CSE 109: Systems Programming

Fall 2018

Program 2: **Due on Sunday, September 23rd at 9pm on CourseSite.**

Checkpoint Due: Tuesday, September 18th at 9pm, via Checkpoint submissions.

Collaboration Reminder:

1. You must submit your own work.
2. In particular, you may not:
   (a) Show your code to any of your classmates
   (b) Look at or copy anyone else's code
   (c) Copy material found on the internet
   (d) Work together on an assignment

Assignment: Background

```
1  int value = 5;
2  int *valuePtr = &value;
3  fprintf(stdout, "%p: %d\n", &value, value);
4  fprintf(stdout, "%p: %d\n", &valuePtr, valuePtr);
5
6  value = 10;
7  fprintf(stdout, "%p: %d\n", &value, value);
8  *valuePtr = 20;
9  fprintf(stdout, "%p: %d\n", &value, value);
10 *(valuePtr + 0) = 20;
11 fprintf(stdout, "%p: %d\n", &value, value);
12 valuePtr[0] = 40;
13 fprintf(stdout, "%p: %d\n", &value, value);
```

Here we show an *int* and an *integer pointer* that points to the aforementioned *int*. This output lets us see where these are stored in memory as well as whatever values they may have.

It then shows 4 different ways to change the *int*. All four are equivalent to each other.

It is also important to understand that pointers can be used with array notation. In fact, the pointer does not know if it refers to one thing or to the first of many things - this is why *main* receives *argc* as a function argument in addition to *argv* so we know how many things *argv* is actually referring to.

```
1  valuePtr = (int *)malloc(10 * sizeof(int));
2  valuePtr[0] = 10;
3  *valuePtr = 20;
4  *(valuePtr + 0) = 30;
5
6  valuePtr[4] = 10;
7  *(valuePtr + 4) = 30;
8
9  free(valuePtr);
10 valuePtr = NULL;
```

This code (as a continuation of the prior code) reassigns the value of *valuePtr*. Firstly, now *valuePtr* no longer knows the address of *value* and secondly, *valuePtr* is now pointing to newly allocated space we can use. *valuePtr* has been given a value that refers to a memory address that holds ten integers, the result of the malloc, or memory allocate, function.

The code then shows three ways to change the first location that *valuePtr* points to. Two of those methods can be used to change the other locations that are accessible from *valuePtr*. Note that nothing stops us from going out-of-bounds or even going backwards. It is our responsibility to use the data correctly or to write code that makes sure we use it correctly.

When we dereference a pointer, we use its type to tell us what to look at and its value to tell us where to go in memory. Dereferencing a pointer gives us what it directly points to. If the pointer actually refers to more than one thing, we have to use array notation (indexing) or pointer arithmetic. Both are shown in the example. (Side note: Pointer arithmetic is very slightly different than addition/subtraction, thinking of it in array notation may be easier at this point).

It is possible for us to access something such as *valuePtr[20]*. However, this may crash with a *segmentation fault* or give you some unexpected data as a result.

In the above code, take a look at *valuePtr*, both the value it contains and where it lives. Add some more *malloc*s and get an idea of what is going on. *malloc* gives us *dynamic memory*, this is in contrast to what we get when we make arrays otherwise (they'll be declared on the *stack* instead).

```
1 int array[10];
2 int i = 0;
3 for(i = 0; i < 10; i++)
4 {
5     array[i] = 5 + i;
6 }
7 fprintf(stdout, "%p %p %p %p\n", array, &array, array[0], &array[0]);
```

The output of the third snippet of code may not quite be what you would expect. This is a nuance of array type behavior.

An array type is "mostly" a pointer, but it isn't dynamically allocated.

```
1 int *arrayPtr = &array;       // Other options possible
```

Notice that *array* and *&array* are equivalent, this is an exception to the rule of the addressing/referencing operator.

There is also some weirdness with determining the size of the array. Since an array is "mostly" a pointer, we would expect that applying the *sizeof* function to it would tell us the size of a pointer. However, it does not. *sizeof(array)* is this case will give us 40. This often leads programmers into thinking that they can use *sizeof* to determine the size of "non-static" arrays, this is not the case. The *sizeof* function should not be used with names and should only be used the types, such as *sizeof(int)*, other usage is prone to mistakes (an exception would be with the usage of C++'s auto type, but that's a different situation).

