Programming report
Week 4 Assignments C++

Jaime Betancor Valado
Christiaan Steenkist
Remco Bos
Diego Ribas Gomes

October 7, 2016

Assignment 31, Pointer Questions

Here are the answers to the questions posed in assignment 31.

1, Difference between pointer variables and arrays

The memory allocation of a pointer is dynamic and that of an array is static, can't be changed. The fundamental difference between them is that a pointer is a variable storing the memory addres of the variable to which it points, and the array is a chunk of variables allocated in one place of memory.

2, Difference between pointer and reference variables

In the reference case we can't re-asign them after the assigment statement and in pointers we can make them point to any other place anytime. As an implication pointers can point to nothing but references need to refer to an object. We can say that a reference is an alias to another objec and a pointer stores the memory address of that object.

3, How element [3][2] is reached

Part a, for variable int array[20][30]

In this case, the word array is an alias for the memory address of array[0][0] and when you write array[3][2] the compiler replace that labe with the corresponding memory address without the need of making further computations.

Part b, for variable int *pointer[20]

When we tell the compiler to search for the location [3][2] the pointer addresses the memory location in which the first element is stored in memory, but for that it needs an intermediate step. We can say that the pointer points to a point that points to the first element of array. Finally the pointer arithmetic begins. In the figure below, [pointer] is the pointer label that points to the memory address of the ptr 0. The pointers ptr point to the first elements of the corresponding row.

```
[pointer]
    |
    v
[ptr 0]  -> [int 0] . . . [int 29]
    .
    .
    .
[prt 19] -> [int 0] . . . [int 29]
```

4, What does pointer arithmetic mean?

Is the operations that you can do with pointers. You can add, substract, increment or decrement them. For example if you have a type pointer and you want to locate the next type variable to which pointer points you can do ++pointer.

5-1, Why accesing an element in an array using only a pointer is preferred over using an index expression.

They are preferred over index expression because they don't hide anything to the programer, they express better what the programmer intends and also because in the past the pointer expression was fast proccessed than the array index expression.

5-2, Why is preferred to use pointer-loops over loops in which the control variable is a type variable?

When you are reading or writting an array inside a loop and you don't know when its last position is, is better to use a pointer comparison instead of an int comparison, the loop will stop when pointer points to a null termination.

Assignment 32, Pointer and array arithmatic

Table 1 has the arithmatic of the pointers and arrays as well as the descriptions of why it works out the way it does.

Assignment 33, The Strings class

Because c-style strings are overrated we made a `Strings` class to store multiple strings for us in the form of char pointers. In `String` objects strings can be stored and retrieved using the `store` and `at` function. The `size` function shows how many strings the object contains.

`at` functions

The `at` member functions should return modifiable or non-modifiable strings. In the case that a string literal is stored then when using at a non-modifiable string should be returned. When saving normal strings or arrays a modifiable string should be returned.

Table 1: Pointer Arithmatic

	Definition:	Rewrite:	Pointers:	Semantics:
x.	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 (*)
a.	int x[8];	x[3] = x[2];	*(x + 3) = *(x + 2)	x + 3 and x + 2 points to the location of the 3th and 2nd int beyond x. Then we assign the value of the 2nd one to the 3th one using the reference operator(*)
b.	char *argv[8];	cout <<argv[2];	cout <<*(argv + 2);	We have a pointer to a pointer of a variable char (array of ”strings”). (argv + 2) points to the location of the 2nd set of chars beyond argv. And the print it.
c.	int x[8];	&x[10] - &x[3];	10 - 3	In this case we are substracting two memory addresses of the vector x in positiosn 10 and so the difference in memory positions will be 10 - 3 = 7.
d.	char *argv[8];	argv[0]++;	(*argv)++	(argv + 0) points to the location of the set of characters argv. Then, in that ”string” we increment a position and delete the first character of that ”string”.
e.	char *argv[8];	argv++[0];	Error	This does not compile.
f.	char *argv[8];	++argv[0];	++(*argv)	(argv + 0) points to the location of the set of characters argv. Then, in that ”string” we increment a position and delete the first character of that ”string”. The difference between d. and e. is that the increment in this is after the statement.
g.	char **argv;	++argv[0][2];	Error	Does not compile

Code Listings

Listing 1: `main.h`

```
extern char **environ;
```

Listing 2: `main.cc`

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

using namespace std;

int main(int argc, char **argv)
{
  Strings strings1(argc, argv);
  Strings strings2(environ);

  if (strings2.size() >= 5)
    strings1.addString(strings2.at(4));

  for (size_t index = 0; index != strings1.size(); ++index)
  {
    cout << strings1.at(index) << '\n';
  }
}
```

Listing 3: `strings.ih`

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

using namespace std;
```

