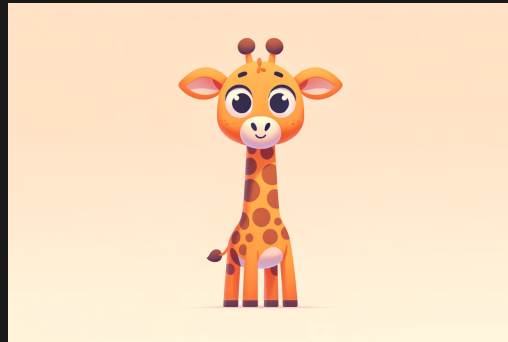


COMP6771

Templates Intro

Lecture 8.1

Author(s): Hayden Smith



[\(Download as PDF\)](#)

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.

```
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 }
```

function-templates.cpp



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 }
```

[function-templates2.cpp](#)

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



Function Templates

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

T is a template type parameter

template <typename> is the template parameter list



Type And Nontype Parameters

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

```
1 #include <iostream>
2 #include <array>
3 #include <vector>
4
5 int main () {
6
7     const auto num = std::size_t{3};
8     std::array<int, num>{1, 2};
9 }
```

type-non-type.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 }
```

type-non-type2.cpp



Class Templates

How we would currently make a Stack of multiple types?

```
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 };
```

Problems: Administrative nightmare, lexical complexity

Compiler deduces T and size from a



Class Templates

Creating our first

```
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 "./class-template-start2.cpp"
27
28 #endif // STACK_H
```

class-template-start1.h

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

class-template-start2.cpp



Class Templates

Creating our first

```
1 #include <iostream>
2 #include <string>
3
4 #include "../class-template-start2.cpp"
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 }
```

[class-template-start-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 code?

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

inc-com-min.cpp

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

inc-com-min.h

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

inc-com-main.cpp

```
clang++ min.cpp main.cpp -o main
```



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.

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

inc-com-min.h

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

inc-com-main.cpp

```
clang++ min.cpp main.cpp -o main
```



Inclusion Compilation Model

- Alternative: Explicit instantiations

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

inc-com-min.h

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

inc-com-minexp.cpp

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

inc-com-main.cpp



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

```
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 template <typename T>
14 auto stack<T>::pop() -> void {
15     items_.pop_back();
16 }
17
18 template <typename T>
19 auto stack<T>::push(const T& i) -> void {
20     items_.push_back(i);
21 }
22
23 auto main() -> int {
24     stack<int> s;
25     s.push(5);
26 }
```

`inc-com-stack.cpp`



Static Members

Each template instantiation has its own set of static members

```
1 #include <iostream>
2 #include <vector>
3
4 template<typename T>
5 class stack {
6 public:
7     stack();
8     ~stack();
9     auto push(T&) -> void;
10    auto pop() -> void;
11    auto top() -> T&;
12    auto top() const -> const T&;
13    static int num_stacks_;
14
15 private:
16     std::vector<T> stack_;
17 };
18
19 template<typename T>
20 int stack<T>::num_stacks_ = 0;
21
22 template<typename T>
23 stack<T>::stack() {
24     num_stacks_++;
25 }
26
27 template<typename T>
28 stack<T>::~~stack() {
29     num_stacks_--;
30 }
31
32 auto main() -> int {
33     stack<float> fs;
34     stack<int> is1, is2, is3;
35     std::cout << stack<float>::num_stacks_ << "\n";
36     std::cout << stack<int>::num_stacks_ << "\n";
37 }
```

static.cpp

Friends

Each stack instantiation has one unique instantiation of the friend

```
1 #include <string>
2 #include <iostream>
3 #include <vector>
4
5 template<typename T>
6 class stack {
7 public:
8     auto push(T const&) -> void;
9     auto pop() -> void;
10
11     friend auto operator<<(std::ostream& os, stack<T> const& s) -> std::ostream& {
12         return os << "My top item is " << s.stack_.back() << "\n";
13     }
14
15 private:
16     std::vector<T> stack_;
17 };
18
19 template<typename T>
20 auto stack<T>::push(T const& t) -> void {
21     stack_.push_back(t);
22 }
23
24 auto main() -> int {
25     stack<std::string> ss;
26     ss.push("Hello");
27     std::cout << ss << "\n";
28
29     stack<int> is;
30     is.push(5);
31     std::cout << is << "\n";
32 }
```

