Metaprogramming with templates

This assignment gives you some praxis in writing meta-functions to enrich your understanding of meta-programming. It's graded G or VG.

1 Functional programming

1.1 Addition

1.1.1 A short example

Write a recursive function adding two numbers using only the operations +1 and -1 (i.e. x+1 or y-1). One solution is:

```
int AddFun(int x, int y) {
    if (x == 0)
        return y;
    else
        return AddFun(x - 1, y + 1);
}
```

1.1.2 To do: define a meta-function doing the same:

We start by defining the general case:

```
template <int X, int Y>
struct Add {
    static const int value = Add<X - 1, Y + 1>::value;
};
```

To get the result we can write:

```
    int x2 = Add<3, 5>::value;
```

To get this to work you need to do a specialization of the struct Add to stop the recursion when X is 0.

1.2 Ackermann function

Ackermann function is an old example of a recursive function, interesting because it is impossible to rewrite with only if statements and only one call on Ackerman (or more theoretical: it cannot be rewritten to a primitive recursive function). Definition:

$$A(m,n) = \begin{cases} n+1 & \text{if } m = 0 \\ A(m-1,1) & \text{if } m > 0 \text{ and } n = 0 \\ A(m-1, A(m,n-1)) & \text{if } m > 0 \text{ and } n > 0. \end{cases}$$

Define Ackermann as a meta-function. Ackermann<1, 0>::value should be 2. Ackermann<0, 1>::value should be 2. Ackermann<3, 4>::value should be 125. Ackermann<4, 0>::value is 13. Ackermann<4, 1>::value do not compile "recursive type or function dependency context too complex" which really just is that the compilers stack got to big. The answer is 65533 = 2^{16} - 3.

2 Type manipulation

2.1 SimplifyType

Do a meta function to ”simplify” types by taking away the last modifier (i.e. make `int*` be converted to `int`). The general definition for the case when there is no modifier to take away is:

```
template<class T>
struct SimplifyType {
    using type = T;
};
```

You should add specializations for removing const, & and *. []??????????????
If you want you can shorten how much you need to write by defining:

```
template<class T>
using SimplifyType_t = typename SimplifyType<T>::type;
```

and the use “`SimplifyType_t<T>`“ instead of “`typename SimplifyType<T>::type` ”.
This is the standard way of writing in STL.

Exempel på vad SimplifyType ger: (AssertSame kollar att dess argument har samma typ).

```
    AssertSame(int, SimplifyType_t<int*>);
    AssertSame(int**const*, SimplifyType_t<int**const**>);
    AssertSame(const int, SimplifyType_t<const int&>);
```

Ett knep för att se vilken typ ni fått fram om det inte stämmer är att skriva

```
    SimplifyType_t<const int&> x = 0;
```

och stega i programmet, i debuggern kan ni se vilken typ x fick.

2.2 BaseType

BaseType fungerar liknande men skalar bort alla saker tills den når typen ”längst in”. Den ska även kunna skala bort "T[]" och "T[5]". Där 5 kan vara vilket heltal som helst.

```
    AssertSame(int, BaseType_t<int*>);
    AssertSame(int, BaseType_t<int***&>);
    AssertSame(int, BaseType_t<const int** const **&>);
```

2.3 Removing all consts in a type. (Only for VG)

Write a struct RAC which removes all const within a type, it's enough if you handle * and const. `"const int const * const * const"` should result in `"int * *"`.
As parallel look on the recursive function for removing all "c" in a string:

```
string RemoveC(string input, string output) {
    if (input == "")
        return output;
    else if (input[0] == 'c')
        return RemoveC(input.substr(1), output);
    else
        return RemoveC(input.substr(1), output + input[0]);
}

string RemoveC(string input) {
    return RemoveC(input, "");
}
```

3 Få fram typen på ett uttryck (expression)

Vi vill ha en funktion som summerar alla värden i en Container t.ex.:

```
template <class CONT>
auto Sum2(const CONT& c) {
    typename CONT::value_type sum{};
    for (auto i : c)
        sum += i;
    return sum;
}
```

men nu ska vi som övning hantera detta utan att använda ” ::value_type”.
För att få tag på värdetypen i containern så kan vi titta på värdet av första elementet i den:

```
*std::begin(c)
```

Typoperatorn decltype kan sen ge oss typen: `decltype(*c.begin())`
För att testa vad för typ skriv:

```
decltype(*std::begin(c)) x = (*std::begin(c));
```

och stega i i dubuggern.
Ett tips är att i <type_traits> finns std::remove_const och std::remove_reference.

Anropet av DoSum är:

```
std::vector<int> v = { 1,2,3,-9 };
int sum = DoSum(v);
```

4 SFINAE

Select algorithm depending on if the data is noexcept copyable. Noecept copyable means that you can do a copy of the object without any exeption throw. Example is float that is noexcept copyable and std::string that is not (since it may make an heap allocation when copying). Sometimes you want to give strong exception guaranty and then the algorithm used can depend on if it is possible to copy the data without exceptions (it's even more common with moveable but we take copyable now). The reason for this will be clear later in the course so you just have to accept it for now.
The STL has a struct for helping with this:

std::is_nothrow_copy_constructible<T>::value is true if T is noexcept copy constructible.

Use this and SFINAE to write a template function

```
bool NoThrowCopyConstructible(...) ...
```

it should answer true and use value call or false and reference call depending on std::is_nothrow_copy_constructible.
OBS! You have to write two function definitions and use SFINAE to select between them.