Lab on Pointers in C

This lab involves playing with a number of C programs to solve some problems involving pointers. There are 4 different problems, work on as many as you can. If you are a pointer guru, try your hand at the segvhunt (exercise 5). Exercise 5 goes beyond what we expect you to know at the moment ...

The exercises here should help make sure you understand all about pointers. It's important to make sure you understand things (get help if you are stuck!).

Exercise 1

```
/* p1.c

Write a short C program that declares and initializes (to any value you like) a
double, an int, and a char. Next declare and initialize a pointer to each of
the three variables. Your program should then print the address of, and value
stored in, and the memory size (in bytes) of each of the six variables.

Use the “0x%x” formatting specifier to print addresses in hexadecimal. You
should see addresses that look something like this: "0xbfe55918". The initial
characters "0x" tell you that hexadecimal notation is being used; the remainder
of the digits give the address itself.

Use the sizeof operator to determine the memory size allocated for each
variable.
*/
```

Exercise 2

```
/* p2.c

Find out (add code to print out) the address of the variable x in foo1, and the
variable y in foo2. What do you notice? Can you explain this?


*/
```

```
#include <stdio.h>

void foo1(int xval)
{
  int x;
  x = xval;

  /* print the address and value of x here */
}

void foo2(int dummy)
{
  int y;

  /* print the address and value of y here */
}

int main()
{
  foo1(7);
  foo2(11);
  return 0;
}
```

Exercise 3

```
/* p3.c

The program below uses pointer arithmetic to determine the size of a 'char'
variable. By using pointer arithmetic we can find out the value of 'cp' and
the value of 'cp+1'. Since cp is a pointer, this addition involves pointer
arithmetic: adding one to a pointer makes the pointer point to the next
element of the same type.

For a pointer to a char, adding 1 really just means adding 1 to the address,
but this is only because each char is 1 byte.

    1. Compile and run the program and see what it does.
    2. Write some code that does pointer arithmetic with a pointer to an
       int and determine how big an int is.
    3. Same idea – figure out how big a double is, by using pointer
       arithmetic and printing out the value of the pointer before and
       after adding 1.
    4. What should happen if you added 2 to the pointers from exercises
       1 through 3, instead of 1? Use your program to verify your
       answer.

*/
```

```
#include <stdio.h>

int main( )
{
  char c = 'Z';
  char *cp = &c;

  printf("cp is 0x%08x\n", cp);
  printf("The character at cp is %c\n", *cp);

  /* Pointer arithmetic - see what cp+1 is */
  cp = cp+1;
  printf("cp is 0x%08x\n", cp);
  /* Do not print *cp, because it points to
     memory not allocated to your program */

  return 0;
}
```

Exercise 4

```
/* p4.c
swap_nums seems to work, but not swap_pointers. Fix it.
*/
```

```
#include <stdio.h>

void swap_nums(int *x, int *y)
{
  int tmp;

  tmp = *x;
  *x = *y;
  *y = tmp;
}

void swap_pointers(char *x, char *y)
{
  char *tmp;

  tmp = x;
  x = y;
  y = tmp;
}

int main()
{
  int a,b;
  char *s1,*s2;

  a = 3; b=4;
  swap_nums(&a,&b);
  printf("a is %d\n", a);
  printf("b is %d\n", b);

  s1 = "I should print second";
  s2 = "I should print first";
  swap_pointers(s1,s2);
  printf("s1 is %s\n", s1);
  printf("s2 is %s\n", s2);

  return 0;
}
```

Exercise 5

For those who want more!

