

C++ Course
Assignment 6

Contents

Exercise 43	1
strings.h	1
strings5.cc	2
Exercise 44	2
strings.h	2
strings.ih	4
add.cc	4
capacity.cc	4
count.cc	4
destroy.cc	4
fill.cc	5
rawpointers.cc	5
release.cc	5
reserve.cc	5
resize.cc	6
safeat.cc	7
strings1.cc	7
strings2.cc	7
strings3.cc	7
strings4.cc	8
strings5.cc	8
swap.cc	8
filter.h	8
filter.ih	9
display.cc	9
empty.cc	9
filter1.cc	10
firstnonempty.cc	10
lastnonempty.cc	10
Exercise 45	10
strings.h	10
strings.ih	12
add.cc	12
capacity.cc	12
count.cc	12
destroy.cc	12
fill.cc	13
rawpointers.cc	13
release.cc	13
reserve.cc	13
resize.cc	14
safeat.cc	15
strings1.cc	15
strings2.cc	15
strings3.cc	15
strings4.cc	16
strings5.cc	16
swap.cc	16
filter.h	16

filter.ih	17
display.cc	17
empty.cc	17
filter1.cc	18
firstnonempty.cc	18
lastnonempty.cc	18
Exercise 46	18
Exercise 47	18
cpu.h	18
dereference.cc	20
memoryreturn.cc	20
readoperand.cc	20
registerreturn.cc	20
valuereturn.cc	20
start.cc	20
errorwrap.cc	21
execute.cc	21
stp.cc	21
store.cc	21
storememory.cc	22
storeregister.cc	22
storevalue.cc	22
Exercise 48	22
Data Model	22
csv.h	23
main.cc	23
Exercise 49	23
csv.h	24
csv.ih	24
allocate.cc	25
clear.cc	25
csv1.cc	25
data.cc	25
doublesize.cc	26
lastline.cc	26
main.cc	26
read.cc	26
read1.cc	27
release.cc	27

Exercise 43

Problem statement. *Fix the memory leak in the 'Strings' class.*

Solution. Because our own implementation of 'Strings' was not perfect, we instead modified the official solution provided in the answers of set 5.

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();
20      Strings(int argc, char *argv[]);
21      Strings(char *environLike[]);
22      Strings(std::istream &in);
23
24      void swap(Strings &other);
25
26      size_t size() const;
27      std::string const *data() const;
28      POD release();
29
30      std::string const &at(size_t idx) const; // for const-objects
31      std::string &at(size_t idx); // for non-const objects
32
33      void add(std::string const &next); // add another element
34
35  private:
36      void fill(char *ntbs[]); // fill prepared d_str
37
38      std::string &safeAt(size_t idx) const; // private backdoor
39      std::string *enlarge();
40      void destroy();
41
42      static size_t count(char *environLike[]); // # elements in env.like
43
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
67  #endif

```

strings5.cc

```
1  #include "strings.ih"           // using namespace std;
2
3  Strings::~Strings()
4  {
5      delete[] d_str;           // 0 pointer allowed
6  }
```

Exercise 44

Problem statement. Use double pointers in *Strings* class.

Solution. Because our own implementation of 'Strings' was not perfect, we instead modified the official solution provided in the answers of set 5.

strings.h

```
1  #ifndef INCLUDED_STRINGS_
2  #define INCLUDED_STRINGS_
3
4  #include <iosfwd>
5
6  class Strings
7  {
8      size_t      d_size;      // number of stored strings
9      size_t      d_capacity = 1; // number of strings that can be stored
10     std::string  **d_arrayStr = nullptr; // pointer to pointers of string data
11
12     public:
13         struct POD
14         {
15             size_t      size;
16             std::string **str;
17         };
18
19         Strings();
20         ~Strings();
21         Strings(int argc, char *argv[]);
22         Strings(char *environLike[]);
23         Strings(std::istream &in);
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         size_t const capacity() const; // return allocated memory in nr of strings
37
38         void reserve(size_t const newCapacity); // reserves memory to new size
39
40         void resize(size_t const newCapacity); // resizes and initializes
41
42     private:
43         void fill(char *ntbs[]); // fill prepared d_str
44
45 }
```

```

46         std::string &safeAt(size_t idx) const;           // private backdoor
47
48         void destroy();
49
50         static size_t count(char *environLike[]);        // # elements in env.like
51
52
53
54         std::string** rawPointers(size_t nrPointers); // creates initialized array of
55                                                         // pointers to strings
56     };
57
58     inline size_t Strings::size() const                 // potentially dangerous practice:
59     {                                                    // inline accessors
60         return d_size;
61     }
62
63
64     inline std::string **const Strings::data() const
65     {
66         return d_arrayStr;
67     }
68
69     inline std::string const &Strings::at(size_t idx) const
70     {
71         return safeAt(idx);
72     }
73
74     inline std::string &Strings::at(size_t idx)
75     {
76         return safeAt(idx);
77     }
78
79
80 #endif

```

strings.ih

