

COMP6771

Advanced C++ Programming

Week 8.1

Advanced Templates

Default Members

```
1 #include <vector>
2
3 template<typename T, typename CONT = std::vector<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 -> T const&;
12    static int num_stacks_;
13
14 private:
15     CONT stack_;
16 };
17
18 template<typename T, typename CONT>
19 int stack<T, CONT>::num_stacks_ = 0;
20
21 template<typename T, typename CONT>
22 stack<T, CONT>::stack() {
23     num_stacks_++;
24 }
25
26 template<typename T, typename CONT>
27 stack<T, CONT>::~~stack() {
28     num_stacks_--;
29 }
```

demo801-default.h

- 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 <iostream>
2
3 #include "../demo801-default.h"
4
5 auto main() -> int {
6     auto fs = stack<float>{};
7     stack<int> is1, is2, is3;
8     std::cout << stack<float>::num_stacks_ << "\n";
9     std::cout << stack<int>::num_stacks_ << "\n";
10 }
```

demo801-default.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`

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

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!**

Our 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 { stack_.push_back(t); }
9     auto top() -> T& { return stack_.back(); }
10    auto pop() -> void { stack_.pop_back(); }
11    auto size() const -> int { return stack_.size(); };
12    auto sum() -> int {
13        return std::accumulate(stack_.begin(), stack_.end(), 0);
14    }
15 private:
16     std::vector<T> stack_;
17 };
```

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

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

demo802-partial.h

```
1 #include "../demo802-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 }
```

demo802-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 }
```

demo803-explicit.cpp

Type Traits

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

```
1 #include <iostream>
2 #include <limits>
3
4 auto main() -> int {
5     std::cout << std::numeric_limits<double>::min() << "\n";
6     std::cout << std::numeric_limits<int>::min() << "\n";
7 }
```

This is what <limits>
might look like

```
1 template <typename T>
2 struct numeric_limits {
3     static auto min() -> T;
4 };
5
6 template <>
7 struct numeric_limits<int> {
8     static auto min() -> int { return -INT_MAX - 1; }
9 }
10
11 template <>
12 struct numeric_limits<float> {
13     static auto min() -> int { return -FLT_MAX - 1; }
14 }
```

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 }
```

demo804-typetraits1.cpp

Two more examples

- Below are STL type trait examples for a specialisation and partial specialisation
- This is a *good* example of partial specialisation
- http://en.cppreference.com/w/cpp/header/type_traits

```
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 }
```

demo805-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 }
```

demo806-typetraits3.cpp

Where it's useful

- Below are STL type trait examples
- http://en.cppreference.com/w/cpp/header/type_traits

```
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 }
```

demo807-typetraits4.cpp

Variadic Templates

These are the instantiations that will have been generated

```
1 #include <iostream>
2 #include <typeinfo>
3
4 template <typename T>
5 auto print(const T& msg) -> void {
6     std::cout << msg << " ";
7 }
8
9 template <typename A, typename... B>
10 auto print(A head, B... tail) -> void {
11     print(head);
12     print(tail...);
13 }
14
15 auto main() -> int {
16     print(1, 2.0f);
17     std::cout << "\n";
18     print(1, 2.0f, "Hello");
19     std::cout << "\n";
20 }
```

demo808-variadic.cpp

```
1 auto print(const char* const& c) -> void {
2     std::cout << c << " ";
3 }
4
5 auto print(float const& b) -> void {
6     std::cout << b << " ";
7 }
8
9 auto print(float b, const char* c) -> void {
10     print(b);
11     print(c);
12 }
13
14 auto print(int const& a) -> void {
15     std::cout << a << " ";
16 }
17
18 auto print(int a, float b, const char* c) -> void {
19     print(a);
20     print(b, c);
21 }
```

Member Templates

- Sometimes templates can be too rigid for our liking:
 - Clearly, this *could* work, but doesn't by default

```
1 #include <vector>
2
3 template <typename T>
4 class stack {
5 public:
6     auto push(T& t) -> void { stack._push_back(t); }
7     auto top() -> T& { return stack_.back(); }
8 private:
9     std::vector<T> stack_;
10 };
11
12 auto main() -> int {
13     auto is1 = stack<int>{};
14     is1.push(2);
15     is1.push(3);
16     auto is2 = stack<int>{is1}; // this works
17     auto ds1 =
18     stack<double>{is1}; // this does not
19 }
```

Member Templates

Through use of member templates, we can extend capabilities

```
1 #include <vector>
2
3 template <typename T>
4 class stack {
5 public:
6     explicit stack() {}
7     template <typename T2>
8     stack(stack<T2>&);
9     auto push(T t) -> void { stack_.push_back(t); }
10    auto pop() -> T;
11    auto empty() const -> bool { return stack_.empty(); }
12 private:
13     std::vector<T> stack_;
14 };
15
16 template <typename T>
17 T stack<T>::pop() {
18     T t = stack_.back();
19     stack_.pop_back();
20     return t;
21 }
22
23 template <typename T>
24 template <typename T2>
25 stack<T>::stack(stack<T2>& s) {
26     while (!s.empty()) {
27         stack_.push_back(static_cast<T>(s.pop()));
28     }
29 }
```

```
1 auto main() -> int {
2     auto is1 = stack<int>{};
3     is1.push(2);
4     is1.push(3);
5     auto is2 = stack<int>{is1}; // this works
6     auto ds1 =
7     stack<double>{is1}; // this does not
8 }
```

Template Template Parameters

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

- 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 }
```

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

```
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 };;
```

```
1 auto main(void) -> int {
2     stack<int, std::vector<int>>> s1;
3     int i1 = 1;
4     int i2 = 2;
5     s1.push(i1);
6     s1.push(i2);
7     while (!s1.empty()) {
8         std::cout << s1.top() << " ";
9         s1.pop();
10    }
11    std::cout << "\n";
12 }
```

```
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 };;
```

```
1 #include <iostream>
2 #include <vector>
3
4 auto main(void) -> int {
5     auto s1 = stack<int, std::vector>{};
6     s1.push(1);
7     s1.push(2);
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**.

type parameter

non-type parameter

```
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 }
```

call parameters

Implicit Deduction

- Non-type parameters: Implicit conversions behave just like normal type conversions
- Type parameters: Three possible implicit conversions
- ... others as well, that we won't go into

```
1 // array to pointer
2 template <typename T>
3 f(T* array) {}
4
5 int a[] = { 1, 2 };
6 f(a);
```

```
1 // const qualification
2 template <typename T>
3 f(const T item) {}
4
5 int a = 5;
6 f(a); // int => const int;
```

```
1 // conversion to base class
2 // from derived class
3 template <typename T>
4 void f(base<T> &a) {}
5
6 template <typename T>
7 class derived : public base<T> { }
8 derived<int> d;
9 f(d);
```

Explicit Deduction

If we need more control over the normal deduction process, we can explicitly specify the types being passed in

```
1 template<typename T>
2 T min(T a, T b) {
3     return a < b ? a : b;
4 }
5
6 auto main() -> int {
7     auto i = int{0};
8     auto d = double{0};
9     min(i, static_cast<int>(d)); // int min(int, int)
10    // min<int>(i, d); // int min(int, int)
11    min(static_cast<double>(i), d); // double min(double, double)
12    min<double>(i, d); // double min(double, double)
13 }
```

demo811-explicitdeduc.cpp

Feedback

