# COMP6771
# Advanced C++ Programming

## Week 8.1
## Advanced Templates

# Default Members

```cpp
#include <vector>

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_--;
}
```

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

```cpp
#include <iostream>

#include "./demo801-default.h"

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";
}
```

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

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

template <typename T>
class stack {
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 -> int { return stack_.size(); };
        auto sum() -> int {
                return std::accumulate(stack_.begin(), stack_.end(), 0);
        }
private:
        std::vector<T> stack_;
};
```

```cpp
auto main() -> int {
        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:
	auto push(T* t) -> void { stack_.push_back(t); }
	auto top() -> T* { return stack_.back(); }
	auto pop() -> void { stack_.pop_back(); }
	auto size() const -> int { 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_;
};
```

demo802-partial.h

```cpp
#include "./demo802-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";
}
```

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&

```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";
}
```

demo803-explicit.cpp

# Type Traits

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

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

auto main() -> int {
	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 auto min() -> T;
};

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

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

# 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";
}
```

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

```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";
}
```

demo805-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";
}
```

demo806-typetraits3.cpp

# 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>
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);
}
```

demo807-typetraits4.cpp

# Variadic Templates

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

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

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

auto main() -> int {
        print(1, 2.0f);
        std::cout << "\n";
        print(1, 2.0f, "Hello");
        std::cout << "\n";
}
```

demo808-variadic.cpp

These are the instantiations that will have been generated

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

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

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

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

auto print(int a, float b, const char* c) -> void {
        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

```
#include <vector>

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

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

# Member Templates

Through use of member templates, we can extend capabilities

```cpp
#include <vector>

template <typename T>
class stack {
public:
        explicit stack() {}
        template <typename T2>
        stack(stack<T2>&);
        auto push(T t) -> void { stack_.push_back(t); }
        auto pop() -> T;
        auto empty() const -> bool { 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()));
        }
}
```

demo809-membertemp.cpp

```cpp
auto main() -> int {
        auto is1 = stack<int>{};
        is1.push(2);
        is1.push(3);
        auto is2 = stack<int>{is1}; // this works
        auto ds1 =
        stack<double>{is1}; // this does not work
     // until we do the changes on the left
}
```

# Template Template Parameters

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

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
}
```

Ideally we can just do:

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

auto main(void) -> int {
  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:
	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_;
};
```

```cpp
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";
}
```

```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_;
};
```

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

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

demo810-temptemp.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**.

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
- ... others as well, that we won't go into

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

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

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

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

```cpp
// 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;
}

auto main() -> int {
	auto i = int{0};
	auto d = double{0};
	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)
}
```

demo811-explicitdeduc.cpp

# Feedback