Memory & C

CS61C Spring 2017

Contents

1 What is memory?

In this class, when we say memory we are referring to dynamic random-access memory (DRAM), which are large blocks of electronic cells, with each cell representing either a zero or one depending on the voltage across the cell's capacitor. Memory is where programs store data. When you create a list in Python or a HashMap in Java, memory is where you store these variables!

More importantly for this class, memory is structured as bytes (units of 8 bits) of data. Each byte is labeled or addressed with a number from 0 to N, where N is just the number of bytes of total memory your machine has. In this class, we will generally assume our machines to have 2^{32} bytes of memory, or $N = 2^{32}$ (for reasons explained in later units). This addressing is called *byte-addressing*, where each byte of data can be found at a specific address. The data stored in memory can be of many different data types, such as a `char`, an `int`, or a pointer:

- `char` represented by 1 byte of data, it has $2^8 = 256$ different values. We convert the byte's numerical value (0 to 255) to a character literal (e.g. 'A', 'B', 'C'), using the ASCII table.
- `int` represented by 4 bytes of data, the value of the integer is the 32 bit number (in 2's complement) corresponding to the 4 bytes.
- *pointer* represented by 4 bytes of data, this data type specifically stores addresses. The 32 bit number corresponding to the 4 bytes is interpreted as an unsigned number (from 0 to $2^{32} - 1$) that is the address for (i.e. points to) a specific byte of data in memory.

For example, let's say we had the following 4 bytes of data: `0x87617661`. If we interpret these 4 bytes, as 4 char's then we would have 'ç', 'a', 'v', 'a'. If we interpret them as an integer, then we would have the signed 32 bit number: `0b 1000 0111 0110 0001 0111 0110 0110 0001` = -2,023,655,839. If we interpret them as a pointer, then these 4 bytes are the address of byte number 2,271,311,457.

In the diagram below, the memory addresses of the bytes are on the left hand side, while the actual values of those bytes are in the table. The offsets at the top simply indicate that the exact address of that byte is the memory address on the left + the offset matching the byte's column. For example, the byte in the 0th row, 3rd column of the table is at address `0x7FFFFFFC + 3 = 0x7FFFFFFF` (you're going to get really good at hex in this class), with value 0x04.

	x0	x1	x2	x3	
0x7FFFFFFC	00	00	00	04	int x = 4;
0x7FFFFFF8	FF	FF	FF	FF	int y = -1;
0x7FFFFFF4	7F	FF	FF	FC	int* p = 0x7FFFFFFC;
0x7FFFFFF0	00	AB	CD	EF	garbage
0x7FFFFFEC	'6'	'1'	'C'	'!'	(string literal) "ILUV61C!"
0x7FFFFFE8	'I'	'L'	'U'	'V'	Note: in memory, each character is actually a numerical value
0x7FFFFFE4	7F	FF	FF	E8	char* s = 0x7FFFFFE8;
	...				

Figure 1: An example memory contents.

2 The Layout of Memory

Now that we know what memory looks like, we can see how memory is actually laid out or organized for a program. The 2^{32} bytes of memory are divided into 4 main sections: the stack, the heap, the data segment, and the code segment. An example memory layout is shown in Figure 4.

2.1 The Code Segment

This is where the actual code that comprises the program is stored. Your computer will actually know how to execute your program by reading the data at this segment of memory. Like all data in memory, your program is stored as just bytes of data. Later in this class, you'll learn how your program is compiled into assembly language, assembled into binary, and then loaded into this segment.

2.2 The Data Segment

This is where statically declared data is stored. By static, we mean variables/data that are declared outside of your program's main function and that are accessible to all methods/scopes of your program. The data here are often constants that are shared by all parts of your program (although like in Java, static does not mean constant).

2.3 The Stack

This is where local variables are stored. By local, we mean variables that are declared within the scope of a function. Since these variables are only necessary for a function call, the data in this segment is not preserved after the function call finishes. This means that after a function call ends, the memory allocated for these local variables will be reclaimed and the corresponding data can (and likely will) be overwritten in later function calls. For example, when you change the value of a local variable inside a function, that change in value only persists within the scope of the function.

