Jaap van der Leest
Timon van der Berg
October 25, 2017

C++ Course
Assignment 6

Contents

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

```
#ifndef INCLUDED_STRINGS_
#define INCLUDED_STRINGS_

#include <iosfwd>

class Strings
{
    size_t d_size;
```

```cpp
    std::string *d_str;

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

        Strings();
        ~Strings();
        Strings(int argc, char *argv[]);
        Strings(char *environLike[]);
        Strings(std::istream &in);

        void swap(Strings &other);

        size_t size() const;
        std::string const *data() const;
        POD release();

        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_str

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

        static size_t count(char *environLike[]); // # elements in env.like

};

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

inline std::string const *Strings::data() const
{
    return d_str;
}

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
```

strings5.cc

```
#include "strings.ih"           // using namespace std;

Strings::~Strings()
{
    delete[] d_str;             // 0 pointer allowed
}
```

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

```
#ifndef INCLUDED_STRINGS_
#define INCLUDED_STRINGS_

#include <iosfwd>

class Strings
{
    size_t              d_size;    // number of stored strings
    size_t      d_capacity = 1;    // number of strings that can be stored
    std::string   **d_arrayStr = nullptr;    // pointer to pointers of string data

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

        Strings();
        ~Strings();
        Strings(int argc, char *argv[]);
        Strings(char *environLike[]);
        Strings(std::istream &in);

        void swap(Strings &other);

        size_t size() const;
        std::string **const data() const;
        POD release();

        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

        size_t const capacity() const;                // return allocated memory in nr of strings

        void reserve(size_t const newCapacity);       // reserves memory to new size

        void resize(size_t const newCapacity);        // resizes and initializes


    private:
        void fill(char *ntbs[]);                      // fill prepared d_str

        std::string &safeAt(size_t idx) const;        // private backdoor

        void destroy();

        static size_t count(char *environLike[]);     // # elements in env.like



        std::string** rawPointers(size_t nrPointers); // creates initialized array of
                                                      // pointers to strings
};

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


inline std::string **const Strings::data() const
{
    return d_arrayStr;
}

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
```

strings.ih

```
#include "strings.h"

#include <istream>
#include <string>

using namespace std;
```

add.cc

```
#include "strings.ih"

void Strings::add(string const &next)

{
    string *strPointer = new string(next);  // store new string address in pointer

    if (d_size >= d_capacity)
    {
        d_capacity <<= 1;                   // multiply d_capacity by 2

        reserve(d_capacity);                // reserve memory for strings
    }

    d_arrayStr[d_size] = strPointer;        // store new pointer in array
    ++d_size;
}

```

capacity.cc

```
#include "strings.ih"


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

count.cc

```
#include "strings.ih"

// static
size_t Strings::count(char *environLike[])
{
    size_t nElements = 0;

    while (*environLike++ != 0)      // visit all defined elements
        ++nElements;                 // inc. counter if one's found

    return nElements;
}
```

destroy.cc

```
#include "strings.ih"

void Strings::destroy()
{
    delete d_arrayStr;  // delete the array of pointers but not what they point to
}
```

fill.cc

```
#include "strings.ih"

void Strings::fill(char *ntbs[])
{
    for (size_t index = 0; index != d_size; ++index)
        *d_arrayStr[index] = string(ntbs[index]);
}
```

rawpointers.cc

```
#include "strings.ih"

string** Strings::rawPointers(size_t nrPointers)
{
    string **newArray = new string*[nrPointers];    // create new pointer
    for (size_t idx = 0; idx != nrPointers; ++idx)  // initialize array with pointers to initialized strings
        newArray[idx] = new string;

    return newArray;
}
```

release.cc

```
#include "strings.ih"

