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.
- The set of default template arguments accumulates over all declarations of a given template.

```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";
    auto fl = stack<float, std::list<float>>{};
}
```

1 fs

3 is1, is2, is3.

demo801-default.cpp

```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";
    auto fl = stack<float, std::list<float>>{};
}
```

auto fs = stack<float, std::unordered_set<float>>{};

0

3

因为 fs 的 type 是 stack<float, std::unordered_set<float>>, 每一种 type 有一个自己的 static variable.

```cpp
#include <list>
#include <unordered_set>

#include "./demo801-default.h"

auto main() -> int {
    auto fs = stack<float, std::unordered_set<float>>{};
    auto fs2 = stack<float, std::vector<float>>{};
    auto fs3 = stack<float>{};
    stack<int> is1, is2, is3;
    std::cout << stack<float, std::unordered_set<float>>::num_stacks_ << "\n";
    std::cout << stack<float>::num_stacks_ << "\n";
    std::cout << stack<int>::num_stacks_ << "\n";
    int j = 5;
    is1.push(j);
}
```

same

1 fs

2 fs2, fs3

3 is1, is2, is3.

看一下直播 !!! ???

```cpp
template<class T, class U = int> class A;
template<class T = float, class U> class A;

template<class T, class U> class A {
   public:
       T x;
       U y;
};

A<> a;
a.x ?
a.y?
```

```cpp
template<class T, class U = char> class A
{
public:
    T x;
    U y;
};

int main()
{
    A<char> a;
    A<int, int> b;
    std::cout<<sizeof(a)<<std::endl;
    std::cout<<sizeof(b)<<std::endl;
    return 0;
}
```

If one template parameter has a default argument, then all template parameters following it must also have default arguments.

```cpp
template<class T = char, class U, class V = int> class X { };

template<class T = char, class U, class V = int, class W> class X
//not allowed
```

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

Specialized case: Different min for different types.

```
auto min(auto a, auto b) {
    return a < b ? a : b;
}

auto min(std::string a, std::string b) {
    return a.size() < b.size() ? a : b;
}
```

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";
}
```

Because T* is still generic.

Partial Specialisation

whole picture:
template <typename T>
class stack<T> {
}

Redefine things. → template <typename T>
class stack<T*> {
}

- 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_;
};
```

Sum pointers doesn't make sense, preserve the semantic!!!
Sum what it point to.

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";
}
```

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

demo802-partial.cpp

Explicit Specialisation

On a concreate types (No T anymore)

```cpp
#include <iostream>

template <typename T>
void fun(T a) {
  std::cout << "The main template fun(): "
            << a << std::endl;
}

template<> // may be skipped, but prefer overloads
void fun(int a) {
  std::cout << "Explicit specialisation for int type: "
            << a << std::endl;
}

int main() {
  fun<char>('a');
  fun<int>(10);
  fun<float>(10.14);
}
```

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

template<typename T>
T add_all(const std::vector<T> &list) {
        T accumulator = {};
        for (auto& elem:list){
        accumulator += elem;
    }

    return accumulator;
}

template<>
T add_all(const std::vector<std::string> &list) {
        std::string accumulator = {};
        for (const std::string& elem : list)
        for (const char& chr : elem)
                accumulator += elem;
    }

        return accumulator;
}

int main() {
  std::vector<int> ivec = {4,3,2,4};
  std::vector<double> dvec = {4.0,3.0,2.0,4.0};
  std::vector<string> svec = {"abc", "bcd"};
  std::cout << add_all(ivec) << std::endl;
  std::cout << add_all(dvec) << std::endl;
  std::cout << add_all(svec) << std::endl;
}
```

Class Template Specialisation

```
#include <iostream>

template <class T>
class Test {
// Data members of test
public:
  Test() {
    // Initialization of data members
    cout << "General template object \n";
  }

  // Other methods of Test
};

template <>
class Test<int> {
public:
  Test() {
    // Initialization of data members
    std::cout << "Class template specialisation\n";
  }
};

int main() {
  Test<int> a;
  Test<char> b;
  Test<float> c;

  return 0;
}
```

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

不是特别明白，可以看看 Youtube 视频.

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

Compile time template metafunctions that returns the info about types.

Ask compiler some question? is it int, fun, class or pointer or will it throw exception?

