Week 2 - II

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. The objects that have been allocated in the process of creating the array of objects simply no longer exist after the exception is thrown, as it goes out of scope. This is why the naming of the try-catch block makes sense, since it did not succeed in its attempt, the original situation is reverted.

../14/maxfour/maxfour.h

```
1  #ifndef INCLUDED_MAXFOUR_
2  #define INCLUDED_MAXFOUR_
3
4  #include <cstddef>
5  #include <string>
6  #include <iostream>
7
8  class MaxFour
9  {
10   private:
11     inline size_t static s_objCount = 0;
12     std::string *d_content;               // Example to allow for memory allocation
13
14   public:
15     MaxFour();
16     ~MaxFour();
17 };
18
19 #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")) // Example to illustrate new/delete in try
7  {
8    ++s_objCount;
9    if (s_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. The program is as simple as it can be, it just creates one local variable. However, as it needs to store a rather long string, the object allocates more memory of its own. If the string were shorter, the exit call would not lead to problems, but now it does. Namely, the destructor of the class string is not called, and its own allocated memory is not freed. Therefore, this is not a proper way to exit a program. It is also not very descriptive, as the user has no idea what went wrong. Not only can an exception show the user a string that explains what prompted the program to end, but it also constitutes a normal end of the program, which would call destructors as per usual. Lastly, implementing an exception handler forces the programmer to think about the logical flow of a program, and what should actually happen if an exceptional situation arises.

../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 = false;
6
7    string leakingString{ "feighsdefighsdlfghsidfgsidfgisdifgbdsfgbyusdfgh" };
8
9    if (exceptionalCondition)
10     exit(3);
11 }
```

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 for the strings stored by the object by the `rawPointers()` function, but the object is not constructed properly, and thus would not be destroyed 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, as the object has not been fully constructed and will therefore also not be destroyed, which would dealloate the memory. However, if it does not, and does not reach the point where it calls the default constuctor, no memory is allocated, and therefore the original situation is restored. 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.