COMP6771
Advanced C++ Programming

Week 7.1
Templates Intro

In this lecture

Why?

- Understanding compile time polymorphism in the form of templates helps understand the workings of C++ on generic types

What?

- Templates
- Non-type parameters
- Inclusion exclusion principle
- Classes, statics, friends

Polymorphism & Generic Programming

- **Polymorphism:** Provision of a single interface to entities of different types
- Two types - :
  - Static (our focus):
    - Function overloading
    - Templates (i.e. generic programming)
      - std::vector<int>
      - std::vector<double>
  - Dynamic:
    - Related to virtual functions and inheritance - see week 9
- **Genering Programming:** Generalising software components to be independent of a particular type
  - STL is a great example of generic programming

Function Templates

Without generic programming, to create two logically identical functions that behave in a way that is independent to the type, we have to rely on function overloading.

```cpp
#include <iostream>

auto min(int a, int b) -> int {
        return a < b ? a : b;
}

auto min(double a, double b) -> double{
        return a < b ? a : b;
}

auto main() -> int {
        std::cout << min(1, 2) << "\n"; // calls line 1
        std::cout << min(1.0, 2.0) << "\n"; // calls line 4
}
```

demo701-functemp1.cpp

Explore how this looks in Compiler Explorer

Function Templates

- Function template: Prescription (i.e. instruction) for the compiler to generate particular instances of a function varying by type
  - The generation of a templated function for a particular type T only happens when a call to that function is seen during compile time

```cpp
#include <iostream>

template <typename T>
auto min(T a, T b) -> T {
        return a < b ? a : b;
}

auto main() -> int {
        std::cout << min(1, 2) << "\n"; // calls int min(int, int)
        std::cout << min(1.0, 2.0) << "\n"; // calls double min(double, double)
}
```

demo702-functemp2.cpp

Explore how this looks in Compiler Explorer

Some Terminology

template type parameter

template parameter list

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

Type and Nontype Parameters

- **Type parameter:** Unknown type with no value
- **Nontype parameter:** Known type with unknown value

```cpp
#include <array>
#include <iostream>

template<typename T, std::size_t size>
auto findmin(const std::array<T, size> a) -> T {
	T min = a[0];
	for (std::size_t i = 1; i < size; ++i) {
		if (a[i] < min)
			min = a[i];
	}
	return min;
}

auto main() -> int {
	std::array<int, 3> x{3, 1, 2};
	std::array<double, 4> y{3.3, 1.1, 2.2, 4.4};
	std::cout << "min of x = " << findmin(x) << "\n";
	std::cout << "min of x = " << findmin(y) << "\n";
}
```

demo703-nontype1.cpp

Compiler deduces **T** and **size** from **a**

Type and Nontype Parameters

- The above example generates the following functions at compile time
- What is "code explosion"? Why do we have to be weary of it?

```cpp
auto findmin(const std::array<int, 3> a) -> int {
        int min = a[0];
        for (int i = 1; i < 3; ++i) {
                if (a[i] < min)
                        min = a[i];
        }
        return min;
}

auto findmin(const std::array<double, 4> a) -> double {
        double min = a[0];
        for (int i = 1; i < 4; ++i) {
                if (a[i] < min)
                        min = a[i];
        }
        return min;
}
```

demo704-nontype2.cpp

Class Templates

- How we would currently make a Stack type
- Issues?
  - Administrative nightmare
  - Lexical complexity (need to learn all type names)

```cpp
class int_stack {
public:
        auto push(int&) -> void;
        auto pop() -> void;
        auto top() -> int&;
        auto top() const -> const int&;
private:
        std::vector<int> stack_;
};
```

```cpp
class double_stack {
public:
        auto push(double&) -> void;
        auto pop() -> void;
        auto top() -> double&;
        auto top() const -> const double&;
private:
        std::vector<double> stack_;
};
```