```

1  #include "strings.h"
2
3  #include <istream>
4  #include <string>
5
6  using namespace std;

```

add.cc

```

1  #include "strings.ih"
2
3  void Strings::add(string const &next)
4  {
5
6      string *strPointer = new string(next); // store new string address in pointer
7
8      if (d_size >= d_capacity)
9      {
10         d_capacity <= 1;                // multiply d_capacity by 2
11
12         reserve(d_capacity);            // reserve memory for strings
13     }
14
15     d_arrayStr[d_size] = strPointer;    // store new pointer in array

```

```
16     ++d_size;
17 }
18
```

capacity.cc

```
1  #include "strings.ih"
2
3
4  size_t const Strings::capacity() const
5  {
6      return d_capacity;
7  }
```

count.cc

```
1  #include "strings.ih"
2
3  // static
4  size_t Strings::count(char *environLike[])
5  {
6      size_t nElements = 0;
7
8      while (*environLike++ != 0)    // visit all defined elements
9          ++nElements;              // inc. counter if one's found
10
11     return nElements;
12 }
```

destroy.cc

```
1  #include "strings.ih"
2
3  void Strings::destroy()
4  {
5      delete d_arrayStr; // delete the array of pointers but not what they point to
6  }
```

fill.cc

```
1  #include "strings.ih"
2
3  void Strings::fill(char *ntbs[])
4  {
5      for (size_t index = 0; index != d_size; ++index)
6          *d_arrayStr[index] = string(ntbs[index]);
7  }
```

rawpointers.cc

```
1  #include "strings.ih"
2
3  string** Strings::rawPointers(size_t nrPointers)
4  {
5      string **newArray = new string*[nrPointers]; // create new pointer
6      for (size_t idx = 0; idx != nrPointers; ++idx) // initialize array with pointers to initialized strings
7          newArray[idx] = new string;
8  }
```

```
9     return newArray;
10 }
```

release.cc

```
1  #include "strings.ih"
2
3  Strings::POD Strings::release()
4  {
5      POD ret{d_size, d_arrayStr};          // initialize the POD for the caller
6
7      d_size = 0;
8      d_capacity = 1;
9      d_arrayStr = nullptr;
10
11     return ret;
12 }
```

reserve.cc

```
1  #include "strings.ih"
2
3
4  void Strings::reserve(size_t const newCapacity)
5  {
6      string **newArray = nullptr;
7
8      if (newCapacity >= d_capacity)
9      {
10         d_capacity = newCapacity;
11         newArray = new string*[d_capacity];          // not initialized
12
13         for (size_t idx = 0; idx != d_size; ++idx)    // copy old pointers
14             newArray[idx] = d_arrayStr[idx];
15     }
16
17     else if (newCapacity < d_size)                    // if new array is too small
18     {
19         newArray = new string*[d_capacity];          // not initialized
20
21         for (size_t idx = 0; idx != newCapacity; ++idx) // copy old pointers
22             newArray[idx] = d_arrayStr[idx];
23         for (size_t idx = newCapacity; idx != d_size; ++idx) // delete pointers and
24             delete d_arrayStr[idx];                  // strings outside new array
25     }
26
27     else // if newCapacity is between d_size and d_capacity
28     {
29         newArray = new string*[d_capacity];          // not initialized
30
31         for (size_t idx = 0; idx != d_size; ++idx)    // copy old pointers
32             newArray[idx] = d_arrayStr[idx];
33     }
34
35     destroy();          // delete old array of pointers, not the string data
36
37     d_arrayStr = newArray;
38 }
```

resize.cc

```
1  #include "strings.ih"
2
3
4  void Strings::resize(size_t const newCapacity)
5  {
6      string **newArray = nullptr;
7
8      if (newCapacity >= d_capacity)
9      {
10         d_capacity = newCapacity;
11         newArray = new string*[d_capacity];           // not initialized
12
13         for (size_t idx = 0; idx != d_size; ++idx)    // copy old pointers
14             newArray[idx] = d_arrayStr[idx];
15         for (size_t idx = d_size; idx != d_capacity; ++idx) // initialize the rest
16             newArray[idx] = new string;
17     }
18
19     else if (newCapacity < d_size)                    // if new array is too small
20     {
21         newArray = new string*[d_capacity];           // not initialized
22
23         for (size_t idx = 0; idx != newCapacity; ++idx) // copy old pointers
24             newArray[idx] = d_arrayStr[idx];
25         for (size_t idx = newCapacity; idx != d_size; ++idx) // delete pointers and
26             delete d_arrayStr[idx];                    // strings outside new array
27     }
28
29     else                                              // if newCapacity is between d_size and d_capacity
30     {
31         newArray = new string*[d_capacity];           // not initialized
32
33         for (size_t idx = 0; idx != d_size; ++idx)    // copy old pointers
34             newArray[idx] = d_arrayStr[idx];
35         for (size_t idx = d_size; idx != d_capacity; ++idx) // initialize the rest
36             newArray[idx] = new string;
37     }
38
39     destroy();                                       // delete old array of pointers, not the string data
40
41     d_arrayStr = newArray;
42 }
43
44
45 // the function seems a lot like the reserve function
46 // the difference is the initialisation of the string objects when enlarging
47 // this doesn't seem to make much sense but is according the exercise.
```

safeat.cc