Stack vs Heap: Dynamic Allocation

Whenever we declare variables they only exist on the stack and therefore have a limited lifetime (based on the closing braces in our source code). The data contained in those variables can not be guaranteed to persist. Globals, which we won't allow, work because of the brace rule, they do not die until the appropriate closing right brace occurs, which is never.

The only way to have persistent data is to use dynamic memory via *malloc* and its related functions. In Java, non-primitive types use *new* in place of *malloc*. Remember that non-primitive types in Java are essentially pointers (which are on the stack and limited in lifetime) but with less control over how they can be used.

We never have *name* access to dynamic memory. I mean, *int x = 5* gives us memory on the stack that we have a name for. There can be other

names that refer to the same value of 5 but x is the designation we initially provided. If I change x, that 5 will also change.

When we create dynamic memory, we create a pointer on our stack that has a value that is the location of some dynamic memory that we can use. We only have the location and the permission to use it. If we were to reassign the pointer, the dynamic memory would not be affected - the pointer would just change. Dereferencing allows us to use this information to make a change in our dynamic space (also called the heap).

Functions:

When we provide arguments to a function, our arguments are copied and provided to the desired function. This is called pass-by-value (We will see others in Programming Languages and we will see pass-by-reference in C++). Because only copies of the original data were provided, the original arguments can not be modified by the function. Since Java is actually passing disguised pointers, it appears that we are changing our arguments in Java but this is not the case.

This doesn't mean we can't manipulate data. Pointers are our friend.

```
1 void function(int first, int *second)
2 {
3     first = 10;
4     *second = 20;
5 }
6
7 int x = 1, y = 2;
8 function(x, &y);
```

After *function* is called, x will be 1 and y will be 20. Neither value that was provided to *function* had a durable change - that is, no change to *first* nor *second* survived the termination of *function*. y will be changed but y was not given to *function*, only its address was given. Use of the address allowed us to invoke a change but *&y*, which was passed, did not change.

We can give a function access to any of our memory, whether it be on the stack or dynamic, by passing pointers. In this assignment, we will see that we will store pointers in dynamic memory for use later on.

Functions can not return access to memory that was on the stack (for that function). They can only return strict values (C++ will allow us to return references which will be confusing later on). This means that whatever the return type of the function, we take the function's return value, copy it

into an object/variable that matches the return type and give it to whoever called the function. The caller does not get the exact data the function returned, it gets a copy of it - same idea as when we pass arguments to a function. Since pointers are just values, it doesn't matter if we copy them. The act of dereferencing does not care whether the value is the original or a copy, it just cares that the value is identical.

Resizing:

Array types can not be resized - even though they are essentially const pointers (formally, they *decay to pointers*). They can not be reassigned to a new array either. The following code is invalid:

```
1 int array[10];
2 int array2[20];
3 array = array2;
```

However, pointers can be used like arrays and they can be reassigned to point to a different location. The amount of space available at that location can not be resized (we can't change the size of what we were given) but we can create a new allocation of the desired size, copy data from the old allocation to the new allocation, reassign the pointer to the new allocation and return the old allocation back to the heap (*free*).

There is a *realloc* function that 'reallocates' memory but it can not guarantee success and has to be used carefully. In this class, we will not use that function, instead, we will use *malloc*, *free* and copying data semantics.

```
1 int array[10];
2 int *arrayPtr = (int *)malloc(10 * sizeof(int));
```

In the example above, *array* can not be changed. *arrayPtr* can always be reassigned to the result of a different *malloc*.

Style:

In class, we follow the Allman style of braces and indentation.

1. **Review the Style document on Coursesite**

Assignment: Preparation

1. Make a *Prog2* directory in your class folder.

   Make sure to chmod 700 your *cse109* directory if you haven't already.

2. All source code files must have the comment block as shown at the end of this document. All files must be contained in your *Prog2* directory.