Listing 4: `strings.h`

```
#ifndef STRINGS_H_
#define STRINGS_H_

#include <iostream>

class Strings
{
  private:
    char **d_str;
    size_t d_size;

  public:
    Strings(int argc, char **argv);
```

```
    Strings(char **environ);

    size_t size();
    std::string at(size_t index);
    std::string const at(size_t index) const;

    void addString(std::string newString);
    void addString(char *charArray);
};

#endif
```

Listing 5: addstring1.cc

```
#include "strings.ih"

void Strings::addString(string newString)
{
  char **n_str = new char *[d_size + 1];

  for (size_t index = 0; index != d_size; ++index)
  {
    n_str[index] = d_str[index];
  }
  n_str[d_size] = &newString[0];

  delete d_str;
  d_str = n_str;
  ++d_size;
}
```

Listing 6: addstring2.cc

```
#include "strings.ih"

void Strings::addString(char *charArray)
{
  char **n_str = new char *[d_size + 1];

  for (size_t index = 0; index != d_size; ++index)
  {
    n_str[index] = d_str[index];
  }
  n_str[d_size] = &charArray[0];

  delete d_str;
  d_str = n_str;
  ++d_size;
}
```

Listing 7: at1.cc

```
#include "strings.ih"

string Strings::at(size_t index)
{
  if (index >= 0 && index < d_size)
    return d_str[index];
  string newString = "null";
  return newString;
}
```

Listing 8: at2.cc

```
#include "strings.ih"

string const Strings::at(size_t index) const
{
  if (index >= 0 && index < d_size)
    return d_str[index];
  return "null";
}
```

Listing 9: size.cc

```
#include "strings.ih"

size_t Strings::size()
{
  return d_size;
}
```

Listing 10: strings1.cc

```
#include "strings.ih"

Strings::Strings(int argc, char **argv)
:
  d_size(argc)
{
  d_str = new char *[argc];
  for (size_t index = 0; index != d_size; ++index)
  {
    d_str[index] = argv[index];
  }
}
```

Listing 11: strings2.cc

```
#include "strings.ih"
```

```
Strings::Strings(char **environ)
:
  d_size(0)
{
  while (*environ != NULL)
  {
    addString(*environ);
  }
}
```

Assignment 34, Strings swapping

For this exercise a member function was to be added to the `Strings` class that swaps the contents of two `Strings` objects. This is done by declaring storage variables in the scope of the member function and swapping the variables like swapping water between cups.

Code Listings

Listing 12: `strings.h`

```
#ifndef STRINGS_H
#define STRINGS_H

#include<iostream>

class Strings
{
  private:
    char **d_str;
    size_t d_size;

  public:
    Strings(int argc, char **argv);
    Strings(char **environ);

    size_t size();
    std::string at(size_t index);
    std::string const at(size_t index) const;

    void addString(std::string newString);
    void addString(char *charArray);
    void stringsSwap(Strings &objectA, Strings &objectB);
};

#endif
```

Listing 13: `stringsswap.cc`

```
#include "strings.ih"
```

```
void Strings::stringsSwap(Strings &objectA, Strings &objectB)
{
    size_t storedSize = objectA.d_size;
    objectA.d_size = objectB.d_size;
    objectB.d_size = storedSize;

    char **storedArray = objectA.d_str;
    objectA.d_str = objectB.d_str;
    objectB.d_str = storedArray;
}
```

Assignment 35, Pimpl

The goal of this exercise was to use and learn about the Pimpl method.

The library

The command used to compile was `main.cc to main.o: g++ -Wall -std=c++14 -c main.cc`. The command used to make the library and link main.o was `ar rsv libdata.a main.o`.

The makefile

```
program : display.o read.o data.ih data.h
  g++ -Wall -std=c++14 -o program display.o read.o \
    -L/home/user/Documents/Programming/C++-Part-1/Week-4/Ex35 -
   ldata -s
  mv program ..

display.o : data.h data.ih display.cc
  g++ -Wall -std=c++14 -c display.cc

read.o : data.h data.ih read.cc
  g++ -Wall -std=c++14 -c read.cc
```

First results

Here are the results of the first library.

- Object 1: value is: 1
- Object 2: value is: 2
- Object 3: value is: 3
- Object 4: value is: 4

After uncommenting the definition of `d_text` in data.h and its use in read.cc the program gave a segmentation fault.

