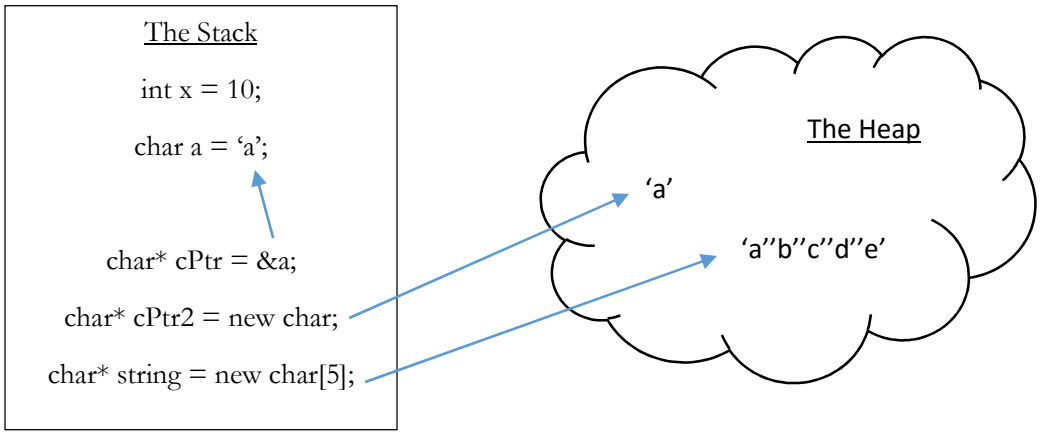
Programming Notes 11: Dynamic Memory

As usual, more examples of the notes are in the example program.

1. The stack vs. the Heap

So far, all the data you've used in your programs is stored in the stack, which is the default storage area for program data. However, the stack is rather small and won't allow you to hold much data. Using dynamic memory, you can take advantage of the heap, which is a much, much larger portion of your computer's memory. Essentially, the heap can store as much data as there is memory in your computer, although how the operating system manages it is somewhat more complicated. To use the heap, you can't directly store values in it, so you must access it only through pointers, as seen in the diagram below.



2. Dynamic Memory

So, as suggested by the previous diagram, you use dynamic memory through the reserved word "new." "New" will essentially look through your computer's memory to find a place which enough space to hold whatever data you specify, will specify that your program can use it, and will finally return a pointer to that memory. This is called memory allocation. To specify the amount of memory you want to allocate, you specify a datatype after the keyword new. This tells your program how many bytes of memory it needs to allocate, and also tells it what kind of pointer it should return. Assuming your computer has not run out of memory, you will always get back a valid pointer, but as usual, the actual value your pointer points to will be garbage. Example:

int* dynamInt = new int;

Creates an integer on the heap, which dynamInt points to

char* dynamChar = new char;

Creates a character on the heap, which dynamChar points to

Now, if you could only allocate single values, the heap wouldn't be very useful (at least under what we've learned so far)—you can allocate an array of the specified datatype simply by adding brackets with the size of the array after the datatype. This will give you a pointer to the start of the array. Remember how pointers can work in almost exactly the same way as arrays? This is where that is so useful, as your pointer is the array, you have no other way to access it. This makes it particularly important for you to mind how you move the pointer around. For example, if you iterate through the array by moving your pointer, you must have some way to get back to the beginning, for reasons discussed in the next section. Example:

```
char* dynamString = new char[50];
```

Creates a c-string of length 50 (50 chars) on the heap, which dynamString points to.

3. Memory Leaks

As mentioned previously, if you move a pointer to dynamic memory away from the memory, you have no way of getting back to it. And if you do this, the memory doesn't simply go away—it's still out there, somewhere in memory, but you have no way to get to it. This is called a memory leak, and is a bug. Memory leaks are especially bad if you are allocating leaked memory multiple times, as the amount of memory leaked goes up and up until either your program is shut down or your computer runs out of memory and freezes until you can shut down the program manually.

