COMP6771 19T2 (https://webcms3.cse.unsw.edu.au/COMP6771/19T2)

Week 09 ▾ Tutorial ▾ Sample Answers ▾

Advanced C++ Programming (https://webcms3.cse.unsw.edu.au/COMP6771/19T2)

1. When would we specialise classes? Why do we not specialise functions?

   - We do not specialise functions because:
     1. Specialising functions can have unintended behaviour - see lecture notes for more detail
     2. We already have this capability in the form of function overloading
   - We specialise classes when we need to either:
     1. preserve a particular semantic for something that would not work otherwise
     2. make an optimisation for a specific type (e.g. case of std::vector<bool> having a huge space optimisation.)

2. Define your own type trait, from scratch, for is_a_pointer, to be used in an application like this

```
template <typename T>
printPointer(T t) {
  if constexpr (traits::is_a_pointer<T>::value) {
        std::cout << *t << "\n";
  } else {
        std::cout << t << "\n";
  }
}
```

   Ensure that your type trait is wrapped in a trait namespace.

```
namespace traits {

template <typename T>
struct is_a_pointer {
  static constexpr bool value = false;
}

template <typename T>
struct is_a_pointer<T*> {
  static constexpr bool value = true;
}

}
```

3. Use type traits in the std namespace to produce your own composition type trait that returns true if the type passed in is an integer or floating point. It should be used as follows

```
template <typename T>
if (is_real_number<T>::value) {
  std::cout << "Is real number" << "\n";
}
```

```
template <typename T>
struct is_real_number {
  static bool type = std::is_integral<T>::type || std::is_floating_point<T>::type;
}

// Alternative solution - the solution above won't compile in some instances
template <typename T>
using is_real_number = std::is_arithmetic<T>;
```

4. What is an xvalue?

   An rvalue that is about to expire (e.g. the result of std::move).

5. What is an prvalue?

   An rvalue that is not an xvalue. Since std::move and other rvalue reference semantics were introduced in C++11, prior to C++11 we did not have these more granular definitions of rvalues

6. What are the inferred types for each of the following?

```
int main() {
        int i = 5;
    int& k = i;

        decltype(i) x;
    decltype(k) y;
    decltype(std::move(i)) z;
    decltype(4.2);

}
```

```
int main() {
        int i = 5;
    int& k = i;

        decltype(i) x; // int; - variable
    decltype(k) y; // int& - lvalue
    decltype(std::move(i)) z; // int&& - xvalue
    decltype(4.2);   // int - prvalue

}
```

7. What does the binding table for lvalues/rvalues look like?

	lvalue	const lvalue	rvalue	const rvalue
template T&&	True	True	True	True
T&	True	False	False	False
const T&	True	True	True	True
T&&	False	False	True	False

8. This code currently doesn't work as the implementation for my_make_unique is incomplete. Complete it through the addition of using std::forward as well as variadic types. To compile with this code, you will need to use types.h which can be found HERE (https://github.com/cs6771/comp6771/blob/master/lectures/week8/forwarding/types.h) in the github.

```
template <typename T>
auto my_make_unique(T item) {
  return std::unique_ptr<T>{new T{item}};
}

int main() {
  MyClass myClass{"MyClass"};
  std::cout << *my_make_unique<MyClass>(myClass) << "\n";
  std::cout << *my_make_unique<MyClass>(std::move(myClass)) << "\n";
  NonCopyable nonCopyable{"NonCopyable"};
  std::cout << *my_make_unique<NonCopyable>(std::move(nonCopyable)) << "\n\n";

  // Use the size constructor.
  std::cout << my_make_unique<std::vector<MyClass>>(5U)->size() << "\n";

  // Use the size-and-value constructor.
  MyClass base{"hello"};
  std::cout << my_make_unique<std::vector<MyClass>>(6U, base)->size() << "\n";
}
```

```
template <typename T, typename... Args>
auto my_make_unique(Args&&... args) {
  return std::unique_ptr<T>{new T{std::forward<Args>(args)...}};
}

int main() {
  MyClass myClass{"MyClass"};
  std::cout << *my_make_unique<MyClass>(myClass) << "\n";
  std::cout << *my_make_unique<MyClass>(std::move(myClass)) << "\n";
  NonCopyable nonCopyable{"NonCopyable"};
  std::cout << *my_make_unique<NonCopyable>(std::move(nonCopyable)) << "\n\n";

  // Use the size constructor.
  std::cout << my_make_unique<std::vector<MyClass>>(5U)->size() << "\n";

  // Use the size-and-value constructor.
  MyClass base{"hello"};
  std::cout << my_make_unique<std::vector<MyClass>>(6U, base)->size() << "\n";
}
```

9. Convert this normal C++ program using constexpr to a program using template meta-programming

```
#include <iostream>

constexpr int factorial (int n) {
  if (n <= 1) {
        return 1;
  }
  return n * factorial(n - 1);
}

int main() {
  std::cout << factorial(6) << std::endl;
}
```

```
#include <iostream>

template<int n> struct Factorial {
  static constexpr int val = Factorial<n-1>::val * n;
};

template<> struct Factorial<0> {
  static constexpr int val = 1; // must be a compile-time constant
};

int main() {
  std::cout << Factorial<6>::val << std::endl;
}
```