Class Templates

Creating our first class template

```cpp
// stack.h
#ifndef STACK_H
#define STACK_H

#include <iostream>
#include <vector>

template<typename T>
class stack {
public:
	friend auto operator<<(std::ostream& os, const stack& s) -> std::ostream& {
		for (const auto& i : s.stack_)
			os << i << " ";
		return os;
	}
	auto push(T const& item) -> void;
	auto pop() -> void;
	auto top() -> T&;
	auto top() const -> const T&;
	auto empty() const -> bool;

private:
	std::vector<T> stack_;
};

#include "./demo705-classtemp.tpp"

#endif // STACK_H
```

demo705-classtemp-main.h

```cpp
#include "./demo705-classtemp.h"

template<typename T>
auto stack<T>::push(T const& item) -> void {
	stack_.push_back(item);
}

template<typename T>
auto stack<T>::pop() -> void {
	stack_.pop_back();
}

template<typename T>
auto stack<T>::top() -> T& {
	return stack_.back();
}

template<typename T>
auto stack<T>::top() const -> T const& {
	return stack_.back();
}

template<typename T>
auto stack<T>::empty() const -> bool {
	return stack_.empty();
}
```

demo705-classtemp-main.tpp

https://en.cppreference.com/w/cpp/language/friend#Template_friends

Class Templates

```cpp
#include <iostream>
#include <string>

#include "./demo705-classtemp.h"

int main() {
        stack<int> s1; // int: template argument
        s1.push(1);
        s1.push(2);
        stack<int> s2 = s1;
        std::cout << s1 << s2 << '\n';
        s1.pop();
        s1.push(3);
        std::cout << s1 << s2 << '\n';
        // s1.push("hello"); // Fails to compile.

        stack<std::string> string_stack;
        string_stack.push("hello");
        // string_stack.push(1); // Fails to compile.
}
```

demo705-classtemp-main.cpp

Class Templates

Default rule-of-five (you don't have to implement these in this case)

```
template <typename T>
stack<T>::stack() { }

template <typename T>
stack<T>::stack(const stack<T> &s) : stack_{s.stack_} { }

template <typename T>
stack<T>::stack(Stack<T> &&s) : stack_(std::move(s.stack_)); { }

template <typename T>
stack<T>& stack<T>::operator=(const stack<T> &s) {
  stack_ = s.stack_;
}

template <typename T>
stack<T>& stack<T>::operator=(stack<T> &&s) {
  stack_ = std::move(s.stack_);
}

template <typename T>
stack<T>::~stack() { }
```

Inclusion compilation model

- What is wrong with this?
- g++ min.cpp main.cpp -o main

min.h

```cpp
template <typename T>
auto min(T a, T b) -> T;
```

min.cpp

```cpp
template <typename T>
auto min(T a, T b) -> int {
        return a < b ? a : b;
}
```

main.cpp

```cpp
#include <iostream>

auto main() -> int {
        std::cout << min(1, 2) << "\n";
}
```

Inclusion compilation model

- When it comes to templates, we include definitions (i.e. implementation) in the .h file
  - This is because template definitions need to be known at **compile time** (template definitions can't be instantiated at link time because that would require an instantiation for all types)
- Will expose implementation details in the .h file
- Can cause slowdown in compilation as every file using min.h will have to instantiate the template, then it's up the linker to ensure there is only 1 instantiation.

min.h

```cpp
template <typename T>
auto min(T a, T b) -> T {
  return a < b ? a : b;
}
```

main.cpp

```cpp
#include <iostream>

auto main() -> int {
  std::cout << min(1, 2) << "\n";
}
```

Inclusion compilation model

- Alternative: Explicit instantiations
- Generally a bad idea

min.h

```cpp
template <typename T>
T min(T a, T b);
```

min.cpp

```cpp
template <typename T>
auto min(T a, T b) -> T {
        return a < b ? a : b;
}

template int min<int>(int, int);
template double min<double>(double, double);
```

main.cpp

```cpp
#include <iostream>

auto main() -> int {
        std::cout << min(1, 2) << "\n";
        std::cout << min(1.0, 2.0) << "\n";
}
```

