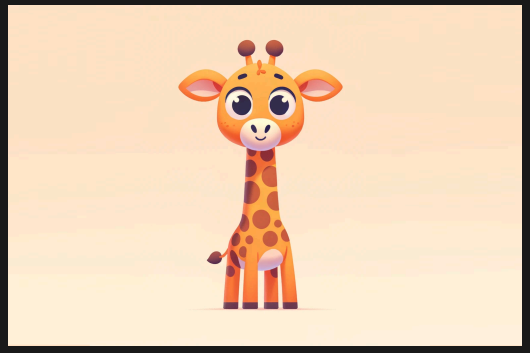
COMP6771

🚗 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.

```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
}
```

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

```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)
}
```

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 parmeter**: 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";
}
```

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

```h
// 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 "./class-template-start2.cpp"

#endif // STACK_H
```

class-template-start1.h

```cpp
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();
}
```

class-template-start2.cpp

🍇 Class Templates

Creating our first

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

#include "./class-template-start2.cpp"

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.
}
```

class-template-start-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 code?

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

inc-com-min.cpp

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

inc-com-min.h

```cpp
#include <iostream>
#include "./inc-com-min.h"

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

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.

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

inc-com-min.h

```cpp
#include <iostream>
#include "./inc-com-min.h"

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

inc-com-main.cpp

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

Inclusion Compilation Model

- Alternative: Explicit instantiations

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

inc-com-min.h

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

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

inc-com-minexp.cpp

```cpp
#include <iostream>
#include "./inc-com-min.h"

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

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

```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);
}

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

inc-com-stack.cpp

Static Members

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

```cpp
#include <iostream>
#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_--;
}

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";
}
```

static.cpp

🤝 Friends

Each stack instantiation has one unique instantiation of the friend

```cpp
#include <string>
#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);
}

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

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

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

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

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

private:
  CONT stack_;
};

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

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

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

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

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

```cpp
#include <vector>
#include <iostream>
#include <numeric>

template <typename T>
class stack {
public:
  auto push(T t) -> void { vec_.push_back(t); }
  auto top() -> T& { return vec_.back(); }
  auto pop() -> void { vec_.pop_back(); }
  auto size() const -> std::size_t { return vec_.size(); };
  auto sum() -> int {
    return std::accumulate(vec_.begin(), vec_.end(), 0);
  }
private:
  std::vector<T> vec_;
};
```

stack.h

```cpp
#include "./stack-partial.h"

auto main() -> int {
  auto i1 = 6771;
  auto i2 = 1917;

  auto s1 = stack<int*>{};
  s1.push(&i1);
  s1.push(&i2);
  std::cout << s1.size() << " ";
  std::cout << s1.top() << " ";
  std::cout << s1.sum() << "\n";
}
```

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 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.

```cpp
#include <numeric>

#include "./stack.h"

template <typename T>
class stack<T*> {
public:
  auto push(T* t) -> void { stack_.push_back(t); }
  auto top() -> T* { return stack_.back(); }
  auto pop() -> void { stack_.pop_back(); }
  auto size() const -> std::size_t { return stack_.size(); };
  auto sum() -> int{
    return std::accumulate(stack_.begin(),
          stack_.end(), 0, [] (int a, T *b) { return a + *b; });
  }
private:
  std::vector<T*> stack_;
};
```

stack-partial.h

```cpp
#include "./stack-partial.h"

auto main() -> int {
  auto i1 = 6771;
  auto i2 = 1917;

  auto s1 = stack<int*>{};
  s1.push(&i1);
  s1.push(&i2);
  std::cout << s1.size() << " ";
  std::cout << s1.top() << " ";
  std::cout << s1.sum() << "\n";
}
```

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&

```cpp
#include <iostream>

template <typename T>
struct is_void {
  static bool const val = false;
};

template<>
struct is_void<void> {
  static bool const val = true;
};

auto main() -> int {
  std::cout << is_void<int>::val << "\n";
  std::cout << is_void<void>::val << "\n";
}
```

specialise-explicit.cpp

Type Traits

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

This is what <limits> might look like

