Jaap van der Leest
Timon van der Berg
October 19, 2017

C++ Course
Assignment 5

Exercise 34

1. Differences pointers and arrays
While the addresses to which a pointer points can be changed, an array will, after creation, always be represented by the same block of adresses in memory. This memory is reserved for the array.
A pointer does not reserve the memory to which it points.

2. Differences between pointers and references
References have to be initialized during decleration and are constant thereafter, whereas pointers can be declared without initialization and may be changed to point somewhere else after initialization. Pointers can point nowhere, references always point somewhere.
Pointers have to be dereferenced explicitely, whereas references are implicitely dereferenced.
References can be bound to anonymous values, whereas pointers can not.

3.
I suppose tt is a function.

answer:
a. int array[20][30]
element [3][2] is reached by looking at row 3 and column 2 of an array.
It is arranged like this:

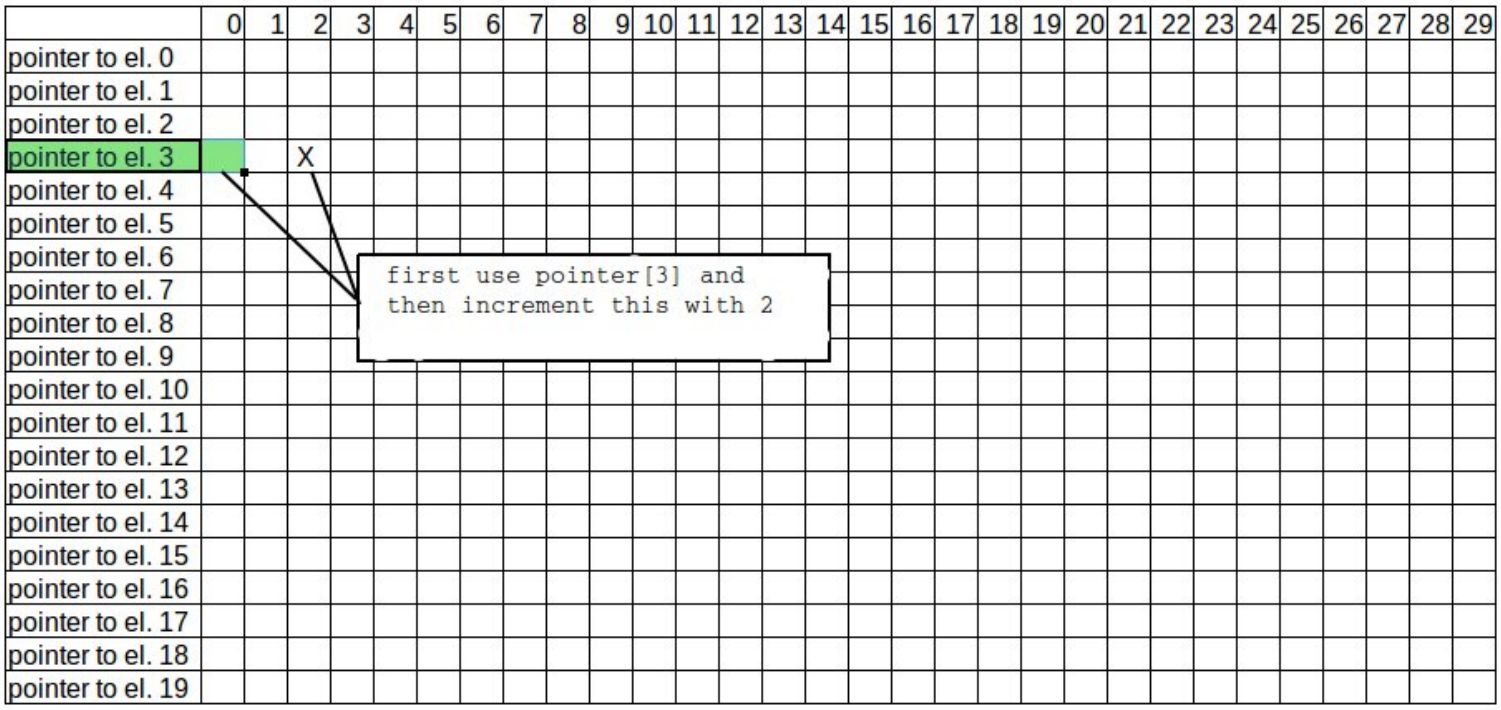
int array[20][30] column

Row / column	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29
0																														
1																														
2																														
Row 3			X																											
4																														
5																														
6																														
7																														
8																														
9																														
10																														
11																														
12																														
13																														
14																														
15																														
16																														
17																														
18																														
19																														

