Week 1: II

Exercise 1

../1/main.ih

```
1  #include <iostream>
2
3  namespace First
4  {
5    enum Enum
6    {
7      FIRST
8    };
9
10   void fun(First::Enum symbol)
11   {
12     std::cout << "First::fun called" << '\n';
13   }
14 }
15
16 namespace Second
17 {
18   void fun(First::Enum symbol)
19   {
20     std::cout << "Second::fun called" << '\n';
21   }
22 }
```

../1/main.cc

```
1  #include "main.ih"
2
3  int main(int argc, char const **argv)
4  {
5    fun(First::FIRST);
6  }
```

Why is First::fun called?
The namespace of the function that is called is determined by the argument used in the function call. In that case, this is First, and therefore First::fun() is called.

How would Second::fun be called?
By explicitly calling it using its namespace: `Second::fun()`.

How is the `<< operator` simplified by a Koenig lookup?
As the `<< operator` from the standard library is the only defined or declared within the bounds of the program that takes the arguments as specified, there is no confusion as to what the intended use of the `<< operator` is. Therefore, it can be used simply as is typically seen in programs, rather than its full form `std::operator<<(std::cout, "string")`. If there are multiple different possibilities as to the desired operator (i.e. function), then it would not work like this.

What happens if another `fun(First::Enum)` is defined above `main()`?
This is an example of what was described before. The call to `fun()` is now ambiguous, because there are two functions that take an enum variable from the namespace First named fun. Hence, the compiler does not know which of the two to choose; it is not evident. Put differently, while it is possible to define two functions with the same name within the same scope, the choice between them must be able to be made based on 'contextual clues', in this case the argument list.

Exercise 2

Why doesn't this work?
First of all, str is not yet defined, so it cannot be passed to promptGet. Secondly, and more to the point, the function promptGet is to return a boolean variable, but the getline function returns a pointer, not a boolean value. Up to now, we have seen getline used in such loops directly (i.e. while(getline()), but this only works because it is converted to fit a boolean comparison.

Change promptGet's body so that the code does compile.
One option would be to alter the return function to make use of a simple if-statement:

```
1  bool promptGet(istream &in, string &str)
2  {
3    cout << "Enter a line or ^D\n";      // ^D signals end-of-input
4
5    if (getline(in, str))
6      return true;
7
8    return false;
9  }
```

Without changing promptGet's body, change promptGet so that the code does compile.
As the return type of getline is &istream (in this case), we can change the header of promtGet to accord, as follows:

```
1  istream &promptGet(istream &in, string &str)
2  {
3    cout << "Enter a line or ^D\n";      // ^D signals end-of-input
4
5    return (getline(in, str));
6  }
```

Now, the implicit operators as mentioned before are used in main to allow for their use in the while-statement.

Exercise 3

../3/main.cc

```
#include "main.ih"

void show(Strings const &str, size_t from, size_t to)
{
    for (; from != to; ++from)
        cout << str[from] << '\n';
}

int main()
{
    Strings str{ environ };

    cout << "The 5th environment definition: " << str[4] << "\n"
            "The 4th character of that definition: " << str[4][3] << '\n';

    show(str, 0, 4);            // show 4 strings
}
```

../3/strings/strings.ih

```
#include "strings.h"
#include <string>
#include <cstring>              // -> memcopy, in swap.cc



using namespace std;
```

../3/strings/stringshandin.h

```
#ifndef INCLUDED_STRINGS_
#define INCLUDED_STRINGS_

#include <iosfwd>

class Strings
{
    size_t d_size = 0;
    size_t d_capacity = 1;
    std::string **d_str;            // now a double *

    public:
        Strings();

        Strings(char **environLike);

        ~Strings();

        size_t size() const;

        void add(std::string const &next);          // add another element

        std::string &operator[](size_t idx);       //index operators
        std::string const &operator[](size_t idx) const;

    private:

        std::string **storageArea();                // to store the next str.
        void destroy();
        std::string **enlarged();                   // to d_capacity
        static std::string **rawPointers(size_t nPointers);
```

```
32
33          std::string &element(size_t idx) const;
34  };
35
36  inline size_t Strings::size() const          // potentially dangerous practice:
37  {                                            // inline accessors
38      return d_size;
39  }
40
41
42
43  #endif
```

