CSci 1113, Spring 2018
Lab Exercise 11 (Week 12): Pointers and Dynamic Memory

The explicit memory management capability of C/C++ is one reason that the language remains popular nearly half a century after its creation. Using dynamic memory requires an understanding of *indirection*: the idea of manipulating a variable (*pointer variable*) that refers to another variable. We say that the pointer variable *points* to the dynamically created memory. In this lab exercise we begin our exploration of dynamic memory management.

Warm-up

Complete the following paper/pencil exercises. First, work on them individually. Then you and your partner should discuss your answers and make any needed corrections. Then both of you should discuss your results with one of your TAs:

1) Pointer basics

a) Declare two (type `double`) pointer variables named `d_var` and `d_array`:

b) Write C++ statements to dynamically create a double-precision floating-point variable and store its address in `d_var.` Also dynamically create an array of 10 double-precision floating-point values and store its address in `d_array`:

c) Write C++ statements to input a value for `d_var` (i.e., a value that `d_var` points to) from the console and then display it:

d) Write C++ statements to initialize the 10 double values in the dynamically allocated array to 1.0 :

e) Now write C++ statements to de-allocate the memory (i.e. using the *delete* operator) pointed by `d_var` and `d_array`:

2) Pointer management

a) Show the output of the following code segment:

```
int a(1);
int b(2);
int *p1, *p2;
p1 = &a;
p2 = &b;
*p1 = *p2;
*p2 = 10;
cout << *p1 << ' ' << b << ' ' << a << endl;
```

b) Describe the problem with this function:

```
void foo()
{  int *array = new int[100];
   for(int i = 0; i < 100; i++)
   {
            //do something
   }
}
```

3) Pointers with classes

a) A user-defined class named `Timer` has a constructor that takes two integer parameters to initialize *hour* and *minute* data members. Write a single C++ statement to create an object of the `Timer` class using *dynamic memory allocation* and assign it to a pointer variable named `timePtr`. It should call the constructor with two parameters. Use values of 10 and 20 for *hour* and *minute* respectively.

b) Write the definition for a function named `randArray` that takes a single integer argument, `n`, and returns a dynamically allocated array of `n` pseudo-random integers. You may assume the pseudo-random number generator has been previously declared and seeded. (i.e. you do not need srand(time(0)) or an include.)

c) Now write C++ statements to call the `randArray` function to construct a 100 element array, then print the array values to the display (one per line) and delete the dynamically allocated array.

Stretch

1) Momentum

Momentum is defined as the product of an item's *mass* and its *velocity*. *Mass* is a scalar value, whereas *velocity* is generally expressed as a vector quantity with three components. The product of the scalar *mass* and the *velocity* yields *momentum* as a vector quantity.

Write a function named `momentum` that will accept as arguments a (i) one-dimensional *velocity* array with three values (type `double`) (i.e. a 3d vector) and (ii) a *mass* (type `double`), and return a *dynamically allocated* array representing the *momentum*. Note the *momentum* is determined by multiplying the scalar *mass* by each element of the vector *array*.

Test your `momentum` function by constructing a short `main` program that will ask the user to input values for the velocity and mass from the console, and then display the momentum.

Workout

1) Arrays of Dynamic Items

Write a program to determine the average momentum of a collection of items with random velocities and masses. Do this using the following outline:

1. Construct a function named `randVec` that will take no arguments and return a dynamically allocated 3-element array of doubles. Each element in the array should be a randomly generated value in the range -100.0 through +100.0.
2. Using `randVec` and your `momentum` function from the previous part, generate *momentum* vectors for 1000 items, each of which has a random *velocity* (as described above) and a randomly generated *mass* in the range 1.0 through 10.0. Save the *momentum* vectors using a suitable array of pointers.
3. Determine and display the average momentum vector of the items using a *for* loop. [Hint: the average should be done component by component.]

