Lab#5 – Due Wednesday, February 25, at the start of class

Purpose: To develop familiarity with C++ pointer variables

Introduction: In this lab, you will learn by experimentation the answers to some questions about working with C++ pointer variables. For each of the questions below, you will either write a short miniprogram that tries out what you are experimenting with, or you will run one furnished by the professor. In either case, you will first make a guess as to what the program will do, and will then carefully record your observations as to what it actually did. It is essential that you record your observations on your writeup form carefully and precisely. Do not do this hastily!

For uniformity, we will make most of our observations using pointers to an 8-byte Node, declared as follows. However, what we learn is generally applicable to pointers of any type. (Some of the results, though, will be specific to the system we are using - a different system may give a different result.) The following class definition should be placed inside your header file for lab5...

```
#include <cstring>
#define NODE_SIZE 8
class Node
{
        public:
                Node()
                { }
                // Mutator - set the value to some string
                void setValue(const char * value)
                { strcpy(_value, value); }

                // Accessor for value as a string
                const char * getValue()
                { return _value; }

                // Accessor for ASCII code for ith individual character of the value
                int getCode(int i)
                { return (int) _value[i]; }
        private:
                char _value[ NODE_SIZE ];
};
```

A pointer to such a node can be declared like this:

```
Node * p;
```

In addition to being able to access the contents of the node using the accessors getValue() and getChar(), the C++ IO library supports printing out the value of a pointer (the address it contains). The following code does this - printing the value as an unsigned integer

```
cout << (unsigned) p << endl; // Prints value of p - an address
```

Before Coming to Lab: Study chapter 5 in your textbook

Specific Requirements
Determine, either by writing a suitable mini-program or by running one furnished by the professor, what happens when you do each of the following. On your writeup form, first record your guess as to what you think your answer will be, then run your program and record what actually happened. If you are unsure what is wanted for any question, show your program to the professor before proceeding. Be sure your response on your writeup form answers the question that was asked! Create one program – that contains all of your "mini-programs" and name it *lab5.cpp*, (place in a project called lab5).

Download the lab write up from our class website.

Your program will contain all of the code for your program – simply comment out the lines that are not currently being used – name each section according to the question – Q1, Q2, Q3, etc.

Important: Include an output statement at the end of your program (after you try whatever you are trying) to see whether the program completes normally or terminates due to a crash.

1. What is the value of NULL (what address value does a null pointer contain)?

2. What happens if you try to dereference a pointer whose value is NULL? (Recall that to dereference a pointer is to follow it to the object it points to. You can use either setValue() or getCode() to do this - do not use getValue() for this experiment, because the C++ run-time protects you from seeing what is really happening when a pointer to a character string is null!

3. When you create a new node, what are the initial contents of the node? (Hint: to see this, create a new node and then output the individual characters using getCode(). Do not use getValue(), because not all ASCII codes correspond to a recognizable printing character.) (Note: your observation here is not the final word on this matter. See also question 8.)

4. What is the relationship between the amount of storage requested and the amount of storage actually allocated? (Is there some additional overhead associated with each node?) To find out the actual amount of storage allocated, one can compare the values of the pointers returned by two successive new operations for the same size node - their difference is generally the size of the node that was allocated the first time. For this question, the professor has written a program for you. (To allow for requesting different size nodes, we allocate an array of char directly rather than using the Node class.) This program can be run by /gc/cs212/lab5/lab5_4

   a. Try this program with a variety of request sizes between 1 and 70. (The program will continue to loop – until an input of "-1"is given.) It should be possible, by careful experimentation, to discover the exact rule our system uses for determining how much storage to actually allocate. Keep trying different node sizes until you are able to determine the rule precisely. (Record the request sizes your tried, and the actual allocated sizes you got, on your writeup form, along with the rule your discovered.)

   (Note: if you were to try this on a non-Linux platform, or with Linux on another architecture, or with a different version of Linux you might get a different answer!)

   b. Now try a size of zero. What size node is actually allocated?

   A Listing of this program follows:

```
    // CS212 Lab 5 question 4.
#include <iostream>
static const char ESC = 27;
using namespace std;
nt main(int argc, char *argv[])
{
        char *c1;
        char *c2;
        int requestSize;
        cout << "Size of node to request (To end: type -1)? ";
        cin >> requestSize;
        while (requestSize!=-1)
        {

                c1 = new char[requestSize];
                c2 = new char[requestSize];

                cout << "First address: " << hex << (unsigned) c1;
                cout << endl << "Second address: " << (unsigned) c2;
                cout << endl << "Size actually allocated: " << dec << (c2 - c1);
                cout << endl;
                delete[] c2;
                delete[] c1;
```

```
                cout << endl << "Size of node to request (To end: type -1)? ";
                cin >> requestSize;
        }
}
```

5. What happens if you try to apply delete to a pointer whose value is NULL?

6. Does the implementation of delete change the value of the pointer variable used? (Does the pointer variable contain the same address after delete that it contained before?)

7. If you have a dangling reference to a node that has been deleted, and you dereference this pointer, can you still access the node through this pointer? (Recall that a dangling reference is a pointer that still points to a deleted node.)

   a. Are you able to access the deleted node at all?

   b. If you can access the node, has the value stored in the node been changed? (Hint: try storing a known value in the node before you delete it; then print out the value after delete to see if it has been changed.)