Assignment

1. Create the file *prog2.c*, this is where source code will go for this section.

2. This program will take in text as input and will not prompt the user in any way (you can use stderr if you want).

   It will also take a command line argument, X. X must be between 20 and 100000, inclusive. If it is not provided, assume it is 100. If the command line argument is invalid, print an error message to standard error and return or exit 1 to end the program.

   X will be used for the size of *buckets*.

3. For each word entered by the user, you will store it in memory (specifically, in a *bucket*) and when they are done you will print out each word - the method will be explained further down.

   If a word entered already exists, you will remove it from the *bucket*, see below.

   When using redirected input, EOF will occur to tell you that the user is done. To simulate this without redirected input you can use Ctrl+D. You will have to look into how to handle your input when there is nothing left.

4. Normally, in order to track each of the words, they would be stored using a *char ***. Each *char ** is a word that would be allocated memory from *malloc* and the overall *char *** would expand to hold more words as needed.

   You **will not** directly *malloc* for each individual word, the words will be placed in the *buckets*.

   You may use a malloc temporarily to hold the word while transferring it to the bucket. Make sure to deallocate it.

5. For this assignment, you will put the words into *buckets* you create. As well as handle removals.

6. Make sure not to leak memory, use *valgrind*!

Bucket

By storing each of the words into a *bucket*, the strings themselves will be stored more compactly than if we were to do a *malloc* for each string.

(a) Specifically, we will *malloc* a *bucket* of size X, the command line argument. These *bucket* will be *char **.

(b) We will maintain a group of these *buckets*. Therefore, we need a *char ***.

(c) We need to keep track of how many *buckets* we have.

(d) We need to keep track of how much space each *bucket* has left.

(e) When we want space for a word, we scan through our *bucket group* in the order each *bucket* was created - don't try to optimize the storage here, it'll break the output later. If we find a *bucket* that has enough space for the word and its associated null character, we then store the word in the *bucket* in the next available location. **Except** if the amount of remaining space available would be exactly 1 (since it would become wasted space). Finally, we update how much space the *bucket* has left and return a pointer that refers to the newly stored word.

Don't try to optimize the storage within the *buckets*, return the first available location within the *bucket*.

(f) If we can not find a spot for the word, allocate a new *bucket*, track it and provide that *bucket* for storage.

If it is impossible to fit a word (with its null character), ignore that word and move on. Do not track it, nor place it in any bucket. If you just need to make another *bucket*, then do so.

(g) If the word already exists in a *bucket*, remove the word from the *bucket*. When doing so, shift any words afterwards to overwrite the space created by the removed word. Then adjust any size tracking of the *bucket*.

If it is repeated again, you would add it since it doesn't exist.

Output

(a) When outputting, print each *bucket* on their own line.

(b) Print *bucket*s in the order they were allocated.

(c) When printing a *bucket*, the first thing to print is an integer (width 6) of how much space has been used in the *bucket* followed by ": ".

Consider the null characters to be used space - this includes the trailing null.

It is possible for a *bucket* to end up empty, in that case, you would print no words but retain the leading space from ": ".

(d) Then, print each word stored in the *bucket*, separated by spaces. Do not print any leading space.

It is possible that this order is the same as the order of the input but will usually be different.

Example: Large Malloc

Let us say the second part is executed with a command line argument of 20 and the input is the following five words: "apple", "crunchy", "potato", "sandwich" and "puff". Let "_" denote the null character.

1. We request 5 bytes for "apple". We look for 6 bytes. There is no room, since we have no *buckets*. Create a *bucket* of size 20. We have 1 *bucket*. *Bucket* 1 has 20 bytes left. Fulfill request by providing *Bucket1[0]*, *Bucket1* has 14 bytes left. *Bucket1* contains "apple_"

2. We request 7 bytes for "crunchy". We look for 8 bytes. We look through the *buckets*. *Bucket1* has 14 bytes left. Provide *Bucket1*[6] and now has 6 bytes left. The *bucket* now contains "apple_crunchy_"

