

## Week 6

### Exercise 53

../53/strings/strings.h

```

1 #ifndef INCLUDED_STRINGS_
2 #define INCLUDED_STRINGS_
3
4 #include <iosfwd>
5
6 class Strings
7 {
8     size_t d_capacity = 1;
9     size_t d_size = 0;
10    std::string **d_pPstrings = 0;
11
12    public:
13        struct POD
14        {
15            size_t size;
16            std::string **str;
17        };
18
19        Strings();
20        Strings(size_t argc, char const *argv[]);
21        Strings(char *environLike[]); // Not const because of testing script
22        Strings(std::istream &in);
23        ~Strings();
24
25        void swap(Strings &other);
26
27        size_t size() const;
28        size_t capacity() const; // New addition
29        std::string* const *data() const;
30        POD release();
31        POD d_POD();
32
33        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

```

```

1  #include "strings.ih"
2
3  void Strings::add(string const &next)
4  {
5      if(d_size + 1 > d_capacity) // If there is no room for the new addition
6          reserve(d_size + 1);    // Create new room
7
8      d_pPstrings[d_size] = new std::string{ next }; // Add the new string
9      ++d_size; // Increase size
10 }

```

```

1 #include "strings.ih"
2
3 Strings::Strings(size_t argc, char const *argv[])
4 {
5     d_pPstrings = rawPointers(1); // Create first memory
6     for (size_t index = 0; index != argc; ++index)
7         add(argv[index]);
8 };

```

```
1 #include "strings.ih"
2
3 Strings::Strings(char *environLike[])
4 {
5     d_pPstrings = rawPointers(1); // Create first memory
6     for (size_t index = 0; environLike[index] != 0; ++index)
7         add(environLike[index]);
8 };
```

DRY, delete!

../53/strings/c\_istream.cc

```

1 #include "strings.ih"
2
3 Strings::Strings(istream &in)
4 {
5     d_pPstrings = rawPointers(1); // Create first memory
6     string line;
7     while (getline(in, line))
8     {
9         add(line);
10        if (line.empty())
11            break;
12    }
13 };

```

```
../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 }
```

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

../53/strings/destroy.cc

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

../53/strings/rawPointers.cc

```
1 #include "strings.ih"
2
3 string** Strings::rawPointers(size_t nPointers)
4 {
5     return (new string*[nPointers]); // Return pointer to new array of raw pointers
6 };
```

*clever naming to keep your mind on the fact that we're allocating pointers here, not strings*

*IRE: don't allocate twice. just compute new d\_capacity*

../53/strings/reserve.cc

```
1 #include "strings.ih"
2
3 void Strings::reserve(size_t newCapacity)
4 {
5     while (d_capacity < newCapacity) // Keep doubling while capacity is still low
6     {
7         size_t oldcapacity = d_capacity; // Old capacity needed to transfer pointers
8         d_capacity *= 2; // Double capacity when needed
9
10        string **tmp = rawPointers(d_capacity); // Create new pointer to raw pointers
11        for (size_t idx = 0; idx != oldcapacity; ++idx) // Transfer over old pointers
12            tmp[idx] = d_pPstrings[idx];
13
14        destroy(); // Destroy old pointer
15        d_pPstrings = tmp; // Assign old pointer to new location
16    }
17 };
```

*d\_size should be enough*

../53/strings/resize.cc

```
1 #include "strings.ih"
2
3 void Strings::resize(size_t newSize)
4 {
5     string newAddition = ""; // Empty string to use for filling
6     if (newSize > d_size) // If newSize is larger than current size
7         for (size_t index = 0; index != newSize - d_capacity; ++index) // Fill
8             add(newAddition);
9     else // (or Flow)
10        if (newSize < d_size) // If smaller
11            for (size_t index = d_size; index != newSize - 1; index--)
12                delete d_pPstrings[index]; // Delete those strings (as in d_tor)
13
14    d_size = newSize; // Set new size to indicated size
15    // If newSize == d_size, this is SF, but better than including
```

```
16          // it in every if-statement
17 }
```

## Exercise 55

../55/strings/strings.h

```
1  #ifndef INCLUDED_STRINGS_
2  #define INCLUDED_STRINGS_
3
4  #include <iosfwd>
5
6  class Strings
7  {
8      size_t d_capacity = 1;           // Rather start capacity at 0, see reserve
9      size_t d_size = 0;
10     std::string *d_pPstrings = 0;
11
12     public:
13         struct POD
14         {
15             size_t      size;
16             std::string *str;
17         };
18
19         Strings();
20         Strings(size_t argc, char const *argv[]);
21         Strings(char *environLike[]);
22         Strings(std::istream &in);
23         ~Strings();
24
25         void swap(Strings &other);
26
27         size_t size() const;
28         size_t capacity() const;
29         std::string* const *data() const;
30         POD release();
31         POD d_POD();
32
33         std::string const &at(size_t idx) const;
34         std::string &at(size_t idx);
35
36         void add(std::string const &next);
37
38     private:
39         void fill(char *ntbs[]);
40         void resize(size_t newSize);
41         std::string* rawMemory(size_t nNewPointers);
42         void reserve(size_t newCapacity);
43
44         std::string &safeAt(size_t idx) const;
45         std::string *enlarge();
46         void destroy();
47 };
48
49 inline size_t Strings::size() const
50 {
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 }
```

*Same issues as previous:*

- use member initializers,
- delegat
- safe hedge around d\_capacity being 1 without actual allocation.

