

COMP6771

Advanced C++ Programming

Week 7.1

Templates Intro

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 8
- **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.

```
1 #include <iostream>
2
3 auto min(int a, int b) -> int {
4     return a < b ? a : b;
5 }
6
7 auto min(double a, double b) -> double{
8     return a < b ? a : b;
9 }
10
11 auto main() -> int {
12     std::cout << min(1, 2) << "\n"; // calls line 1
13     std::cout << min(1.0, 2.0) << "\n"; // calls line 4
14 }
```

demo701-functemp1.cpp

Explore how this looks in [Compiler Explorer](#)

Function Templates

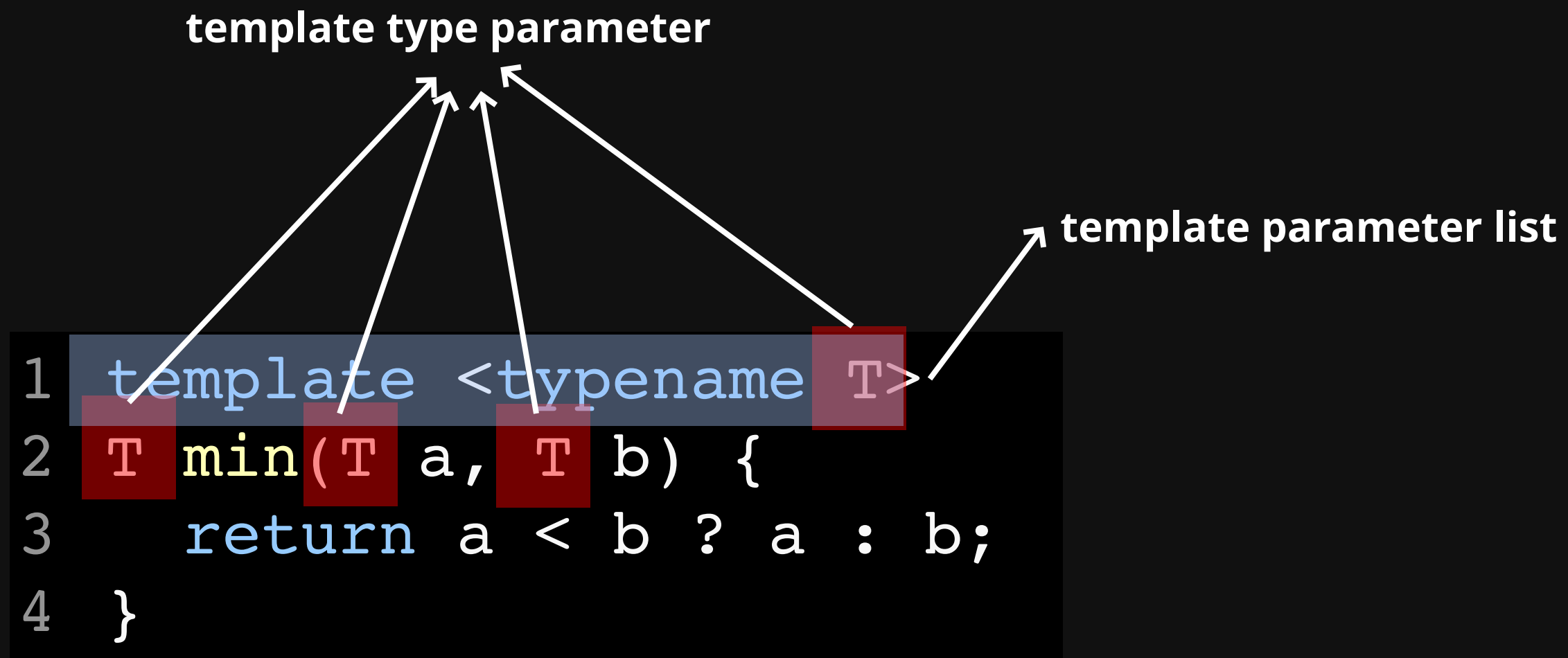
- Function template: Prescription (i.e. instruction) for the compiler to generate particular instances of a function varying by type
 - The generation of a templated function for a particular type T only happens when a call to that function is seen during compile time

```
1 #include <iostream>
2
3 template <typename T>
4 auto min(T a, T b) -> T {
5     return a < b ? a : b;
6 }
7
8 auto main() -> int {
9     std::cout << min(1, 2) << "\n"; // calls int min(int, int)
10    std::cout << min(1.0, 2.0) << "\n"; // calls double min(double, double)
11 }
```

demo702-functemp2.cpp

Explore how this looks in [Compiler Explorer](#)

