CSCI 2271 Computer Systems
Class 6: 2/1/16

1. Dealing with Non-Void Functions

- Now suppose function g returns a value.
  - See the second page of the handout.
  - The C compiler treats a non-void function the same as if it were a void function having an extra, pointer-valued first argument.
  - That is:

```
x = g(e1, e2);
```

  - is treated as if it were this:

```
g(&x, e1, e2);
```

- Go through the code on the second page of the handout.
  - The changes are in boldface.

2. Pointer Arithmetic

- Look at the file *ptrArithmetic.c*.
  - This code allocates some variables of different types, and prints the values of *&x* and *(&x) + 1* for them.
- Note that, unlike integers, these values do not necessarily differ by 1. They differ by the size of the variable.
- The reasoning is as follows:
  - If x is an integer, then &x is the address of an integer.
  - We want (&x)+1 to also be an address of another variable, so we skip over the bytes of x.
- Thus for any variable z, (&z)+1 skips over the bytes of z.
  - Thus, the size of z determines how many bytes get skipped.
- This process is called *pointer arithmetic*.

3. Abusing Pointer Arithmetic

- Pointer arithmetic gives us the power to access the contents of a stack frame, arbitrarily.
  - Once we have the address of a variable in a frame, we can use arithmetic to access the other addresses in the frame.
  - This lets us access the stack frame in unintended ways.
- For example, look at the file *ptrAbuse1.c*.
  - The first part prints out the addresses of some variables c, s, and i, where c is a char, s is a short, and i is an int.
  - Running the code shows that the variables are laid out sequentially, in the order c, s, i.
- Consider the second part.

  - It uses pointer arithmetic to access the contents of s and i as chars, using pointer arithmetic from c.
  - The first *printf* statement prints "B A".
  - These two characters are taken from the first and second halves of s. The hex value 41 is the character A, and the hex value 42 is the character B.
  - Note that the characters are retrieved "backwards" from what you might expect. This is because the virtual machine uses "little-endian" integer representation instead of the "big-endian" representation.
  - The second *printf* statement prints "D C B A".
  - These characters are taken from the four bytes of i. Note that they are also backwards.
- Now consider the third part.
  - The *printf* statement prints "4344 4142". Why?
  - We know that the bytes of i are backwards, namely 44 43 42 41.
  - When we read the first two bytes as a short, we get 44 43.
  - But these get interpreted as a little-endian number, that is the hex number 4344.

4. Pointers

- We have seen that you can't store address values in integer variables.
- In C, you must store address values in *pointer* variables.
- For example:

```
int x = 1;
float y = 2.0;
int *px = &x;
float *py = &y;
printf("%p  %p  %p  %p\n", px, py, px+1, py+1);

// the following are not allowed because of data typing
py = &x;
py = px;
```

- The syntax for declaring pointers is a bit subtle.
  - The declaration *int *p* says (reading it backwards) that p is a pointer to an int.
  - That is, the * belongs to the variable name, not the type name.
  - Consider the following example:

```
int x, y;
int *px, py;
int *qx, *qy;
```

  - The second line says that px is a pointer, but py is an integer.
  - If you want both to be pointers, you must use the declaration of line 3.

5. Dereferencing a Pointer

- So far, pointers are interesting but not that useful. What makes them useful is the * operator.
- If p is a pointer variable, then *p is the value stored at the address held by p.
  - This is called *dereferencing* the pointer.
- For example:

```
int x;
int *p = &x;
*p = 3;  //this changes the value of x
```

- Note that this code changed x without mentioning x.
- Also note that lines 2 and 3 give two very different uses for the "*" symbol.
  - Don't get confused by this.

6. Some Simple Examples

```
int x = 2;
int y = 3;
int *p;
int *q;

p = &x;      // p points to x
*p = y;      // assign 3 to x
y = *p + 1; // assign 4 to y
q = p;       // q also points to x
p = *q;      // ILLEGAL!
*p = q;      // ILLEGAL!
q = &y;      // q now points to y
*p = *q;     // assign the value of y (i.e. 4) to x
```