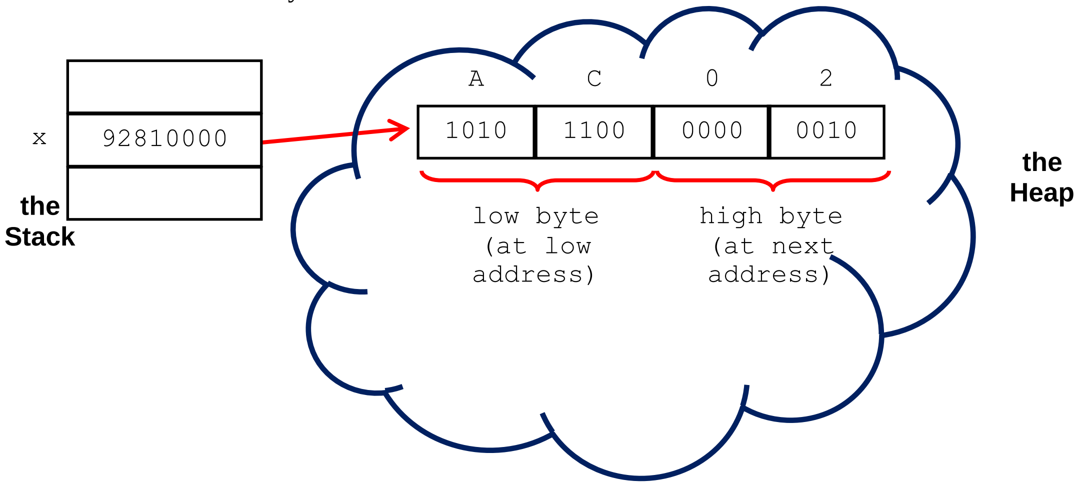
Brief tutorial on using pointer arithmetic and pointer typecasts in C:

First of all, you must understand the effect and uses of a pointer typecast. Consider the following snippet:

```
uint16_t *x = malloc(sizeof(uint16_t));     // 1
*x = 684;                                   // 2
```

Statement 1 causes the allocation of a two-byte region of memory, whose address is stored in the pointer `x`. Statement 2 stores the value 684 (`0x2AC` in hexadecimal, or `0000 0010 1010 1100` in binary) into that two-byte region. Let's assume that the address returned by the call to `malloc()` was `0x00008192`.

So the current situation in memory would be:



(The bytes are stored in little-endian order, just as they would be on any Intel-compatible system.) If you dereference the pointer `x`, you'll obtain the contents of the two bytes beginning at the address stored in `x`. That's because `x` was declared as a pointer to something of type `uint16_t`, and `sizeof(uint16_t)` is 2 (bytes).

Now consider:

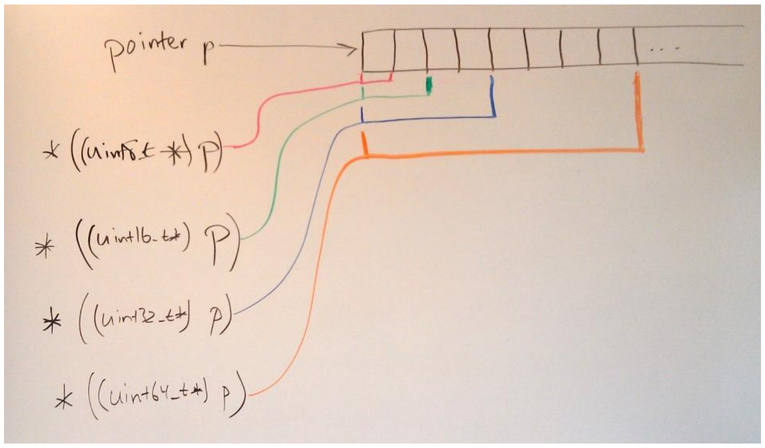
```
uint8_t *y = NULL;                      // 3:  y == 0x00000000
y = (uint8_t*) x;                       // 4:  y == 0x00008192

uint8_t   z = *y;                       // 5:  z == 0xAC    (1 byte)

uint16_t  w = *x;                       // 6:  w == 0x02AC  (2 bytes)
```

The effect of statement 4 is that `y` takes on the same value as `x`; pointer variables are 32 bits wide, regardless of the type of target they may take, and so the value of `x` will fit neatly into `y`. So, why the typecast? Simply that C is somewhat picky about assigning a pointer of one type to a pointer of another type, and the typecast formalizes the logic so that the compiler will accept it. If you dereference `y`, you'll obtain the contents of the single byte at the address stored in `y`, since `sizeof(uint8_t)` is 1. Hence, statement 5 will assign the value `0xAC` or `172` to `z`, but statement 6 will assign the two-byte value `0x02AC` or `684` to the variable `w`.