The Pimpl makefile

Note that for each situation a new library is made.

```
program : display.o read.o dataconstrdestr.o data.ih data.h
  g++ -Wall -std=c++14 -o program display.o read.o dataconstrdestr.
   o \
  -L/home/user/Documents/Programming/C++-Part-1/Week-4/Ex35 -
   ldatapimpl2 -s
  mv program ..

dataconstrdestr.o : data.h data.ih dataconstrdestr.cc
  g++ -Wall -std=c++14 -c dataconstrdestr.cc

display.o : data.h data.ih display.cc
  g++ -Wall -std=c++14 -c display.cc

read.o : data.h data.ih read.cc
  g++ -Wall -std=c++14 -c read.cc
```

Second results

With `d_text` commented out.

- Object 1: value is: 1
- Object 2: value is: 2
- Object 3: value is: 3
- Object 4: value is: 4

With `d_text` in the running code.

- Object 1: value is: 1
- Object 2: value is: 2
- Object 3: value is: 3
- Object 4: value is: 4

Code listings

Listing 14: `data.ih`

```
#include "data.h"

#include <iostream>

using namespace std;
```

```
class Data::DataImp
{
    public:
        bool read();
        void display() const;

    private:
        int d_value = 0;
        std::string d_text;
};
```

Listing 15: data.h

```
#ifndef INCLUDED_DATA_
#define INCLUDED_DATA_

#include <string>

class Data
{
public:
    Data();
    ~Data();
    bool read();
    void display() const;

private:
    class DataImp;
    DataImp* d_pimpl;
};

#endif
```

Listing 16: main.cc

```
#include "main.ih"

int main(int argc, char **argv)
{
    Data *data = new Data();
    size_t count = 0;

    while (data->read())
    {
        cout << "Object " << ++count << ": ";
        data->display();
    }
```

```
    delete data;
}
```

Listing 17: display.cc

```
#include "data.ih"

void Data::display() const
{
    d_pimpl->display();
}

void Data::DataImp::display() const
{
    cout << "value is: " << d_value << '\n';
}
```

Listing 18: read.cc

```
#include "data.ih"

bool Data::read()
{
    d_pimpl->read();
    return cin.good();
}

bool Data::DataImp::read()
{
        d_text.clear();
        cin >> d_value;
        return cin.good();
}
```

Assigment 36, Pimpl questions

Here are the answers to the questions of assignment 36 about assignment 35.

When the program breaks down

The program breaks after creating the new library in step 2 of exercise 35 is because the calling code expects a smaller object size than in the first step where the object was smaller without the additional private member. So memory corruption occurs.

Why the program doesnt break down with the pointer implementation.

After using the pointer to implementation approach the program does not break because a pointer to the implementation of the private member uses the memory address and thus knows from the memory address how much memory the object takes up. Using no pointer and not changing the `main.o` (where an object is constructed) after adding a new private member, the main.o still thinks the object has the same size. This gives a memory corruption.

World-famous library

You can design 2 classes, 1 for the functionality (i.e. public interface) and 1 for the data (data/private members). The data class can use the class with functionality and vice versa. Data members can be added, because you only need to make an object from the functional class. This one remains the same in size, so no memory corruption. Changing the data class does not alter the size of the object made from the functional class. The data class is only used for the functional class and the data members are only accessed by the functional class if necessary.

Assignment 37, inverse identity

We were tasked with making a function that constructs the inverse identity of an array by using an array of pointers that point to the rows of a two-dimensional matrix. The function adheres to the problem description: two for loops, 1 parameter, 3 initialized pointer variables and one assignment statement.

Code listings

Listing 19: `main.h`

```
#include <iostream>

void inv_identity(int (*row)[10]);
```

Listing 20: `main.cc`

```
#include "main.h"

using namespace std;

int main(int argc, char **argv)
{
  int square[10][10];
  int (*row)[10] = square;

  inv_identity(row);
}
```

Listing 21: `inv_identity.cc`

```
#include "main.h"
#include <typeinfo>

using namespace std;
```

```
void inv_identity(int (*row)[10])
{
  for (int (*rowNum)[10] = row; rowNum != row + 10; ++rowNum)
  {
    for (int *colPos = *rowNum, *rowPos = *row + (*rowNum - *row) /
    10; colPos != *rowNum + 10; ++colPos, rowPos += 10) {
      *colPos = rowPos == colPos ? 0 : 1;
    }
  }
}
```