C++ Exercises
Set5

Author(s): Tino Alferink, Robert van Ark, Owen Givlin

–

15:01

October 14, 2024

33

```
This assignment answers some basic questions about pointers.

1) what differences exist between pointer variables and arrays:

A pointer variable stores the location of another variable as their value.
An array is a series of elements stored in a contiguous memory location.
The name of the array stores the location of the 0'th index of the array.
An array is a contiguous block in memory, while a pointer refers to one spot.

2) what differences exist between pointer variables and reference variables

A pointer variable stores the location of another variable as their value,
while a reference variable is an alias for an existing variable.

When a reference variable is initialized, it cannot be changed to point to
another variable, while for a pointer variable this is possible.

A reference variable cannot point to a NULL value, pointer variables can.

3a) how element [3][2] is reached for the variable 'int array[20][30]'

The size of an array[idx] element is 30 integers. To acces element[3],
we jump 3 * int[30] in memory to get that address. then we jump the size of 2
integers to that location to acces element [3][2].

3b) how element [3][2] is reached for the variable 'int *pointer[20]'

The variable pointer is an array of 20 pointers. Firstly, the fourth pointer of
this array is accessed, then this pointer is deferenced such that the array that
it points to is accessed after which the third element is accessed.

4) what is meant by 'pointer arithmetic'

Pointer arithmetic is doing calculations with addresses of variables
to obtain another variable address in memory. Such as increasing/decreasing its
value, + n/- n, shift to the address n places ahead/before the original pointer.
```

34

```
In this assignment we learn to give classes their own data.
```

Listing 1: `copycat.ih`

```
#include "copycat.h"
#include <cstring>

//using namespace std;
```

Listing 2: copycat.h

```
#ifndef COPYCAT_INCLUDED_
#define COPYCAT_INCLUDED_

#include <cstddef>

extern char **environ;

class CopyCat
{
    size_t d_size;
    char **d_data;

    public:
        CopyCat();                              //inline ; copies environ
        CopyCat(int argc, char **argv);         // 1
        CopyCat(char **data);                   // 2 ; copies environ-like variable

    private:
        static size_t const nElements(char const *const *data);
        static char **ntbsCopy(char const *const *data);
        static char *duplicate(char const *const src);
};

inline CopyCat::CopyCat()
:
    CopyCat(environ)
{}

#endif
```

Listing 3: copycat1.cc

```
#include "copycat.ih"

CopyCat::CopyCat(int argc, char **argv)
:
    d_size{ static_cast<size_t>(argc) },
    d_data{ ntbsCopy(argv) }
{}
```

Listing 4: copycat2.cc

```
#include "copycat.ih"

CopyCat::CopyCat(char **data)
:
    d_size{ nElements(data) },
    d_data{ ntbsCopy(data) }
{}
```

35

Purpose of this exercise: Learn to store an (apriori unknown) number of data elements in a class.

Listing 5: charcount/charcount.ih

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

using namespace std;
```

Listing 6: charcount/charcount.h

```
#ifndef INCLUDED_CHARCOUNT_
#define INCLUDED_CHARCOUNT_
```

```
#include <iosfwd>
#include <cstddef>

struct CharCount
{
    enum Action                     // enum to decide action
    {
        APPEND,
        INC,
        INSERT
    };

    struct Char
    {
        char ch;                    // the character itself
        size_t count;               // amount of occurences of char
    };

    struct CharInfo                 // char frequency of all chars
    {
        Char *charPtr = 0;          // points to Char objects
        size_t nCharObj = 0;        // number of Char objects stored
    };

    CharInfo d_charInfo;

    public:
        void count(std::istream &in);   // count occurences of characters
        CharInfo const info();          // return d_charInfo

    private:
        Action locate(char ch);         // return Action for character
        void append(char ch);           // append character to end of charPtr
        void insert(char ch);           // insert character in charPtr
        void increment(char ch);        // increment the count of character
};

