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).

```
int main()
{
  Class *pointer;
  if (boolAppropriate)
    pointer = new Class;

  if (boolNoLongerNecessary)
    delete pointer;
}
```

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.

```
int main()
{
  char block[10 * sizeof(ExClass)]; // Block of memory necessary later

  if (boolNeedsTypeA)
    for (size_t idx = 0; index != 10; ++index)  // Create ten 'ExClass's in the
      ExClass *sEC = new(block + idx * sizeof(ExClass)) // space, using constructor
                         ExClass(typeA);                  // A.

  if (boolNeedsTypeB)
    for (size_t idx = 0; index != 10; ++index)  // Create ten 'ExClass's in the
      ExClass *sEC = new(block + idx * sizeof(ExClass)) // space, using constructor
                         ExClass(typeB);                  // B.

  sEC->~ExClass();        // Destruct the data
  delete[] block;         // Deallocate the memory
}
```

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.

```
int main()
{
  string *block = static_cast<string *>(operator new(8 * sizeof(std::string)));

  string *pNames = new(block + 5) std::string("John");

  pNames->~string();

  operator delete(block);
}
```

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/strings/strings.h

```cpp
#ifndef INCLUDED_STRINGS_
#define INCLUDED_STRINGS_

#include <iosfwd>

class Strings
{
  size_t d_capacity = 1;
  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
    std::string &at(size_t idx);                 // for non-const objects

    void add(std::string const &next);           // add another element

  private:
    void fill(char *ntbs[]);                     // fill prepared d_pPstrings
    void resize(size_t newSize);                 // New addition
    std::string** rawPointers(size_t nNewPointers); // New addition
    void reserve(size_t newCapacity);            // New addition

    std::string &safeAt(size_t idx) const;       // private backdoor
    std::string *enlarge();
    void destroy();
};

inline size_t Strings::size() const          // potentially dangerous practice:
{                                            // inline accessors
  return d_size;
}

inline size_t Strings::capacity() const
{
  return d_capacity;
}

inline std::string const &Strings::at(size_t idx) const
{
  return safeAt(idx);
}
```

```

inline std::string &Strings::at(size_t idx)
{
  return safeAt(idx);
}

#endif
```

../53/strings/add.cc

```
#include "strings.ih"

void Strings::add(string const &next)
{
  if(d_size + 1 > d_capacity) // If there is no room for the new addition
    reserve(d_size + 1);      // Create new room

  d_pPstrings[d_size] = new std::string{ next };  // Add the new string
  ++d_size; // Increase size
}
```

../53/strings/c_argcargv.cc

```
#include "strings.ih"

Strings::Strings(size_t argc, char const *argv[])
{
  d_pPstrings = rawPointers(1); // Create first memory
  for (size_t index = 0; index != argc; ++index)
    add(argv[index]);
};
```

../53/strings/c_environlike.cc

```
#include "strings.ih"

Strings::Strings(char *environLike[])
{
  d_pPstrings = rawPointers(1); // Create first memory
  for (size_t index = 0; environLike[index] != 0; ++index)
    add(environLike[index]);
};
```

../53/strings/c_istream.cc

```
#include "strings.ih"

Strings::Strings(istream &in)
{
  d_pPstrings = rawPointers(1); // Create first memory
  string line;
  while (getline(in, line))
  {
    add(line);
    if (line.empty())
      break;
  }
};
```

../53/strings/d_tor.cc

```
#include "strings.ih"

Strings::~Strings()
```

```
{
  for (size_t index = 0; index != d_size; ++index)  // For each element
    delete d_pPstrings[index];  // Delete that element (also call its destructors)
  destroy();  // Call original destroy(); one last time
}
```

The destroy(); function is unchanged from ex. 52 (already checked), but included for convenience.

../53/strings/destroy.cc

```
#include "strings.ih"

void Strings::destroy()
{
  delete[] d_pPstrings;
}
```

../53/strings/rawPointers.cc

```
#include "strings.ih"

string** Strings::rawPointers(size_t nPointers)
{
  return (new string*[nPointers]);  // Return pointer to new array of raw pointers
};
```

../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
    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 = "";  // Empty string to use for filling
  if (newSize > d_size) // If newSize is larger than current size
    for (size_t index = 0; index != newSize - d_capacity; ++index)  // Fill
      add(newAddition);

  if (newSize < d_size) // If smaller
    for (size_t index = d_size; index != newSize - 1; index--)
      delete d_pPstrings[index]; // Delete those strings (as in d_tor)

  d_size = newSize; // Set new size to indicated size
                    // If newSize == d_size, this is SF, but better than including
                    // it in every if-statement
}
```

Exercise 55

../55/strings/strings.h

