Week 6

Exercise 50

Exercise 51

New / delete variants

New variable or array
Allocates new memory sized appropriately to the type and dimensions specified. Thereafter it will attempt to construct and initialise these objects. Finally, it will return a pointer to the start of the memory allocated to these objects. The advantage and consideration of constructing an object (array) is that one has control over when it is deleted (if at all).

```
1  int main()
2  {
3    Class *pointer;
4    if (boolAppropriate)
5      pointer = new Class;
6
7    if (boolNoLongerNecessary)
8      delete pointer;
9  }
```

Without the use of new (i.e. Class newClass), the scope of the newly constructed class would simply be limited to within the if statement, and thereafter destroyed. However, now the object persists until manually deleted. As such, the creation and destruction do not take place unless necessary. Another example would be in the creation of arrays with dimensions that are not previously established.

Placement new
Allows for 'filling' an already allocated piece of memory, possibly with another object or objects than the one one / those that it was originally allocated for. In other words, it allows us to simply construct objects in memory previously allocated. This could be useful when imagining a shared or transient storage location in memory that can be easily located, whether it contains characters, integers, or whatever else.

```
1  int main()
2  {
3    char block[10 * sizeof(ExClass)]; // Block of memory necessary later
4
5    if (boolNeedsTypeA)
6      for (size_t idx = 0; index != 10; ++index)  // Create ten 'ExClass's in the
7        ExClass *sEC = new(block + idx * sizeof(ExClass)) // space, using constructor
8                          ExClass(typeA);                 // A.
9
10   if (boolNeedsTypeB)
11     for (size_t idx = 0; index != 10; ++index)  // Create ten 'ExClass's in the
12       ExClass *sEC = new(block + idx * sizeof(ExClass)) // space, using constructor
13                         ExClass(typeB);                 // B.
14
15   sEC->~ExClass();        // Destruct the data
16   delete[] block;         // Deallocate the memory
17 }
```

In this example, we know that a block of memory sized to fit ten 'ExClass's is required, but not yet which constructor will be used to fill it. Hence, it can already be allocated, but left empty until that decision is made. In the end,

Operator new
This allocates raw memory sized to fit a specified number of specified objects.

```
1  int main()
2  {
3    string *block = static_cast<string *>(operator new(8 * sizeof(std::string)));
4
5    string *pNames = new(block + 5) std::string("John");
6
7    pNames->~string();
8
9    operator delete(block);
10 }
```

Without actually filling this memory, it is rather useless on its own. Instead, it can be viewed as a preferable alternative to placement new discussed previously. Even without repeating/rewriting the loop, it is already obvious that the notation is simpler: there is no continuous evaluation of sizeof throughout the loop. Instead, just the offset index is used.

Exercise 52

As can be seen, the new addition consists of a destructor *Strings*() defined in the header and implemented in *dtor.cc*. In turn, this destructor calls the previously-defined *destroy*() (see *destoy.cc*). The destructor is automatically called when the object goes out of scope, so this ensures that the memory allocated by the objects is deleted in such a situation (as it would be before enlarging it).

../52/strings/strings.h

```
1  #ifndef INCLUDED_STRINGS_
2  #define INCLUDED_STRINGS_
3
4  #include <iosfwd>
5
6  class Strings
7  {
8    size_t d_size;
9    std::string *d_str;
10
11   public:
12     struct POD
13     {
14       size_t      size;
15       std::string *str;
16     };
17
18     Strings();
19     Strings(int argc, char *argv[]);
20     Strings(char *environLike[]);
21     Strings(std::istream &in);
22
23     ~Strings(); // New addition: destructor.
24
25     void swap(Strings &other);
26
27     size_t size() const;
28     std::string const *data() const;
29     POD release();
30
31     std::string const &at(size_t idx) const;    // for const-objects
32     std::string &at(size_t idx);                // for non-const objects
33
34     void add(std::string const &next);          // add another element
35
36   private:
37     void fill(char *ntbs[]);                    // fill prepared d_str
38
39     std::string &safeAt(size_t idx) const;      // private backdoor
40     std::string *enlarge();
41     void destroy();
42
43     static size_t count(char *environLike[]);   // # elements in env.like
44 };
45
46 inline size_t Strings::size() const         // potentially dangerous practice:
47 {                                           // inline accessors
48   return d_size;
49 }
50
51 inline std::string const *Strings::data() const
52 {
53   return d_str;
54 }
55
56 inline std::string const &Strings::at(size_t idx) const
57 {
```

```
58    return safeAt(idx);
59  }
60
61  inline std::string &Strings::at(size_t idx)
62  {
63    return safeAt(idx);
64  }
65
66  #endif
```

../52/strings/dtor.cc

```
1   #include "strings.ih"
2
3   Strings::~Strings()
4   {
5     destroy();
6   }
```

../52/strings/destroy.cc

```
1   #include "strings.ih"
2
3   void Strings::destroy()
4   {
5     delete[] d_str;
6   }
```