```cpp
#include <iostream>
#include <limits>

template <typename T>
struct numeric_limits {
  static auto min() -> T;
};

template <>
struct numeric_limits<int> {
  static auto min() -> int { return -99999999 - 1; }
};

template <>
struct numeric_limits<double> {
  static auto min() -> double { return -999999999.99999 - 1.0; }
};

auto main() -> int {
  std::cout << std::numeric_limits<double>::min() << "\n";
  std::cout << std::numeric_limits<int>::min() << "\n";
}
```

tt-limits.cpp

Type Traits

Traits allow generic template functions to be parameterised

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

template<typename T, std::size_t size>
T findMax(const std::array<T, size>& arr) {
  T largest = std::numeric_limits<T>::min();
  for (auto const& i : arr) {
    if (i > largest)
      largest = i;
  }
  return largest;
}

auto main() -> int {
  auto i = std::array<int, 3>{-1, -2, -3};
  std::cout << findMax<int, 3>(i) << "\n";
  auto j = std::array<double, 3>{1.0, 1.1, 1.2};
  std::cout << findMax<double, 3>(j) << "\n";
}
```

typetraits1.cpp

Type Traits

- Below are STL type trait examples for a specialisation and partial specialisation
- This is a good example of partial specialisation

```cpp
#include <iostream>

template <typename T>
struct is_void {
  static const bool val = false;
};

template<>
struct is_void<void> {
  static const bool val = true;
};

auto main() -> int {
  std::cout << is_void<int>::val << "\n";
  std::cout << is_void<void>::val << "\n";
}
```

typetraits2.cpp

```cpp
#include <iostream>

template <typename T>
struct is_pointer {
  static const bool val = false;
};

template<typename T>
struct is_pointer<T*> {
  static const bool val = true;
};

auto main() -> int {
  std::cout << is_pointer<int*>::val << "\n";
  std::cout << is_pointer<int>::val << "\n";
}
```

typetraits3.cpp

Type Traits

- Below are STL type trait examples for a specialisation and partial specialisation
- This is a good example of partial specialisation

```cpp
#include <iostream>
#include <type_traits>

template<typename T>
auto testIfNumberType(T i) -> void {
  if (std::is_integral<T>::value || std::is_floating_point<T>::value) {
    std::cout << i << " is a number"
              << "\n";
  }
  else {
    std::cout << i << " is not a number"
              << "\n";
  }
}

auto main() -> int {
  auto i = int{6};
  auto l = long{7};
  auto d = double{3.14};
  testIfNumberType(i);
  testIfNumberType(l);
  testIfNumberType(d);
  testIfNumberType(123);
  testIfNumberType("Hello");
  auto s = "World";
  testIfNumberType(s);
}
```

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?

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

auto main(void) -> int {
  stack<int, std::vector<int>> s1;
  s1.push(1);
  s1.push(2);
  std::cout << "s1: " << s1 << "\n";

  stack<float, std::vector<float>> s2;
  s2.push(1.1);
  s2.push(2.2);
  std::cout << "s2: " << s2 << "\n";
  //stack<float, std::vector<int>> s2; :O
}
```

🍴 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

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

template <typename T, typename CONT>
class stack {
public:
  auto push(T t) -> void { stack_.push_back(t); }
  auto pop() -> void { stack_.pop_back(); }
  auto top() -> T& { return stack_.back(); }
  auto empty() const -> bool { return stack_.empty(); }
private:
  CONT stack_;
};

auto main(void) -> int {
  stack<int, std::vector<int>> s1;
  int i1 = 1;
  int i2 = 2;
  s1.push(i1);
  s1.push(i2);
  while (!s1.empty()) {
    std::cout << s1.top() << " ";
    s1.pop();
  }
  std::cout << "\n";
}
```

template-template-before.cpp

Template Template Parameters

After

```cpp
#include <iostream>
#include <vector>
#include <memory>

template <typename T, template <typename...> typename CONT>
class stack {
public:
  auto push(T t) -> void { stack_.push_back(t); }
  auto pop() -> void { stack_.pop_back(); }
  auto top() -> T& { return stack_.back(); }
  auto empty() const -> bool { return stack_.empty(); }
private:
  CONT<T> stack_;
};

#include <iostream>
#include <vector>

auto main(void) -> int {
  auto s1 = stack<int, std::vector>{};
  s1.push(1);
  s1.push(2);
}
```

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.

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

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 conversations behave just like normal type conversions

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

👂 Feedback

Or go to the form here.