```cpp
#ifndef INCLUDED_STRINGS_
#define INCLUDED_STRINGS_

#include <iosfwd>

class Strings
{
  size_t d_capacity = 1;        // Rather start capacity at 0, see reserve
  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
    std::string &at(size_t idx);                // for non-const objects

    void add(std::string const &next);          // add another element

  private:
    void fill(char *ntbs[]);                    // fill prepared d_pPstrings
    void resize(size_t newSize);                // New addition
    std::string* rawMemory(size_t nNewPointers); // New addition
    void reserve(size_t newCapacity);           // New addition

    std::string &safeAt(size_t idx) const;      // private backdoor
    std::string *enlarge();
    void destroy();
};

inline size_t Strings::size() const          // potentially dangerous practice:
{                                            // inline accessors
  return d_size;
}

inline size_t Strings::capacity() const
{
  return d_capacity;
}

inline std::string const &Strings::at(size_t idx) const
{
  return safeAt(idx);
}
```

```

inline std::string &Strings::at(size_t idx)
{
  return safeAt(idx);
}

#endif
```

../55/strings/add.cc

```
#include "strings.ih"

void Strings::add(string const &next)
{
  if(d_size + 1 > d_capacity) // If there is no room for the new addition
    reserve(d_size + 1);      // Create new room

  new(d_pPstrings + d_size ) std::string{ next };  // Add the new string
  ++d_size; // Increase size
}
```

../55/strings/c_argcargv.cc

```
#include "strings.ih"

Strings::Strings(size_t argc, char const *argv[])
{
  d_pPstrings = rawMemory(1); // Create first memory
  for (size_t index = 0; index != argc; ++index)
    add(argv[index]);
};
```

../55/strings/c_environlike.cc

```
#include "strings.ih"

Strings::Strings(char *environLike[])
{
  d_pPstrings = rawMemory(1); // Create first memory
  for (size_t index = 0; environLike[index] != 0; ++index)
    add(environLike[index]);
};
```

../55/strings/c_istream.cc

```
#include "strings.ih"

Strings::Strings(istream &in)
{
    d_pPstrings = rawMemory(1); // Create first memory
    string line;
    while (getline(in, line))
    {
    add(line);
    if (line.empty())
      break;
    }
};
```

../55/strings/destroy.cc

```
#include "strings.ih"

void Strings::destroy()
```

```
{
  for (string *end = d_pPstrings + d_size; end-- != d_pPstrings; )
    end->~string();
  operator delete(d_pPstrings);
}
```

../55/strings/d_tor.cc

```
#include "strings.ih"

Strings::~Strings()
{
  destroy();  // Call original destroy(); one last time
}
```

../55/strings/rawMemory.cc

```
#include "strings.ih"

string* Strings::rawMemory(size_t nPointers)
{
  string *tmp = static_cast<string *>(operator new(nPointers * sizeof(string)));
  return tmp;
};
```

../55/strings/reserve.cc

```
#include "strings.ih"

void Strings::reserve(size_t newCapacity)
{
  while (d_capacity < newCapacity)  // Keep doubling while capacity is still low
  {
    d_capacity *= 2;  // Double capacity when needed

    string *tmp = rawMemory(d_capacity); // Create new pointer to raw memory
    for (size_t idx = d_size; idx--; )  // Transfer over old strings
      new(tmp + idx) string{ d_pPstrings[idx] };

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

../55/strings/resize.cc

```
#include "strings.ih"

void Strings::resize(size_t newSize)
{
  string newAddition = "";  // Empty string to use for filling
  if (newSize > d_size) // If newSize is larger than current size
    for (size_t index = 0; index != newSize - d_capacity; ++index)  // Fill
      add(newAddition);

  if (newSize < d_size) // If smaller
    for (string *end = d_pPstrings + d_size; end-- != d_pPstrings + newSize; )
      end->~string();

  d_size = newSize; // Set new size to indicated size
                    // If newSize == d_size, this is SF, but better than including
                    // it in every if-statement
}
```

Exercise 56

Type	***Time***
real	0m1,401s
user	0m1,240s
sys	0m0,160s

Table 1: Original

Type	***Time***
real	0m1,519s
user	0m1,381s
sys	0m0,137s

Table 2: Double pointers

Type	***Time***
real	0m17,604s
user	0m17,485s
sys	0m0,112s

Table 3: Placement new

Tables 1 through 3 display the time it took to run the respective programs. Note that the original implementation was timed on another machine because it was taking so long.

It is clear that using double pointers makes the program run fastest. Intuitively, this also makes sense. For both the original implementation as well as the one using placement new one, when creating room for new strings, the extant strings have to be copied entirely. The latter implementation at least uses a doubling algorithm, but still, copying strings is just not very efficient. Simply copying over pointers to already existing objects seems like a much better idea, because depending on the length of said strings, they could be very large - at least much larger than a mere pointer. z