Week 6

Exercise 50

../50–3/main.ih

```
#include <iostream>

#include "charcount/charcount.h"

using namespace std;

void showChar(char ch);
```

../50–3/main.cc

```
#include "main.ih"

int main()
{
    CharCount cc;
                                                // process and count all chars
    cout << "processed " << cc.count(cin) << " characters\n";

    CharCount::CharInfo const &info = cc.info();

    for (size_t idx = 0; idx != info.nChar; ++idx)
    {
        showChar(info.ptr[idx].ch);
        cout << ": " << info.ptr[idx].count << " times\n";
    }
    cout << "capacity is " << cc.capacity() << '\n';
}
```

../50–3/showChar.cc

```
#include "main.ih"

void showChar(char ch)
{
    cout << "char ";

    switch (ch)
    {
        case '\n':
            cout << "'\\n'";
        break;

        case '\t':
            cout << "'\\t'";
        break;

        case ' ':
            cout << "' '";
        break;

        default:
            if (isprint(ch))
                cout << '\'' << ch << '\'';
            else
                cout << static_cast<size_t>( static_cast<unsigned char>(ch) );
        break;
    }
}
```

../50–3/charcount/charcount.h

```
#ifndef INCLUDED_CHARCOUNT_
#define INCLUDED_CHARCOUNT_

#include <iosfwd>

class CharCount
{
  enum capacities
  {
    MAXSIZE = 255, //maximum size since no more ASCII values possible
    STARTSIZE = 8, //a very small file might only contain one word
  };               //in which case 8 seperate ASCII values might be enough

  enum Action
    {
        APPEND = 0,
        INSERT = 1,
        ADD = 2
    };

    public:
        struct Char
        {
            char ch;
            size_t count;
        };

        struct CharInfo
        {
            Char *ptr;
            size_t nChar;
        };


    private:

        size_t d_cap = MAXSIZE;
        size_t d_size = STARTSIZE;

        CharInfo d_info =
        {
          static_cast<CharCount::Char *>(
          operator new(d_size * sizeof(CharCount::Char)))
          , 0
        };

        //allocating raw memory block for the array of Char Objects

        static void (CharCount::*s_action[])(char ch, size_t idx);

        //declares the array of pointers so it can reach private member
        //functions (add, insert, append)

    public:
        ~CharCount(); //defining destructor so that the used memory is
                      //freed at the end of main.
        size_t count(std::istream &in);
        CharInfo const &info() const;
        size_t const capacity() const;

    private:
        void process(char ch);
        //calls locate and then the appropriate action
        //(append, insert or add)
```

```
65          Action locate(size_t *idx, char ch);
66          //locates the index of the current char and returns the appropriate
67          //action
68          void append(char ch, size_t idx); // inserts char at nChar
69          void insert(char ch, size_t idx); // inserts char at index
70                                            // and calls transfer
71
72          void add(char ch, size_t idx);
73                                            //increases count by 1 of already
74                                            //existing char object
75          void transfer(Char *dest, size_t begin, size_t end);
76          //moves all chars from begin to an index 1 higher, starting at the
77          //highest index
78          void enlarge(); //allocates a new raw block of memory, twice the size
79                          //of the previous memory
80          void destroy();
81                          //frees the memory of the Char objects and afterwards
82                          //the memory used by the pointer itself.
83
84          CharCount::Char rawCapacity() const;
85
86          //returns the current raw capacity
87
88  };
89
90  inline CharCount::CharInfo const &CharCount::info() const
91  {
92      return d_info; //returns a reference to charinfo object
93  }
94
95  inline CharCount::Char CharCount::rawCapacity() const
96  {
97      return *(d_info).ptr; //returns current raw capacity
98  }
99  inline size_t const CharCount::capacity() const
100 {
101     return d_size;
102 }
103
104
105 #endif
```

../50–3/charcount/charcount.ih

```
1  #include "charcount.h"
2
3  #include <iostream>
4  using namespace std;
```