Additionally, unlike the other segments, the stack grows downwards, from higher addresses to lower addresses, meaning that new data is stored at lesser addresses than previously stored data. For example, in Program I and Figure 4, `x` was declared before `s_static`, but `x` is stored at the higher address `0xFFFFFFFC` while `y` would be stored at the lower address `0xFFFFFFF8`.

2.4 The Heap

This is where dynamically allocated data is stored. By dynamically allocated, we generally refer to variables/memory that is allocated using `malloc()`. Unlike data in the stack, data in the heap will persist after a function call ends and is not automatically reclaimed. Therefore, programs must explicitly use `free()` to free the memory in the heap when that data is no longer needed.

3 Important Data Types and Concepts in C

Unlike many languages, C allows programmers to interact directly with memory. This isn't a C class, but the language is very useful to know while learning memory (and many other concepts later in this class). There are many data types in C, but some are particularly common.

3.1 Pointers

A pointer is just a memory address where data of a specific data type is stored. In C, a pointer is declared by: `{data type} *{pointer name}`, where data type is the data type of the data at the pointer's address, and pointer name is just the name of the pointer variable. For example, `int* x` would declare a pointer to an int, such that the value of `x` is the address of some integer. Additionally, `int* x` is the same as `int *x`, but the latter is used more often for clarity [1]. `NULL` is commonly used as the default value of a pointer when the pointer's actual values has not yet been set.

Pointers are useful for passing *references* to data in between functions. Since in C, arguments are passed by value, if you were to pass an array of 1,000 elements to a function, then that entire array (of thousands of bytes) would be copied into a parameter in the function call (in stack memory). Whereas, if you were to pass a pointer to that array instead to the function, only the pointer (of just 4 bytes) would be copied into a parameter. Passing a pointer into a function is similar to pass by reference in other languages, like Java and Python.

There are also two operators regularly used with pointers, the `&` and `*` operators. The `&` operator takes the address of a given variable. For example, following Program II in Figure 3, `y` would evaluate to 4 but `&y` would evaluate to the memory address where `y` is stored.

The `*` or dereference operator can be thought of as the inverse of `&`. Given a pointer or memory address, `*` will dereference that pointer by going to the corresponding address in memory and returning the actual value stored at that address. For example, following Program II, `y_addr` would contain some address, say `0xbfffffec`, while `*y_addr` would return 4 (since the value in memory stored at `0xbfffffec` is 4). `*y` would error out, as `y` is an `int` not a `int*`.

Also, when using the `*` operator, make sure that the pointer you are attempting to dereference is a valid address (i.e. the address of an existing variable or allocated stack/heap/static memory). Dereferencing an invalid pointer, either with value `NULL` or an address to unmapped/unallocated memory, will result in a *segmentation fault*.

3.2 Pointer Arithmetic

In C, integers can be added and subtracted from pointers in order to move a number of bytes from a given address. Specifically, `{pointer of data type} + k`, would return `{the address stored in the pointer} + {k * sizeof(data type) bytes}`.

For example, following Program II, `y_addr + 2` would take the address stored in `y_addr`, `0xbfffffec`, and then add 2 * `sizeof(int)` = 2 * 4 bytes = 8 bytes to that address, to ultimately return `0xbfffffec + 8 = 0xbffffff4`.

[1] If you needed to declare two pointer variables, you may write `int* p1, p2;`. However, this will actually evaluate to `int* p1; int p2;`. Instead, if you write `int *p1, *p2` you will get the intended behavior `int* p1; int* p2;`

```
int z = -1;
int main(int argc, char* argv) {
  int x = 3 * sizeof(int);
  char* s_static = "61C";
  char s_stack[4];
  s_stack = "61B".

  int *heap_arr = malloc(x);
  heap_arr[0] = 1;
  heap_arr[1] = 2;
  heap_arr[2] = 4;
}
```