```
1  #include "strings.ih"
2
3  namespace {
4      string empty;
5  }
6
7  std::string &Strings::safeAt(size_t idx) const
8  {
9      if (idx >= d_size)
10      {
11          empty.clear();
```



```
12         return empty;
13     }
14
15     return *(d_arrayStr[idx]);
16 }
```

strings1.cc

```
1  #include "strings.ih"
2
3  Strings::Strings()
4  :
5      d_size(0),
6      d_arrayStr(rawPointers(d_capacity))
7  {}
```

strings2.cc

```
1  #include "strings.ih"
2
3  Strings::Strings(int argc, char *argv[])
4  :
5      d_size(argc),
6      d_capacity(argc),
7      d_arrayStr(rawPointers(d_capacity))
8  {
9      fill(argv);           // fill the newly created array
10 }
```

strings3.cc

```
1  #include "strings.ih"
2
3  Strings::Strings(char *environLike[])
4  :
5      d_size(count(environLike)),
6      d_capacity(d_size),
7      d_arrayStr(rawPointers(d_capacity))
8
9  {
10     fill(environLike);     // fill the newly created array
11 }
```

strings4.cc

```
1  #include "strings.ih"
2
3  Strings::Strings(istream &in)
4  :
5      d_size(0),
6      d_arrayStr(new string *[d_capacity])
7
8  {
9      string line;
10
11     while (getline(in, line))
12         add(line);
13 }
```

strings5.cc

```
1  #include "strings.ih"                // using namespace std;
2
3  Strings::~Strings()
4  {
5      for (size_t idx = 0; idx != d_size; ++idx) // delete all strings by calling
6          delete d_arrayStr[idx];              // their destructor and free their memory
7
8      delete(d_arrayStr);                    // delete pointer to array of pointers
9  }
```

swap.cc

```
1  #include "strings.ih"
2
3  void Strings::swap(Strings &other)
4  {
5      string **tmp = d_arrayStr;
6      d_arrayStr = other.d_arrayStr;
7      other.d_arrayStr = tmp;
8
9      size_t size = d_size;
10     d_size = other.d_size;
11     other.d_size = size;
12 }
```

filter.h

```
1  #ifndef INCLUDED_FILTER_
2  #define INCLUDED_FILTER_
3
4  #include <iosfwd>
5  #include "../strings/strings.h"
6
7  class Filter
8  {
9      Strings d_lines;
10
11     public:
12         Filter(std::istream &in);
13
14         void display() const;
15
16     private:
17
18         static bool empty(std::string const &str);
19
20         static size_t firstNonEmpty(size_t size, std::string **const str);
21         static size_t beyondLastNonEmpty(size_t size, std::string **const str);
22
23 };
24
25 #endif
```

filter.ih

```
1  #include "filter.h"
2
3  #include <iostream>
```

```
4
5 using namespace std;
```

display.cc

```
1 #include "filter.ih"
2
3 void Filter::display() const
4 {
5     size_t size = d_lines.size();           // get number and contents
6     string **const str = d_lines.data();
7
8                                           // print fm first non empty through
9                                           // last non empty
10    for (size_t index = firstNonEmpty(size, str),
11         end = beyondLastNonEmpty(size, str);
12         index != end;
13         ++index
14    )
15        cout << *str[index] << '\n';
16
17 }
```

empty.cc

```
1 #include "filter.ih"
2
3 // static
4 bool Filter::empty(string const &str)
5 {
6     // find_first_not_of(" \t") returns index -> not empty,
7     // so:
8     // find_first_not_of(" \t") != npos      -> not empty
9     // so:
10    // find_first_not_of(" \t") == npos      -> empty
11
12    return str.find_first_not_of(" \t") == string::npos;
13 }
```

filter1.cc

```
1 #include "filter.ih"
2
3 Filter::Filter(istream &in)
4 :
5     d_lines(in)
6 {}
```

firstnonempty.cc

```
1 #include "filter.ih"
2
3 // static
4 size_t Filter::firstNonEmpty(size_t size, string **const str)
5 {
6     size_t idx = 0;
7
8                                           // skip initial empty lines
9     while (idx != size && empty(*str[idx]))
10         ++idx;
```

