Week 2

Exercise 11

../11/main.ih

```
#include <iostream>
#include <string>
#include <exception>

namespace Icmbuild
{
    extern char version[];
    extern char years[];
    extern char author[];
};

void usage(std::string const &progname);

using namespace std;
```

../11/main.cc

```
#include "main.ih"


int main(int argc, char **argv)
{
  string line;

  while (true)
  {
    try{

      while (true)
        {
            cout << "please enter a number: ";
            getline(cin, line);
            if (line == "q")
            {
                cout << "leaving...\n";
                return 0;
            }
            cout << stod(line) << '\n';
        }
    }

    catch (...)
    {
      cout << '`' << line << "' is not a number" << '\n';
    }
  }
}
```

Exercise 12

../12/main.cc

```
#include "main.ih"

namespace
{
    char version[] = "1.00.00";

    Arg::LongOption longOptions[] =
    {
        Arg::LongOption{"required", Arg::Required},
        Arg::LongOption{"debug"},
        Arg::LongOption{"filenames", 'f'},
        Arg::LongOption{"help", 'h'},
        Arg::LongOption{"version", 'v'},
    };
    auto longEnd = longOptions + size(longOptions);
}

int main(int argc, char **argv)
try
{
    Arg &arg = Arg::initialize("vha:bf:", longOptions, longEnd, argc, argv);

    cout << arg.basename() << '\n';

    arg.versionHelp(usage, version, 1);

    for (size_t idx = 0; idx != arg.nArgs(); ++ idx)
        cout << "arg " << idx << ": " << arg.arg(idx) << '\n';

    cout << "n options: " << arg.nOptions() << "\n"
            "n Args: " << arg.nArgs() <<
            '\n';

    string value;
    size_t count = arg.option(&value, 'a');

    cout <<
        "option a count: " << count << ", value: " << value << "\n"
        "option b count: " << arg.option('b') <<
        '\n';

    count = arg.option(&value, 'f');

    cout <<
        "option f count: " << count << ", value: " << value <<
        '\n';

    count = arg.option(&value, "required");

    cout <<
        "option debug count: " << arg.option(0, "debug") << "\n"
        "option required count: " << count << ", value: " << value <<
        '\n';
}

catch (char ch)
{
  if (ch == 1)
    return 1;
  throw; //not the error we intended to catch so handle somewhere else
}

```

```
63 catch (...)
64 {
65     cout << "done\n";
66 }
```

../12/arg/arg1.cc

```
1  #include "arg.ih"
2
3      // for the time being: exit is called on errors.
4
5  Arg::Arg(char const *optstring, int argc, char **argv)
6  {
7      string opts{ prepareArg(optstring, argv[0]) };
8
9      while (true)                    // process all options
10     {
11                                     // get the next option
12         switch (int opt = getopt(argc, argv, opts.c_str()))
13         {
14             case -1:        // all options processed: get the arguments
15                 copyArgs(argv + optind, argv + argc);
16             return;
17
18             case ':':
19                 cerr << "Option value missing for `-" << optopt << "'\n";
20             throw char(1);
21
22             case '?':
23                 cerr << "Unknown option `-" << optopt << "'\n";
24             throw char(1);
25
26             default:
27                 d_option.add(opt);
28             break;
29         }
30     }
31 }
```

../12/arg/arg2.cc

```
1  #include "arg.ih"
2
3      // for the time being: exit is called on errors.
4
5  Arg::Arg(char const *optstring,
6           LongOption const *const begin, LongOption const *const end,
7           int argc, char **argv)
8  {
9      string opts{ prepareArg(optstring, argv[0]) };
10
11                                     // create array of n structs for long option
12                                     // specifications (the final one must be 0,
13                                     //  getoptlong requirement)
14     OptStructArray optStructs{ static_cast<size_t>(end - begin + 1) };
15     fillLongOptions(optStructs.get(), optstring, begin, end);
16
17     int longOptionIndex;        // receives the index of the long options
18     while (true)
19     {                            // get the next option
20         switch (int opt = getopt_long(argc, argv, opts.c_str(),
21                                       optStructs.get(), &longOptionIndex))
22         {
23             case -1:        // all options processed: get the arguments
24                 copyArgs(argv + optind, argv + argc);
```

```
25              return;
26
27             case ':':
28                  cerr << "Option value missing for `--" << optopt << "'\n";
29              exit(1);
30
31              case '?':
32                  cerr << "Unknown option `--" << optopt << "'\n";
33              exit(1);
34
35              case 0:
36                  opt = longOptionChar(begin[longOptionIndex]);
37                  if (opt == 0)
38                      break;
39              [[fallthrough]];
40
41              default:
42                  d_option.add(opt);
43              break;
44          }
45      }
46  }
```

