Homework 1: Fun with C and C++

CS-240: Data Structures: Fall 2015

The goal of this assignment is to provide you with concrete exposure to some C++ programming syntax and concepts that we covered in class, but not necessarily in the Labs. You should work *individually* on this assignment, but you are allowed to use a computer to "try out" any code segments I've asked you to write. In fact, I highly recommend that you do!

1. Pointers [15 pts]

Consider the following code:

```
int i = 123;
int *ip;  int
**ipp;

// Mystery assignment statements here
 cout << "Testing " << **ipp <<
endl;
```

(a) *[5 pts]* Which mystery assignment statements, using only `ip`, `ipp`, and `i`, will result in the following line being printed:

```
Testing 123
```

```
ip = &i;
ipp = &ip;
```

(b) *[10 pts]* Explain very specifically, in your own words, exactly what your mystery statements are doing. You should use the terms "pointer" *and* "address" in your answer.

Let ip become the pointer of integer i, so ip stores the address of i, using &i. Let ipp become the pointer of ip, then ipp stores the address of ip, using &ip to get it. When print out **ipp, get the content stored in address *ipp, which is ip, and the content is i.

2. Addresses, `sizeof()`, the ++ Operator, and Casting [25 pts]

During class, we talked briefly about *casting*, the `sizeof()` function, and the representation of different C++ built-in data types within a running C++ program. On a lab machine (or on a `remote.cs.binghamton.edu` machine), copy the following program into a file called `cast.cpp`. (Remove the line numbers.)

```
 1: #include <iostream>
2: using namespace std;
3:
 4: int main() {
 5:    int i;
 6:    char myCharArray[51] = "      This string right here contains exactly 50 chars. ";
```

```
 7:    double myDoubleArray[4] = {100, 101, 102, 103};
8:
 9:    char *cp, *cbp; 10:
double *dp, *dbp; 11:
12:    dp = &myDoubleArray[0];
13:    dbp = myDoubleArray;
14:    cp = &myCharArray[0];
15:    cbp = myCharArray; 16:
17:    while ((cp - cbp) < sizeof(myCharArray)) {
18:      cp++; dp++;
19:    }
20:    cout << "Without cast: " << (dp - dbp) << endl;
21:    cout << "      Cast 1: " << ((int *) dp - (int *) dbp) << endl;
22:    cout << "      Cast 2: " << ((int) dp - (int) dbp) << endl;
23: }
```

In this program, the two "`(int *)`"'s on Line 21, and the two "`(int)`"'s on Line 22 are called *casts*, and they tell C++ to consider the `dp` and `dbp` variables as pointers to `int`s, and then as `int`s, rather than as whatever type they are declared as (in this case, pointers to `double`s). The "`sizeof()`" call returns the number of bytes used to store the variable or type passed as a parameter to the function. Play around with the `sizeof()` function before answering this question. (Write a small program and pass different types and variables to `sizeof()` to see what it returns.) Make sure you know what sizeof() will return on Line 17.

(a) *[5 pts]* Write a single line that when typed into a shell on a lab machine successfully compiles this program into a file called `cast.exe`. (Not "make"… please run g++ directly.). If your first attempt does not work, read the output from the failed attempt and try again.

g++ -fpermissive -w -o cast.exe cast.cpp

(b) *[5 pts]* Exactly what will this program print?!

Without cast: 51
Cast 1: 102
Cast 2: 408

Assume for parts (c) and (d) that `myCharArray` gets placed at memory address 1000 (decimal), and `myDoubleArray` gets placed at memory address 2000 (also decimal).

(c) *[5 pts]* What are the decimal values of the following variables when the program reaches Line 16? Format your answer as follows:

i: ____0___
dp:___2000____
dbp:___2000____
cp:__1000_____
cbp:_____1000__

(d) *[5 pts]* What are the decimal values of each variable when the program reaches Line 23?

i: _____0__
dp:___2051____

dbp:__2000_____
cp:__1051_____
cbp:__1000_____

(e) *[5 pts]* Explain *why* the program prints what it does. In particular, why do Lines 20, 21, and 22 print different values? (Hint: You should use the word "overloading" in your explanation... but be much more specific than that.)

line 20 stores the actuall address of the double element. There are 51 elements so after the while loop the difference between the dp and dbp will be 51.

line 21 treat the elements as int, so the compiler think (int *) dp stores the address of int element. Since a double takes 2 int's space, so the difference will be doubled.

line 22 converted the address with 8 byte to int which only has 4 byte, Thus I guess the loss precisions caused the difference is much larger.