Why ?: **Optimization: quick sort /merge sort by knowing data type of iterator or const to non**

type trait is a simple struct template that contains a member constant, which in turn holds the answer to the question the type trait asks or the transformation it performs

Useful in conditional compilation or

you can apply transformation i.e. add or remove const

handy when writing libraries that make use of template

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

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

```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; }
}
```

This is what <limits> might look like

Advantage: All at compile time, nothing run time

Trait Categoris:

Numeric_Limit<>:: min,max, is_signed, is_integer

Type Category: is_array<T>::value, is_pointer, is_enum

Type Properties: is_const

Type Traits

Traits allow generic template functions to be parameterised

```
#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

```
#include <type_traits>

class GFG {
};

// main method
auto main() -> int {
{
    std::cout << alignment_of<GFG>::value << std::endl;
    std::cout << alignment_of<int>() << std::endl;
    std::cout << alignment_of<double>() <<std::endl;

    return 0;
}
```

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

```cpp
template<template T>
typename std::remove_refernce<T>::
{return value;}

```

const T&	**const T**	
T &	T	
T&&	T	do more than conversion
const T	T	

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

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

void algorithm_signed  (int i)      { /*...*/ }
void algorithm_unsigned(unsigned u) { /*...*/ }

template <typename T>
void algorithm(T t) // act as dispatcher
{
    if constexpr(std::is_signed<T>::value)
        algorithm_signed(t);
    else
    if constexpr (std::is_unsigned<T>::value)
        algorithm_unsigned(t);
    else
        static_assert(std::is_signed<T>::value
                    || std::is_unsigned<T>::value, "Must be signed or unsigned!

}

int main(){

algorithm(3);       // T is int, include algorithm_signed()

unsigned x = 3;
algorithm(x);       // T is unsigned int, include algorithm_unsigned()

algorithm("hello"); // T is string, build error!
}
```

Type Recursion

How can printf take any # of arguments

Variadic Templates

These are the instantiations that will have been generated

```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";
}
```

Δ meaning many typename Bs.

Bs

Recursion.

demo808-variadic.cpp

Create at compile time.

```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);
}
```

Title Text

Subtitle

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


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

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

Functions/member functions of a class that deal with other kind of templates.

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()));
        }
}
```

```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
}
```

Can't do template <typename T, typename T2>

type of new stack.

demo809-membertemp.cpp

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;
}
```

C++ figure it out
std::vector<int>

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

- When compiler tries to find a template to match the template template argument, it only considers primary class templates.
- (A *primary template* is the template that is being specialized.) The compiler will not consider any partial specialization even if their parameter lists match that of the template template parameter.

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

template <class T, template <class...> class C, class U>
C<T> All_V(const C<U> &c) {
   C<T> result(c.begin(), c.end());
   return result;
}

int main() {
   std::vector<float> vf = {1.2, 2.6, 3.7};
   auto vi = All_V<int>(vf);
   for(auto &&i: vi) {
      std::cout << i << std::endl;
   }
}
```

The compiler considers the partial specializations based on a template template argument once you have instantiated a specialization based on the corresponding template template parameter.

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**.

```cpp
std::vector<int> v{1,2,3};
std::vector v{1,2,3};
```

type paremeter

non-type parameter [nullptr, **integral values**, lvalue references, pointer, enumerations, and floating-point values]

call parameters

```cpp
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;
}
```

```cpp
template <auto N>             // (1)
class MyClass{
    ....
};

template <int N>             // (2)
class MyClass<N> {
    ....
};


MyClass<'x'> myClass1;       // (3)
MyClass<2017>  myClass2;     // (4)
```

Automatic type deduction for non-type templates can also be applied to variadic templates

```cpp
template <auto... ns>
class VariadicTemplate{ .... };

template <auto n1, decltype(n1)...
class TypedVariadicTemplate{ .... }
```

Implicit Deduction

- We may omit any template argument that the compiler can determine or deduce by the usage and context of that template function call.
- The compiler tries to deduce a template argument by comparing the type of the corresponding template parameter with the type of the argument used in the function call. The two types that the compiler compares (the template parameter and the argument used in function call) must be of certain structure in order for template argument deduction to work.
- 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

```
f(1,2);   // f<int>(1,2);
f(1.1,2.2);  f<double>(1.1,2.2);
f('a', 'b'); f<char>(char a, char b);
f(2, 'b');  ?
f(2, 2.2);  ?  ?

//
template<typename T>
  T f(T a, T b)
     return a+b;

template<typename T1, typename T2>
  T f(T1 a, T2 b)
     return a+b;

```

```
// 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(a); // 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

- casting the argument to follow same type
- explicitly stating type of T, preventing compiler attempeting to deduce
- specifiying in function template that parameter may be of different type and let it on compiler to figure it out.

```cpp
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