COMP6771 Week 8.1

Advanced Templates

Default Members

```cpp
#include <vector>

template <typename T, typename Cont = std::vector<T>>
class Stack {
 public:
  Stack();
  ~Stack();
  void push(T&);
  void pop();
  T& top();
  const T& top() const;
  static int numStacks_;
 private:
  Cont stack_;
};

template <typename T, typename Cont>
int Stack<T, Cont>::numStacks_ = 0;

template <typename T, typename Cont>
Stack<T, Cont>::Stack() { numStacks_++; }

template <typename T, typename Cont>
Stack<T, Cont>:: ~Stack() { numStacks_--; }
```

- 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 <iostream>

#include "lectures/week7/stack.h"

int main() {
  Stack<float> fs;
  Stack<int> is1, is2, is3;
  std::cout << Stack<float>::numStacks_ << "\n";
  std::cout << Stack<int>::numStacks_ << "\n";
}
```

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

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

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

```cpp
int main() {
  int i1 = 6771;
  int i2 = 1917;

  Stack<int> s1;
  s1.push(i1);
  s1.push(i2);
  std::cout << s1.size() << " ";
  std::cout << s1.top() << " ";
  std::cout << s1.sum() << "\n";
}
```

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

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

```cpp
int main() {
  int i1 = 6771;
  int i2 = 1917;
  Stack<int*> s2;
  s2.push(&i1);
  s2.push(&i2);
  std::cout << s2.size() << " ";
  std::cout << *(s2.top()) << " ";
  std::cout << s2.sum() << "\n";
}
```

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&

```
#include <iostream>

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

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

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

Quiz

What is the relationship between these two functions?

Not as trivial as you might think

```
template <typename C>
void print_front(const C& c) {
  std::cout << c.front() << "\n";
}

template <typename T>
void print_front(T* t) {
  std::cout << *t << "\n";
}
```

Quiz

- This is an overload (**not** a specialisation)
- This is a good thing (function specialisations are bad)
  - For more details why, see http://www.gotw.ca/publications/mill17.htm

```
template <typename C>
void print_front(const C& c) {
  std::cout << c.front() << "\n";
}

template <typename T>
void print_front(T* t) {
  std::cout << *t << "\n";
}
```

Type Traits

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

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

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

This is what <limits> might look like

```cpp
template <typename T>
struct numeric_limits {
  static T min();
};

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

template <>
struct numeric_limits<float> {
  static int min() { return -__FLT_MAX__ - 1; }
}
```

Type Traits

- Traits allow generic template functions to be parameterised

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

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

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

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

```cpp
#include <iostream>

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

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

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

```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;
};

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

Where it's useful

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

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

template <typename T>
void testIfNumberType(T i) {
  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";
  }
}

int main() {
  int i = 6;
  long l = 7;
  double d = 3.14;
  testIfNumberType(i);
  testIfNumberType(l);
  testIfNumberType(d);
  testIfNumberType(123);
  testIfNumberType("Hello");
  std::string s = "World";
  testIfNumberType(s);
}
```

Variadic Templates

```cpp
#include <iostream>
#include <typeinfo>

template <typename T>
void print(const T& msg) {
  std::cout << msg << " ";
}

template <typename A, typename... B>
  void print(A head, B... tail) {
  print(head);
  print(tail...);
}

int main() {
  print(1, 2.0f);
  std::cout << std::endl;
  print(1, 2.0f, "Hello");
  std::cout << std::endl;
}
```

- These are the instantiations that will have been generated

```cpp
void print(const char* const& c) {
  std::cout << c << " ";
}

void print(const float& b) {
  std::cout << b << " ";
}

void print(float b, const char* c) {
  print(b);
  print(c);
}

void print(const int& a) {
  std::cout << a << " ";
}

void print(int a, float b, const char* c) {
  print(a);
  print(b, c);
}
```

Member Templates

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

```cpp
#include <vector>

template <typename T>
class Stack {
 public:
  void push(T& t) { stack._push_back(t); }
  T& top() { return stack_.back(); }
 private:
  std::vector<T> stack_;
};

int main() {
  Stack<int> is1;
  is1.push(2);
  is1.push(3);
  Stack<int> is2{is1}; // this works
  Stack<double> ds1{is1}; // this does not
}
```

Member Templates

- Through use of member templates, we can extend capabilities

```cpp
template <typename T>
class Stack {
 public:
  explicit Stack() {}
  template <typename T2>
  Stack(Stack<T2>&);
  void push(T t) { stack_.push_back(t); }
  T pop();
  bool empty() const { return stack_.empty(); }
 private:
  std::vector<T> stack_;
};

template <typename T>
T Stack<T>::pop() {
  T t = stack_.back();
  stack_.pop_back();
  return t;
}

template <typename T>
template <typename T2>
Stack<T>::Stack(Stack<T2>& s) {
  while (!s.empty()) {
    stack_.push_back(static_cast<T>(s.pop()));
  }
}
```

```cpp
int main() {
  Stack<int> is1;
  is1.push(2);
  is1.push(3);
  Stack<int> is2{is1}; // this works
  Stack<double> ds1{is1}; // this does not
}
```

Template Template Parameters

```cpp
template <typename T, template <typename> Cont>
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?

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

int main(void) {
  Stack<int, std::vector<int>> s1;
  s1.push(1);
  s1.push(2);
  std::cout << "s1: " << s1 << std::endl;

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

Ideally we can just do:

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

int main(void) {
  Stack<int, std::vector> s1;
  s1.push(1);
  s1.push(2);
  std::cout << "s1: " << s1 << std::endl;

  Stack<float, std::vector> s2;
  s2.push(1.1);
  s2.push(2.2);
  std::cout << "s2: " << s2 << std::endl;
}
```

Template Template Parameters

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

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

```cpp
int main(void) {
  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";
}
```

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

template <typename T, template<typename, typename = std::allocator<T>> class Cor
class Stack {
 public:
  void push(T t) { stack_.push_back(t); }
  void pop() { stack_.pop_back(); }
  T& top() { return stack_.back(); }
  bool empty() const { return stack_.empty(); }
 private:
  Cont<T> stack_;
};
```

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

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

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 paremeter

non-type parameter

call parameters

```
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;
}
```

Implicit Deduction

- Non-type parameters: Implicit conversions behave just like normal type conversions
- Type parameters: Three possible implicit conversions

```
// array to pointer
template <typename T>
f(T* array) {}

int a[] = { 1, 2 };
f(a);
```

```
// const qualification
template <typename T>
f(const T item {}

int a = 5;
f(5); // int => const int;
```

```
// conversion to base class
//   from derived class
template <typename T>
void f(Base<T> &a) {}

template <typename T>
class Derived : public Base<T> { }
Derived<int> d;
f(d);
```

Explicit Deduction

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

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

int main() {
  int i; double d;
  min(i, static_cast<int>(d)); // int min(int, int)
  min<int>(i, d); // int min(int, int)
  min(static_cast<double>(i), d); // double min(double, double)
  min<double>(i, d); // double min(double, double)
}
```