*But, we bis in idem.*

*// Not const because of testing script*

*// New addition*

*// for const-objects*

*// for non-const objects*

*// add another element*

*// fill prepared d\_pPstrings*

*// New addition*

*// New addition*

*// New addition*

*// private backdoor*

*// potentially dangerous practice:*

*// inline accessors*

```
63
64 inline std::string &Strings::at(size_t idx)
65 {
66     return safeAt(idx);
67 }
68
69 #endif
```

../55/strings/add.cc

```
1 #include "strings.ih"
2
3 void Strings::add(string const &next)
4 {
5     if(d_size + 1 > d_capacity) // If there is no room for the new addition
6         reserve(d_size + 1);    // Create new room
7
8     new(d_pPstrings + d_size ) std::string{ next }; // Add the new string
9     ++d_size; // Increase size
10 }
```

../55/strings/c.argcargv.cc

```
1 #include "strings.ih"
2
3 Strings::Strings(size_t argc, char const *argv[])
4 {
5     d_pPstrings = rawMemory(1); // Create first memory
6     for (size_t index = 0; index != argc; ++index)
7         add(argv[index]);
8 };
```

../55/strings/c.environlike.cc

```
1 #include "strings.ih"
2
3 Strings::Strings(char *environLike[])
4 {
5     d_pPstrings = rawMemory(1); // Create first memory
6     for (size_t index = 0; environLike[index] != 0; ++index)
7         add(environLike[index]);
8 };
```

../55/strings/c.istream.cc

```
1 #include "strings.ih"
2
3 Strings::Strings(istream &in)
4 {
5     d_pPstrings = rawMemory(1); // Create first memory
6     string line;
7     while (getline(in, line))
8     {
9         add(line);
10        if (line.empty())
11            break;
12    }
13 };
```

../55/strings/destroy.cc

```
1 #include "strings.ih"
2
3 void Strings::destroy()
```

```
4 {
5     for (string *end = d_pPstrings + d_size; end-- != d_pPstrings; )
6         end->~string();
7     operator delete(d_pPstrings);
8 }
```

../55/strings/d\_tor.cc

```
1 #include "strings.ih"
2
3 Strings::~Strings()
4 {
5     destroy(); // Call original destroy(); one last time
6 }
```

../55/strings/rawMemory.cc

```
1 #include "strings.ih"
2
3 string* Strings::rawMemory(size_t nPointers)
4 {
5     string *tmp = static_cast<string *>(operator new(nPointers * sizeof(string)));
6     return tmp;
7 };
```

../55/strings/reserve.cc

```
1 #include "strings.ih"
2
3 void Strings::reserve(size_t newCapacity)
4 {
5     while (d_capacity < newCapacity) // Keep doubling while capacity is still low
6     {
7         d_capacity *= 2; // Double capacity when needed
8
9         string *tmp = rawMemory(d_capacity); // Create new pointer to raw memory
10        for (size_t idx = d_size; idx--; ) // Transfer over old strings
11            new(tmp + idx) string{ d_pPstrings[idx] };
12
13        destroy(); // Destroy old pointer
14        d_pPstrings = tmp; // Assign old pointer to new location
15    }
16 };
```

../55/strings/resize.cc

```
1 #include "strings.ih"
2
3 void Strings::resize(size_t newSize)
4 {
5     string newAddition = ""; // Empty string to use for filling
6     if (newSize > d_size) // If newSize is larger than current size
7         for (size_t index = 0; index != newSize - d_capacity; ++index) // Fill
8             add(newAddition);
9
10    if (newSize < d_size) // If smaller
11        for (string *end = d_pPstrings + d_size; end-- != d_pPstrings + newSize; )
12            end->~string();
13
14    d_size = newSize; // Set new size to indicated size
15                    // If newSize == d_size, this is SF, but better than including
16                    // it in every if-statement
17 }
```

Exercise 56

<i>Type</i>	<i>Time</i>
real	25m36,605s
user	25m25,435s
sys	0m11,168s

Table 1: Original

<i>Type</i>	<i>Time</i>
real	0m1,519s
user	0m1,381s
sys	0m0,137s

Table 2: Double pointers

<i>Type</i>	<i>Time</i>
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 than the others, 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.