8. If you create a node of a given size and then delete it, and then use new to get a new node of the same size, is the node you just deleted "recycled"? (Do you get back a pointer to the same node that you just deleted?)

   a. Do you get back a recycled node? (Hint: you can determine this by writing out the value of the pointer before you do the delete, and then again after you do the new.)

   b. If so, are the contents of the node cleaned out, or does the new node contain the value that was in the node before it was recycled? (Hint: try storing a known value in the node before you delete it; then print out the value in the newly created node.)

   c. Combining the above with your answer to question 3, what - if anything - can you say about the initial contents of a node created by new?

9. What happens if you try to delete the same node twice (say through two different pointers that both point to the same node)?

   (Note: deleting the same node twice is erroneous and is normally the result of an error in the program. However, different implementations of the standard libraries may do different things in the face to this error. If you were to try this on a non-Linux platform, or with Linux on another architecture, or with a different version of Linux you might get a different answer!)

10. Does deleting the same node twice have any adverse consequences on integrity of the storage management subsystem? For this question, professor Bjork has written a program for you:

```
/* CS212 Lab 5 question 10. Does deleting the same node twice have any
* adverse consequences on the integrity of the storage management
* subsystem?
* Copyright (c) 2001, 2003 - Russell C. Bjork, updated by Steve Brinton (2009)
*
*/

#include <iostream> using namespace std;
 -- Declaration for Node class (same as above) omitted to conserve space
int main(int argc, char * argv [])
{
// Create a node, then delete it twice.
Node * p = new Node();
cout << hex << (unsigned)p << endl;
delete p;
delete p;
// Create two new nodes, and store values into them
Node * q = new Node();
q -> setValue("Node #1");
Node * r = new Node();
r -> setValue("Node #2");
// Write out the node contents and the pointer values
cout << "Contents are: " << q -> getValue() << ", "
                << r -> getValue() << endl;
cout << "Pointers are: " << q << ", " << r << endl;
}
```

The above program can be run by typing /gc/cs212/lab5/lab5_10 into a terminal connected to moses.

What happens when you run it? How do you explain the program's behavior?

(Note: This use to be more exciting a couple of years ago when no error was reported which means you use to be able to “double free” something from the heap)

11. What happens if you try to store a value into a node that is bigger than the size of the node? For this question, place the following code in your program:

```
Node *p = new Node();
Node *q = new Node();
q -> setValue("Node #2");
p ->setValue("This is one long node");

cout << "Node 1: " << p->getValue() << endl;
cout << "Node 2: " << q->getValue() << endl;
```

What happens when you run it? How do explain the program's behavior?

Repeat the observation above and use a node that is size 16...use these statements:

```
q -> setValue("Node #2 is small");
p ->setValue("This is one long node, isn't it?");
```

What happened this time? Does it support your explanation for the previous behavior?

12 . What happens if you write a program that goes into an infinite loop creating new nodes by using new (without ever deleting any nodes)?

a. What happens when the program finally runs out of available memory?

b. About how many nodes of size 8 can you actually create? (Hint: Use a variable to keep a count of nodes created. Print the value of the counter and the value of the pointer only when the counter is divisible by 1,000,000 to prevent creating a lot of output! Even so, you will need to be patient!)

c. About how much memory does it appear your program is allowed to use? (Hint: remember that when you request a node of size 8 bytes, 16 bytes are actually allocated!)

d. Use the free command to print out the amount of memory available on your computer. The output of free will look something like this - where xxxxxxx stands for some actual value that will be printed:

```
                 total       used       free     shared    buffers     cached
Mem:          xxxxxxx    xxxxxxx    xxxxxxx    xxxxxxx    xxxxxxx    xxxxxxx
-/+ buffers/cache:  xxxxxxx    xxxxxxx
Swap:         xxxxxxx    xxxxxxx    xxxxxxx
```

- The values printed are in units of kilobytes (1024 bytes).
- The first row represents physical memory - thus the total shown in this row is the total physical memory available on the machine.

- The final row represents swap space - i.e. virtual memory - so the total shown in this row is the total amount of disk space available for use as an extension of physical memory.

  What relationship is there between the total amount of memory you were able to allocate by repeated new operations and the memory (physical + virtual) available on the system? Where do you think the rest of the memory has gone?

Concluding Essay

After completing the above, each member of the team must individually write a brief essay (1-2 paragraphs) on the following topic:

> In what ways does the behavior of C++ pointers substantiate the claim that C++ gives you just enough rope to shoot yourself in the foot ? In your essay, you should pull together potential pitfalls to the programmer arising from as many as possible of the experiments you did in lab. Where possible, explicitly contrast the behavior of C++ and Java. (But don t manufacture problems - there are enough there without your having to try to hard to find them!)

Note: the differences between C++ and Java are in part a matter of priorities of the language designers. The C++ designers opted for run time efficiency, the Java designers for reducing the risk of programmer error.

Turn In: Place the essay for each team member, the completed write up form, and the source code file into a properly named folder and then place it into the class drop box.