Functional programming in C++
Meta programming (including traits)

shan@iha.dk
V1.2

Introduction

In the exercise we will endolge ourselves in meta programming both the class and their functional countaparts.

Prerequisites

BE sure to have read the chapters in the book regarding this topic...

Exercise 1 Static and Dynamic polymorphism

Discuss the terms *static* and *dynamic* polymorphism in relation to templates likes:

- STL Containers
- Meta programming as a general concept in C++

Exercise 2 Binary with only binary digits

In the binary code example as seen in listing 2.1 one can supply a binary value that actually is not a binary value and the compiler will still compute a result.

Listing 2.1: struct Binary<> and its usage

```cpp
template<size_t N>
struct Binary
{
    static const size_t value = Binary<N/10>::value << 1 | N%10;
};

template<>
struct Binary<0>
{
    static const size_t value = 0;
};

std::cout << "Binary<1011>::value = " << Binary<1021>::value << std::endl;
    /* Should generate a compiler error, but does not! */
```

In this exercise the goal is to modify the existing solution such that a compiler error is generated if *a none binary value* is supplied.

Exercise 3 Typelists

Typelists (*TL*) can be used for numerous different things, in this exercise they are to be developed for two overall reasons.

1. In the course of developing the supportive algorithms one becomes more familiar with the concept and how it works.
2. In a concret scenario a limitation is desired on which types a given template class may be parameterized with.

AARHUS UNIVERSITY
SCHOOL OF ENGINEERING

Functional programming in C++
Meta programming (including traits)

shan@iha.dk
V1.2

Exercise 3.1 Write your own TL

In the setup proposed the typelist base template is as in listing 3.1.

Listing 3.1: Basic TypeList relevant structs

```
struct NullType{};

template<typename L, typename R>
struct TypeList
{
    typedef L First;
    typedef R Rest;
};
```

Discuss the concept behind and present examples complete with descriptions on how they are to be used and understood. This includes having a *TL* consisting of numerous types.

To ease usage create macros that make it easier and more readable. They could be like in listing 3.2.

Listing 3.2: Macros for generating typelists

```
typedef TYPELIST1(int) IntTL;
typedef TYPELIST4(char, int, long, short) MixTL;
```

Note that the succeeding digit denotes the number of types in the *TL*. In the first the '1' denotes one type, whereas in the second the '4' denotes four types.

Exercise 3.2 Helpers

Exercise 3.2.1 struct IsSame<>

Create a template that checks to see if two types are the same. The following code snippets must be work:

Listing 3.3: Usage example for IsSame<>

```
/* Must be true */
std::cout << "IsSame<int, int>::value=" << IsSame<int, int>::value << std::
    endl;

/* Must be false */
std::cout << "IsSame<int, char>::value=" << IsSame<int, char>::value << std
    ::endl;
```

Exercise 3.2.2 struct Contains<>

Contains iterates through a *TL* to find a specific type. The following code snippets must be work:

Listing 3.4: Usage example for Contains<>

```cpp
typedef TYPELIST3(int, char, long) TL;
/* Must be true */
std::cout << "Contains<TL, int>::value=" << Contains<TL, int>::value << std
    ::endl;

/* Must be false */
std::cout << "Contains<TL, std::string>::value=" << Contains<TL, std::
    string>::value << std::endl;
```

Exercise 3.2.3 struct AtIndex<>

AtIndex returns the type at the given index The following code snippets must be work:

Listing 3.5: Usage example for AtIndex<>

```cpp
typedef TYPELIST3(long, char, int) TL;
/* Must be true */
std::cout << "IsSame<typename AtIndex<TL, 2>::type, int>::value" << IsSame<
    typename AtIndex<TL, 2>::type, int>::value << std::endl;

/* Must be false */
std::cout << "IsSame<typename AtIndex<TL, 2>::type, char>::value" << IsSame
    <typename AtIndex<TL, 2>::type, char>::value << std::endl;
```

Exercise 3.2.4 struct PrintIT<>

Recursively traverses the *TL* and prints out the types it contains. Use something appropriately like[1]: `typeid(int).name()`.

Exercise 3.2.5 struct Remove<> - The challenger!!!

In this exercise the idea is to remove a type and thus generate a new *TL* that does not contain some specified type.

The following code snippets must be work:

Listing 3.6: Usage example for Remove<>

```cpp
typedef TYPELIST3(long, char, int) TL;
/* Must be false */
std::cout << "Contains<typename Remove<TL, int>::type, int>::value" <<
    Contains<typename Remove<TL, int>::type, int>::value << std::endl;

/* Must be true, semi test to verify that the typelist still contains a
    char */
std::cout << "Contains<typename Remove<TL, char>::type, int>::value" <<
    Contains<typename Remove<TL, char>::type, int>::value << std::endl;
```

[1]Obviously adapted, T NOT `int` and so on.

Exercise 3.3 Limiting parameterized types for MyArray

Use `BOOST_STATIC_ASSERT()` to limit the types that MyArray may be parameterized with. The types allowed are:

- `int`
- `short`
- `long`
- `char`

Two solutions are sought that can ensure that the `MyArray` is only parameterizable by the above denoted types.:

1. Using our own *TL* and helpers
2. Using `boost::mpl` & `boost::traits`[2]

Note: Instead of using *boost* to solve this, one could use *C++11* instead. If you want to use features from *C++11* you should have a look at `static_assert()` and file `type_traits`.

Exercise 4 Transfering input parameters as effective as possible

A classic approach to transfer input parameters in a template function is to write `const T&`, however this is not the most effective approach for all types.

Consider what you believe to be the optimal approach for the following types and why.

- `int`, `short`, `float` etc. (whats their classification name?)
- Regular class - such as the `std::string`
- Simple struct or class with no state.
- A reference variable

A Hint: Transfer could be one of these where `T` is the input parameter type in question

- `T` - The type it self
- `const T&` - The type having had added `const` and `&`.

Devise a scheme such that you can write the following *and* ensure optimal parameter transference, according what you determined above:

[2]boost has similar and better functionality. It is there for a simple matter of finding the corresponding functionality in boost and employing them in the exercise.

Listing 4.1: Optimal transference of input parameters

```
template<typename T, size_t N>
class MyStack
{
public:
  /* Some constructor + destructor doing something sensible */
  void push(typename ParamType<T>::type t)
  {
    t_[index_] = t;
    ++index_;
  }

private:
   T t_[N];
   size_t index_;
};
```

The class template type `ParamType<>` used for this particular job could look something like in listing 4.2.

Listing 4.2: Scaffolding for template class `ParamType`

```
template<typename T>
struct ParamType : boost::mpl::eval_if< /* some kind of 'or' expression */,
                                        /* whatever type to be returned if
                                           true */,
                                        /* whatever type to be returned if
                                           false */>
{};
```

A print-out of `typeinfo` in the member method `push()` of the type `t` might be very usefull, to verify what type actually made it. Be vary of what you want to query about e.g. about which type classification `T` is. Based on this you want to return the type itself `T` or a `const &` edition of it. Do note that *if* the type in it self is a reference type, one cannot add another reference...