../50–3/charcount/add.cc

```
1  #include "charcount.ih"
2
3  void CharCount::add(char ch, size_t idx)
4  {
5      ++d_info.ptr[idx].count;
6  }
```

../50–3/charcount/append.cc

```
1  #include "charcount.ih"
2
3  void CharCount::append(char ch, size_t idx)
4  {
5      insert(ch, d_info.nChar);
```

```
}
//appendix character at the end, adding index so the array of pointers
//to members works.
```

../50–3/charcount/count.cc

```
#include "charcount.ih"

size_t CharCount::count(istream &in)
{
    size_t nChars = 0;

    char ch;

    while (in.get(ch))
    {
        ++nChars;
        process(ch);                    // add ch to the set of characters
    }

    return nChars;
}
```

../50–3/charcount/data.cc

```
#include "charcount.ih"

void (CharCount::*CharCount::s_action[])(char ch, size_t idx) =
{
    &CharCount::append,
    &CharCount::insert,
    &CharCount::add,
};

//array of pointers to memberfunctions
```

../50–3/charcount/destroy.cc

```
#include "charcount.ih"

void CharCount::destroy()
{
  for (Char *end = d_info.ptr + d_size; end-- != d_info.ptr; )
      end->~Char(); //destructs the Char Objects

  operator delete(d_info.ptr); //destructs the pointer to the Char Objects
}
```

../50–3/charcount/destructor.cc

```
#include "charcount.ih"

CharCount::~CharCount()
{
  destroy();  //Class destructor
}
```

../50–3/charcount/enlarge.cc

```
#include "charcount.ih"


void CharCount::enlarge()
{
```

```
6     if  ((d_size <<= 1) > d_cap) //checks if there is capacity for another Char
7       d_size = d_cap;            //object and doubles the capacity if needed
8                                  //if it is then larger than its cap its set
9                                  //to the maximum capacity instead
10    CharCount::Char *tmp = static_cast<CharCount::Char *>(
11      operator new(d_size * sizeof(CharCount::Char)));
12      //allocates a new block of raw memory the size of the current capacity times
13      //the size of a Char Object.
14    for  (size_t index = d_size; index--; )
15      new(tmp + index) CharCount::Char{ d_info.ptr[index] };
16      //copies the old objects into the newly created one
17    destroy(); //destructs the old objects and the pointer to them
18    d_info.ptr = tmp; //sets the ptr to the newly created raw block of memory
19  }
```

../50–3/charcount/insert.cc

```
1   #include "charcount.ih"
2
3   void CharCount::insert(char ch, size_t idx)
4   {
5       if  (d_size == d_info.nChar + 1)
6         enlarge();                    //increase size of memory if needed
7
8       Char *&ptr = d_info.ptr;
9                                       // transfer the rest
10      transfer(ptr + d_info.nChar + 1, idx, d_info.nChar);
11
12      ptr[idx] = Char{ ch, 1 };       // insert the new element
13
14      ++d_info.nChar;                 // added new element
15      d_info.ptr = ptr;               // point at the new Char array
16  }
```

../50–3/charcount/locate.cc

```
1   #include "charcount.ih"
2
3   CharCount::Action CharCount::locate(size_t *destIdx, char ch)
4   {
5       size_t uCh = static_cast<unsigned char>(ch); //converts the current char to
6       //an unsigned char such that the objects are sorted in the desired order
7
8       for (size_t idx = 0; idx != d_info.nChar; ++idx)
9       {
10          size_t value = static_cast<unsigned char>(
11                                  d_info.ptr[idx].ch
12                              ); //as above to compare the stored chars in the
13                                 //Char Objects to the current char
14
15          if (uCh > value) //if current char is greater continue to the next
16              continue;    //stored Char object
17
18          *destIdx = idx;  //otherwise we now know where to insert the new Char
19                           //object
20          return uCh == value ?
21                      ADD       //if it is equal it already exists and we
22                  :             //want to add 1 to the Char Objects char counter.
23                      INSERT;   //otherwise we want to insert the new Char Object
24      }                         //at the current index
25
26      return APPEND;            // if no value greater than the current chars
27                                //value is found we want to append the new Char
28                                //Object to the Char Object array.
29  }
```