Here's a brief summary:



Note how we can use the type of a pointer to determine how many bytes of memory we obtain when we dereference that pointer, as well as how those bytes are interpreted. This can be really useful.

The second thing to understand is how pointer arithmetic works. Here is a simple summary:

```
T *p, *q;       // Take T to represent a generic type.
. . .           // Assume p gets assigned a target in here.
q = p + K;      // Let K be an expression that evaluates to an integer.
```

Then the value of `q` will be: `p + K * sizeof(T)`. Note well: this is very dangerous unless you understand how to make use of it. In some respects, this is really quite simple; maybe too simple. The essential thing you must always remember is that if you want to move a pointer by a specific number of bytes, it's simplest if the pointer is a `char*` or `uint8_t*`, since the arithmetic will then provide you with byte-level control of the pointer's logical position in memory.

The following loop would walk the pointer `y` through the bytes of the target of `x` (the `uint16_t*` seen earlier):

```
uint8_t *y = (uint8_t*) x;

uint32_t bytesRead = 0;

while ( bytesRead < sizeof(uint16_t) ) {

   printf("%"PRIx8"\n", *y);  // print value of current byte in hex;
                              //    'x' causes output in hex;
                              //    'X' capitalizes the hex digits

   ++y;                   // step to next byte of target of x
   ++bytesRead;
}
```

You can also achieve the same effect by applying an offset to the pointer instead of incrementing the pointer:

```
. . .  // same code as before

   printf("%"PRIx8"\n", *(y + bytesRead));  // y + bytesRead points to a
                                             //    location bytesRead bytes
                                             //    past where y points

   ++bytesRead;           // increment your offset counter
}
```

The second approach works because `y + bytesRead` is a `uint8_t*` that "walks" through memory byte-by-byte as `bytesRead` is incremented.

You might want to experiment with this a bit…

Now for copying values from memory into your variables (which are also in memory, of course)... The simplest approach is use appropriate variable types and pointer typecasts. Suppose that the `uint8_t` pointer `p` points to some location in memory and you want to obtain the next four bytes and interpret them as an `int32_t` value; then you could try these:

```
int32_t  N = *p;                // NO.  This takes only 1 byte!

int32_t *q = (int32_t*) p;      // Slap an int32_t* onto the location;
int32_t  N = *q;                //    so this takes 4 bytes as desired.

int32_t  N = *((int32_t*) p);   // Or do it all in one fell swoop.
```

The last form is the most idiomatic in the C language; it creates a temporary, nameless `int32_t*` from `p` and then dereferences it to obtain the desired value. Note that this doesn't change the value of `p`, and therefore does not change where `p` points to in memory. So, if you wanted to copy the next few bytes you'd need to apply pointer arithmetic to move `p` past the bytes you just copied:

```
p = p + sizeof(int32_t);  // Since p is a uint8_t*, this will move p forward
                          //    by exactly the size of an int32_t.
```

One final C tip may be of use. The C Standard Library includes a function that will copy a specified number of bytes from one region of memory into another region of memory: `memcpy()`. You can find a description of how to use the function in any decent C language reference, including the C tutorial linked from the Resources page of the course website.

The C Standard, section 6.5.6 says:

> When an expression that has integer type is added to or subtracted from a pointer, the result has the type of the pointer operand. If the pointer operand points to an element of an array object, and the array is large enough, the result points to an element offset from the original element such that the difference of the subscripts of the resulting and original array elements equals the integer expression. In other words, if the expression **`P`** points to the *i*-th element of an array object, the expressions **`(P)+N`** (equivalently, **`N+(P)`**) and **`(P)-N`** (where **`N`** has the value *n*) point to, respectively, the *i*+*n*-th and *i*−*n*-th elements of the array object, provided they exist. Moreover, if the expression **`P`** points to the last element of an array object, the expression **`(P)+1`** points one past the last element of the array object, and if the expression **`Q`** points one past the last element of an array object, the expression **`(Q)-1`** points to the last element of the array object. If both the pointer operand and the result point to elements of the same array object, or one past the last element of the array object, the evaluation shall not produce an overflow; otherwise, the behavior is undefined. If the result points one past the last element of the array object, it shall not be used as the operand of a unary **`*`** operator that is evaluated.