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 53

../53/main.ih

```
#include <iostream>

#include "filter/filter.h"

extern char **environ;

using namespace std;
```

../53/main.cc

```
#include "main.ih"

int main()
{
  for (size_t iter = 0; iter != 1000; ++iter)
  {
    Strings env(environ);

    for (size_t rept = 0; rept != 100; ++rept)
    {
      for (char **ptr = environ; *ptr; ++ptr)
        env.add(*ptr);
    }
  }
}
```

../53/strings/strings.h

```
#ifndef INCLUDED_STRINGS_
#define INCLUDED_STRINGS_

#include <iosfwd>

class Strings
{
  size_t d_capacity = 0;
  size_t d_size = 0;
  std::string **d_pPstrings = 0;

  public:
    struct POD
    {
      size_t      size;
      std::string *str;
    };

    Strings();
    Strings(size_t argc, char const *argv[]);
    Strings(char *environLike[]);           // Not const because of testing script
    Strings(std::istream &in);
    ~Strings();

    void swap(Strings &other);

    size_t size() const;
    size_t capacity() const;                    // New addition
    std::string* const *data() const;
    POD release();
    POD d_POD();

    std::string const &at(size_t idx) const;    // for const-objects
```

```
34      std::string &at(size_t idx);                // for non-const objects
35
36      void add(std::string const &next);          // add another element
37
38    private:
39      void fill(char *ntbs[]);                    // fill prepared d_pPstrings
40      void resize(size_t newSize);                // New addition
41      std::string** rawPointers(size_t nNewPointers); // New addition
42      void reserve(size_t newCapacity);           // New addition
43
44      std::string &safeAt(size_t idx) const;      // private backdoor
45      std::string *enlarge();
46      void destroy();
47  };
48
49  inline size_t Strings::size() const         // potentially dangerous practice:
50  {                                           // inline accessors
51    return d_size;
52  }
53
54  inline size_t Strings::capacity() const
55  {
56    return d_capacity;
57  }
58
59  inline std::string const &Strings::at(size_t idx) const
60  {
61    return safeAt(idx);
62  }
63
64  inline std::string &Strings::at(size_t idx)
65  {
66    return safeAt(idx);
67  }
68
69  #endif
```

../52/strings/add.cc

```
1   #include "strings.ih"
2
3   void Strings::add(string const &next)
4   {
5       string *tmp = enlarge();            // make room for the next string,
6                                           // tmp is the new string *
7
8       tmp[d_size] = next;                 // store next
9
10      destroy();                          // return old memory
11
12      d_str = tmp;                        // update d_str and d_size
13      ++d_size;
14  }
```

The destroy(); function is unchanged, but included for convenience.

../53/strings/d_tor.cc

```
1   #include "strings.ih"
2
3   Strings::~Strings()
4   {
5     for (size_t index = 0; index != d_size; ++index)  // For each element
6       delete d_pPstrings[index];  // Delete that element (also call its destructors)
7     destroy();  // Call original destroy(); one last time
8   }
```

../53/strings/rawPointers.cc

```
#include "strings.ih"

string** Strings::rawPointers(size_t nPointers)
{
  return (new string*[nPointers]);
};
```

../53/strings/reserve.cc

```
#include "strings.ih"

void Strings::reserve(size_t newCapacity)
{
  while (d_capacity < newCapacity)  // Keep doubling while capacity is still low
  {
    size_t oldcapacity = d_capacity;  // Old capacity needed to transfer pointers
    if (d_capacity == 0)  // Rather start with no capacity to keep things clean
      d_capacity = 1;     // So, for first time, capacity must be set to 1
    else
      d_capacity *= 2;  // Double capacity when needed

    string **tmp = rawPointers(d_capacity); // Create new pointer to raw pointers
    for (size_t idx = 0; idx != oldcapacity; ++idx) // Transfer over old pointers
      tmp[idx] = d_pPstrings[idx];

    destroy();  // Destroy old pointer
    d_pPstrings = tmp;  // Assign old pointer to new location
  }
};
```

../53/strings/resize.cc

```
#include "strings.ih"

void Strings::resize(size_t newSize)
{
  string newAddition = "";
  if (newSize > d_size)
    for (size_t index = 0; index != newSize - d_capacity; ++index)
      add(newAddition);

  if (newSize < d_size)
    for (size_t index = d_size; index != newSize - 1; index--)
      delete d_pPstrings[index];

  d_size = newSize;
}
```