Program specialization and metaprogramming

Laboratory work No. 5

Metafunctions and metafunctors

1. Aims

To get acquainted with advanced concepts of metaprogramming in C++ using Standard Template Library (STL).

2. Tasks

Learn how to higher order functions (metafunctions) and higher order functors (metafunctors).

3. Work

The function is a mapping of parameter(s) to the result. It must be side-effect-free, that is, you always get the same result when you provide same parameters to the same function.

Higher order function is a function, which has another function either as a of parameters or as a result.

Consider the following example:

```
bool GreaterThan(int i, int j) {
    return i > j;
}
void SortDescend(std::vector<int>& v) {
    std::sort(begin(v), end(v), &GreaterThan);
}
```

Here std::sort can be considered a higher-order function (not really, it isn't pure) since the third parameter is a function.

A **lazy meta function** is implemented as a template specialization with a nested type specifier called 'type' which usually depends on the input parameters in some way.

```
using t = typename std::add_pointer<int>::type;
```

An **eager meta function** is implemented as an alias which resolves to the resulting type directly.

```
using t = std::add_pointer_t<int>;
```

A **wrapper** is a class template which does not contain a nested ::type.

```
using t = std::tuple<int>;
```

A **meta closure** is an optionally templated class containing a nested eager meta function named f (or fn, invoke etc. in other libraries). A meta closure takes two sets of arguments, one fixed set

as part of the class template and one dynamic set in the nested eager meta function.

```
using pred = meta::quote<std::add_pointer_t>;

//invoke in the library
using t = typename pred::template invoke<int>;
```

A continuation capable meta closure or just **continuation** for short is special case of a meta closure of which the last fixed parameter is its self a continuation.

```
template <typename M, typename C = identity>
struct same_as {
    template <typename T>
    using f = typename C::template f<std::is_same<M, T>>;
};

template<typename C = identity>struct add_pointer {
    template<typename T>
    using f = typename C::template f<typename std::add_pointer<T>::type>;
};
```

The difference between runtime functions and runtime functors lies in the fact that functors essentially take two sets of parameters. The first set is passed to the constructor and the second set is passed to the overloaded function operator. This allows us to embed information in a functor which is unknown to the caller. The overloaded function call operator inside the functor has access to both the parameters it was called with as well as the parameters it was constructed with, otherwise it is pretty much a normal function. Lets refer to the parameters the functor was constructed with the "fixed" parameters because the caller cannot change them.

```
struct RuntimeAdd {
    int i_;
    RuntimeAdd(int i) :i_{ i } {}
    int operator()(int i) {
        return i + i_;
    }
};

//construct a functor which adds 4 to its argument
RuntimeAdd add4{ 4 };
int result = add4(3);
```

What is the value of the result variable?

We can achieve a similar pattern by nesting two function templates:

```
template<typename T, T V_Value>
struct Value {
    using Type = Value < T, V_Value >;
    static const T value = V_Value;
};

template<int I>
using Int = Value < int, I >;

template<typename T_Fixed>
struct Add {
```

```
    template<typename T>
    struct Apply : Int<T_Fixed::value + T::value> {};
};


//create a metafunctor which adds 4 to its argument
using Add4 = Add<Int<4>>;
using Result = typename Add4::template Apply<Int<3>>::Type;

std::cout << Result::value << std::endl;
```

What is printed by the program?

The following example demonstrates the application of the functor:

```
template<typename T>
typename T::RetType f(T in) {
    return in();
}

long f(long i) {
    return i + 1;
}

struct MyFunctor {
    using RetType = int;
    int operator()() {
        return 42;
    }
};
```

What is printed by this program?

```
    std::cout << f(MyFunctor{}) << std::endl;
    std::cout << f(3) << std::endl;
```

4. Report

Prepare a report of your work in this lab using a standard word processing application. The report is a single chapter of your final semester report. Final semester report will have to be uploaded to course Moodle page.

The content of report:

1. Title
2. Aims and tasks
3. Work done, described in steps and illustrated by screenshots and written or modified source code. Provide comments what you have done and what were the results (program outputs). Present answers to the questions given in the lab description.
4. Conclusions – what you have learned?