friend.cpp



Default Members

- 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

```
1 #include <vector>
2 #include <iostream>
3
4 template<typename T, typename CONT = std::vector<T>>
5 class stack {
6 public:
7     stack();
8     ~stack();
9     auto push(T&) -> void;
10    auto pop() -> void;
11    auto top() -> T&;
12    auto top() const -> T const&;
13    static int num_stacks_;
14
15 private:
16     CONT stack_;
17 };
18
19 template<typename T, typename CONT>
20 int stack<T, CONT>::num_stacks_ = 0;
21
22 template<typename T, typename CONT>
23 stack<T, CONT>::stack() {
24     num_stacks_++;
25 }
26
27 template<typename T, typename CONT>
28 stack<T, CONT>::~~stack() {
29     num_stacks_--;
30 }
31
32 auto main() -> int {
33     auto fs = stack<float>{};
34     stack<int> is1, is2, is3;
35     std::cout << stack<float>::num_stacks_ << "\n";
36     std::cout << stack<int>::num_stacks_ << "\n";
37 }
```

default-members.cpp



Specialisation

- The templates we've defined so far are completely generic
- There are two ways we can redefine our generic types for something more specific:
 - **Partial specialisation:**
 - Describing the template for another form of the template
 - T^*
 - `std::vector<T>`
 - **Explicit specialisation**
 - Describing the template for a specific, non-generic type
 - `std::string`
 - `int`



Specialisation

When to specialise

- You need to preserve existing semantics for something that would not otherwise work
 - `std::is_pointer` is partially specialised over pointers
- You want to write a type trait
 - `std::is_integral` is fully specialised for `int`, `long`, etc.
- There is an optimisation you can make for a specific type
 - `std::vector<bool>` is fully specialised to reduce memory footprint



Specialisation

When NOT to specialise

- Don't specialise functions
 - A function cannot be partially specialised
 - Fully specialised functions are better done with overloads
 - Herb Sutter has an article on this
 - <http://www.gotw.ca/publications/mill17.htm>
- You think it would be cool if you changed some feature of the class for a specific type
 - People assume a class works the same for all types
 - Don't violate assumptions!



A Stack Template

- Here is our stack template class
 - stack.h
 - stack_main.cpp

```
1 #include <vector>
2 #include <iostream>
3 #include <numeric>
4
5 template <typename T>
6 class stack {
7 public:
8     auto push(T t) -> void { vec_.push_back(t); }
9     auto top() -> T& { return vec_.back(); }
10    auto pop() -> void { vec_.pop_back(); }
11    auto size() const -> std::size_t { return vec_.size(); };
12    auto sum() -> int {
13        return std::accumulate(vec_.begin(), vec_.end(), 0);
14    }
15 private:
16    std::vector<T> vec_;
17 };
```

stack.h

```
1 #include "./stack-partial.h"
2
3 auto main() -> int {
4     auto i1 = 6771;
5     auto i2 = 1917;
6
7     auto s1 = stack<int*>{};
8     s1.push(&i1);
9     s1.push(&i2);
10    std::cout << s1.size() << " ";
11    std::cout << s1.top() << " ";
12    std::cout << s1.sum() << "\n";
13 }
```

stack_main.cpp



Partial Specialisation

- In this case we will specialise for pointer types.
 - Why do we need to do this?
- You can partially specialise classes
 - You cannot partially specialise a particular function of a class in isolation
- The following is a fairly standard example, for illustration purposes only. Specialisation is designed to refine a generic implementation for a specific type, not to change the semantic.

```
1 #include <numeric>
2
3 #include "../stack.h"
4
5 template <typename T>
6 class stack<T*> {
7 public:
8     auto push(T* t) -> void { stack_.push_back(t); }
9     auto top() -> T* { return stack_.back(); }
10    auto pop() -> void { stack_.pop_back(); }
11    auto size() const -> std::size_t { return stack_.size(); };
12    auto sum() -> int{
13        return std::accumulate(stack_.begin(),
14                               stack_.end(), 0, [] (int a, T *b) { return a + *b; });
15    }
16 private:
17    std::vector<T*> stack_;
18 };
```