b. int *pointer[20]
element [3][2] is reached by taking the pointer to the first element of row 3,

so pointer[3], then adding 2 to the pointer. C++ knows the size of the element it points to
so just incrementing the pointer is sufficient to point to the next element, regardless
of the size of the element in the array.
this is arranged like this:

int *pointer[20]



4. Pointer Arithmetic
Incrementing a pointer increases its adress by a value equal
to the size of the pointer type. This increment ++ptr can be also be written
ptr = ptr + 1. In the same way we can write ptr = ptr + 3 to move three of
<typesize of ptr> to the right. Substraction is defined analogously.
Thus pointer arithmetic is just arithmetic on the adress to which the
pointer points. We can also
- compare pointers to pointers
- compare pointers to 0 or nullptr (pointer that does not point
anywhere)
- substract pointer from pointers (tells you how much memory is in between)

5. explain why accessing an element in an array using only a pointer variable is
to be preferred over using an index expression.
the use of pointers is more flexible.
e.g. If we want to access a multidimensional array of unknown size this would be
fairly problematic in a function if we want to give the function the array
as parameter. Because at the definition of the function we need to know (at least
part of the) array dimensions.
Using pointers we can point to just the start of the array. Giving the function
the actual dimensions at runtime it can calculate the appropriate positions of
the different values.

Exercise 35

We apologize for any damage to the eyes of the reader caused by the following table.

---
definition: rewrite: pointers: semantics:

---

int x[8]; x[4] *(x + 4) x + 4 points to the location of the 4th int beyond x. That element is reached using the dereference operator (*)

int x[8]; x[3] = x[2]; *(x + 3) = *(x + 2) x can be considered a pointer to the first element of the array x. We reach the fourth and third element by pointer arithmetic.

char *argv[8]; cout << argv[2]; cout << *(argv+2) argv is an array of 8 pointers to ints. The cout statement first produces the pointer, i.e. it's adress value. argv can be considered a pointer to the first pointer in the array. If we increase it by 2 we obtain a pointer to the third pointer in the array. Dereferencing gives us the pointer in that position.

int x[8]; &x[10] - &x[3]; (x + 10) - (x + 3) x is an array of 8 ints. The & operator here tries to return the adress of the 11th and 4th elements. The 11th element does not exist. For a pointer this is 'not a problem', since it will just read the memory that is there without caring if this is actually part of the array or in fact contains an int. What actually happens is the subtraction of 1 address from another. Therefore what is stored at the addresses is irrelevant in this example. Subtracting the 2 addresses will always result in 7.

char *argv[8]; argv[0]++; (*argv)++ argv is an array of pointers to the first chars of null terminated byte strings. argv[0] is a pointer to the first char of the first null terminated byte string. Incrementing argv[0] means that argv[0] now points to the second char of the first null terminated byte string. This might be dangerous because you now have lost a reference to the beginning of the string. So memory leaks can easily occur.

*argv gives the first first element of argv, i.e. the first pointer in the array. Incrementing means it now points to to the memory following that char, which also is the second char of the null terminated byte string.

char *argv[8]; argv++[0]; *argv++ argv[0] is processed in a stmnt. after that argv (so does argv[0]) points to what was argv[1].

char *argv[8]; ++argv[0]; +++argv first argv[0] (so does argv) points to the second char in the first ntbs of the array. Then the stmnt is processed.

char **argv; ++argv[0][2]; +++(*argv + 2) argv is a pointer to a pointer to a char. In this case it points to the first argument (program name) from that it points to the 3rd char. The value of this char will be increased by 1. So an A becomes a B. Thereafter the stmnt is processed.