To make sure you don't leak memory, you must always be wary of what you're doing with your pointers as not to lose any dynamic memory, and always delete your memory when you are done using it. One of the most common sources of memory leakage is if you allocate some in a function, but never delete it. Because the pointer to the memory only exists in the function, the pointer will be deleted when the function ends, and the memory will be lost. Then, if the function is called again, it will create more memory and lose that, and on and on. However, there is a solution! To delete dynamically allocated memory, you simply use the keyword "delete." "Delete" will deallocate the memory that your pointer points to, making it available to other programs on your computer. However, there are several caveats you must keep in mind when deleting memory. First, if you call delete on a pointer that does not point to dynamic memory, you will get a segmentation fault. Second, once you delete the memory, it does not actually change where the pointer points—but now it will be pointing to some random place in memory and will also cause a segmentation fault if you use it. Because of this, you should always set the pointer to "NULL" (or zero) after you delete it. This is useful because later in your program, you can test if it is valid or not simply by checking that it is not Finally, to delete dynamically allocated arrays, you simply add empty brackets directly onto the end of the delete keyword ("delete[]"). This give the third caveat—the pointer must be pointing at the beginning of the array, or else your program will only delete a portion of the memory and the part not deleted will be leaked. Examples:

```
void func() {
  char* dynamStr = new char[50];

  //Do processing

  //Without this line, the string memory would be leaked
  delete[] dynamStr;
```

```
    //While this is technically not necessary, as the pointer will be gone when the function ends, but
    is a good habit to get into
    dynamStr = NULL;
}
```

4. Why "Dynamic"? Also, exactly sized c-strings.

You may be wondering why this memory is called "dynamic" memory, as there isn't really anything dynamic about it. Well, it's because, unlike arrays, in which you must specify a constant size for them at compile time (before your program is run), dynamic memory can be allocated with variable values. This means that you can make arrays of some size that's specified at runtime, or when your program is run. For example, you could ask the user how many values they want to create, and create exactly that many, rather than having to allocate some array of a maximum size and likely wasting a ton of memory that simply isn't needed. Another example is using exactly sized c-strings (meaning you only allocate the exact number of characters in the string). Of course, if you're inputting a c-string, you don't know beforehand how many characters you will be receiving, so you still have to first input the text into a maximum sized buffer, and then copy it into an exactly sized dynamic string. To do this, you need a string length function, which will take in a character pointer (to a string) and return the length of the string. However, the length should not include the null character that marks the end of the string. Finally, you also need a string copy function which will copy each character from one c-string to another. Example:

```
cout << "How many numbers will you input: ";
cin >> numVals;

int* values = new int[numVals];

for(int i = 0; i < numVals; i++)
  cin >> values[i];
```

OR

```
cout << "Enter a word: ";
cin >> buffer;

char* exactStr = new char[strlen(buffer) + 1];
strcpy(exactStr,buffer);

cout << "Exactly sized word: " << exactStr << endl;
```

5. Dynamic Memory and Functions

Finally, I've previously mentioned allocating dynamic memory in functions, and said you must always remember to delete your memory before the end of the function. Well, that's not strictly true. The golden rule of preventing memory leaks is to always to delete your memory before you lose access to it, and not everything in a function is lost when it ends. I'm speaking, of course, of returned values and reference parameters. A function can return a pointer to dynamic memory just like any other pointer, and in that case it would not make sense to delete the memory within the

function, as the returned value would be useless. Reference parameters, which would be pointers passed by reference, can work the same way—if you are setting a reference parameter to point to a new bit of dynamic memory, it can still be accessed elsewhere and most likely should not be deleted within the function. However, again, always be wary of moving around dynamic memory pointers—if a pointer to dynamic memory is passed to a function, that function can delete the memory, and if it does not set it to some other memory (which means the pointer must be passed by reference), the pointer will no longer be valid in main, and you might end up with a segmentation fault. Example:

```
char* allocateString(int size) {
  char* str = new char[size];
  return str;
}
```

The dynamic memory should not be deallocated within this function, as it is returned, preserving a way to access it so it can be deleted elsewhere.

```
void resizeStr(char*& dest, int size) {
  delete[] dest;
  dest = new char[size];
}
```

This function will delete the memory pointed to by "dest," and reallocates it with the new size. It should not be deleted because as it is a reference parameter, if will be pointing to the new memory in main, and can be deleted later.