Figure 2: Program I

```
struct bear {
  char* name;
  struct bear* buddy;
};

int main(int argc, char* argv) {
  int y = 4;
  int *y_addr = &y;

  int arr[4];
  arr[0] = 1;
  arr[1] = 2;
  arr[2] = 3;
  arr[3] = 4;
  int* a = &(arr[0]);

  int n = sizeof(struct bear);
  struct bear stack_bear;
  struct bear* heap_bear = malloc(n);

  stack_bear.name = "Golden␣Bear";
  stack_bear.buddy = heap_bear;
  heap_bear->name = "Oski";
  heap_bear->buddy = &stack_bear;
}
```

Figure 3: Program II

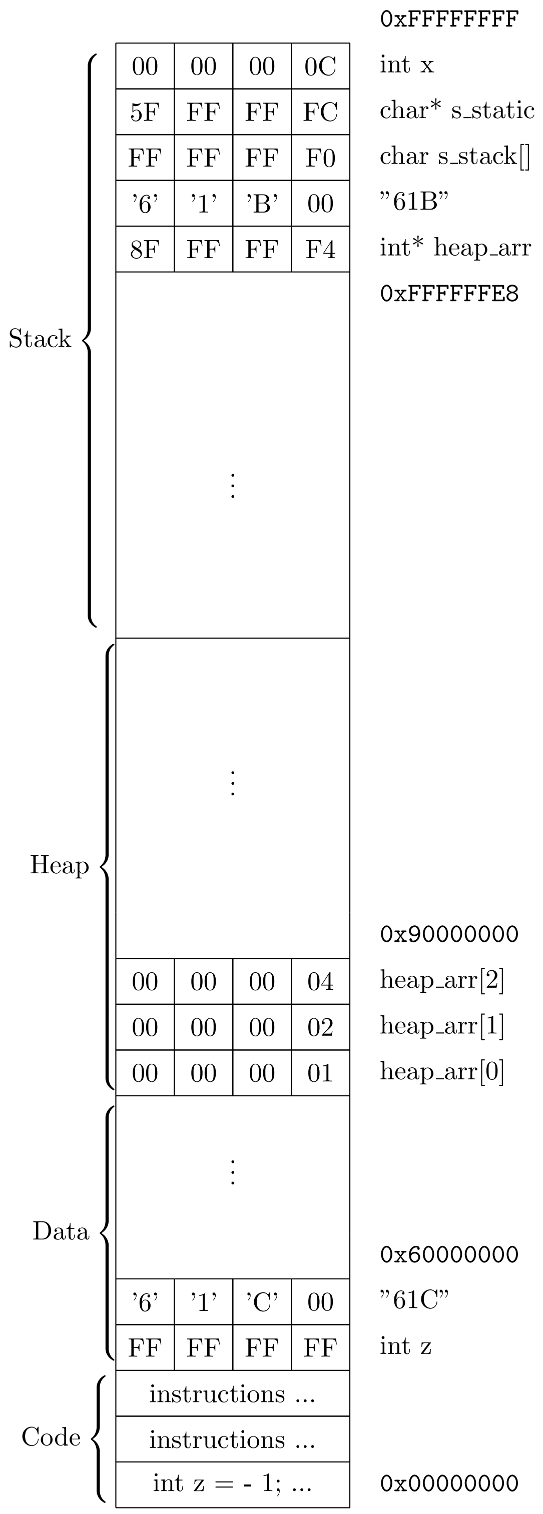

Figure 4: Memory after Program I

3.3 Arrays

An array is a list of elements similar to an array in Java, where all of the elements must be of the same data type and the length of the array is fixed at declaration. They are declared as {`data type`} `arr[length]` and their contents can be accessed with `arr[index]`. (see Program II)

Like many data types in C, arrays can be thought of as a special type of pointer, that points to the 0th element of the array. For example, in Program II, `arr == a`.