../3/strings/operatorIndex.cc

```
1   #include "strings.ih"
2
3   string &Strings::operatorIndex(size_t idx) const
4   {
5     if (idx < d_size)
6       return *d_str[idx];
7
8     return "Out of bounds"; //throw would be nice, like in the annotations
9                             //but that is not covered at this points.
10  }
```

../3/strings/indexoperator1.cc

```
1   #include "strings.ih"
2
3   string &Strings::operator[](size_t idx)
4   {
5     return operatorIndex(idx);
6   }
```

breaks the principle of no surprises.

../3/strings/indexoperator2.cc

```
1   #include "strings.ih"
2
3   string const &Strings::operator[](size_t idx) const
4   {
5     return operatorIndex(idx);
6   }
```

03

Exercise 4

../4/main.cc

```
1  #include "main.ih"
2
3  int main(int argc, char **argv)
4  {
5      Strings str2;
6
7      cin >> str2; //insert lines ast string objects, each line being a sepate
8                   //string.
9
10     cout << '\n' << str2;
11 }
```

../4/strings/strings.h

```
1  #ifndef INCLUDED_STRINGS_
2  #define INCLUDED_STRINGS_
3
4  #include <iosfwd>
5  #include <string>
6
7  class Strings
8  {
9      friend std::ostream &operator<<(std::ostream &out, Strings const &rvalue);
10     friend std::istream &operator>>(std::istream &in, Strings &rvalue);
11
12     size_t d_size = 0;
13     size_t d_capacity = 1;
14     std::string **d_str;            // now a double *
15
16     public:
17         Strings();
18
19         Strings(int argc, char *argv[]);
20         Strings(char **environLike);
21
22         ~Strings();
23
24         size_t size() const;
25         size_t capacity() const;
26         std::string const &at(size_t idx) const;     // for const-objects
27         std::string &at(size_t idx);                 // for non-const objects
28
29         void add(std::string const &next);           // add another element
30
31         void resize(size_t newSize);
32         void reserve(size_t newCapacity);
33
34         void swap(Strings &other);
35
36
37     private:
38         std::ostream &insertInto(std::ostream &out) const;//performing the
39                                                           //insertion
40         std::istream &extractFrom(std::istream &in);//performing the extraction
41
42         std::string &safeAt(size_t idx) const;       // private backdoor
43         std::string **storageArea();                 // to store the next str.
44         void destroy();
45         std::string **enlarged();                    // to d_capacity
46         static std::string **rawPointers(size_t nPointers);
47
```

```
48  };
49
50
51  inline size_t Strings::size() const          // potentially dangerous practice:
52  {                                            // inline accessors
53      return d_size;
54  }
55
56  inline size_t Strings::capacity() const      // potentially dangerous practice:
57  {                                            // inline accessors
58      return d_capacity;
59  }
60
61  inline std::string const &Strings::at(size_t idx) const
62  {
63      return safeAt(idx);
64  }
65
66  inline std::string &Strings::at(size_t idx)
67  {
68      return safeAt(idx);
69  }
70
71
72  inline std::ostream &operator<<(std::ostream &out, Strings const &rvalue)
73  {
74    return rvalue.insertInto(out);
75  }
76
77  inline std::istream &operator>>(std::istream &in, Strings &rvalue)
78  {
79    return rvalue.extractFrom(in);
80  }
81
82
83  #endif
```

../4/strings/strings.ih

```
1  #include "strings.h"
2  #include <string>
3  #include <cstring>                // -> memcopy, in swap.cc
4  #include <iostream>
5
6  using namespace std;
```

../4/strings/extractFrom.cc

```
1  #include "strings.ih"
2
3  std::istream &Strings::extractFrom(std::istream &in)
4  {
5    string line;
6    while(getline(in, line))
7    {
8      add(line);
9    }
10   return in;
11 }
```