stack-partial.h

```
1 #include "../stack-partial.h"
2
3 auto main() -> int {
4     auto i1 = 6771;
5     auto i2 = 1917;
6
7     auto s1 = stack<int*>{};
8     s1.push(&i1);
9     s1.push(&i2);
10    std::cout << s1.size() << " ";
11    std::cout << s1.top() << " ";
12    std::cout << s1.sum() << "\n";
13 }
```

stack_main-partial.cpp



Explicit Specialisation

- Explicit specialisation should only be done on classes.
- `std::vector<bool>` is an interesting example and here too `std::vector<bool>::reference` is not a `bool&`

```
1 #include <iostream>
2
3 template <typename T>
4 struct is_void {
5     static bool const val = false;
6 };
7
8 template<>
9 struct is_void<void> {
10     static bool const val = true;
11 };
12
13 auto main() -> int {
14     std::cout << is_void<int>::val << "\n";
15     std::cout << is_void<void>::val << "\n";
16 }
```

[specialise-explicit.cpp](#)



Type Traits

- **Trait:** Class (or class template) that characterises a type

This is what `<limits>` might look like

```
1 #include <iostream>
2 #include <limits>
3
4 template <typename T>
5 struct numeric_limits {
6     static auto min() -> T;
7 };
8
9 template <>
10 struct numeric_limits<int> {
11     static auto min() -> int { return -99999999 - 1; }
12 };
13
14 template <>
15 struct numeric_limits<double> {
16     static auto min() -> double { return -999999999.99999 - 1.0; }
17 };
18
19 auto main() -> int {
20     std::cout << std::numeric_limits<double>::min() << "\n";
21     std::cout << std::numeric_limits<int>::min() << "\n";
22 }
```

tt-limits.cpp



Type Traits

Traits allow generic template functions to be parameterised

```
1 #include <array>
2 #include <iostream>
3 #include <limits>
4
5 template<typename T, std::size_t size>
6 T findMax(const std::array<T, size>& arr) {
7     T largest = std::numeric_limits<T>::min();
8     for (auto const& i : arr) {
9         if (i > largest)
10            largest = i;
11     }
12     return largest;
13 }
14
15 auto main() -> int {
16     auto i = std::array<int, 3>{-1, -2, -3};
17     std::cout << findMax<int, 3>(i) << "\n";
18     auto j = std::array<double, 3>{1.0, 1.1, 1.2};
19     std::cout << findMax<double, 3>(j) << "\n";
20 }
```

typetraits1.cpp



Type Traits

- Below are STL [type trait](#) examples for a specialisation and partial specialisation
- This is a good example of partial specialisation

```
1 #include <iostream>
2
3 template <typename T>
4 struct is_void {
5     static const bool val = false;
6 };
7
8 template<>
9 struct is_void<void> {
10     static const bool val = true;
11 };
12
13 auto main() -> int {
14     std::cout << is_void<int>::val << "\n";
15     std::cout << is_void<void>::val << "\n";
16 }
```

typetraits2.cpp

```
1 #include <iostream>
2
3 template <typename T>
4 struct is_pointer {
5     static const bool val = false;
6 };
7
8 template<typename T>
9 struct is_pointer<T*> {
10     static const bool val = true;
11 };
12
13 auto main() -> int {
14     std::cout << is_pointer<int*>::val << "\n";
15     std::cout << is_pointer<int>::val << "\n";
16 }
```