../50–3/charcount/process.cc

```
1  #include "charcount.ih"
2
3
4  void CharCount::process(char ch)
5  {
6      size_t idx;
7      Action loc_action = locate(&idx, ch);
8      (this->*s_action[loc_action])(ch, idx);
9  }
10
11 //defines array of pointers to member functions
12 //locate finds the index of the current character and what action to take
13 //this action and index is then passed to the array of pointers
```

../50–3/charcount/transfer.cc

```
1  #include "charcount.ih"
2
3  void CharCount::transfer(Char *dest, size_t begin, size_t end)
4  {
5      for (; begin - 1 != end; --end)
6          *dest-- = move(d_info.ptr[end]);
7  }
```

Exercise 51

New / delete variants

New variable or array
Allocates new memory sized appropriately to the type and dimensions specified. Thereafter it will attempt to construct and initialise these objects. Finally, it will return a pointer to the start of the memory allocated to these objects. The advantage and consideration of constructing an object (array) is that one has control over when it is deleted (if at all...).

```
int main()
{
  Class *pointer;
  if (boolAppropriate)
    pointer = new Class;
    // or
    pointer = new Class[5];

  if (boolNoLongerNecessary)
    delete pointer;
    // or
    delete[] pointer;
}
```

Without the use of new (i.e. Class newClass), the scope of the newly constructed class would simply be limited to within the if statement, and thereafter destroyed. However, now the object persists until manually deleted. As such, the creation and destruction do not take place unless necessary. Another example would be in the creation of arrays with dimensions that are not previously established, and re-pointing a pointer to the new array.

Operator and placement new
Allows for the allocation of raw memory that can be cast to fit a certain amount of a specific type. It is useful if one already has a reasonable idea of how much memory is needed for a specific application. Thereafter, a pointer to these memory locations is returned.

```
#include <memory>

void destroy()
{
  for (string *end = d_storage + d_size; end-- != d_memory)
    end->~string();
  operator delete(d_memory);
}

int main()
{
  string *tmp = static_cast<string *>(operator new(d_capacity * sizeof(string *)));
  // Raw memory is allocated, tmp points there

  for(size_t idx = d_size; idx--; )
    new(tmp + idx) string{ d_storage[idx] }
  // Old strings copied into newly allocated memory

  destroy();
  // Old memory deleted
  d_memory = tmp;
  // d_memory now points to new memory, which contains the initial strings
  new (d_memory + d_size) std::string{ newString };
  ++d_size;
  // Placement new adds a new string to the memory
  // ...
  destroy(); // one last time
}
```

In this example, we know that a block of memory sized to fit *d_capacity* strings is required, but not yet which constructor will be used to fill it (which comes later). Hence, it can already be allocated, but left empty

until that decision is made (or even if it is not made at all).

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
```

../53/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     d_pPstrings[d_size] = new std::string{ next };  // Add the new string
9     ++d_size; // Increase size
10  }
```

../53/strings/c_argcargv.cc

```
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   };
```

../53/strings/c_environlike.cc

```
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   };
```

../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  };
```

../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 };
```

../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
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[]);           // 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* rawMemory(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
```

../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

Type	*Time*
real	25m36,605s
user	25m25,435s
sys	0m11,168s

Table 1: Original

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

Table 2: Double pointers

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

Table 3: Placement new

Tables 1 through 3 display the time it took to run the respective programs. Note that the original implementation was timed on another machine 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) ;
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 founc 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 {                      //to functions
27     REGISTER = 0,           // register index
28     MEMORY = 1,             // memory location (= index)
29     VALUE = 2,              // direct value
30     SYNTAX = 3,             // syntax error while specifying an operand
31 };
32
33
34 #endif
```