../12/arg/initialize1.cc

```
1   #include "arg.ih"
2
3   // static
4   Arg &Arg::initialize(char const *optstring, int argc, char **argv)
5   {
6       if (s_arg)                          // only 1 Arg object is allowed
7       {
8           cerr << "Arg::initialize(): already initialized\n";
9           throw char(1);
10      }
11
12      s_arg = new Arg(optstring, argc, argv); // construct the object
13
14      return *s_arg;                          // return its reference.
15  }
```

../12/arg/initialize2.cc

```
1   #include "arg.ih"
2
3   // static
4   Arg &Arg::initialize(char const *optstring,
5                   LongOption const *const begin,
6                   LongOption const *const end,
7                   int argc, char **argv)
8   {
9       if (s_arg)                          // only 1 Arg object is allowed
10      {
11          cerr <<  "Arg::initialize(): already initialized\n";
12          throw char(1);
13      }
14
15                                          // construct the object
16      s_arg = new Arg(optstring, begin, end, argc, argv);
17
18      return *s_arg;                      // return its reference
19  }
```

../12/arg/instance.cc

```
1   #include "arg.ih"
```

```
2
3   // static
4   Arg &Arg::instance()
5   {
6       if (not s_arg)                  // instance requires an initialized object
7       {
8           cerr << "Arg::instance(): not yet initialized";
9           throw char(1);
10      }
11
12      return *s_arg;                  // return its reference if available
13  }
```

../12/arg/setoptiontype.cc

```
1   #include "arg.ih"
2
3   int Arg::setOptionType(string const &optString,
4                          LongOption const &longOption)
5   {
6       string::size_type pos = optString.find_first_of(longOption.optionChar());
7
8       if (pos == string::npos)
9       {
10          cerr << "Arg::setOptionType()" << ": short option `" <<
11                  static_cast<char>(longOption.optionChar()) << "' not found\n";
12          throw char(1);
13      }
14
15      return optString[pos + 1] == ':' ? Required : None;
16  }
```

../12/arg/versionhelp.cc

```
1   #include "arg.ih"
2
3       // using exit for the time being...
4
5   void Arg::versionHelp(void (*usage)(string const &progname),
6           char const *version, size_t minArgs, int helpFlag,
7           int versionFlag) const
8   {
9       if (option(versionFlag) && !option(helpFlag))       // show the version
10      {                                                   // on request, unless
11          cout << basename() << " V" << version << '\n';  // -h was requested
12          throw char(1);
13      }
14
15      if (nArgs() < minArgs || option(helpFlag))  // provide usage on request
16      {                                           // or if too few arguments
17          usage(basename());
18          throw char(1);
19      }
20  }
```

Exercise 13

using throw rethrows the existing exception object. So no new objects are made
(Source https/en.cppreference.com/w/cpp/language/throw)

../13/main.ih

```
#include <iostream>
#include <string>
#include "test/test.ih"

namespace Icmbuild
{
    extern char version[];
    extern char years[];
    extern char author[];
};

void usage(std::string const &progname);

using namespace std;
```

../13/main.cc

```
#include "main.ih"

int main(int argc, char **argv)
try
{
  test testone(3);
  test testtwo = testone;

  cout << testtwo.counter() << '\n'; //counter is 2 here

  throw(testtwo);  //this throw results in a counter of 4 so 2 more copies
                   //are made. (4 without the rethrowing)
  throw(&testone); //this throw ( if the line above is removed )
                   //results in a counter of 2, so no more copies are made
}

catch (test object)
{
    cout << object.counter() << '\n';
    return 1;
}


catch (test *object)
{
    cout << (*object).counter() << '\n';
    return 1;
}

catch (...)
{
    return 1;
}
```

../13/test/test.ih

```
#include "test.h"
#include <iostream>
//#define CERR std::cerr << __FILE__": "

using namespace std;
```

../13/test/test.h

```
#ifndef INCLUDED_TEST_
#define INCLUDED_TEST_


class test
{
        inline static int s_counter = 0;

        int d_value = 0;

    public:
        test();
        test(int value);
        test(test &copy);
        ~test();

        static int counter();
        int value();

    private:
};

#endif
```

../13/test/counter.cc

```
#include "test.ih"

int test::counter()
{
  return s_counter;
}
```

../13/test/test1.cc

```
#include "test.ih"

test::test()
//:
{
  ++s_counter;
}
```

../13/test/test2.cc

```
#include "test.ih"

test::test(test &copy)
:
  d_value(copy.d_value)
{
  ++s_counter;
}
```

../13/test/test3.cc

```
#include "test.ih"

test::test(int value)
:
  d_value(value)
{
```

```
7     ++s_counter;
8   }
```

../13/test/value.cc

```
1   #include "test.ih"
2
3   int test::value()
4   {
5     return d_value;
6   }
```