../4/strings/insertInto.cc

```
1  #include "strings.ih"
2
3  std::ostream &Strings::insertInto(std::ostream &out) const
```

```
4 {
5   for (size_t idx = 0; idx < d_size; ++idx)
6         out << *d_str[idx] << '\n';
7
8   return out;
9 }
```

§ 4

Exercise 6

../6/msg.h

```
1  #ifndef INCLUDED_MSGH
2  #define INCLUDED_MSGH
3
4  enum class Msg
5  {
6      NONE    = 0      ,
7      DEBUG   = 1 << 0,
8      INFO    = 1 << 1,
9      NOTICE  = 1 << 2,
10     WARNING = 1 << 3,
11     ERR     = 1 << 4,
12     CRIT    = 1 << 5,
13     ALERT   = 1 << 6,
14     EMERG   = 1 << 7,
15     ALL     = (1 << 8) - 1
16 };
17
18 #endif
19
20 inline int valueOf(Msg message)          // Return int representation of enum
21 {
22   return static_cast<int>(message);
23 };
24 inline Msg enumOf(int enumInt)           // Return enum representation of int
25 {
26   return static_cast<Msg>(enumInt);
27 };
28
29 inline Msg operator&(Msg a, Msg b)       // AND
30 {
31   return enumOf(valueOf(a) & valueOf(b));
32 };
33 inline Msg operator~(Msg a)              // NOT
34 {
35   return enumOf(~valueOf(a));
36 };
37 inline Msg operator|(Msg a, Msg b)       // OR
38 {
39   return enumOf(valueOf(a) | valueOf(b));
40 };
41 inline Msg operator^(Msg a, Msg b)       // XOR
42 {
43   return enumOf(valueOf(a) ^ valueOf(b));
44 };
45
46 inline Msg operator==(Msg a, Msg b)      // Equal to
47 {
48   return enumOf(valueOf(a) == valueOf(b));
49 };
50 inline Msg operator!=(Msg a, Msg b)      // Not equal to
51 {
52   return enumOf(valueOf(a) != valueOf(b));
53 };
54 inline Msg operator<(Msg a, Msg b)       // Smaller than
55 {
56   return enumOf(valueOf(a) < valueOf(b));
57 };
58 inline Msg operator>(Msg a, Msg b)       // Larger than
59 {
60   return enumOf(valueOf(a) > valueOf(b));
61 };
62 inline Msg operator<=(Msg a, Msg b)      // Smaller or equal to
```

```
63 {
64   return enumOf(valueOf(a) <= valueOf(b));
65 };
66 inline Msg operator>=(Msg a, Msg b)        // Larger or equal to
67 {
68   return enumOf(valueOf(a) >= valueOf(b));
69 };
```

../6/main.ih

```
1 #include "msg.h"
2
3 #include <iostream>
4
5 void show(Msg message);
6
7 using namespace std;
```

../6/main.cc

```
1 #include "main.ih"
2
3 int main(int argc, char const **argv)
4 {
5   show(Msg::NONE);
6   show(Msg::NONE | Msg::EMERG);
7   show(Msg::ALERT | Msg::CRIT);
8   show(Msg::ALL   & (Msg::ERR | Msg::WARNING));
9   show(~Msg::NOTICE);
10 }
```

../6/show.cc

```
1 #include "main.ih"
2
3 char const* msgN[10] =
4 {
5   "DEBUG",     // 1
6   "INFO",      // 2
7   "NOTICE",    // 4
8   "WARNING",   // 8
9   "ERR",       // 16
10  "CRIT",      // 32
11  "ALERT",     // 64
12  "EMERG",     // 128
13  "ALL",       // 255
14  "NONE"
15 };
16
17 void show(Msg message)
18 {
19   if (valueOf(message) == 0)            // Seperate case for NONE
20     std::cout << msgN[9] << ' ';
21
22   for (size_t idx = 0; idx != 9; ++idx) // Loop to identify codes
23   {
24     if (valueOf(message) == (1 << idx)) // Shift performed here to use idx in []
25     {
26       std::cout << msgN[idx] << ' ';
27       break;
28     }
29   }
30   std::cout << '\n';
31 }
```

Output of the given program:

```
1 NONE
2 EMERG
3
4
5
6
```