ECE 222 System Programming Concepts
Lecture 6 – Pointers

Pointers are a difficult tool to master. They are the source of many coding errors and bugs, leading to a number of flaws and security problems. Why then do we use them?

1) Passing values to/from a function.

[Use example program func.c]

Consider the memory map for the given code:

label	address (byte)	value
numerator	1000-1003	
denominator	1004-1007	
dividend	1008-1011	
remainder	1012-1015	
x	2000-2003	
y	2004-2007	
d	2008-2011	
r	2012-2015	

When the code executes, the values 9 and 2 go into x and y, then the function call happens. What does that do? It makes a copy of the given parameters, and places them in the local memory locations for the function.

[Place 9, 2, 2008, 2012 in the appropriate locations.]

Then the function code is executed. Using the pointer to dereference the address, the results are placed in the memory locations for d and r. Notice that those results are never in any local memory location for the function, they are stored only in the original main variables.

2) Pointers are arrays, and vice versa.

[Use example program ap.c]
[Have the class read the code, to figure out what it does, while building the map]

label	address (byte)	value
array[0]	1000-1007	10.0
[1]	1008-1015	11.0
[2]	1016-1023	12.0
[3]	1024-1031	13.0
[4]	1032-1039	14.0

d_ptr	1040-1043	1000
value	1044-1051	-3.3
i	1052-1055	
offset	1056-1059	2

This program demonstrates two ways to display an address, hexadecimal and base10. The code %p stands for pointer (address), but displays in hexadecimal. I prefer base 10, so I use %u which stands for unsigned integer.

> [After demonstrating the program, and in particular the assignment statement at the bottom, have the class write an equivalent statement using a pointer.]

Answer: *(d_ptr+offset)=value;

3) Dynamic memory allocation.

When declaring variables in a program, you are using **static allocation**. This means that the size (in bytes) of the variables is known before the program runs. Upon starting execution of the program, the O/S finds a place in memory for all the variables in an area called “the stack”. The entire time that the program is running, the size of this area does not change.

Sometimes, a program does not know how much memory it needs prior to execution. For example, in reading a dictionary, the program might not know how big an array is needed to store all the words before the particular dictionary file has been selected, and read. In this case we need to use dynamic memory allocation. The basic statement is as follows:

```
double *array;
array = (double *)malloc(10*sizeof(double));
        ^^^^^^^   ^^^^^^  ^^^^^^^^^^^^^^
        typecast  request  how many bytes?
                   to O/S
```

The O/S manages a large pool of memory, called “virtual memory” (VM), that is much larger than the stack. A malloc() call asks the O/S for a chunk of memory from VM of the given size. The address of that chunk of memory, if available, is returned:

label	address (byte)	value
array	1000-1003	100000
VM	100000-…	

When the program is done with that memory, it should use the free() function to return ownership of the memory to the O/S:

free(array);

What does the sizeof() function do? It returns the size, in bytes, of a variable or type.

[Use example program sizes.c to test students.]

Once memory is dynamically allocated, it can be used just like static memory. This means that even though we allocated it using a pointer, we can access it using a pointer or an array! (Remember, they are interchangeable most of the time.)

[Use example program dyn.c to demonstrate.] [Show memory map using VM.]

4) Double pointers.

What is a double pointer? For example:

double **ptr;

Break it down piece by piece. *ptr is an address of a double. The * symbol means "address of", so **ptr must mean "address of address of a double".

How big, in bytes, is a double pointer? It's still just an address, so 4 bytes.

Why would we use one? Suppose we wanted to pass an address to a function, and have that function allocate memory to it.

[Use example program dyn2.c] [Memory map will look like:]

label	address (byte)	value
numbers	1000-1003	100000
i	1004-1007	7
listsize	2000-2003	7
list	2004-2007	1000
VM	100000-…	

Look at the mixed array/pointer reference:

*(list)[i]=10+i;

How could that be rewritten using only pointers? [Make class think it through.]

Answer: *((*list)+i)=10+i;