```
11     return idx;
12 }
```

lastnonempty.cc

```
1  #include "filter.ih"
2
3  // static
4  size_t Filter::beyondLastNonEmpty(size_t size, string **const str)
5  {
6      size_t idx = size;
7      // skip all empty lines at the end
8      while (idx-- && empty(*str[idx]))
9          ;
10
11     return idx + 1;          // idx at the last non-empty line,
12                             // but we must be beyond
```

Exercise 45

Problem statement. *Something something something*
Solution.

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      size_t d_capacity = 1;
10     std::string *d_str;
11
12     public:
13         struct POD
14         {
15             size_t      size;
16             std::string *str;
17         };
18
19         Strings();
20         ~Strings();
21         Strings(int argc, char *argv[]);
22         Strings(char *environLike[]);
23         Strings(std::istream &in);
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         size_t const capacity() const;              // return allocated memory in nr of strings
```

```

37
38     private:
39         void fill(char *ntbs[]);                // fill prepared d_str
40
41         std::string &safeAt(size_t idx) const;    // private backdoor
42
43         void destroy();                          // frees memory
44
45         static size_t count(char *environLike[]); // # elements in env.like
46
47         std::string* rawStrings(size_t nrPointers); // allocates memory for strings
48
49         void reserve(size_t const newCapacity);    // reserve memory
50
51         void resize(size_t const newCapacity);     // resize capacity
52
53 };
54
55 inline size_t Strings::size() const                // potentially dangerous practice:
56 {                                                  // inline accessors
57     return d_size;
58 }
59
60 inline std::string const *Strings::data() const
61 {
62     return d_str;
63 }
64
65 inline std::string const &Strings::at(size_t idx) const
66 {
67     return safeAt(idx);
68 }
69
70 inline std::string &Strings::at(size_t idx)
71 {
72     return safeAt(idx);
73 }
74
75
76 #endif

```

strings.ih

```

1  #include "strings.h"
2  #include <iostream>
3  #include <string>
4
5  using namespace std;

```

add.cc

```

1  #include "strings.ih"
2
3  void Strings::add(string const &next)
4  {
5      if (d_size == d_capacity)
6      {
7          d_capacity <= 1;                // multiply d_capacity by 2
8
9          reserve(d_capacity);           // reserve memory for strings
10     }
11

```

```
12     d_str[d_size] = string(next); // store new pointer in array
13
14     ++d_size;
15 }
```

capacity.cc

```
1 #include "strings.ih"
2
3
4 size_t const Strings::capacity() const
5 {
6     return d_capacity;
7 }
```

count.cc

```
1 #include "strings.ih"
2
3 // static
4 size_t Strings::count(char *environLike[])
5 {
6     size_t nElements = 0;
7
8     while (*environLike++ != 0) // visit all defined elements
9         ++nElements;           // inc. counter if one's found
10
11     return nElements;
12 }
```

destroy.cc

```
1 #include "strings.ih"
2
3 void Strings::destroy()
4 {
5     operator delete[](d_str); // frees the allocated memory but doesn't
6                               // delete the strings
7 }
```

fill.cc

```
1 #include "strings.ih"
2
3 void Strings::fill(char *ntbs[])
4 {
5     for (size_t index = 0; index != d_size; ++index)
6         d_str[index] = ntbs[index];
7 }
```

rawpointers.cc

```
1 #include "strings.ih"
2
3 string* Strings::rawStrings(size_t nrPointers)
4 {
5     string *newArray = static_cast<string *>(operator new[](nrPointers * sizeof(string)));
6 }
```

```
7     return newArray;
8 }
```

release.cc

```
1  #include "strings.ih"
2
3  Strings::POD Strings::release()
4  {
5      POD ret{ d_size, d_str };          // initialize the POD for the caller
6
7      d_size = 0;                        // reinitialize our data members
8      d_str = 0;
9
10     return ret;
11 }
```

reserve.cc

```
1  #include "strings.ih"
2
3
4  void Strings::reserve(size_t const newCapacity)
5  {
6      string *newArray = nullptr;
7
8      if (newCapacity >= d_capacity)
9      {
10         cerr << "reserve line 10 \n";
11         d_capacity = newCapacity;
12         cerr << "reserve line 12 \n";
13         newArray = rawStrings(d_capacity);          // not initialized
14         cerr << "reserve line 14 \n";
15         for (size_t idx = 0; idx != d_size; ++idx)  // copy old pointers
16             newArray[idx] = d_str[idx];
17         cerr << "reserve line 17 \n";
18     }
19
20     else if (newCapacity < d_size)                  // if new array is too small
21     {
22         newArray = rawStrings(d_capacity);          // not initialized
23
24         for (size_t idx = 0; idx != newCapacity; ++idx) // copy old pointers
25             newArray[idx] = d_str[idx];
26         for (size_t idx = newCapacity; idx != d_size; ++idx) // delete pointers and
27             d_str[idx].~string();                    // strings outside new array
28     }
29
30     else                                             // if newCapacity is between d_size and d_capacity
31     {
32         newArray = rawStrings(d_capacity);          // not initialized
33
34         for (size_t idx = 0; idx != d_size; ++idx)  // copy old pointers
35             newArray[idx] = d_str[idx];
36     }
37
38     destroy();                                     // delete old array of pointers, not the string data
39
40     d_str = newArray;
41 }
```