Strings::POD Strings::release()
{
    POD ret{d_size, d_arrayStr};       // initialize the POD for the caller

    d_size = 0;
    d_capacity = 1;
    d_arrayStr = nullptr;

    return ret;
}
```

reserve.cc

```
#include "strings.ih"


void Strings::reserve(size_t const newCapacity)
{
    string **newArray = nullptr;

    if (newCapacity >= d_capacity)
    {
        d_capacity = newCapacity;
        newArray = new string*[d_capacity];                  // not initialized

        for (size_t idx = 0; idx != d_size; ++idx)           // copy old pointers
            newArray[idx] = d_arrayStr[idx];
    }

    else if (newCapacity < d_size)                           // if new array is too small
    {
        newArray = new string*[d_capacity];                  // not initialized

        for (size_t idx = 0; idx != newCapacity; ++idx)      // copy old pointers
            newArray[idx] = d_arrayStr[idx];
        for (size_t idx = newCapacity; idx != d_size; ++idx) // delete pointers and
            delete d_arrayStr[idx];                          // strings outside new array
    }

    else                    // if newCapacity is between d_size and d_capacity
    {
        newArray = new string*[d_capacity];                  // not initialized

        for (size_t idx = 0; idx != d_size; ++idx)           // copy old pointers
            newArray[idx] = d_arrayStr[idx];
    }

    destroy();                       // delete old array of pointers, not the string data

    d_arrayStr = newArray;
}
```

resize.cc

```
#include "strings.ih"


void Strings::resize(size_t const newCapacity)
{
    string **newArray = nullptr;

    if (newCapacity >= d_capacity)
    {
        d_capacity = newCapacity;
        newArray = new string*[d_capacity];                  // not initialized

        for (size_t idx = 0; idx != d_size; ++idx)           // copy old pointers
            newArray[idx] = d_arrayStr[idx];
        for (size_t idx = d_size; idx != d_capacity; ++idx)  // initialize the rest
            newArray[idx] = new string;

    }

    else if (newCapacity < d_size)                           // if new array is too small
    {
        newArray = new string*[d_capacity];                  // not initialized

        for (size_t idx = 0; idx != newCapacity; ++idx)      // copy old pointers
            newArray[idx] = d_arrayStr[idx];
        for (size_t idx = newCapacity; idx != d_size; ++idx) // delete pointers and
            delete d_arrayStr[idx];                          // strings outside new array
    }

    else                    // if newCapacity is between d_size and d_capacity
    {
        newArray = new string*[d_capacity];                  // not initialized

        for (size_t idx = 0; idx != d_size; ++idx)           // copy old pointers
            newArray[idx] = d_arrayStr[idx];
        for (size_t idx = d_size; idx != d_capacity; ++idx)  // initialize the rest
            newArray[idx] = new string;
    }

    destroy();                       // delete old array of pointers, not the string data

    d_arrayStr = newArray;
}


// the function seems a lot like the reserve function
// the difference is the initialisation of the string objects when enlarging
// this doesn't seem to make much sense but is according the exercise.
```

safeat.cc

```
#include "strings.ih"

namespace {
    string empty;
}

std::string &Strings::safeAt(size_t idx) const
{
    if (idx >= d_size)
    {
        empty.clear();
        return empty;
    }

    return *(d_arrayStr[idx]);
}
```

strings1.cc

```
#include "strings.ih"

Strings::Strings()
:
    d_size(0),
    d_arrayStr(rawPointers(d_capacity))
{}
```

strings2.cc

```
#include "strings.ih"

Strings::Strings(int argc, char *argv[])
:
    d_size(argc),
    d_capacity(argc),
    d_arrayStr(rawPointers(d_capacity))
{
    fill(argv);          // fill the newly created array
}
```

strings3.cc

```
#include "strings.ih"

Strings::Strings(char *environLike[])
:
    d_size(count(environLike)),
    d_capacity(d_size),
    d_arrayStr(rawPointers(d_capacity))

{
    fill(environLike);       // fill the newly created array
}
```

strings4.cc

```
#include "strings.ih"

Strings::Strings(istream &in)
:
    d_size(0),
    d_arrayStr(new string *[d_capacity])

{
    string line;

    while (getline(in, line))
        add(line);
}
```

strings5.cc

```
#include "strings.ih"                          // using namespace std;

Strings::~Strings()
{
    for (size_t idx = 0; idx != d_size; ++idx) // delete all strings by calling
        delete d_arrayStr[idx];                // their destructor and free their memory

    delete(d_arrayStr);                        // delete pointer to array of pointers
}
```

swap.cc

