Week 1: II

Exercise 2

../2/conversion.h

```
#ifndef _CONVERSIONT
#define _CONVERSIONT

template <typename OutputT, typename InputT>  // Two types
OutputT as(InputT &inputVar)                   // Return outT, input inT
{
  return static_cast<OutputT>(inputVar);      // Cast inT to outT
};

#endif
```

../2/main.ih

```
#define ERR(msg) printf("%s : %d", (msg), __LINE__)

#include "conversion.h"

#include <iostream>     // For testing/printing

using namespace std;
```

../2/main.cc

```
#include "main.ih"

int main(int argc, char const **argv)
{
  int chVal = 'X';
  cout << as<char>(chVal) << '\n';
}
```

Exercise 3

../3/rawCapacity.h

```
#ifndef _RAWCAPACITYT
#define _RAWCAPACITYT

#include <cstddef>                // For size_t

template <typename TypeT>         // One var type
TypeT* rawCapacity(size_t noVars) // Return pointer to specified type
{
  return static_cast<TypeT*>( operator new(noVars * sizeof(TypeT)) );
};                                  // P to array of noVars var type

#endif
```

../3/main.ih

```
#define ERR(msg) printf("%s : %d", (msg), __LINE__)

#include "rawCapacity.h"

#include <string>   // For the example used
#include <iostream> // For printing (testing)

using namespace std;
```

../3/main.cc

```
#include "main.ih"

int main(int argc, char const **argv)
{
  string *pStringArray = rawCapacity<string>(10); // Room for 10 strings
  operator delete(pStringArray);
}
```

Exercise 7

../7–T/exception/exception.h

```
#ifndef INCLUDED_EXCEPTION_
#define INCLUDED_EXCEPTION_

#include <string>
#include <sstream>
#include <exception>

class Exception: public std::exception
{
  std::string d_what;

  public:
    Exception() = default;

    char const *what() const noexcept(true) override;

    template <typename InputT>
    friend Exception &&operator<<(Exception &&in, InputT anyT);

};

#include "inserterT.h"

#endif
```

../7–T/exception/exception.ih

```
#include "exception.h"

using namespace std;
```

../7–T/exception/inserterT.h

```
#ifndef INCLUDED_INSERTERT_
#define INCLUDED_INSERTERT_

template <typename InputT>
Exception &&operator<<(Exception &&in, InputT anyT)
{
  std::ostringstream ss;
  ss << anyT;
  in.d_what += ss.str();
  return std::move(in);
}

#endif
```

../7–T/exception/what.cc

```
#include "exception.ih"

char const *Exception::what() const noexcept(true)
{
  return d_what.c_str();
}
```

../7–T/main.ih

```
#define ERR(msg) printf("%s : %d", (msg), __LINE__)

```

```
3  #include "exception/exception.h"
4  #include <iostream>
5
6  using namespace std;
```

../7–T/main.cc

```
1  #include "main.ih"
2
3  int main(int argc, char **argv)
4  try
5  {
6    size_t exSizet = 5;
7
8    throw Exception{} << "insert anything that's ostream-insertable: "
9                      << "strings" << ", values "
10                     << exSizet << " etc., etc." << argc;
11 }
12
13 catch (exception const &ex)
14 {
15   cout << ex.what() << '\n';
16 }
```

Exercise 9

• Why is the scope resolution operator required when calling max()?
Because there is a function called `std::max` in the standard namespace. Since we are specifying the use of that namespace, simply calling `max()` will result in the compiler assuming that we want to use that one. Instead, by using the operator `::` we instead force a call to the function in the global namespace.

• When compiling this function the compiler complains (...) Why doesn't the compiler generate a max(double, double) function?
Because the types of the two inputs (3.5 and 4) are different. The one is a double, and the other an int (as far as the compiler is concerned, at least). Hence, there are two different types input to the template, which is only constructed to take two inputs of the *same* type.

• Assume we add a function (...) to the source. Explain why this solves the problem.
Now, this function is chosen over the template, as non-template functions are preferred if both are available. Then, though implicit conversion, the previously integer variable is 'forced' into being a double for use in the function.

• Assume we would then call ::max('a', 12). Which max() function is then used and why?
The non-template function, again. As before, the template is only designed to take in two arguments of the same type. The template would not even work in this case. Hence, the non-template version of `max()` is used, whereby 'a' is converted to a `double` with a value of 97.

• Remove the additional max function. (...), how can we get the compiler to compile the source properly?
One option is to allow for arguments of two different types in the template header and argument list, as shown below. Note that the parameters are passed by value, to avoid returning a reference to a temporary value. The return type can be either the type of the left or right argument, accomplished by using the `auto` specifier.

```
1 template <typename lTypeT, typename rTypeT>
2 inline auto const max(lTypeT const left, rTypeT const right)
```

• Specify a general characteristic of the answer to the previous question (i.e., can the approach always be used or are there certain limitations?).
This approach works as long as the two types can be compared using the `larger-than operator`. More generally, if the types used are compatible in the operations performed in the template, it should work. For example, a string will not work; the compiler will try all conversions it can think of to make the types compatible in this manner, but it cannot find a suitable conversion.