resize.cc

```
1  #include "strings.ih"
2
3
4  void Strings::resize(size_t const newCapacity)
5  {
6      string *newArray = nullptr;
7
8      if (newCapacity >= d_capacity)
9      {
10         d_capacity = newCapacity;
11         newArray = rawStrings(d_capacity);           // not initialized
12
13         for (size_t idx = 0; idx != d_size; ++idx)   // copy old pointers
14             newArray[idx] = d_str[idx];
15         for (size_t idx = d_size; idx != d_capacity; ++idx) // initialize the rest
16             newArray[idx] = string();
17     }
18
19     else if (newCapacity < d_size)                  // if new array is too small
20     {
21         newArray = rawStrings(d_capacity);           // not initialized
22
23         for (size_t idx = 0; idx != newCapacity; ++idx) // copy old pointers
24             newArray[idx] = d_str[idx];
25         for (size_t idx = newCapacity; idx != d_size; ++idx) // delete pointers and
26             d_str[idx].~string();                     // strings outside new array
27     }
28
29     else // if newCapacity is between d_size and d_capacity
30     {
31         newArray = rawStrings(d_capacity);           // not initialized
32
33         for (size_t idx = 0; idx != d_size; ++idx)   // copy old pointers
34             newArray[idx] = d_str[idx];
35         for (size_t idx = d_size; idx != d_capacity; ++idx) // initialize the rest
36             newArray[idx] = string();
37     }
38
39     destroy(); // delete old array of pointers, not the string data
40
41     d_str = newArray;
42 }
43
44
45 // the function seems a lot like the reserve function
46 // the difference is the initialisation of the string objects when enlarging
47 // this doesn't seem to make much sense but is according the exercise.
```

safeat.cc

```
1  #include "strings.ih"
2
3  namespace {
4      string empty;
5  }
6
7  std::string &Strings::safeAt(size_t idx) const
8  {
9      if (idx >= d_size)
10      {
11         empty.clear();
```



```
12         return empty;
13     }
14
15     return d_str[idx];
16 }
```

strings1.cc

```
1  #include "strings.ih"
2
3  Strings::Strings()
4  :
5      d_size(0),
6      d_str(rawStrings(d_capacity))
7  {}
```

strings2.cc

```
1  #include "strings.ih"
2
3  Strings::Strings(int argc, char *argv[])
4  :
5      d_size(argc),
6      d_capacity(d_size),
7      d_str(rawStrings(d_capacity))
8  {
9      fill(argv);
10 }
```

strings3.cc

```
1  #include "strings.ih"
2
3  Strings::Strings(char *environLike[])
4  :
5      d_size(count(environLike)),
6      d_capacity(d_size),
7      d_str(rawStrings(d_capacity))
8  {
9      fill(environLike);
10 }
```

strings4.cc

```
1  #include "strings.ih"
2
3  Strings::Strings(istream &in)
4  :
5      d_size(0),
6      d_str(rawStrings(d_capacity))
7  {
8      string line;
9      while (getline(in, line))
10         add(line);
11 }
```

strings5.cc

```
1  #include "strings.ih"                // using namespace std;
2
3  Strings::~Strings()
4  {
5      for (size_t idx = 0; idx != d_size; ++idx) // call all destructors of strings
6          d_str[idx].~string();
7
8      operator delete[](d_str);          // delete allocated memory
9  }
```

swap.cc

```
1  #include "strings.ih"
2
3  void Strings::swap(Strings &other)
4  {
5      string *tmp = d_str;
6      d_str = other.d_str;
7      other.d_str = tmp;
8
9      size_t size = d_size;
10     d_size = other.d_size;
11     other.d_size = size;
12 }
```

filter.h

```
1  #ifndef INCLUDED_FILTER_
2  #define INCLUDED_FILTER_
3
4  #include <iosfwd>
5  #include "../strings/strings.h"
6
7  class Filter
8  {
9      Strings d_lines;
10
11     public:
12         Filter(std::istream &in);
13
14         void display() const;
15
16     private:
17
18         static bool empty(std::string const &str);
19
20         static size_t firstNonEmpty(size_t size, std::string const *str);
21         static size_t beyondLastNonEmpty(size_t size, std::string const *str);
22
23 };
24
25 #endif
```

filter.ih

```
1  #include "filter.h"
2
3  #include <iostream>
```

```
4
5 using namespace std;
```

display.cc

```
1 #include "filter.ih"
2
3 void Filter::display() const
4 {
5     size_t size = d_lines.size();           // get number and contents
6     string const *str = d_lines.data();
7
8                                           // print fm first non empty through
9                                           // last non empty
10    for (size_t index = firstNonEmpty(size, str),
11         end = beyondLastNonEmpty(size, str);
12         index != end;
13         ++index
14    )
15        cout << str[index] << '\n';
16
17 }
```

empty.cc

```
1 #include "filter.ih"
2
3 // static
4 bool Filter::empty(string const &str)
5 {
6     // find_first_not_of(" \t") returns index -> not empty,
7     // so:
8     // find_first_not_of(" \t") != npos      -> not empty
9     // so:
10    // find_first_not_of(" \t") == npos      -> empty
11
12    return str.find_first_not_of(" \t") == string::npos;
13 }
```