```
#include "strings.ih"

void Strings::swap(Strings &other)
{
    string **tmp = d_arrayStr;
    d_arrayStr = other.d_arrayStr;
    other.d_arrayStr = tmp;

    size_t size = d_size;
    d_size = other.d_size;
    other.d_size = size;
}
```

filter.h

```
#ifndef INCLUDED_FILTER_
#define INCLUDED_FILTER_

#include <iosfwd>
#include "../strings/strings.h"

class Filter
{
    Strings d_lines;

    public:
        Filter(std::istream &in);

        void display() const;

    private:

        static bool empty(std::string const &str);

        static size_t firstNonEmpty(size_t size, std::string **const str);
        static size_t beyondLastNonEmpty(size_t size, std::string **const str);


};

#endif
```

filter.ih

```
#include "filter.h"

#include <iostream>
```

```

using namespace std;
```

display.cc

```
#include "filter.ih"

void Filter::display() const
{
    size_t size = d_lines.size();          // get number and contents
    string **const str = d_lines.data();

                                           // print fm first non empty through
                                           // last non empty
    for (size_t index = firstNonEmpty(size, str),
            end = beyondLastNonEmpty(size, str);
            index != end;
                ++index
    )
        cout << *str[index] << '\n';

}
```

empty.cc

```
#include "filter.ih"

// static
bool Filter::empty(string const &str)
{
    // find_first_not_of(" \t") returns index -> not empty,
    //  so:
    // find_first_not_of(" \t") != npos       -> not empty
    //  so:
    // find_first_not_of(" \t") == npos       -> empty

    return str.find_first_not_of(" \t") == string::npos;
}
```

filter1.cc

```
#include "filter.ih"

Filter::Filter(istream &in)
:
    d_lines(in)
{}
```

firstnonempty.cc

```
#include "filter.ih"

// static
size_t Filter::firstNonEmpty(size_t size, string **const str)
{
    size_t idx = 0;
                                        // skip initial empty lines
    while (idx != size && empty(*str[idx]))
        ++idx;

    return idx;
}
```

lastnonempty.cc

```
#include "filter.ih"

// static
size_t Filter::beyondLastNonEmpty(size_t size, string **const str)
{
    size_t idx = size;
                                         // skip all empty lines at the end
    while (idx-- && empty(*str[idx]))
        ;

    return idx + 1;                      // idx at the last non-empty line,
}                                        // but we must be beyond
```

Exercise 45

Problem statement. *Something something something*
Solution.

strings.h

```
#ifndef INCLUDED_STRINGS_
#define INCLUDED_STRINGS_

#include <iosfwd>

class Strings
{
    size_t d_size;
    size_t d_capacity = 1;
    std::string *d_str;

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

        Strings();
        ~Strings();
        Strings(int argc, char *argv[]);
        Strings(char *environLike[]);
        Strings(std::istream &in);

        void swap(Strings &other);

        size_t size() const;
        std::string const *data() const;
        POD release();

        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

        size_t const capacity() const;               // return allocated memory in nr of strings

    private:
        void fill(char *ntbs[]);                    // fill prepared d_str

        std::string &safeAt(size_t idx) const;      // private backdoor

        void destroy();                             // frees memory

        static size_t count(char *environLike[]);   // # elements in env.like

        std::string* rawStrings(size_t nrPointers); // allocates memory for strings

        void reserve(size_t const newCapacity);     // reserve memory

        void resize(size_t const newCapacity);      // resize capacity

};

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

inline std::string const *Strings::data() const
{
    return d_str;
}

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
```

strings.ih

```
#include "strings.h"
#include <iostream>
#include <string>

using namespace std;
```

add.cc

```
#include "strings.ih"

void Strings::add(string const &next)
{
    if (d_size == d_capacity)
    {
        d_capacity <<= 1;          // multiply d_capacity by 2

        reserve(d_capacity);       // reserve memory for strings
    }

    d_str[d_size] = string(next);  // store new pointer in array

    ++d_size;
}
```

capacity.cc

```
#include "strings.ih"