#endif
```

Listing 7: charcount/count.cc

```
#include "charcount.ih"

void CharCount::count(istream &cin)
{
    char ch;
    while(cin.get(ch))              // get character from stdin
    {
        switch (locate(ch))         // choose Action according to locate
        {
        case APPEND:
            append(ch);
            break;
        case INSERT:
            insert(ch);
            break;
        case INC:
            increment(ch);
            break;
        default:
            break;
        }
    }
}
```

Listing 8: charcount/info.cc

```
#include "charcount.ih"

CharCount::CharInfo const CharCount::info()
```

```
{
    return d_charInfo;
}
```

Listing 9: charcount/locate.cc

```
#include "charcount.ih"

CharCount::Action CharCount::locate(char ch)
{

    if (d_charInfo.charPtr == 0     // no char saved || char > last element
        || ch > d_charInfo.charPtr[d_charInfo.nCharObj - 1].ch)
        return APPEND;

    for (size_t idx = 0; idx < d_charInfo.nCharObj; ++idx)
    {                               // char already saved
        if (ch == d_charInfo.charPtr[idx].ch)
            return INC;
    }
    return INSERT;                  // char < last element and not saved

}
```

Listing 10: charcount/append.cc

```
#include "charcount.ih"

void CharCount::append(char ch)
{
                                    // initialize array for new size
    Char *tmp = new Char[d_charInfo.nCharObj + 1];
                                    // copy chars
    for (size_t idx = 0; idx < d_charInfo.nCharObj; ++idx)
        tmp[idx] = d_charInfo.charPtr[idx];
                                    // append ch
    tmp[d_charInfo.nCharObj] = Char{ch, 1};

    ++d_charInfo.nCharObj;          // increment char count

    delete[] d_charInfo.charPtr;
    d_charInfo.charPtr = tmp;       // set tmp as new charPtr
}
```

Listing 11: charcount/increment.cc

```
#include "charcount.ih"

void CharCount::increment(char ch)
{
    for (size_t idx = 0; idx < d_charInfo.nCharObj; ++idx)
    {                               // found Char to increment
        if (ch == d_charInfo.charPtr[idx].ch){
            ++d_charInfo.charPtr[idx].count;
            break;
        }

    }
}
```

Listing 12: charcount/insert.cc

```
#include "charcount.ih"

void CharCount::insert(char ch)
{
                                    // initialize array for new size
    Char *tmp = new Char[d_charInfo.nCharObj + 1];
```

```
    size_t idx = 0;                 // copy chars before ch
    while(idx != d_charInfo.nCharObj && ch > d_charInfo.charPtr[idx].ch)
    {
        tmp[idx] = d_charInfo.charPtr[idx];
        ++idx;
    }

    tmp[idx] = Char{ch, 1};         // insert ch in tmp
                                    // copy chars after ch
    for (; idx < d_charInfo.nCharObj; ++idx)
        tmp[idx + 1] = d_charInfo.charPtr[idx];

    d_charInfo.nCharObj += 1;       // increment char count

    delete[] d_charInfo.charPtr;
    d_charInfo.charPtr = tmp;       // set tmp as new charPtr
}
```

Listing 13: main.ih

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

using namespace std;

void showChar(CharCount::Char charObj);
```

Listing 14: showchar.cc

```
#include "main.ih"

namespace {
    void showDefaultChar(char ch)   // helper function
    {
        if (isprint(ch))            // printable chars
            cout << "'" << ch << "'";
        else                        // non-printable chars
            cout << static_cast<int>(ch);
    }
}

void showChar(CharCount::Char charObj)
{
    cout << "char ";
    switch (charObj.ch)
    {
        case '\n':                  // print special case
            cout << "'\\n'";
            break;
        case '\t':
            cout << "'\\t'";
            break;
        default:                    // default case
            showDefaultChar(charObj.ch);
            break;
    }
    cout << ": " << charObj.count << " times\n";
}
```