filter1.cc

```
1 #include "filter.ih"
2
3 Filter::Filter(istream &in)
4 :
5     d_lines(in)
6 {}
```

firstnonempty.cc

```
1 #include "filter.ih"
2
3 // static
4 size_t Filter::firstNonEmpty(size_t size, string const *str)
5 {
6     size_t idx = 0;
7
8     // skip initial empty lines
9     while (idx != size && empty(str[idx]))
10         ++idx;
```

```
11     return idx;
12 }
```

lastnonempty.cc

```
1  #include "filter.ih"
2
3  // static
4  size_t Filter::beyondLastNonEmpty(size_t size, string const &str)
5  {
6      size_t idx = size;
7      // skip all empty lines at the end
8      while (idx-- && empty(str[idx]))
9          ;
10
11     return idx + 1;          // idx at the last non-empty line,
12                             // but we must be beyond
```

Exercise 46

Problem statement. [gi](#) Solution. [go](#)

Exercise 47

Problem statement. Replace the switches in the 'CPU' class using function pointers.

Solution. Because our own implementation of CPU was imperfect, we used the official solutions for Exercise 31. Our modified header is found below, followed by any new or modified helper functions. Everything not shown is assumed to be the unchanged.

cpu.h

```
1  #ifndef INCLUDED_CPU_
2  #define INCLUDED_CPU_
3
4  #include "../tokenizer/tokenizer.h"
5  #include "../memory/memory.h"
6  #include "../enums/enums.h"
7
8  class Memory; // Jaap: why this?
9
10 class CPU
11 {
12     enum
13     {
14         NREGISTERS = 5,          // a..e at indices 0..4, respectively
15         LAST_REGISTER = NREGISTERS - 1
16     };
17
18     struct Operand
19     {
20         OperandType type;
21         int value;
22     };
23
24     Memory &d_memory;
25     Tokenizer d_tokenizer;
26
27     int d_register[NREGISTERS];
28
29     public:
```

```

30     CPU(Memory &memory);
31     void start();
32
33 private:
34     bool error();
35
36
37
38
39     void stp();
40     static void (CPU::*execute[])();
41     void errorwrap();
42
43     int dereference(Operand const &value);
44     static int (CPU::*readOperand[])(Operand const &value);
45     int valueReturn(Operand const &value);
46     int memoryReturn(Operand const &value);
47     int registerReturn(Operand const &value);
48
49     bool rvalue(Operand &lhs);
50     bool lvalue(Operand &lhs);
51
52
53
54
55     bool operands(Operand &lhs, Operand &rhs);
56
57     bool twoOperands(Operand &lhs, int &lhsValue, int &rhsValue);
58
59
60
61
62
63
64
65
66
67
68
69
70
71 };
72
73 #endif

```

// show 'syntax error', and prepare for the next input line
// return a value or a register's or memory location's value
// helpers for start
// retrieve an rvalue operand
// retrieve an lvalue operand
// determine 2 operands, lhs must be an lvalue
// store a value in register or memory
// assign a value
// add values
// subtract values
// multiply values
// divide values (remainder: last reg.)
// div a b computes a /= b, last reg: %
// negate a value
// display a value

dereference.cc

```

1  #include "cpu.ih"
2
3  int CPU::dereference(Operand const &value)
4  {
5      return (this->*readOperand[value.type])(value);
6  }

```

memoryreturn.cc

```

1  #include "cpu.ih"
2
3  int CPU::memoryReturn(Operand const &value)
4  {
5      return d_memory.load(value.value);
6  }

```

readoperand.cc

```
1  #include "cpu.ih"
2
3  int (CPU::*CPU::readOperand[])(Operand const &value) =    // order as in enums.h
4  {
5      nullptr,          // padding for syntax, will never be called
6      &CPU::valueReturn,
7      &CPU::registerReturn,
8      &CPU::memoryReturn
9  };
```

registerreturn.cc

```
1  #include "cpu.ih"
2
3  int CPU::registerReturn(Operand const &value)
4  {
5      return d_register[value.value];
6  }
```

valuereturn.cc

```
1  #include "cpu.ih"
2
3  int CPU::valueReturn(Operand const &value)
4  {
5      return value.value;
6  }
```

start.cc

```
1  #include "cpu.ih"
2
3  void CPU::start()
4  {
5      while (true)
6      {
7          (this->*execute[d_tokenizer.opcode()])();
8          d_tokenizer.reset();
9      }
10 }
```

errorwrap.cc

```
1  #include "cpu.ih"
2
3  void CPU::errorwrap()
4  {
5      error();
6  }
```

execute.cc

```
1  #include "cpu.ih"
2
3  void (CPU::*CPU::execute[])() =    // order as in enums.h
4  {
```

```

5      &CPU::errorwrap,
6      &CPU::mov,
7      &CPU::add,
8      &CPU::sub,
9      &CPU::mul,
10     &CPU::div,
11     &CPU::neg,
12     &CPU::dsp,
13     &CPU::stp
14 };