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

count.cc

```
#include "strings.ih"

// static
size_t Strings::count(char *environLike[])
{
    size_t nElements = 0;

    while (*environLike++ != 0)      // visit all defined elements
        ++nElements;                 // inc. counter if one's found

    return nElements;
}
```

destroy.cc

```
#include "strings.ih"

void Strings::destroy()
{
    operator delete[](d_str);    // frees the allocated memory but doesn't
                                 // delete the strings
}
```

fill.cc

```
#include "strings.ih"

void Strings::fill(char *ntbs[])
{
    for (size_t index = 0; index != d_size; ++index)
        d_str[index] = ntbs[index];
}
```

rawpointers.cc

```
#include "strings.ih"

string* Strings::rawStrings(size_t nrPointers)
{
    string *newArray = static_cast<string *>(operator new[](nrPointers * sizeof(string)));

    return newArray;
}
```

release.cc

```
#include "strings.ih"

Strings::POD Strings::release()
{
    POD ret{ d_size, d_str };       // initialize the POD for the caller

    d_size = 0;                     // reinitialize our data members
    d_str = 0;

    return ret;
}
```

reserve.cc

```
#include "strings.ih"


void Strings::reserve(size_t const newCapacity)
{
    string *newArray = nullptr;

    if (newCapacity >= d_capacity)
    {
        cerr << "reserve line 10 \n";
        d_capacity = newCapacity;
        cerr << "reserve line 12 \n";
        newArray = rawStrings(d_capacity);                   // not initialized
        cerr << "reserve line 14 \n";
        for (size_t idx = 0; idx != d_size; ++idx)           // copy old pointers
            newArray[idx] = d_str[idx];
        cerr << "reserve line 17 \n";
    }

    else if (newCapacity < d_size)                           // if new array is too small
    {
        newArray = rawStrings(d_capacity);                   // not initialized

        for (size_t idx = 0; idx != newCapacity; ++idx)      // copy old pointers
            newArray[idx] = d_str[idx];
        for (size_t idx = newCapacity; idx != d_size; ++idx) // delete pointers and
            d_str[idx].~string();                            // strings outside new array
    }

    else                    // if newCapacity is between d_size and d_capacity
    {
        newArray = rawStrings(d_capacity);                   // not initialized

        for (size_t idx = 0; idx != d_size; ++idx)           // copy old pointers
            newArray[idx] = d_str[idx];
    }

    destroy();                           // delete old array of pointers, not the string data

    d_str = newArray;
}
```

resize.cc

```cpp
#include "strings.ih"


void Strings::resize(size_t const newCapacity)
{
    string *newArray = nullptr;

    if (newCapacity >= d_capacity)
    {
        d_capacity = newCapacity;
        newArray = rawStrings(d_capacity);                   // not initialized

        for (size_t idx = 0; idx != d_size; ++idx)           // copy old pointers
            newArray[idx] = d_str[idx];
        for (size_t idx = d_size; idx != d_capacity; ++idx)  // initialize the rest
            newArray[idx] = string();

    }

    else if (newCapacity < d_size)                           // if new array is too small
    {
        newArray = rawStrings(d_capacity);                   // not initialized

        for (size_t idx = 0; idx != newCapacity; ++idx)      // copy old pointers
            newArray[idx] = d_str[idx];
        for (size_t idx = newCapacity; idx != d_size; ++idx) // delete pointers and
            d_str[idx].~string();                            // strings outside new array
    }

    else                    // if newCapacity is between d_size and d_capacity
    {
        newArray = rawStrings(d_capacity);                   // not initialized

        for (size_t idx = 0; idx != d_size; ++idx)           // copy old pointers
            newArray[idx] = d_str[idx];
        for (size_t idx = d_size; idx != d_capacity; ++idx)  // initialize the rest
            newArray[idx] = string();
    }

    destroy();                       // delete old array of pointers, not the string data

    d_str = newArray;
}


// the function seems a lot like the reserve function
// the difference is the initialisation of the string objects when enlarging
// this doesn't seem to make much sense but is according the exercise.
```

