CS 240 Laboratory 6/7
Pointers and Introduction to gdb/valgrind

- Predict results of pointer code
- Write some pointer code
- Analyze incorrect code
- Start to use GNU debugger ***gdb***
  - see what is going on "inside" a program while it executes
  - display values of variables and examine contents of memory
  - understand the effect of your programs on the hardware of the system
  - Start to use **Valgrind** memory error detection tool to indicate problems with memory allocation/deallocation and access

Pointers

A *pointer* is a variable that contains the address of another variable.

Since a pointer contains the address of an item, it is possible to access the item "indirectly" through the pointer. For example,

```
int x;
int* px = &x;
*px = 2; //changes the value of x to 2
```

means *px* contains the address of *x*, or "points" to *x*.

Similarly,

```
int y = *px;
```

means that *y* gets the value stored at the address in *px* (the value *px* "points" to).

Array notation can also be used interchangeably with pointer notation in C. For example,

```
int x;
int* px = &x;
px[0] = 2;  //changes the value of  x to 2
```

Pointer Arithmetic

If *p* is a pointer, then *p++* increments *p* to point to the next element of whatever kind of object *p* points to. So, the actual number by which p gets increments is a multiple of the size in bytes of the object pointed to.

```
int* p;
p++;
```

results in *p* being incremented by the size of an integer in bytes on the particular machine on which the operation is performed. If the word size is 32 bits, p is incremented by 4. If the word size is 64 bits, p is incremented by 8.

If you add an integer **x** to a pointer, you are expressing an address that is **x** number of values away from the pointer, where the values are the size in bytes of the type of value being pointed to by the pointer. For example,

```
int x;
int* p = &x;
int* q;
q = p + 1;  //the address  q is 1 integer, or 4 bytes,
            // away from the address  p
```

So, the pointer/address **q** is calculated with:

p + (1*scaling factor)

The scaling factor for an **int** is 4 (meaning that an integer uses 4 bytes).

If you subtract one pointer from another, you are determining how many values of the type being pointed to are between the two pointers/addresses. Assuming the above code has been executed:

q – p // returns 1 (which is a special type of variable in C, *ptr_diff*), which indicates there is 1 integer between the two pointers/addresses

The actual addresses of **q** and **p** are actually 4 bytes apart. If you perform a subtraction of the actual addresses, then you would have to divide by the scaling factor to determine the result. For example,

given **q = 0x00000004** and **p = 0x00000000**

q-p is calculated by (0x00000004 – 0x00000000)/4 = 1

Multiple Dereferencing and Memory Models

The following declaration allocates space in memory for an array of *pointers* (specifically, 3 *pointers* to *char*s):

char* commandA[3];

Assuming ***commandA*** is a local variable of a function, the variable itself and the space for this array, will be:

- allocated at compile time
- assigned to addresses in the *stack* area of memory,

The ***commandA*** array could also be declared *dynamically* using the ***malloc*** function:

char** commandA = (char**)malloc(3 * sizeof(char*));

The space for the array would then be:

- allocated at run-time, when the statement is executed
- assigned to addresses in the *heap* area of memory

You can dereference more than once with the use of multiple operators (remember that arrays and pointer can be used interchangeably). For example:

char** commandPtr = commandA;

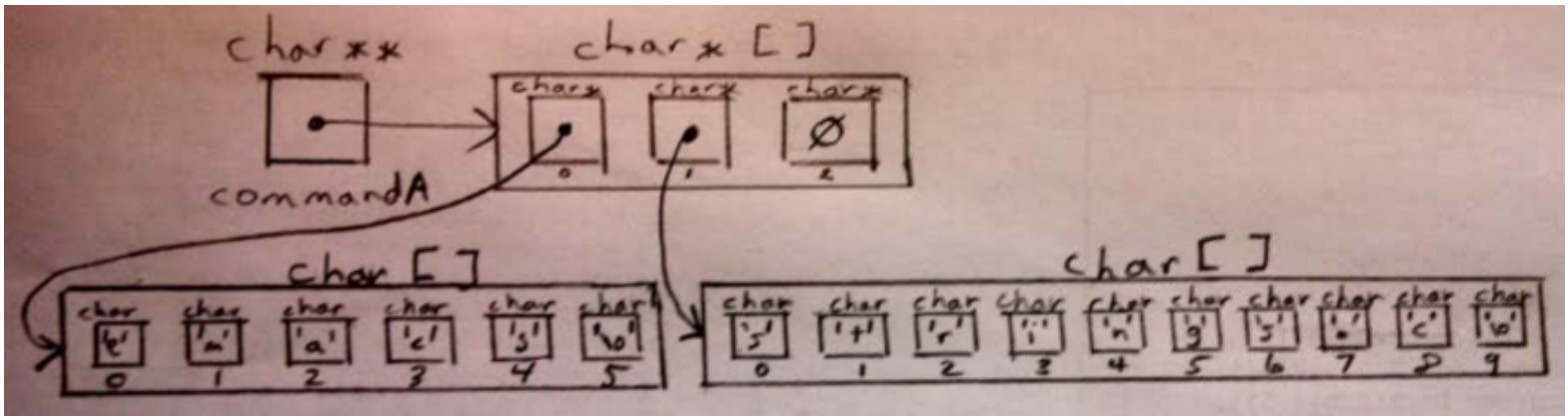
The following statements can be used to initialize strings (arrays of characters terminated by a null character):

commandA[0] = "emacs";
commandA[1] = "strings.c";
commandA[2] = NULL;