Exercise 36: Strings Class

The following implements a class `Strings` wrapping an array of a changable number of std::string elements. The member functions requested in 37 and 38 are also shown in the class header. strings/strings.h:

```
#ifndef DEFINE_STRINGS_H
#define DEFINE_STRINGS_H

#include "../struct.h"                          // 'Rel' output struct
#include <string>                               // std::string, size_t
#include <istream>                              // std::istream

class Strings
{
    size_t d_size = 0;                          // number of strings
    std::string *d_str = 0;                     // pointer to array of strings
    std::string const d_emptystring = "";       // yeah.
```

```
    public:
        Strings();
        Strings(int const argc, char *argv[]);
        Strings(char *env[]);
        Strings(std::istream &in);

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

        std::string const &at(size_t index) const; // non-modifiable string
        std::string &at(size_t index);             // modifiable string

        Rel release();                             // return data and clear

        void add(std::string str);                 // add string to object data

        void stringsSwap(Strings& objectB);        // switch contents of 2 strings objects

    private:
        std::string &priv_at(size_t index) const;  // backdoor
};

inline size_t Strings::size() const
{
    return d_size;
}
inline std::string Strings::data() const
{
    return d_str;
}

#endif

// class Strings that can either be initialized with an argc, argv
// pair of arguments, with an environ type of argument,
// or with an input stream.
```

strings/strings.ih:

```
#include <istream>              // std::istream
#include <string>               // string
#include "strings.h"            // 'Strings' class
```

main.cc:

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

int main(int argc, char *argv[], char *envp[])
{
    Strings argstr(argc, argv);
    Strings envstr(envp);
    Strings streamstr(std::cin);
}
```

strings/add.cc:

```
#include "strings.ih"                                   // 'Strings' class

void Strings::add(std::string str)                      // ntbs will be converted
{
    std::string *tmp = new std::string[++d_size];       // array 1 string bigger
    for (size_t index = 0; index != d_size - 1; ++index) // copy
```

```
        *(tmp + index) = *(d_str + index);
    tmp[d_size - 1] = str;                              // add new string
    delete[] d_str;                                     // delete old array
    d_str = tmp;                                        // point to new array
}

// -copy the currently stored strings to a new storage area
// -add the next string to the new storage area
// -destroy the information pointed at by d_str
// -update d_str and d_size so that they refer to the new storage area.
```

strings/strings1.cc:

```
#include "strings.ih"

Strings::Strings()          // default constructor returns null pointer
{
}
```

strings/strings2.cc:

```
#include "strings.ih"                   // 'Strings', string, istream

Strings::Strings(std::istream &is)
{
    std::string str;
    while (std::getline(is, str) && !is.eof())
        add(str);
}
```

strings/strings3.cc:

```
#include "strings.ih"

Strings::Strings(int const argc, char *argv[])
{
    for (int index = 0; index != argc; index++)
    {
        add(argv[index]);
    }
}
// since the number of additions is known
// this could be rewritten to add them all at once
// that would be more efficient (though both env and agrv are usually small)
```

strings/strings4.cc:

```
#include "strings.ih"

Strings::Strings(char *env[])
{
    for (int index = 0; index != sizeof(*env); index ++)
    {
        add(*(env + index));
    }
}
```

Exercise 37

See 36 for the class header. main.cc:

```
#include "filter/filter.h"
#include <iostream>

int main()
{
    Filter fil(std::cin);
    fil.display();
}
```

strings/at1.cc:

```
#include "strings.ih"

std::string const &Strings::at(size_t index) const          // non-modifiable at
{
    std::string const &ref = priv_at(index);
    return ref;
}
```

strings/at2.cc:

```
#include "strings.ih"

std::string &Strings::at(size_t index)          // const ref to modifiable string
{
    std::string &ref = priv_at(index);
    return ref;
}
```

strings/privat_at.cc:

```
#include "strings.ih"

std::string &Strings::priv_at(size_t index) const
{
    std::string &ref = (index > d_size) ? d_emptystring : d_str[index];
    return ref;
}
```

strings/release.cc:

```
#include "strings.ih"                   // 'Strings' class, <string>, <istream>
                                        // 'Rel' struct (via 'Strings')
Rel Strings::release()
{
    Rel out;
    out.data = d_str;
    out.size = d_size;

    d_str = 0;                          // clear instance
    d_size = 0;

    return out;
}

// release data to user, clear class instance
```

The filtering is implemented in a class 'filter'. filter/filter.h:

```
#include <istream>                      // std::istream
#include "../strings/strings.h"         // 'Strings' class
```