safeat.cc

```cpp
#include "strings.ih"

namespace {
    string empty;
}

std::string &Strings::safeAt(size_t idx) const
{
    if (idx >= d_size)
    {
        empty.clear();
        return empty;
    }

    return d_str[idx];
}
```

strings1.cc

```
#include "strings.ih"

Strings::Strings()
:
    d_size(0),
    d_str(rawStrings(d_capacity))
{}
```

strings2.cc

```
#include "strings.ih"

Strings::Strings(int argc, char *argv[])
:
    d_size(argc),
    d_capacity(d_size),
    d_str(rawStrings(d_capacity))
{
    fill(argv);
}
```

strings3.cc

```
#include "strings.ih"

Strings::Strings(char *environLike[])
:
    d_size(count(environLike)),
    d_capacity(d_size),
    d_str(rawStrings(d_capacity))
{
    fill(environLike);
}
```

strings4.cc

```
#include "strings.ih"

Strings::Strings(istream &in)
:
    d_size(0),
    d_str(rawStrings(d_capacity))
{
    string line;
    while (getline(in, line))
        add(line);
}
```

strings5.cc

```
#include "strings.ih"                           // using namespace std;

Strings::~Strings()
{
    for (size_t idx = 0; idx != d_size; ++idx) // call all destructors of strings
        d_str[idx].~string();

    operator delete[](d_str);                  // delete allocated memory
}
```

swap.cc

```
#include "strings.ih"

void Strings::swap(Strings &other)
{
    string *tmp = d_str;
    d_str = other.d_str;
    other.d_str = tmp;

    size_t size = d_size;
    d_size = other.d_size;
    other.d_size = size;
}
```

filter.h

```
#ifndef INCLUDED_FILTER_
#define INCLUDED_FILTER_

#include <iosfwd>
#include "../strings/strings.h"

class Filter
{
    Strings d_lines;

    public:
        Filter(std::istream &in);

        void display() const;

    private:

        static bool empty(std::string const &str);

        static size_t firstNonEmpty(size_t size, std::string const *str);
        static size_t beyondLastNonEmpty(size_t size, std::string const *str);


};

#endif
```

filter.ih

```
#include "filter.h"

#include <iostream>
```

```

using namespace std;
```

display.cc

```
#include "filter.ih"

void Filter::display() const
{
    size_t size = d_lines.size();          // get number and contents
    string const *str = d_lines.data();

                                        // print fm first non empty through
                                        // last non empty
    for (size_t index = firstNonEmpty(size, str),
                end = beyondLastNonEmpty(size, str);
            index != end;
                ++index
    )
        cout << str[index] << '\n';

}
```

empty.cc

```
#include "filter.ih"

// static
bool Filter::empty(string const &str)
{
    // find_first_not_of(" \t") returns index -> not empty,
    //  so:
        // find_first_not_of(" \t") != npos       -> not empty
        //  so:
            // find_first_not_of(" \t") == npos       -> empty

    return str.find_first_not_of(" \t") == string::npos;
}
```

filter1.cc

```
#include "filter.ih"

Filter::Filter(istream &in)
:
    d_lines(in)
{}
```

firstnonempty.cc

```
#include "filter.ih"

// static
size_t Filter::firstNonEmpty(size_t size, string const *str)
{
    size_t idx = 0;
                                        // skip initial empty lines
    while (idx != size && empty(str[idx]))
        ++idx;

    return idx;
}
```

lastnonempty.cc

```
#include "filter.ih"

// static
size_t Filter::beyondLastNonEmpty(size_t size, string const *str)
{
    size_t idx = size;
                                         // skip all empty lines at the end
    while (idx-- && empty(str[idx]))
        ;

    return idx + 1;                      // idx at the last non-empty line,
}                                        // 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 byany new or modified helper functions. Everything not shown is assumed to be the unchanged.

cpu.h

```
#ifndef INCLUDED_CPU_
#define INCLUDED_CPU_

#include "../tokenizer/tokenizer.h"
#include "../memory/memory.h"
#include "../enums/enums.h"

class Memory;  //      Jaap: why this?