3. We request 6 bytes for "potato". We look for 7 bytes. We look through its *buckets*. *Bucket1* has 6 bytes left, not sufficient. There are no other *buckets*. Allocate a new *bucket*, *bucket2*. We now have 2 *buckets*. *Bucket2* has 20 bytes left. Provide *Bucket2[0]*, and now has 13 bytes left. *Bucket1* contains "apple_crunchy_" and *Bucket2* contains "potato_"

4. We request 8 bytes for "sandwich". We look for 9 bytes. We look through the *bucket*. *Bucket1* has 6 bytes left, not sufficient. *Bucket2* has 13 bytes left, use this *bucket*. Provide *Bucket2[7]*, and now has 4 bytes left. *Bucket1* contains "apple_crunchy_" and *Bucket2* contains "potato_sandwich_"

5. We request 4 bytes for "puff". We look for 5 bytes. We look through the *buckets*. *Bucket1* has 6 bytes left, if we provided *Bucket1[14]*, we would end up wasting 1 byte. Therefore, we look at *Bucket2*. *Bucket2* has 4 bytes left, which is not enough. We allocate a new *bucket*, *Bucket3*. We now have 3 *buckets*. *Bucket3* has 20 bytes left. Provide *Bucket3[0]*, and we now have 15 bytes left. *Bucket1* contains "apple_crunchy_", *Bucket2* contains "potato_sandwich_" and *Bucket3* contains "puff ".

The output of this example would be as follows (underscores used to show width - DO NOT PRINT THE UNDERSCORES):

```
____14: apple crunchy
____16: potato sandwich
_____5: puff
```

Checkpointing

1. Each of the following behaviors must be done via a function that is called by *main*. These functions must be prototyped above *main* and be defined after *main*.

2. size_t removeDuplicate(char* line, size_t length, char* duplicate, size_t dupSize) => Removes a duplicate word from the line. Sizes of both are specified as inputs. Returns the resulting size of the line. If the word is not found on the line, no changes are made.

3. size_t* addSize_tElement(size_t* list, size_t* size, size_t* capacity, size_t toAdd) => Adds *toAdd* to the given list at the end of the list and if necessary: creates a new list of a larger size, copying the old list into the new one, freeing the old list, placing the new element and then returning the new list. The original list is destroyed (or should be considered destroyed), the caller must use the returned list instead.

   This function will change *size and may change *capacity.

   When necessary, you should expand the list by (2*capacity)+1.

4. Both functions are due at the Checkpoint time.

5. These functions can call other functions, as long as everything is included in your checkpoint submission.

6. To submit your Checkpoint, copy your *prog2.c* to *checkpoint2.c*, remove the main function from the code. Then you can enter the following command:

   ```
   ~jloew/CSE109/submitCheckpoint.pl 2
   ```

   That is lowercase PL, followed by the number 2.

7. You may submit your Checkpoint a total of **ten** times. This does not reset after the checkpoint is due.

8. Ideally, it will tell you which functions are incorrect. It is possible that the functions are incorrect but pass the checkpoint. Although, they should be mostly correct or completely correct if they pass the checkpoint.

9. The checkpoint will not check for memory corruption or leaks. You will need to handle that yourself.

Testing

1. Make sure your code does not leak memory by running it with Valgrind.

2. Some test cases (*.in files) will be provided in the directory:

   ∼jloew/CSE109/prog2student/

3. A working executable may also be in that directory.

4. The working executable is built to the expected specifications (unless we screwed up). Any confusion in the requirements can potentially be tested by using the working executable.

5. You can generate output files from the working executable and use *diff* to compare them to the results that your code generates.

6. Warnings constitute errors, they are there for a reason, fix them!

7. Valgrind can tell you what lines of code that some errors (the ones it detects) come from. You may find that useful as well.

Submission:

1. Once ready to submit, you can package up the assignment as a .tgz file

   tar -czvf Prog2.tgz Prog2

   You must use this command in the directory that contains the *Prog2* folder, not within the directory.

2. Transfer *Prog2.tgz* to the Program 2 submission area of CourseSite.

Comment Block:

```
/*
    CSE 109: Fall 2018
    <Your Name>
    <Your user id (Email ID)>
    <Program Description>
    Program #2
*/
```