```

stp.cc

```

1  #include "cpu.ih"
2
3  void CPU::stp() // seperate file, add to header
4  {
5  }

```

store.cc

```

1  #include "cpu.ih"
2
3  void CPU::store(Operand const &lhs, int value)
4  {
5      (this->storeValue[lhs.type])(lhs.value, value);
6  }
7  void CPU::storeRegister(int place, int value)
8  {
9      d_register[place] = value;
10 }
11
12 void CPU::storeMemory(int place, int value)
13 {
14     d_memory.store(place, value);
15 }
16
17 void (CPU::*CPU::storeValue[])(int place, int value)
18 {
19     nullptr,
20     nullptr, // these should never be called
21     &CPU::storeRegister,
22     &CPU::storeMemory
23 };

```

storememory.cc

```

1  #include "cpu.ih"
2
3  void CPU::storeMemory(int place, int value)
4  {
5      d_memory.store(place, value);
6  }

```

storeregister.cc

```

1  #include "cpu.ih"
2
3  void CPU::storeRegister(int place, int value)
4  {

```

```

5     d_register[place] = value;
6 }

```

storevalue.cc

```

1  #include "cpu.ih"
2
3  void (CPU::*CPU::storeValue[])(int place, int value)
4  {
5      nullptr,
6      nullptr,                                     // these should never be called
7      &CPU::storeRegister,
8      &CPU::storeMemory
9  };

```

Exercise 48

Problem statement. Design the CSV class header.

Solution.

Data Model

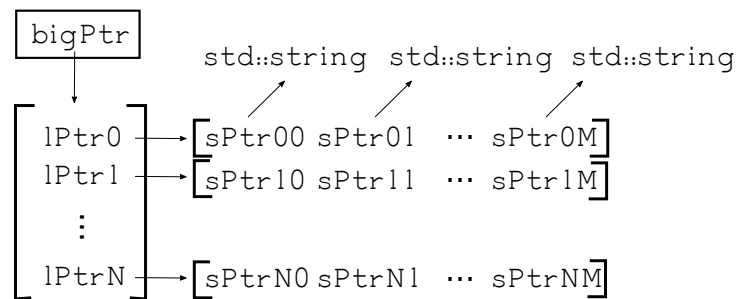


Figure 1: `bigPtr` is a triple pointer. It points to an array of 'line pointers', each of these point to an array of `std::string` pointers representing the comma-separated values. For example: using the notation above we have `bigPtr[1][1] = sPtr11` for the second value on the second line.

csv.h

```

1  #ifndef CSV_HEADER_H
2  #define CSV_HEADER_H
3
4  #include <string>                                     // std::string
5  #include <istream>                                    // std::istream
6
7  class CSV
8  {
9      size_t d_size = 1;                               // number of lines allocated
10     size_t d_nLines = 0;                             // number of lines read
11     size_t d_nFields = 1;                             // number of values per line
12     char d_fieldSep;                                  // field separator (default comma)
13
14     std::string ***bigPtr;                            // pointer to array of line pointers (see also big comment below)
15
16     public:
17         ~CSV();
18         CSV(size_t field, char fieldSep = ',');
19
20         std::string const *const *const *data() const; // return pointer to data
21         std::string const &lastline() const;           // ref last extraction

```



```

22
23     size_t nFields()                const;    // values per line, set in first read
24     size_t size()                  const;    // number of currently stored lines
25
26     size_t read(std::istream &in, size_t nLines = 0); // read lines using read1, return number read
27
28     std::string ***release();        // return pointer to data, move responsibility for data
29                                     // to called. Resets bigPtr but does not erase stored lines.
30     void clear(size_t nFields = 0);  // erase everything
31 private:
32     bool read1(std::istream &in);    // read 1 line, parse for CSV's, set nFields
33 };
34
35 #endif // CSV_HEADER_H
36
37 // Line pointers point to array of pointers
38 // to std::string. i.e. :
39 // bigPtr -> [Lptr0 Lptr1 ... LptrN]
40 // where Lptri -> [strPtri1 strPtri2 ... strPtriM] for i = 1,...,N
41 // where strPtrik -> std::string for k = 1,...,M/
42 // see also the figure in the report.

```

main.cc

```

1  #include "csv.h"
2
3  int main()
4  {
5      CSV file1(5, ',');
6  }

```

Exercise 49

Problem statement. Implement the CSV class member functions.

