}
```

Templates

Note:

The process of replacing the template parameters with the actual ones is called template instantiation.

Each instance of a templated function is a different function created by the compiler.

Class Templates

We can template classes

```
template <typename T>
class MyClass{
public:
    T field1;
    std::vector<T> data;
    MyClass(size_t s);
    void print();
};
```

```
template <typename T>
void MyClass<T>::print(){

    std::cout<<field1<<std::endl;
    for(auto x:data){
        std::cout<<x<<" ";
    }
    std::cout<<std::endl;
}
```

```
MyClass<int> var;
```

Class Templates

What if a member function needs a template too?

Inside the class declaration:

```
template <typename M>
  void printTogetherWithSomething(M p);
```

For the implementation:

```
template <typename T>
template <typename M>
void MyClass<T>::printTogetherWithSomething(M p){
    ...
}
```

Variadic Templates

What if there needs to be many parameters and no one knows how many? This function will print arbitrary number of anything that has operator << defined:

```
void myPrint(){
	std::cout<<std::endl;
}

template <typename T, typename... Types>
void myPrint(T arg, Types... args){
	std::cout<<arg<<" ";
	myPrint(args...);
}
```

Variadic Templates

A more neat way to write the same function (C++17 standard):

```
template <typename T, typename... Types>
void myPrint(T arg, Types... args){
    std::cout<<arg<<" ";
    if constexpr (sizeof...(args)>0){
        myPrint(args...);
    }else{
        std::cout<<std::endl;
    }
}
```

Nontype Template Parameters

Template parameters do not have to only be types:

```
Template <int Something, typename T>
T addSomething(const T& x){

    return T+x;

}
```

Template Specialization

We might need to specialize our templates, for example when working with C library functions which only take certain types (like MPI functions). Suppose we have a class:

```
template <typename T>
class MyClass{
public:
    T x;
    //does something long with x
    //needs to call special function based on T
    void ExampleFunction();

};
```

Template Specialization

There are 2 ways to do this:

before c++17:

```
template<>
void MyClass<int>::ExampleFunction(){
        // ... does something long here  ...
        IntOnlyFunction(x);
}
```

Template Specialization

There are 2 ways to do this:

before c++17:

```
template<>
void MyClass<int>::ExampleFunction(){
        // ... does something long here  ...
        IntOnlyFunction(x);
}
```

Might lead to too much code duplication

Template Specialization

There are 2 ways to do this:

before c++17:

```
template<>
void MyClass<int>::ExampleFunction(){
        // ... does something long here  ...
        IntOnlyFunction(x);
}
```

Might lead to too much code duplication

after c++17:

```
template <typename T>
void MyClass<T>::ExampleFunction(){
        // … does something long here ...
        if constexpr(std::is_same_v<int,T>){
                        IntOnlyFunction(x);
        }
}
```

Template Specialization

There are 2 ways to do this:

before c++17:

```
template<>
void MyClass<int>::ExampleFunction(){
        // ... does something long here  ...
        IntOnlyFunction(x);
}
```

Might lead to too much code duplication

after c++17:

```
template <typename T>
void MyClass<T>::ExampleFunction(){
        // … does something long here ...
        if constexpr(std::is_same_v<int,T>){
                        IntOnlyFunction(x);
        }
}
```

This part is only coded once

Error Novels

Compilers generate “Error Novels” when encountering mistakes in the templated code. Our examples are too small, let’s abuse the << operator and compile this with gcc:

```
struct MyTestStruct{
        int a;
};

int main(int argc, char *argv[]){

        MyTestStruct t;
        t.a=27;
        std::cout<<t<<std::endl;

}
```

Literature

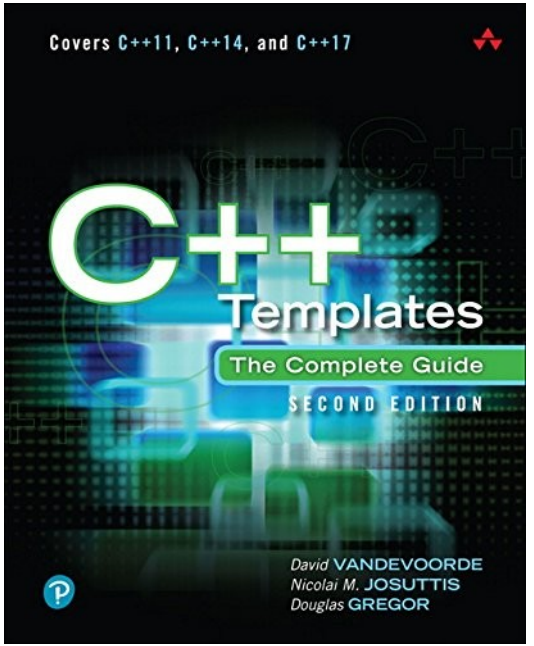
We have learned all the essential parts,

but left out numerous fine details.

Find them all in the wonderful book of

D. Vandevoorde, N. Josuttis and D. Gregor:

"C++ Templates. The complete Guide"



Exercises

1. Implement a function that calculates the sum of elements of a vector containing values of any type that has a `+ operator` defined (do not use STL algorithms even if you know what that is).

2. Implement a class that has 2 fields of any numerical type and a function that returns the sum of both of them and an int or double variable given as a parameter.

3. Implement a function that calculates a sum of any number of integer and double arguments. Test it by calling `myVSum(7, 8.4, 1)` and `myVSum(1.2,4,9.7, 5, 1)`.

4. Write a `constexpr` templated function that takes no parameters except template ones and calculates factorial at compile time. Use `static_assert` to test your function.

A little reminder:

```
using namespace std;
```

A little reminder:

using namespace std;