Listing 15: main.cc

```
#include "main.ih"

using namespace std;

int main()
{
    CharCount charCount;
    charCount.count(cin);           // count chars from stdin
```

```
    CharCount::CharInfo const info = charCount.info();
    for (size_t idx = 0; idx != info.nCharObj; ++idx)
        showChar(info.charPtr[idx]);
}
```

36

Complete the table below, by entering the empty fields of the 'pointers' and 'semantics' columns. The index-expressions should be rewritten without using the index-operator []. In all cases you have to use pointer notations.

definition:	rewrite:
int x[8];	x[2] = x[3];
pointer notation:	*(x + 2) = *(x + 3)
semantics:	x + 2 and x + 3 points to the location of the 2nd and 3rd int beyond x respectively. the elements are both dereferenced so that the content of 3rd int beyond x is assigned to the second int beyond x
char *argv[8];	cout << argv[2];
pointer notation:	cout << *(argv + 2);
semantics:	argv + 2 points to the location of the 2nd char * beyond x, it is dereferenced using '*" and sent to the output stream
int x[8];	&x[10] - &x[3];
pointer notation:	&*(x + 10) - &*(x + 3)
semantics:	x + 10 and x + 3 point to the location of the 10th and 3rd int beyond x respectively. it is dereferenced using '*' to reach the value of the element but then the location is again referenced using the '&' operator. Then the locations are subtracted from eachother to reach &*(x + 10) - &*(x +3) == (x + 10) - (x + 3) == 7
main's argv;	argv++[0];
pointer notation:	*argv++
semantics:	In the given statement, argv is first incremented and then the 0th index is retrieved. This was the 1th index before argv was incremented. This is a result from the memory address that argv holds being incremented by 1. The increment operator has precedence over the deference operator. Hence, we increment argv in pointer notation and then dereference it.
main's argv;	argv[0]++;
pointer notation:	(*argv)++
semantics:	In the given statement, the value at the 0th index is retrieved and then it's incremented by one. This is equivalent to derefencing the memory address and incrementing by one (after using it as it is a postfix).
main's argv;	++argv[0];
pointer notation:	++*argv
semantics:	In the given statement, the index operator has precedence over the prefix increment operator. Hence, the value at the 0th index is retrieved, incremented by 1, and then used. In pointer notation, it is equivalent when there is first an increment operator immediately following the

	derefencing operator and the array name. In that case the dereferencing is done first.
main's argv;	++argv[0][2];
pointer notation:	++*(*argv + 2)
semantics:	In the given statement, again the index operators have precedence. Argv is an array of pointers to pointers. First, the 0th pointer is accessed and dereferenced, after which the third value of the pointed to array is accessed. This is then incremented by one before it is used. In pointer notation this is done by first derefencing the arrays name, obtaining the memory address of the first element that is pointed to by the obtained pointer. Then, two is added obtaining the address of the third element of that character array. Lastly, this is incremented before use.

37

Convert verbal variable and function descriptions to declarations.

1) Declare a variable ptr points to rows of 8 doubles.

```
double (*ptr)[8];
```

2) declare a variable ptr pointing to an array holding the addresses of series of doubles.

```
double **ptr;
```

3) declare a pointer ptr that may point to arrays of immutable pointers to arrays of mutable pointers to immutable std::string objects.

```
std::string const * * const * ptr;
```

4) declare a function fun expecting no parameters, returning a matrix of 6 x 6 std::string objects.

```
std::string (*fun())[6][6];
```

5) declare a function fun as in the previous item, now using StringMat;

```
StringMat fun();
```

6) complete the following function by providing the initialization section of its for-statement (submit the completed function):

```
void process(unsigned begin, unsigned end, char const *const *args)
{
    for
    (
        char const *const *argsBegin = args + begin,
                  *argsEnd = args + end;
        argsBegin != argsEnd;
        ++argsBegin
    )
        process(*argsBegin);
}
```

6) declare a function fun defining a parameter ptr which is an immutable pointer to NTBSs, returning one of these pointers as an immutable NTBS

```
char const *fun(char const *const *ptr);
```

38

In this assignment we argue about the correctness of given code.