typetraits3.cpp



Type Traits

- Below are STL [type trait](#) examples for a specialisation and partial specialisation
- This is a good example of partial specialisation

```
1 #include <iostream>
2 #include <type_traits>
3
4 template<typename T>
5 auto testIfNumberType(T i) -> void {
6     if (std::is_integral<T>::value || std::is_floating_point<T>::value) {
7         std::cout << i << " is a number"
8             << "\n";
9     }
10    else {
11        std::cout << i << " is not a number"
12            << "\n";
13    }
14 }
15
16 auto main() -> int {
17     auto i = int{6};
18     auto l = long{7};
19     auto d = double{3.14};
20     testIfNumberType(i);
21     testIfNumberType(l);
22     testIfNumberType(d);
23     testIfNumberType(123);
24     testIfNumberType("Hello");
25     auto s = "World";
26     testIfNumberType(s);
27 }
```

typetraits4.cpp



Template Template Parameters

- Previously, when we want to have a Stack with templated container type we had to do the following:
 - What is the issue with this?

```
1 #include <iostream>
2 #include <vector>
3
4 auto main(void) -> int {
5     stack<int, std::vector<int>> s1;
6     s1.push(1);
7     s1.push(2);
8     std::cout << "s1: " << s1 << "\n";
9
10    stack<float, std::vector<float>> s2;
11    s2.push(1.1);
12    s2.push(2.2);
13    std::cout << "s2: " << s2 << "\n";
14    //stack<float, std::vector<int>> s2; :0
15 }
```



Template Template Parameters

```
1 template <typename T, template <typename> typename CONT>
2 class stack {}
```

Using this, ideally we can just do:

```
1 #include <iostream>
2 #include <vector>
3
4 auto main(void) -> int {
5     stack<int, std::vector> s1;
6     s1.push(1);
7     s1.push(2);
8     std::cout << "s1: " << s1 << std::endl;
9
10    stack<float, std::vector> s2;
11    s2.push(1.1);
12    s2.push(2.2);
13    std::cout << "s2: " << s2 << std::endl;
14 }
```



Template Template Parameters

Before

```
1 #include <iostream>
2 #include <vector>
3
4 template <typename T, typename CONT>
5 class stack {
6 public:
7     auto push(T t) -> void { stack_.push_back(t); }
8     auto pop() -> void { stack_.pop_back(); }
9     auto top() -> T& { return stack_.back(); }
10    auto empty() const -> bool { return stack_.empty(); }
11 private:
12     CONT stack_;
13 };
14
15 auto main(void) -> int {
16     stack<int, std::vector<int>> s1;
17     int i1 = 1;
18     int i2 = 2;
19     s1.push(i1);
20     s1.push(i2);
21     while (!s1.empty()) {
22         std::cout << s1.top() << " ";
23         s1.pop();
24     }
25     std::cout << "\n";
26 }
```

template-template-before.cpp



Template Template Parameters

After

```
1 #include <iostream>
2 #include <vector>
3 #include <memory>
4
5 template <typename T, template <typename...> typename CONT>
6 class stack {
7 public:
8     auto push(T t) -> void { stack_.push_back(t); }
9     auto pop() -> void { stack_.pop_back(); }
10    auto top() -> T& { return stack_.back(); }
11    auto empty() const -> bool { return stack_.empty(); }
12 private:
13     CONT<T> stack_;
14 };
15
16 #include <iostream>
17 #include <vector>
18
19 auto main(void) -> int {
20     auto s1 = stack<int, std::vector>{};
21     s1.push(1);
22     s1.push(2);
23 }
```

template-template-after.cpp



Template Argument Deduction

Template Argument Deduction is the process of determining the types (of type parameters) and the values of nontype parameters from the types of function arguments.

```
1 template <typename T, int size>
2 T findmin(const T (&a)[size]) {
3     T min = a[0];
4     for (int i = 1; i < size; i++) {
5         if (a[i] < min) min = a[i];
6     }
7     return min;
8 }
```



Template Argument Deduction

Template Argument Deduction is the process of determining the types (of type parameters) and the values of nontype parameters from the types of function arguments.

Non-type parameters: Implicit conversions behave just like normal type conversions

Type parameters: Generally implicitly convert (real rules more complicated)

Feedback



Or go to the [form here](#).