3. Pass by Value vs. Pass by Reference [15 pts]

Consider the following program:

```
 1:#include <iostream>
2:using namespace std;   3:
 4:int func3(int &three) {
 5:   three = 33333;
 6:   return(three);
7:}  8:
 9:int func2(int &two) {
10:   two = 22222;
11:   return func3(two);
12:} 13:
14:int func1(int &one) {
15:   one = 11111;
16:   return func2(one);
17:} 18:
19:int main() {
20:   int zero = 0;
21:   int rval;
22:   rval = func1(zero);
23:   cout << "rval: " << rval << endl;
24:   cout << "zero: " << zero << endl;
25:}
```

(a) *[5 pts]* What will this program print? Explain *specifically* and in your own words **why** it will print what it does.

rval: 3333
zero: 3333

(b) *[5 pts]* Is there any *single character* in the program that you could change to get the output of the program to include the following line?:

```
zero: 22222
```

If so, what character on what line would you change, and why would it work? If you can't change just one *single character* to produce this output, explain why not.

yes.
int func3(int three){

when calling func3, you get a copy of two, which is 22222, as the argument. Thus the zero itself will not be modified.

(c) *[5 pts]* Is there any *single character* in the program that you could change to get the output of the program to include the following line?:

```
rval: 11111
```

If so, what character on what line would you change, and why would it work? If you can't change just one *single character* to produce this output, explain why not.

```
yes.

int func2(int two){

when calling fun2, you get a copy of one, which is 11111, and whenever you implement next functions, it would not change the zero itself.
```

4. Compiling vs. Linking [10 pts]

The *makefiles* we have been using in class call g++ to *compile* code, and also to *link* it (in separate calls). Explain in your own words, the difference between compiling and linking, in terms of creating and using source (.cpp), object (.o), and executable (.exe) files. You don't have to describe *how* compilers and linkers do what they do, which is certainly beyond what we need to understand for this class, but I do want you to describe *what* they do, in terms of their input and output files. Cite an example that assumes that you have (i) files containing a single class definition and implementation in *MyClass.h* and *MyClass.cpp*, and (ii) a single file called *UseMyClass.cpp*, containing "`main()`" and a declaration of an object of type `MyClass`.

The compiler compile our source code (*.cpp in i and ii) to whatever can be read by the machine and create MyClass.o and UseMyClass.o. When linking these to .o file to one and create an executable thing such as *.exe, this is the one that can be run by users.

5. Static vs. Heap vs. Stack Memory [15 pts]

Consider the following class definition:

```
#define MAX_STUDENTS 70;

class CS240
{    private:
      int numStudents;
```

```
      Student students[MAX_STUDENTS];
public:       CS240();
      CS240(int num_students); };
```

Now consider the following declaration statement, which creates a single instance of the CS240 class.

```
CS240 rocks;
```

(a) *[5 pts]* Write the simplest C++ program you can write, which includes this exact declaration statement and causes the "rocks" object to be placed in *static* memory. If the rocks variable, declared exactly this way, would *never* be placed in static memory, explain why not.

CS 240 rocks;
rock = CS240(49);

(b) *[5 pts]* Write the simplest C++ program you can write, which includes this exact declaration statement and causes the "rocks" object to be placed on the *stack*. If the rocks variable, declared exactly this way, would *never* be placed on the stack, explain why not.
void function (CS240 rocks){

CS240 miao = rocks;

}

(c) *[5 pts]* Write the simplest C++ program you can write, which includes this exact declaration statement and causes the "rocks" object to be placed on the *heap*. If the rocks variable, declared exactly this way, would *never* be placed on the heap, explain why not.

CS240 *rockList;
rockList = new CS240[1000];
rockList[0] = rocks;

6. Arrays and Dynamically Allocated Memory [20 pts]

Consider the following program:

```
#include <iostream> using
namespace std;

void showFloatArray(float f1[10])
{    for (int i=0; i < 10; i++)
cout << "  " << f1[i];     cout <<
endl;
}
float *getFloatArrayOne() {    float
*floatArray = new float[10];    for
(int i=0; i < 10; i++)
floatArray[i] = (float) i;
return(floatArray);
}  float *getFloatArrayTwo()
{    float myFloatArray[10];
float *floatArray = myFloatArray;
for (int i=0; i < 10; i++)
```

```
floatArray[i] = (float) i;
return(floatArray);
}  int
main() {
   float *f1 = getFloatArrayOne();
float *f2 = getFloatArrayTwo();
showFloatArray(f1);
showFloatArray(f2);  }
```

(a) *[10 pts]* When I ran this program, I got the following output:

```
  0  1  2  3  4  5  6  7  8  9
  9  4.59163e-41  5.8808e-39  0  4  4  0.00264489  4.59163e-41  8  9
```

So getFloatArrayOne() seems to work well, and getFloatArrayTwo() does not. But I told you in class that you could access and use dynamically allocated arrays the same way you could access and use statically allocated arrays, because a C++ array is a pointer to the first element in the array. Can you explain the different behavior? (I'm not asking you why the values are exactly what they are, or some even seem to be correct... I just want to know why all 10 are not what I set them to be!)

in line
float *floatArray = myFloatArray;
You actually assigned the array to the pointer, not the floatarray itself. Thus it does not work.

(b) *[10 pts]* There's also a memory leak in this program. Assuming that I can fix it so that getFloatArray() does properly return a pointer to a new floating point array, and assuming I want to delete all the heap memory that I allocated (before the program exits), where should I put a delete statement (or statements) to remove the memory leak?

In function getFloatArray (either one or two), delete floatArray before return it.