class CPU
{
    enum
    {
        NREGISTERS = 5,                                                 // a..e at indices 0..4, respectively
        LAST_REGISTER = NREGISTERS - 1
    };

    struct Operand
    {
        OperandType type;
        int value;
    };

    Memory &d_memory;
    Tokenizer d_tokenizer;

    int d_register[NREGISTERS];

    public:
        CPU(Memory &memory);
        void start();

    private:
        bool error();                                                 // show 'syntax error', and prepare for the
                                                                      // next input line
                                                                      // return a value or a register's or
                                                                      // memory location's value

        void stp();                                                   // helpers for start
        static void (CPU::*execute[])();
        void errorwrap();

        int dereference(Operand const &value);
        static int (CPU::*readOperand[])(Operand const &value);
        int valueReturn(Operand const &value);
        int memoryReturn(Operand const &value);
        int registerReturn(Operand const &value);

        bool rvalue(Operand &lhs);                                    // retrieve an rvalue operand
        bool lvalue(Operand &lhs);                                    // retrieve an lvalue operand

                                                                      // determine 2 operands, lhs must be an lvalue
        bool operands(Operand &lhs, Operand &rhs);

        bool twoOperands(Operand &lhs, int &lhsValue, int &rhsValue);

                                                                      // store a value in register or memory
        void store(Operand const &lhs, int value);
        void storeRegister(int place, int value);
        void storeMemory(int place, int value);
        static void (CPU::*storeValue[])(int place, int value);

        void mov();                                                   // assign a value
        void add();                                                   // add values
        void sub();                                                   // subtract values
        void mul();                                                   // multiply values
        void div();                                                   // divide values (remainder: last reg.)
                                                                      // div a b computes a /= b, last reg: %
        void neg();                                                   // negate a value
        void dsp();                                                   // display a value
};

#endif
```

dereference.cc

```
#include "cpu.ih"

int CPU::dereference(Operand const &value)
{
     return (this->*readOperand[value.type])(value);
}
```

memoryreturn.cc

```
#include "cpu.ih"

int CPU::memoryReturn(Operand const &value)
{
    return d_memory.load(value.value);
}
```

readoperand.cc

```
#include "cpu.ih"

int (CPU::*CPU::readOperand[])(Operand const &value) =      // order as in enums.h
{
    nullptr,                                    // padding for syntax, will never be called
    &CPU::valueReturn,
    &CPU::registerReturn,
    &CPU::memoryReturn
};
```

registerreturn.cc

```
#include "cpu.ih"

int CPU::registerReturn(Operand const &value)
{
    return d_register[value.value];
}
```

valuereturn.cc

```
#include "cpu.ih"

int CPU::valueReturn(Operand const &value)
{
    return value.value;
}
```

start.cc

```
#include "cpu.ih"

void CPU::start()
{
    while (true)
    {
        (this->*execute[d_tokenizer.opcode()])();
        d_tokenizer.reset();
    }
}
```

errorwrap.cc

```
#include "cpu.ih"

void CPU::errorwrap()
{
    error();
}
```

execute.cc

```
#include "cpu.ih"

void (CPU::*CPU::execute[])() =                 // order as in enums.h
{
    &CPU::errorwrap,
    &CPU::mov,
    &CPU::add,
    &CPU::sub,
    &CPU::mul,
    &CPU::div,
    &CPU::neg,
    &CPU::dsp,
    &CPU::stp
};
```

stp.cc

```
#include "cpu.ih"

void CPU::stp() // seperate file, add to header
{
}
```

store.cc

```
#include "cpu.ih"

void CPU::store(Operand const &lhs, int value)
{
    (this->*storeValue[lhs.type])(lhs.value, value);
}
void CPU::storeRegister(int place, int value)
{
    d_register[place] = value;
}

void CPU::storeMemory(int place, int value)
{
    d_memory.store(place, value);
}

void (CPU::*CPU::storeValue[])(int place, int value)
{
    nullptr,
    nullptr,                                        // these should never be called
    &CPU::storeRegister,
    &CPU::storeMemory
};
```

storememory.cc