2) Letter Frequency

Write a function that will take a string and return a count of each letter in the string. For example, "my dog ate my homework" contains 3 m's, 3 o's, 2 e's, 2 y's and one each of d, g, a, t, h, w, r and k.

Your function should take a single string argument and return a *dynamically allocated* array of 26 integers representing the count of each of the letters a . . z respectively. Your function should be case *insensitive*, i.e., count 'A' and 'a' as the occurrence of the letter a. [Hint: use the letter to integer conversion functions.] Do not count non-letter characters (i.e., spaces, punctuation, digits, etc.)

Write a program that will solicit a string from the user using `getline`, call your letter frequency function and print out the frequency of each letter in the string. Do not list letters that do not occur at least once.

Example:

```
Enter a string: my dog at my homework
Letter frequency
  a  1
  d  1
  e  1
  g  1
  h  1
  k  1
  m  3
  o  3
  r  1
  t  1
  w  1
  y  2
```

Challenge

1) Buffering

Real-world data from measuring devices or sensors is generally produced *asynchronously* relative to the processing routine(s). The data may appear in bursts that occur faster than can be individually processed. An effective way to handle this situation is to employ a *buffer*. A *buffer* is simply an array into which a data-*generating* process writes data, and from which a separate data-*processing* routine subsequently retrieves it. In the short-term, if the data is being generated at a faster rate than can be processed, the buffer fills while the processing routine catches up. In the long-term (assuming the average processing rate exceeds the average generation rate), the data processing will be able to remain ahead of or keep pace with the generating process.

There are a number of ways to implement buffering. One simple way is to employ a *double-buffering* technique in which a generating process fills one buffer, then provides the address of the filled buffer to the processing routine and allocates a new buffer to begin filling. The processing routine extracts the data from the filled buffer and disposes the buffer when finished, then awaits the next "filled" buffer.

In this problem, we'll simulate a basic *double-buffering* scheme using dynamically allocated arrays of integers. You will need to construct a data *generation* function and a data *processing* function and then randomly call them from a simulation loop in the main program.

1) Construct a function: `double getProb()` that will return a pseudo-random value between 0.0 and 1.0.

2) Construct a function: `int* generateData(int* &inbuf, int &count)` that will simulate asynchronous data generation by obtaining a random number between 0 and 9 (you do not need to use getProb() for this) and saving it in an input buffer. Your function should do the following:

- Generate a random integer between 0 and 9 and add it to the buffer at the array location specified by the count argument. (Note the buffer is passed as a pointer variable reference.)
- Increase the count.
- If the buffer is full: a) return the address of the full buffer, b) reset the count to zero, c) allocate a new buffer and d) save its address in the pointer variable passed as the first argument.
- If the buffer is not full, then return NULL.

3) Construct a function: `void processData(int* &outbuf, int &count, int &total)` to simulate "processing" of the asynchronous data. Do the following:
- If the output buffer pointer is NULL, do nothing, i.e., just return.
- Otherwise: obtain the buffer value at [`count`] and add it to `total`, then increase the count.
- If all elements in the buffer have been exhausted, then reset the count to zero, delete the (dynamically allocated) buffer and set the buffer pointer argument to NULL.

4) Finally, include the following code in a main simulation program that will call the `generateData` and `ProcessData` functions in a loop.

```
const int BUFSIZE=10;
const int ITERATIONS=50;

int main()
{  int *fillbuffer = new int[BUFSIZE];
   int fillcnt=0;
   int *processbuffer = NULL;
   int processcnt=0;
   int tcount = 0;

   for(int i=0; i<ITERATIONS; i++)
   {  int *temp;
      if(getProb() <= 0.40 )
      {  temp = generateData(fillbuffer,fillcnt);
         if( temp != NULL )
            processbuffer = temp;
      }
      if(getProb() <= 0.60)
         processData(processbuffer,processcnt,tcount);
      cout << fillcnt << '\t' << processcnt << endl;
   }

   cout << "Total value: " << tcount << endl;
   return 0;
}
```