Inclusion compilation model

- Lazy instantiation: Only members functions that are called are instantiated
  - In this case, pop() will not be instantiated
- Exact same principles will apply for classes
- Implementations must be in header file, and compiler should only behave as if one Stack<int> was instantiated

main.cpp

```cpp
auto main() -> int {
	stack<int> s;
	s.push(5);
}
```

stack.h

```cpp
#include <vector>

template <typename T>
class stack {
public:
	stack() {}
	auto pop() -> void;
	auto push(const T& i) -> void;
private:
	std::vector<T> items_;
}

template <typename T>
auto stack<T>::pop() -> void {
	items_.pop_back();
}

template <typename T>
auto stack<T>::push(const T& i) -> void {
	items_.push_back(i);
}
```

Static Members

```cpp
#include <vector>

template<typename T>
class stack {
public:
	stack();
	~stack();
	auto push(T&) -> void;
	auto pop() -> void;
	auto top() -> T&;
	auto top() const -> const T&;
	static int num_stacks_;

private:
	std::vector<T> stack_;
};

template<typename T>
int stack<T>::num_stacks_ = 0;

template<typename T>
stack<T>::stack() {
	num_stacks_++;
}

template<typename T>
stack<T>::~stack() {
	num_stacks_--;
}
```

demo706-static.h

Each template instantiation has it's own set of static members

```cpp
#include <iostream>

#include "./demo706-static.h"

auto main() -> int {
	stack<float> fs;
	stack<int> is1, is2, is3;
	std::cout << stack<float>::num_stacks_ << "\n";
	std::cout << stack<int>::num_stacks_ << "\n";
}
```

demo706-static.cpp

Friends

Each stack instantiation has one unique instantiation of the friend

```
#include <iostream>
#include <vector>

template<typename T>
class stack {
public:
	auto push(T const&) -> void;
	auto pop() -> void;

	friend auto operator<<(std::ostream& os, stack<T> const& s) -> std::ostream& {
		return os << "My top item is " << s.stack_.back() << "\n";
	}

private:
	std::vector<T> stack_;
};

template<typename T>
auto stack<T>::push(T const& t) -> void {
	stack_.push_back(t);
}
```

demo707-friend.h

```
#include <iostream>
#include <string>

#include "./stack.h"

auto main() -> int {
	stack<std::string> ss;
	ss.push("Hello");
	std::cout << ss << "\n":

	stack<int> is;
	is.push(5);
	std::cout << is << "\n":
}
```

demo707-friend.cpp

(Unrelated) Constexpr

- We can provide default arguments to template types (where the defaults themselves are types)
- It means we have to update all of our template parameter lists

Constexpr

- Either:
  - A variable that can be calculated at compile time
  - A function that, if its inputs are known at compile time, can be run at compile time

```cpp
#include <iostream>

constexpr int constexpr_factorial(int n) {
	return n <= 1 ? 1 : n * constexpr_factorial(n - 1);
}

int factorial(int n) {
	return n <= 1 ? 1 : n * factorial(n - 1);
}

auto main() -> int {
	// Beats a #define any day.
	constexpr int max_n = 10;
	constexpr int tenfactorial = constexpr_factorial(10);

	// This will fail to compile
	int ninefactorial = factorial(9);

	std::cout << max_n << "\n";
	std::cout << tenfactorial << "\n";
	std::cout << ninefactorial << "\n";
}
```

demo708-constexpr.cpp

Constexpr (Benefits)

- Benefits:
  - Values that can be determined at compile time mean less processing is needed at runtime, resulting in an overall faster program execution
  - Shifts potential sources of errors to compile time instead of runtime (easier to debug)

Feedback