When an array is declared, enough space in memory is allocated to store the contents of the array. Specifically, declaring {`data type`} `arr[length]` would allocate `length * sizeof(data type)` bytes of memory for the contents of `arr`. In Program I, declaring `char s_stack[4]` allocated 4 bytes of stack memory at `0xFFFFFFF0`, that were then filled with '6', '1', 'B', '\0'. Indexing into an array is then just pointer arithmetic, as `arr[index]` is equivalent to `*(arr + index)`.

3.4 Strings

Unlike in most other languages, like Python and Java, strings are not their own data type in C. Instead, strings are simply considered pointers to the 0th char of the string's actual value, with the string's literal value being a contiguous sequence of `char`'s. The end of a string is then indicated by the null terminator, `'\0'`, which is just a byte with value 0x00. For example, `"61C"` and `"61B"` are both null-terminated strings in Figure 4.

Additionally, for strings, compilers generally store string literals in the static segment of memory even if the literal is declared inside a function call (and would normally go into the stack). This optimization is made because if a string literal is used, its value will likely not change during the program. However, one exception to this optimization, is if a string literal is stored in a `char[]` instead of a `char*`. Since an array allocates space for the array's contents, the string literal will be stored in that allocated space in the stack instead of in the static segment. Figure 4 shows the difference between the two with `char* s_static` and `char s_stack[]`.

3.5 Structs

Similar to objects in other languages, structs are used to associate many primitive data types (e.g. chars, ints, pointers) into a new data type. For example, the `struct bear` in Program II, has a string variable for a name, and a pointer to another instance of `struct bear`. In C, each of these "instance variables" is called a member of the struct (e.g. the `buddy` member).

After declaring a struct, you can then access its members using the `.` operator. It's also very common for programs to use pointers to structs. When accessing a member from a struct pointer, the `->` operator is used. For example, following Program II, `heap_bear->name;` would return "Oski" and is equivalent to `(*heap_bear).name;`.

Additionally, in memory, structs are stored as tightly packed, meaning that all the members of the struct are stored together in a contiguous chunk of memory. Figure 5 shows an example of an instance of the bear struct in memory.

	...				
struct bear	'D'	'o'	'e'	'\0'	char* name;
	BF	FF	FF	EC	struct bear* buddy;

Figure 5: An example memory contents.

3.6 malloc()

`malloc` is a function in C that allows you to dynamically allocate memory (i.e. allocate memory from the heap). It takes in one argument, the number of bytes of heap memory that you wish to allocate, and then returns the address of where that allocated block starts. Although this class assumes data type sizes of a 32 bit machine, the number of bytes that represents a given data type can vary between machines, so when specifying the number of bytes to allocate for `malloc`, the `sizeof({data type})` function is used, which returns the number of bytes that the given data type takes. For example, to allocate enough space for an array of four integers: `int* arr = malloc(4 * sizeof(int))`. Figure 6 shows an example of a situation that requires `malloc`: since stack variables do not persist after the end of the function call, a heap variable must be used when creating a return value for a function.

As mentioned earlier, heap memory is not automatically reclaimed and persists beyond a function call. Therefore, at some point before the program finishes, you must call `free()` on the address that you obtained from calling `malloc()`. For example, to free the previous array of four integers: `free(arr);`. Failure to call `free()` will result in a **memory leak**, where the heap memory allocated by `malloc` remains unavailable even though it is no longer in use.

```
struct bear* makeStackOski() {
  struct bear oski;
  oski.name = "Oski"
  oski.buddy = NULL;

  /* Since oski is allocated on the stack,
      its data is not guaranteed to persist
      (i.e. may be garbage) after this function call */
  return &oski;
}

struct bear* makeHeapOski() {
  struct bear* oski = malloc(sizeof(struct bear));
  oski->name = "Oski";
  oski->buddy = NULL;

  /* Since oski is allocated on the heap,
     its data will persist after this function call */
  return oski;
}

void oskiInMemory() {
  struct bear* oskiGood = makeHeapOski();
  struct bear* oskiBad = makeHeapOski();

  /* Do some stuff */

  free(oskiGood); /* Used heap memory is freed */
  return; /* oskiBad hasn't been freed -- memory leak! */

}
```

Figure 6: Using malloc()