Solution.

csv.h

```

1  #ifndef CSV_HEADER_H
2  #define CSV_HEADER_H
3
4  #include <string>                // std::string
5  #include <istream>              // std::istream
6  #include "../csvextractor/csvextractor.h"
7
8  class CSV
9  {
10     size_t d_size;                // number of lines allocated
11     size_t d_nLines;             // number of lines read
12     size_t d_nFields;            // number of values per line
13     char d_fieldSep;             // field separator (default comma)
14     std::string d_lastLine;
15
16     std::string ***bigPtr;        // pointer to array of line pointers (see also big comment below)
17
18 public:
19     CSV(size_t field, char fieldSep = ',');
20
21     std::string const *const *const *data() const;    // return pointer to data
22     std::string const &lastline() const;             // ref last extraction
23

```

```

24     size_t nFields()                const;    // values per line, set in first read
25     size_t size()                  const;    // number of currently stored lines
26
27     size_t read(std::istream &in, size_t nLines = 0); // read lines using read1, return number read
28
29     std::string ***release();        // return pointer to data, move responsibility for data
30                                     // to called. Resets bigPtr but does not erase stored lines.
31     void clear(size_t nFields = 0);  // erase everything
32 private:
33     bool read1(std::istream &in);    // read 1 line, parse for CSV's, set nFields
34     void allocate();
35     void doubleSize();
36 };
37
38 inline size_t CSV::nFields() const
39 {
40     return d_nFields;
41 }
42
43 inline size_t CSV::size() const
44 {
45     return d_size;
46 }
47 #endif // CSV_HEADER_H
48
49 // Line pointers point to array of pointers
50 // to std::string. i.e. :
51 // bigPtr -> [Lptr0 Lptr1 ... LptrN]
52 // where Lptri -> [strPtri1 strPtri2 ... strPtriM] for i = 1,...,N
53 // where strPtrik -> std::string for k = 1,...,M/
54 // see also the figure in the report.
55
56 // - memcpy copies raw bytes

```

csv.ih

```

1  #include "csv.h"
2  #include <string>
3  #include <iostream>
4  using std::string;

```

allocate.cc

```

1  #include "csv.ih"
2
3  void CSV::allocate()
4  {
5      bigPtr = new std::string **[1]; // line array
6  }
7  // allocate me some memory

```

clear.cc

```

1  #include "csv.ih"
2
3  void CSV::clear(size_t nFields) // nFields defaults to 0
4  {
5      // de-allocate all every line array
6      for (string **line = bigPtr[0]; line != bigPtr[d_nLines - 1]; ++line)
7          delete[] line;
8      // de-allocate array of lines

```

```

9     delete[] bigPtr;
10
11     d_size = 1;
12     d_nLines = 0;
13     d_nFields = nFields;
14
15     allocate();
16 }

```

csv1.cc

```

1  #include "csv.ih"
2
3  CSV::CSV(size_t field, char fieldSep)
4      :
5      {
6          d_size(1),          // to allocate: 1 line
7          d_nLines(0),        // 0 lines read so far
8          d_nFields(field),   // to allocate: 'field' fields
9          d_fieldSep(fieldSep), // set field separator, default ','
10         d_lastLine()
11     {
12         bigPtr = new std::string **[1];    // allocate line array
13     }

```

data.cc

```

1  #include "csv.ih"
2
3  std::string const *const *const CSV::data() const
4  {
5      return bigPtr;
6  }

```

doublesize.cc

```

1  #include "csv.h"
2
3  void CSV::doubleSize()
4  {
5      d_size = d_size << 1;
6      allocate();
7  }

```

lastline.cc

```

1  #include "csv.h"
2
3  std::string const &CSV::lastline() const
4  {
5      return d_lastLine;
6  }

```

main.cc

```

1  #include "csv.h"
2
3  int main()
4  {

```

```
5     CSV file1(5, ',');
6 }
```

read.cc

```
1  #include "csv.h"
2
3  size_t CSV::read(std::istream &in, size_t nLines) // nLines defaults to 0
4  {
5      size_t lines = 0;
6
7      if (nLines == 0)
8          while (in.good())                // read all lines
9              {
10                 read1(in);
11                 ++lines;
12             }
13      else
14          while (lines != nLines && in.good()) // read 'nLines' lines
15              {
16                 read1(in);
17                 ++lines;
18             }
19      return lines;
20  }
21
22  // By default, all lines of in are read and are processed by the read1 member.
23  // By specifying a non-zero value for the nLines parameter the specified number of
24  // lines is read from in. Reading stops once in's status is not good. When nLines
25  // is specified as zero, then reading continues until all CSV lines have been processed.
26  // The number of successfully processed lines is returned.
```

read1.cc

```
1  #include "csv.h"
2
3  bool CSV::read1(std::istream &in)
4  {
5      CSVExtractor csvFile(in);                // CSVExtractor takes 1 line from stream 'in'
6
7      if (d_nFields == 0)                      // field count
8          d_nFields = csvFile.nFields();
9
10     if (d_size - d_nLines == 0)              // increase capacity
11         doubleSize();
12
13     return csvFile.parse(bigPtr);
14  }
15
16  // One line is read from in and is parsed for its CSVs. If parsing fails, false is returned.
17  // After successfully calling read1 for the first time all subsequent lines read by read1 must
18  // have the same number of comma separated values as encountered when calling read1 for the first time..
```

release.cc

```
1
```