What is the design flaw in the code?
Since s_finder is a static member instance of PhoneBook, it will be initialized when PhoneBook is accessed for the first time. But the order in which PhoneBook and Finder are initialized is undefined, as they are in separate source files. Thus initialization of s_finder is not guaranteed when PhoneBook tries to use member functions.
How can this flaw be repaired?
There are several ways to fix this. One way is to make PhoneBook static and making Finder non-static. As PhoneBook is a global instance of which we need only 1 this would make sense to do. With PhoneBook being static and having only one instance, we also have one finder instance.
With finder not being a static member of PhoneBook, initialization of finder is not done when PhoneBook is first accessed, but when the PhoneBook instance is created. This way, it is ensured finder exists before it's member functions are used and thus the order of source files is irrelevant.

39

In this exercise we convert the given class Data by using the pimpl design.

Listing 16: data/data.ih

```
#include "data.h"

#include <iostream>

using namespace std;
```

Listing 17: data/data.h

```
#ifndef INCLUDED_DATA_
#define INCLUDED_DATA_

#include <memory>
#include "../dataImpl/dataImpl.h"

class Data
{
    std::unique_ptr<DataImpl> impl;

    public:
        Data() : impl(std::make_unique<DataImpl>()) {};
        ~Data() = default;
        bool read()
        {
            return impl->read();
        }
        void display() const
        {
            impl->display();
        }
};

#endif
```

Listing 18: dataImpl/dataImpl.h

```
#ifndef INCLUDED_DATAIMPL_
#define INCLUDED_DATAIMPL_

#include <string>

class DataImpl
{
```

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

    public:
        bool read();
        void display() const;
};

#endif
```

Firstly we will create library libdata using the following commands:
g++ -std=c++26 -Wall -c data/display.cc -o data/display.o
g++ -std=c++26 -Wall -c data/read.cc -o data/read.o
ar rcs libdata.a data/*.o
Now we have created our library, which will be linked to main.o after compiling main.cc:
g++ -std=c++26 -Wall -c main.cc
g++ main.o -o main -L. -ldata
When running the program using data.in, the following is displayed:
Object 1: value is: 1
Object 2: value is: 2
Object 3: value is: 3
Object 4: value is: 4

After uncommenting the two lines in data.h and read.cc, and creating a new library using the first three commands again, we then link the newly made library version to main.o using the last command mentioned. When trying to run the program we get a Segmentation fault (core dumped).

Next up we make the two lines comments again, after which we change the class to use the pimpl principle. Class Data is changed into DataImpl, and then the new class Data will have a pointer to a new Data instance as the only data member. If we repeat the process again, now we do not get a Segmentation fault, but the code displays the same as before uncommenting changes are made and the program executes correctly.

40

In 40, we will elaborate on our findings in the previous exercise and go into detail on the pImpl approach.

> Why does the program break after introducing the new library?
> The new library has an implementation of Data with a new data member (and it's use) introduced: d_string. The ABI of Data is changed, while main is compiled with the old library. Because of this, the program breaks when linked to the new library.
> Why does the program not break using pImpl?
> Using pImpl, Data contains only a pointer to a DataImpl instance. This way, the ABI of DataImpl can change (by adding d_string), without the ABI of Data changing. This way, the program still runs after linking to the new library.
> Changing a class made without pImpl:
> We can use a combination of versioning and inheritance to make this work. Firstly, the class will remain unchanged, but we do need to ensure that it has a destructor and versioning. The versioning is done by having a static version() function returning a 1.
> Then a new class can be made with the two extra data members, which inherits the first class. This function also has the same version() function that now returns 2.
> Lastly we need a function that returns the instance of the correct class. It takes a version number and returns a pointer to a new instance of the first class, which is the original class or the new class depending on version.