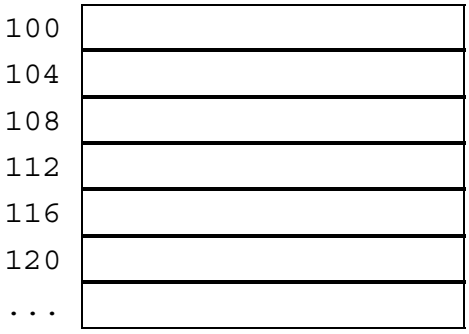
```
/* segvhunt.c

   Find and eliminate all code that generates Segmentation Fault

*/

#include <stdio.h>

int main() {
  char **s;
  char foo[] = "Hello World";

  *s = foo;
  printf("s is %s\n",s);

  s[0] = foo;
  printf("s[0] is %s\n",s[0]);

  return(0);
}
```

Hints:

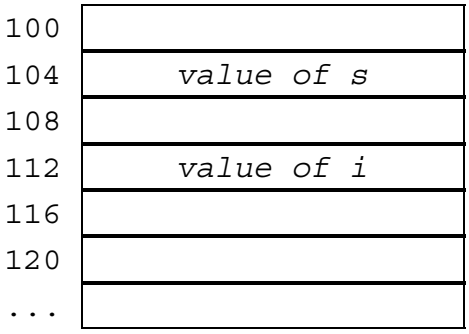
If you are still not comfortable with pointers, you might try the suggestions below to work through things.

Draw a table as a representation of the memory of the computer that is used by your process. Something like this:

Assume each cell in your picture can hold a word (32 bit int or a pointer). Make up some addresses and label the cells with the addresses. Something like this:



For each variable in the C function you are working with, assign the variable to some memory location. For example, we could say that the variable `char *s` is stored in memory location 104 and the variable `int i` is stored in memory location 112:

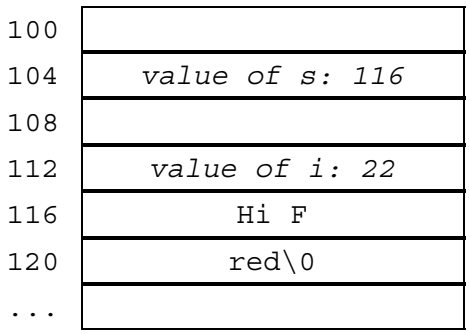


Suppose we also have a string literal in the C function, for example from something like this:

```
s = "Hi Fred";
```

This means that the string `"Hi Fred"` is also somewhere in memory, and that the value of `s` is the address of the first byte in the string. We could pretend that the compiler puts the string in memory starting at address 116. Note that the value of s is now set to 116, which means that the memory location 104 holds the number 116.

We could also assign a value to `i`, for example: `i=22;`. The picture below now represents that state of the effected memory locations:



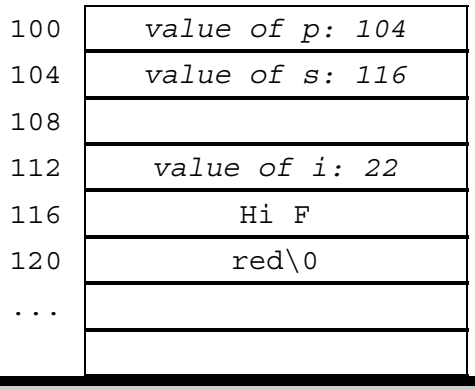
Now keep in mind the following examples of how various C expressions relate to the picture above:

Expression	Description	Value from example
`i`	The value of variable `i`	`22`
`&i`	The address of variable `i`	`112`
`s`	The value of variable `s`	`116`
`&s`	The address of variable `s`	`104`
`*s`	The character at the address `s` (remember that `s` has a value, it's value is an address, so this refers to the character at address 116).	`'H'`
`s[3]`	The character at the address 3 bytes past the address in `s`, in other words, at address `s+3` or `119.`	`'F'`
`s+2`	The address obtained by adding 2 to the value of `s`.	`118`
`*(s+2)`	The character at the address 2 bytes past the address in `s`, in other words, at address `s+2` or `118.`	`' '`

Want more? OK - add this to the picture:

```
char **p = &s;
```

Now `p` is a variable that holds in it the address of `s`. Assume p is placed in memory at address 100:



Expression	Description	Value from example
`p`	The value of variable `p`	`104`
`&p`	The address of variable `p`	`100`
`*p`	The value (of type `char *`) in memory at address `p` (`p` is an address, it's value is 104).	`116`
`*(*p)`	The character at the address `*p`, which is the character at address 116	`'H'`