Week 2

Exercise 11

Exercise 12

Exercise 13

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.