Week 6

Exercise 57

../57/cpu/cpu.h

```
1  //+cpu
2  #ifndef INCLUDED_CPU_
3  #define INCLUDED_CPU_
4
5  #include "../tokenizer/tokenizer.h"    // the Tokenizer is a component of the
6                                         // CPU.
7  #include "../memory/memory.h"
8  // class Memory;                      // Memory only needs to be a declared
9                                         // term
10
11 class CPU
12 {
13     enum
14     {
15         NREGISTERS = 5,                // a.e. at indices 0..4, respectively
16         LAST_REGISTER = NREGISTERS - 1
17     };
18
19     struct Operand
20     {
21         OperandType type;
22         int value;
23     };
24
25     Memory &d_memory;
26     Tokenizer d_tokenizer;
27
28     int d_register[NREGISTERS];
29
30     private:
31         static void (CPU::*s_lhstype[])(int lhsvalue, int value) ;
32         //replaces the switch in store
33         static int (CPU::*s_deref[])(int input_number) ;      consistency
34         //replaces the switch in dereference
35
36     public:
37         CPU(Memory &memory);
38         void run();
39
40     private:
41         bool error();                // show 'syntax error', and prepare for the
42                                         // next input line
43
44         bool execute(Opcode opcode); // perform the action matching opcode
45
46 //+cpu
47                                         // return a value or a register's or
48                                         // memory location's value
49     int dereference(Operand const &value);
50
51     bool rvalue(Operand &lhs); // retrieve an rvalue operand
52     bool lvalue(Operand &lhs); // retrieve an lvalue operand
53
54                                         // determine 2 operands, lhs must be an lvalue
55     bool operands(Operand &lhs, Operand &rhs);
56
57     bool twoOperands(Operand &lhs, int &lhsValue, int &rhsValue);
58
59                                         // store a value in register or memory
60     void store(Operand const &lhs, int value);
```

```
61  //cpu2
62      void mov();           // assign a value
63      void add();           // add values
64      void sub();           // subtract values
65      void mul();           // multiply values
66      void div();           // divide values (remainder: last reg.)
67                          // div a b computes a /= b, last reg: %
68      void neg();           // negate a value
69      void dsp();           // display a value
70
71  //added for 57
72
73      void regStore(int lhsvalue, int input); //stores value in reg
74      int value(int input); //returns value
75      int get_register(int input); //register reserved returns reg value
76
77      void store(int address, int value);
78      int load(int address);
79      //declaring the functions found in memory here so they can be used
80      //in the arrays of pointers to functions.
81
82  };
83
84  inline void CPU::regStore(int lhsvalue, int input)
85  {
86      d_register[lhsvalue] = input;
87  }
88
89  inline int CPU::value(int input)
90  {
91      return input;
92  }
93
94  inline int CPU::get_register(int input)
95  {
96      return d_register[input];
97  }
98
99  #endif
100  //cpu2
```

../57/cpu/data.cc

```
1  #include "cpu.ih"
2
3  int (CPU::*CPU::s_deref[])(int input_number) =
4  {
5      &CPU::get_register,
6      &CPU::load,
7      &CPU::value,
8  };
9
10 void (CPU::*CPU::s_lhstype[])(int lhsvalue, int value) =
11 {
12     &CPU::regStore,
13     &CPU::store,
14 };
15 //ik heb geprobeerd ze local te maken maar dat is me helaas niet gelukt.
16 // dus hier maar in een data file
```

../57/cpu/dereference.cc

```
1  #include "cpu.ih"
2
3
```

```
4
5 int CPU::dereference(Operand const &value)
6 {
7     OperandType loc_type = value.type;
8
9     return (this->* s_deref[static_cast<size_t>(loc_type)])(value.value);
10    //using static cast to get the index from the Operandtype since its
11    //an Enum Class
12 }
```

../57/cpu/store.cc

```
1 #include "cpu.ih"
2
3
4 void CPU::store(Operand const &lhs, int value)
5 {
6     OperandType loc_type = lhs.type;
7
8
9
10    (this->*s_lhstype[static_cast<size_t>(loc_type)])(lhs.value, value);
11    //using static cast since were using Enum Classes
12 }
13 //instead of the switch we now have pointers to functions.
```

../57/enums/enums.h

```
1 #ifndef INCLUDED_ENUMS_
2 #define INCLUDED_ENUMS_
3
4 enum RAM
5 {
6     SIZE = 20
7 };
8
9    // all opcodes recognized by the CPU. They must also be known by the
10    // tokenizer, which is why they are 'escalated' to a separate header file.
11 enum class Opcode
12 {
13     ERR,
14     MOV,
15     ADD,
16     SUB,
17     MUL,
18     DIV,
19     NEG,
20     DSP,
21     STOP,
22 };
23
24    // the various operand types
25 enum class OperandType //altered order to correspond with arrays of pointers
26 {
27     REGISTER = 0,           //to functions
28     MEMORY = 1,             // register index
29     VALUE = 2,               // memory location (= index)
30     SYNTAX = 3,             // direct value
31     // syntax error while specifying an operand
32 };
33
34 #endif
```