```
class Filter
{
    Strings d_String;
    public:
        Filter();
        Filter(std::istream &is);
        void display();
    private:
        void removeWhitespace();
};
```

filter/filter.ih:

```
#include "filter.h"    // 'Filter' class
#include "../struct.h" // output struct Rel
#include <istream>     // std::istream
#include <string>      // std::string
```

filter/display.cc:

```
#include "filter.ih"                    // 'Filter' class, 'Struct', <istream>, <string>
#include <iostream>                     // std::cout

void Filter::display()
{
    std::string whitespace = " \t\n\f\n\r";                                 // could use is_white

    Rel released = d_String.release();                                      // destroys String object
    std::string *out = released.data;
    size_t siz = released.size;

    while (out->find_first_not_of(whitespace) == std::string::npos)         // forward loop
    {
        ++out;
        --siz;
    }

    while((out + siz - 1)->find_first_not_of(whitespace) == std::string::npos) // backward loop
        --siz;

    for (std::string *ptr = out; ptr != out + siz; ++ptr)
        std::cout << *ptr << '\n';
}
```

filter/filter1.cc:

```
#include "filter.ih"

Filter::Filter()
    :
        d_String()
{
    // thats all folks
}
```

filter/filter2.cc:

```
#include "filter.ih"

Filter::Filter(std::istream &is)
    :
        d_String(is)
```

```
{
    // thats all folks
}
```

Exercise 38

main.cc:

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


using namespace std;

int main()                                          // int argc, char *argv[], char* envp[])
{

    string A = "Hello world for the first time\n";
    string B = "Hello world for the second time\n";

    Strings strA;                                   // create first object with first string as data
    strA.add(A);

    Strings strB;                                   // create second object with second string as data
    strB.add(B);

    cout << "before swapping: \n";
    cout << *strA.data();
    cout << *strB.data();


    strA.stringsSwap(strB);                         // call function to swap data.
                                                    // since these are the pointer weeks, no data
                                                    // is swapped really. Only the addresses in the
                                                    // pointers pointing to the data.

    cout << "after swapping: \n";
    cout << *strA.data();
    cout << *strB.data();

}
```

strings/stringsswap.cc:

```
#include "strings.h"

void Strings::stringsSwap(Strings& objectB)
{
    std::string *tempA = d_str;     // store the address of the data of objectA
    d_str = objectB.d_str;          // now switch the addresses
    objectB.d_str = tempA;
}
```

Exercise 39

main.cc:

```
#include "exercise_39.ih"

using namespace std;
```

```
int main()
{
    int square[10][10];             // for the sake of the exercise the array is
                                    // not initialized what normally should be done.
    int (*row)[10] = square;        // pointer to the rows of square

    inv_identity(row);              // fill the matrix


    for (size_t r=0; r != 10 ; ++r)  // display the matrix
    {
        cout << "Rij " << r << ": ";
        for (size_t c=0; c != 10 ; ++c)
            cout << square[r][c];
        cout << '\n';
    }

}
```

exercise39.ih:

```
#include <iostream>     // size_t

#ifdef __cplusplus
    extern "C" {
#endif

void inv_identity(int (*arrayPointer)[10]);

#ifdef __cplusplus
    }
#endif
```

inv_indentity.cc:

```
#include "exercise_39.ih"

using namespace std;

void inv_identity(int (*arrayPointer)[10])
{
    for (int (*row)[10] = arrayPointer; row != arrayPointer + 10; ++row)
    {
        for (int *col = static_cast<int*>(*row),
            *cross = (static_cast<int*>(*row) + (row - arrayPointer));
            col != static_cast<int*>(*row) + 10;
            ++col)

                (col == cross) ? *col = 0 : *col = 1;
    }
}

// explanation:
// int *col = static_cast<int*>(*row):
// *col gets the same address of row. Since row points to 10 ints and col points
// to a single int a conversion is necessary.
//
// *cross = (static_cast<int*>(*row) + (row - arrayPointer):
// *cross gets the address of row. Then the index of row (from the first for loop)
// is added
//
// (col == cross) ? *col = 0 : *col = 1:
// if the addresses of col and cross match, that means that both indices from both
```

```
// for loops are equal. So we have a point on the main diagonal and there should be
// a zero, while all the others should be ones
```