```
#include "cpu.ih"

void CPU::storeMemory(int place, int value)
{
    d_memory.store(place, value);
}
```

storeregister.cc

```
#include "cpu.ih"

void CPU::storeRegister(int place, int value)
{
    d_register[place] = value;
}
```

storevalue.cc

```
#include "cpu.ih"

void (CPU::*CPU::storeValue[])(int place, int value)
{
    nullptr,
    nullptr,                                    // these should never be called
    &CPU::storeRegister,
    &CPU::storeMemory
};
```

Exercise 48

Problem statement. *Design the CSV class header.*
Solution.

Data Model

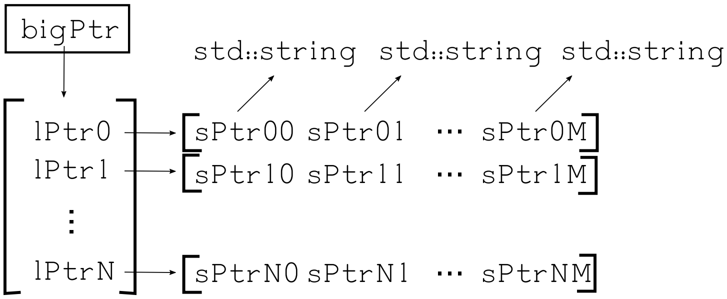

Figure 1: `bigPtr` is a triple pointer. It points to an array of 'line pointers', each of these point to an array of `std::string` pointers representing the comma-seperated values. For example: using the notation above we have `bigPtr[1][1] = sPtr11` for the second value on the second line.

csv.h

```
#ifndef CSV_HEADER_H
#define CSV_HEADER_H

#include <string>                                         // std::string
#include <istream>                                        // std::istream

class CSV
{
    size_t d_size = 1;                                    // number of lines allocated
    size_t d_nLines = 0;                                  // number of lines read
    size_t d_nFields = 1;                                 // number of values per line
    char d_fieldSep;                                      // field seperator (default comma)

    std::string ***bigPtr;                                // pointer to array of line pointers (see also big comment belo

    public:
        ~CSV();
        CSV(size_t field, char fieldSep = ',');

        std::string const *const *const *data() const;    // return pointer to data
        std::string const &lastline()          const;    // ref last extraction

        size_t nFields()                        const;     // values per line, set in first read
        size_t size()                           const;     // number of currently stored lines

        size_t read(std::istream &in, size_t nLines = 0);  // read lines using read1, return number read

        std::string ***release();                          // return pointer to data, move responsibility for data
                                                           // to called. Resets bigPtr but does not erase stored lines.
        void clear(size_t nFields = 0);                    // erase everything
    private:
        bool read1(std::istream &in);                      // read 1 line, parse for CSV's, set nFields
};

#endif // CSV_HEADER_H

// Line pointers point to array of pointers
// to std::string. i.e. :
// bigPtr -> [Lptr0 Lptr1 ... LptrN]
// where Lptri -> [strPtri1 strPtri2 ... strPtriM] for i = 1,...,N
// where strPtrik -> std::string                   for k = 1,...,M/
// see also the figure in the report.
```

main.cc

```
#include "csv.h"

int main()
{
    CSV file1(5, ',');
}
```

Exercise 49

Problem statement. *Implement the CSV class member functions.*
Solution.

csv.h

```
#ifndef CSV_HEADER_H
#define CSV_HEADER_H

#include <string>                                           // std::string
#include <istream>                                          // std::istream
#include "../csvextractor/csvextractor.h"

class CSV
{
    size_t d_size;                                      // number of lines allocated
    size_t d_nLines;                                    // number of lines read
    size_t d_nFields;                                   // number of values per line
    char d_fieldSep;                                  // field seperator (default comma)
    std::string d_lastLine;

    std::string ***bigPtr;                                  // pointer to array of line pointers (see also big comment bel

    public:
        CSV(size_t field, char fieldSep = ',');

        std::string const *const *const *data() const;      // return pointer to data
        std::string const &lastline()           const;      // ref last extraction

        size_t nFields()                        const;      // values per line, set in first read
        size_t size()                           const;      // number of currently stored lines