Some Terminology



Type and Nontype Parameters

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

```
1 #include <array>
2 #include <iostream>
3
4 template<typename T, std::size_t size>
5 auto findmin(const std::array<T, size> a) -> T {
6     T min = a[0];
7     for (std::size_t i = 1; i < size; ++i) {
8         if (a[i] < min)
9             min = a[i];
10    }
11    return min;
12 }
13
14 auto main() -> int {
15     std::array<int, 3> x{3, 1, 2};
16     std::array<double, 4> y{3.3, 1.1, 2.2, 4.4};
17     std::cout << "min of x = " << findmin(x) << "\n";
18     std::cout << "min of x = " << findmin(y) << "\n";
19 }
```

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?

```
1 auto findmin(const std::array<int, 3> a) -> int {
2     int min = a[0];
3     for (int i = 1; i < 3; ++i) {
4         if (a[i] < min)
5             min = a[i];
6     }
7     return min;
8 }
9
10 auto findmin(const std::array<double, 4> a) -> double {
11     double min = a[0];
12     for (int i = 1; i < 4; ++i) {
13         if (a[i] < min)
14             min = a[i];
15     }
16     return min;
17 }
```

demo704-nontype2.cpp

Class Templates

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

```
1 class int_stack {
2 public:
3     auto push(int&) -> void;
4     auto pop() -> void;
5     auto top() -> int&;
6     auto top() const -> const int&;
7 private:
8     std::vector<int> stack_;
9 };
```

```
1 class double_stack {
2 public:
3     auto push(double&) -> void;
4     auto pop() -> void;
5     auto top() -> double&;
6     auto top() const -> const double&;
7 private:
8     std::vector<double> stack_;
9 };
```


Class Templates

- Creating our first class template

```
1 // stack.h
2 #ifndef STACK_H
3 #define STACK_H
4
5 #include <iostream>
6 #include <vector>
7
8 template<typename T>
9 class stack {
10 public:
11     friend auto operator<<(std::ostream& os, const stack& s) -> std::ostream& {
12         for (const auto& i : s.stack_)
13             os << i << " ";
14         return os;
15     }
16     auto push(T const& item) -> void;
17     auto pop() -> void;
18     auto top() -> T&;
19     auto top() const -> const T&;
20     auto empty() const -> bool;
21
22 private:
23     std::vector<T> stack_;
24 };
25
26 #include "../demo705-classtemp.hpp"
27
28 #endif // STACK_H
```

demo705-classtemp-main.h

```
1 #include "../demo705-classtemp.h"
2
3 template<typename T>
4 auto stack<T>::push(T const& item) -> void {
5     stack_.push_back(item);
6 }
7
8 template<typename T>
9 auto stack<T>::pop() -> void {
10     stack_.pop_back();
11 }
12
13 template<typename T>
14 auto stack<T>::top() -> T& {
15     return stack_.back();
16 }
17
18 template<typename T>
19 auto stack<T>::top() const -> const T& {
20     return stack_.back();
21 }
22
23 template<typename T>
24 auto stack<T>::empty() const -> bool {
25     return stack_.empty();
26 }
```

demo705-classtemp-main.hpp

Class Templates

```
1 #include <iostream>
2 #include <string>
3
4 #include "../demo705-classtemp.h"
5
6 int main() {
7     stack<int> s1; // int: template argument
8     s1.push(1);
9     s1.push(2);
10    stack<int> s2 = s1;
11    std::cout << s1 << s2 << '\n';
12    s1.pop();
13    s1.push(3);
14    std::cout << s1 << s2 << '\n';
15    // s1.push("hello"); // Fails to compile.
16
17    stack<std::string> string_stack;
18    string_stack.push("hello");
19    // string_stack.push(1); // Fails to compile.
20 }
```

demo705-classtemp-main.cpp

Class Templates

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

```
1  template <typename T>
2  stack<T>::stack() { }
3
4  template <typename T>
5  stack<T>::stack(const stack<T> &s) : stack_{s.stack_} { }
6
7  template <typename T>
8  stack<T>::stack(Stack<T> &&s) : stack_(std::move(s.stack_)); { }
9
10 template <typename T>
11 stack<T>& stack<T>::operator=(const stack<T> &s) {
12     stack_ = s.stack_;
13 }
14
15 template <typename T>
16 stack<T>& stack<T>::operator=(stack<T> &&s) {
17     stack_ = std::move(s.stack_);
18 }
19
20 template <typename T>
21 stack<T>::~~stack() { }
```