Exercise 14

As can be seen, there are two try blocks in this program. The first is the one in `main()`, which tries to construct an array of ten `MaxFour` objects, and catches any string-type exception, printing its message if encountered. This is just for the benefit of the user. The second is in the default constructor of the class. The constructor first allocates a new string in `d_content`, increments the object counter, and throws an exception if this counter surpasses four objects. In that case, the current object is incomplete, and a destructor would not be called for it at the program's end. This would result in a memory leak with the size of a single object, because after this incomplete objects the array construction is halted. However, a simple `catch(...)` block that follows this try block with the same content as the destructor is enough to delete what has been allocated, effectively constituting object destruction right there and then.
The reason this is so simple is because this manner of programming is very intuitive. The constructor is asked to perform allocation, and if something goes wrong, to revert that allocation.

../14/maxfour/maxfour.h

```
1  #ifndef INCLUDED_MAXFOUR_
2  #define INCLUDED_MAXFOUR_
3
4  #include <cstddef>
5  #include <string>
6
7  class MaxFour
8  {
9    public:
10     MaxFour();
11     ~MaxFour();
12
13   private:
14     inline size_t static objCount = 0;
15     std::string *d_content;
16 };
17
18 #endif
```

../14/maxfour/maxfour.ih

```
1  #include "maxfour.h"
2
3  using namespace std;
```

../14/maxfour/c_maxfour.cc

```
1  #include "maxfour.ih"
2
3  MaxFour::MaxFour()
4  try
5  :
6  d_content(new std::string("hello"))
7  {
8    ++objCount;
9    if (objCount > 4)
10     throw string{ "max. number of objects reached" };
11 }
12 catch (...)
13 {
14   delete d_content;
15 }
```

../14/maxfour/d_maxfour.cc

```
1  #include "maxfour.ih"
2
3  MaxFour::~MaxFour()
```

```
4  {
5    delete d_content;
6  }
```

../14/main.ih

```
1  #include "maxfour/maxfour.h"
2
3  #include <iostream>
4
5  using namespace std;
```

../14/main.cc

```
1  #include "main.ih"
2
3  int main(int argc, char const **argv)
4  try
5  {
6    MaxFour classArray[10];
7  }
8  catch (string message)
9  {
10   cerr << message << '\n';
11 }
```

Exercise 16

Consider the example program below. It is a bit contrived, because it is clear that this would lead to memory leaks. However, if one imagines this type of condition to be part of a much, much larger program, then it can serve as an illustration of why using `exit()` is a bad idea. Say that the string allocation is part of the construction of a certain class. Additionally, its destructor would return this memory, represented by the last line in this example. However, along the way, something horrible happens which should prompt the program to stop. This way of achieving that goal would mean that the destructor is not called, thus causing a memory leak. Furthermore, unless properly embedded in a function that is more descriptive, does not provide any form of information about what went wrong, or why. Throwing an exception in this case would allow for the programming of a deallocation procedure for when such a situation occurs, but more importantly, forces the programmer to think about the proper approach that may even allow the program to continue functioning despite this error. If not, a try/catch block would allow the program to come to a more graceful ending, whereby at least destructors are called.

../16/main.ih

```
1  using namespace std;
2
3  #include "string"
```

../16/main.cc

```
1  #include "main.ih"
2
3  int main(int argc, char const **argv)
4  {
5    bool exceptionalCondition = true;
6
7    string *stringArray;
8    stringArray = new string{ "hello" };
9
10   if (exceptionalCondition)
11     exit(1);
12
13   delete stringArray;
14 }
```

Exercise 17

Is the default constructor's implementation exception safe? If not, how would you change it so that it is exception safe?
It is not. It can be changed similar to the approach that is implemented in the class MaxFour (see exercise 14). In other words, as such:

Listing 1: c_strings.cc

```
1  #include "strings.ih"
2
3  Strings::Strings()
4  try
5  :
6    d_str(rawPointers(1))
7  {}
8  catch (...)
9  {
10   delete d_str;
11 }
```

What happens if the default constructor is called from another constructor (using constructor delegation) in these cases:

- *The default constructor fails and throws an exception*
  The only thing that can fail is the allocation in the initialiser list. Hence, memory would be allocated, but the object is not constructed properly, and thus would not be deconstructed either, leading to memory leaks. However, because of the try/catch block, if an exception occurs, it is caught immediately and the memory subsequently deallocated again.
- *The constructor calling the default constructor using constructor delegation fails and throws an exception*
  That depends on the contents of that second constructor. If it allocates some more memory, that will lead to memory leaks as well. However, if it does not, and does not reach the point where it calls the default constuctor, there is no problem. The exception that it throws can be caught elsewhere and perhaps displayed to the user, but essentially the situation before the constructor is called is restored either way (i.e. rolled back).

Note: these answers assume that the questions pertain to the situation *after* the change suggested above.