        size_t read(std::istream &in, size_t nLines = 0);   // read lines using read1, return number read

        std::string ***release();                           // return pointer to data, move responsibility for data
                                                            // to called. Resets bigPtr but does not erase stored lines.
        void clear(size_t nFields = 0);                     // erase everything
    private:
        bool read1(std::istream &in);                       // read 1 line, parse for CSV's, set nFields
        void allocate();
        void doubleSize();
};

inline size_t CSV::nFields() const
{
    return d_nFields;
}

inline size_t CSV::size() const
{
    return d_size;
}
#endif // CSV_HEADER_H

// Line pointers point to array of pointers
// to std::string. i.e. :
// bigPtr -> [Lptr0 Lptr1 ... LptrN]
// where Lptri -> [strPtri1 strPtri2 ... strPtriM] for i = 1,...,N
// where strPtrik -> std::string                   for k = 1,...,M/
// see also the figure in the report.

// - memcpy copies raw bytes
```

csv.ih

```
#include "csv.h"
#include <string>
#include <iostream>
using std::string;
```

allocate.cc

```
#include "csv.ih"

void CSV::allocate()
{
    bigPtr = new std::string **[1];                              // line array
}
// allocate me some memory
```

clear.cc

```
#include "csv.ih"

void CSV::clear(size_t nFields)                                            // nFields defaults to 0
{
                                                                           // de-allocate all every line array
    for (string **line = bigPtr[0]; line != bigPtr[d_nLines - 1]; ++line)
        delete[] line;
                                                                           // de-allocate array of lines
    delete[] bigPtr;
                                                                           // reset parameters
    d_size = 1;
    d_nLines = 0;
    d_nFields = nFields;
                                                                           // re-allocate (should be private helper)
    allocate();
}
```

csv1.cc

```
#include "csv.ih"

CSV::CSV(size_t field, char fieldSep)
    :
        d_size(1),                   // to allocate: 1 line
        d_nLines(0),                  // 0 lines read so far
        d_nFields(field),             // to allocate: 'field' fields
        d_fieldSep(fieldSep),          // set field seperator, default ','
        d_lastLine()
{
    bigPtr = new std::string **[1];     // allocate line array
}
```

data.cc

```
#include "csv.ih"

std::string const *const *const *CSV::data() const
{
    return bigPtr;
}
```

doublesize.cc

```
#include "csv.h"

void CSV::doubleSize()
{
    d_size = d_size << 1;
    allocate();
}
```

lastline.cc

```
#include "csv.h"

std::string const &CSV::lastline() const
{
    return d_lastLine;
}
```

main.cc

```
#include "csv.h"

int main()
{
    CSV file1(5, ',');
}
```

read.cc

```
#include "csv.h"

size_t CSV::read(std::istream &in, size_t nLines) // nLines defaults to 0
{
    size_t lines = 0;

    if (nLines == 0)
        while (in.good())                         // read all lines
            {
                    read1(in);
                    ++lines;
            }
    else
         while (lines != nLines && in.good())     // read 'nLines' lines
            {
                    read1(in);
                    ++lines;
            }
    return lines;
}

// By default, all lines of in are read and are processed by the read1 member.
// By specifying a non-zero value for the nLines parameter the specified number of
// lines is read from in. Reading stops once in's status is not good. When nLines
// is specified as zero, then reading continues until all CSV lines have been processed.
// The number of successfully processed lines is returned.
```

read1.cc

```
#include "csv.h"

bool CSV::read1(std::istream &in)
{
    CSVextractor csvFile(in);              // CSVextractor takes 1 line from stream 'in'

    if (d_nFields == 0)                    // field count
        d_nFields = csvFile.nFields();

    if (d_size - d_nLines  == 0)           // increase capacity
        doubleSize();

    return csvFile.parse(bigPtr);
}

// One line is read from in and is parsed for its CSVs. If parsing fails, false is returned.
// After successfully calling read1 for the first time all subsequent lines read by read1 must
// have the same number of comma separated values as encountered when calling read1 for the first time..
```

release.cc

```

```