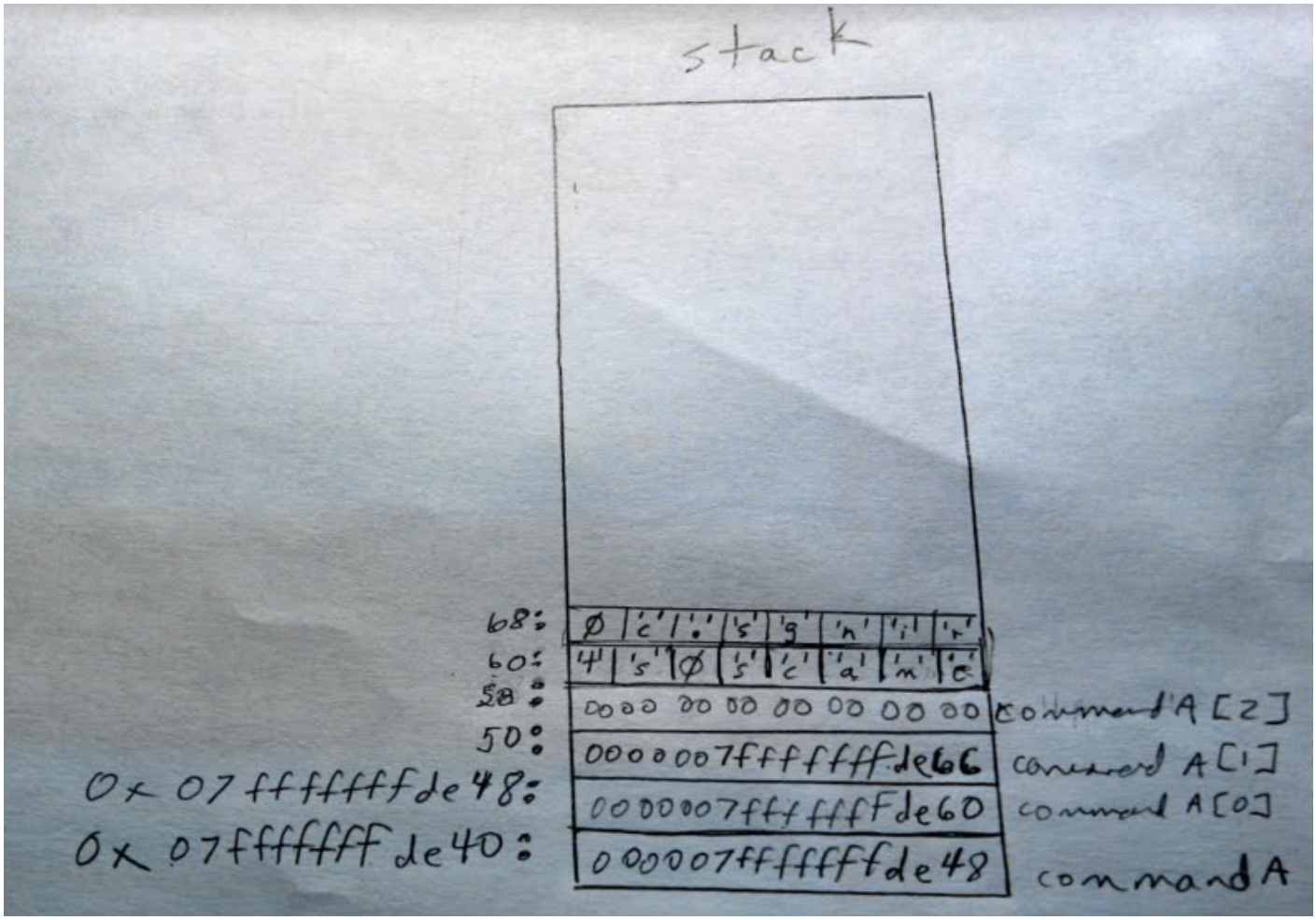
The character for the arrays would be stored in memory in a section which is used for string literals whose values can be understood at compile time.

If the array is being filled with characters during execution of the program, however, then you would have to dynamically allocate an array of size = number of characters + 1(for the terminating null character). In that case, the array would be stored in the *heap*.

You could use the following diagram to model the data (the directed arrows indicate a *pointer*, or *address*):



Another way to understand how memory is organized here is to use our model of memory from lecture (here we are assuming that ***commandA*** and the strings are stored as local variables, in the *stack* area of memory):



GNU Debugger (gdb)

Commands

Can be shortened to a single letter, or repeated by entering <return> at the prompt):

- Compile C program with **–g** option to create debugging information
- Run the program under **gdb**

  $ **gdb testprog**

  (gdb) **run**

- Set breakpoints

  (gdb) **break main**

- Step/next statement by statement through your program

  (gdb) **step**
  (gdb) **next**
  (gdb) **cont** -- continue execution

- Display/print code or values of variables and arguments

  (gdb) **list**
  (gdb) **print x**
  (gdb) **info locals**
  (gdb) **info args**

- (gdb) **`quit`** or **`Ctrl-d`** **`--`** to exit.

- To find a bug:

  1. Set breakpoints at the start of every function
  2. Restart the program and step line-by-line until you locate the problem exactly.
  3. If program is stuck (infinite loop) **`Ctrl-c`** terminates the action of any gdb command that is in progress and returns to the gdb prompt.

- Execute statements/expressions during execution to tweak program execution state

  (gdb) **set var i = 2**

- Display/print binary and hexadecimal representation of variables and arguments

  (gdb) **print /x result** -- uses hex representation
  (gdb) **print /t result** -- uses binary representation