Inclusion compilation model

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

min.h

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

min.cpp

```
1 template <typename T>
2 auto min(T a, T b) -> int {
3     return a < b ? a : b;
4 }
```

main.cpp

```
1 #include <iostream>
2
3 auto main() -> int {
4     std::cout << min(1, 2) << "\n";
5 }
```

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

```
1 template <typename T>
2 auto min(T a, T b) -> T {
3     return a < b ? a : b;
4 }
```

main.cpp

```
1 #include <iostream>
2
3 auto main() -> int {
4     std::cout << min(1, 2) << "\n";
5 }
```

Inclusion Compilation Model

- Alternative: Explicit instantiations
- **Generally a bad idea**

min.h

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

min.cpp

```
1 template <typename T>
2 auto min(T a, T b) -> T {
3     return a < b ? a : b;
4 }
5
6 template int min<int>(int, int);
7 template double min<double>(double, double);
```

main.cpp

```
1 #include <iostream>
2
3 auto main() -> int {
4     std::cout << min(1, 2) << "\n";
5     std::cout << min(1.0, 2.0) << "\n";
6 }
```

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

stack.h

```
1 #include <vector>
2
3 template <typename T>
4 class stack {
5 public:
6     stack() {}
7     auto pop() -> void;
8     auto push(const T& i) -> void;
9 private:
10     std::vector<T> items_;
11 }
12
13 #include "stack.cpp"
```

stack.cpp

```
1 template <typename T>
2 auto stack<T>::pop() -> void {
3     items_.pop_back();
4 }
5
6 template <typename T>
7 auto stack<T>::push(const T& i) -> void {
8     items_.push_back(i);
9 }
```

main.cpp

```
1 auto main() -> int {
2     stack<int> s;
3     s.push(5);
4 }
```

Static Members

```
1 #include <vector>
2
3 template<typename T>
4 class stack {
5 public:
6     stack();
7     ~stack();
8     auto push(T&) -> void;
9     auto pop() -> void;
10    auto top() -> T&;
11    auto top() const -> const T&;
12    static int num_stacks_;
13
14 private:
15     std::vector<T> stack_;
16 };
17
18 template<typename T>
19 int stack<T>::num_stacks_ = 0;
20
21 template<typename T>
22 stack<T>::stack() {
23     num_stacks_++;
24 }
25
26 template<typename T>
27 stack<T>::~~stack() {
28     num_stacks_--;
29 }
```

demo706-static.h

- Each template instantiation has its own set of static members

```
1 #include <iostream>
2
3 #include "../demo706-static.h"
4
5 auto main() -> int {
6     stack<float> fs;
7     stack<int> is1, is2, is3;
8     std::cout << stack<float>::num_stacks_ << "\n";
9     std::cout << stack<int>::num_stacks_ << "\n";
10 }
```

demo706-static.cpp

Friends

- Each stack instantiation has one unique instantiation of the friend

```
1 #include <iostream>
2 #include <vector>
3
4 template<typename T>
5 class stack {
6 public:
7     auto push(T const&) -> void;
8     auto pop() -> void;
9
10    friend auto operator<<(std::ostream& os, stack<T> const& s) -> std::ostream& {
11        return os << "My top item is " << s.stack_.back() << "\n";
12    }
13
14 private:
15     std::vector<T> stack_;
16 };
17
18 template<typename T>
19 auto stack<T>::push(T const& t) -> void {
20     stack_.push_back(t);
21 }
```

demo707-friend.h

```
1 #include <iostream>
2 #include <string>
3
4 #include "../stack.h"
5
6 auto main() -> int {
7     stack<std::string> ss;
8     ss.push("Hello");
9     std::cout << ss << "\n":
10
11     stack<int> is;
12     is.push(5);
13     std::cout << is << "\n":
14 }
```

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

```
1 #include <iostream>
2
3 // This can be called at compile time, or at runtime
4 constexpr int constexpr_factorial(int n) {
5     return n <= 1 ? 1 : n * constexpr_factorial(n - 1);
6 }
7
8 // This may not be called at compile time
9 int factorial(int n) {
10     return n <= 1 ? 1 : n * factorial(n - 1);
11 }
12
13 auto main() -> int {
14     // Beats a #define any day.
15     constexpr int max_n = 10;
16
17     constexpr int tenfactorial = constexpr_factorial(10);
18
19     // This will fail to compile
20     int ninefactorial = factorial(9);
21
22     std::cout << max_n << "\n";
23     std::cout << tenfactorial << "\n";
24     std::cout << ninefactorial << "\n";
25 }
```

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)