## Pointer Manipulations                                    Pointer Casts and Data Accesses

**Viewing Memory**

The contents of a block of memory may be viewed as a collection of hex nybbles indicating the contents of the byte in the memory region; for example, here's a block of 512 bytes:

```
7F1022EC2BEAD1F54E9262897A7E39039EF20A22F84AE7F28A40241BCA9ED049
AFF236DADC07CF2B9932DF9DFDA2D19B94DBBD8D26A47FB0E5A4CBAF429BF8F1
8E2ECC6A359B95CECD746BCA163C66AB1823383EC8B7EEAD5BB95C9E0BD886E2
835B4DB8F7E287C457F28F6D2FF51847185085E008738D632CE901803E9163C1
ECB079E39200A8E9F33757222C6F6944C0EE25C861B22B8D9C2D0DDABE709BAA
6279746515369C086DF32A996393DD238102EBE2B5166F897A7C2B01EDC6AC0D
DA3AC0EF705DF7DD502176B3B453D63556C1170BD8865C1B03871DF04DC9FD27
03BE17731B0E506B30C61FE419F51A6FB7317A8FB8D6AABB5DC7ABAA90A8D293
66E90681F756ED271C0C0C360126A5B85720470FF6F2CA54B975FE4A1ED0DD84
897A7E3903F3D857FFE48D000A32B96252007149F23C9DACB19BF6CF6CD35425
B75AD6F24DAF494C93D64C9E0805005B0671A4F8AD41A45FDC9A2E486E826E25
2CE901803E9163C18102EBE2B5166F89DC440BD8F360758736C2253FC7259ACD
963EC6447F6AA35B05D1A4735412983026A47FB0E5A4CBAF39034754682D0DDA
56B05A4A10CFD14791F60BD8862026B15EECF5DD5798385C6ADCCFBEEE67EE45
17488F2818606FA956F50271152922731518506CB088C81A6597D853FFC79816
0F273E2787ADD1DDA2D34EB7FC712A12897A7E39034754682D0DDAB75AD6FDA2
```

The memory block is a sequence of bytes; we can think of each byte as being at a particular *offset* from the beginning of the memory block. For example, in the first row above, the byte 10 is at offset $1_{10}$, and the byte 4A is at offset $21_{10}$. (Recall that a byte consists of two hex nybbles.)

Another way of thinking about this is that we have an array of bytes, indexed just like the cells of any array, relative to the first byte in the memory block. If we called the array `Data`, then `Data[1]` would be `0x10` (or $16_{10}$) and `Data[21]` would be `0x4A` (or $74_{10}$).

Here is a C function that will print a selected block of bytes from such a memory block, using an array-based view of the necessary logic:

```c
/**  Uses array-based logic to access a specified portion of a region of
 *   memory and print the corresponding bytes to a supplied file stream.
 *
 *   Pre:   Out is open on a file
 *          Base[0] is the first byte of the memory region to be examined
 *          Index is the index of the first relevant byte of the memory region
 *          nToPrint is the number of bytes to be printed
 *   Comments:
 *      This uses array syntax to access the bytes in the memory block.
 */
void showBytesAtIndex(FILE* Out, const uint8_t Base[], uint16_t Index,
                                                       uint8_t nToPrint) {

   for (uint8_t pos = 0; pos < nToPrint; pos++) {

      fprintf(Out, "  %02X", Base[Index + pos]);
   }

   fprintf(Out, "\n");
}
```

Suppose we executed the call:  `showBytesAtIndex(stdout, Data, 34, 6)`

Now, `Data[34]` would be the byte `36` near the beginning of the second row of the display, and the function would print 6 bytes starting there, with two spaces separating the bytes:  `36  DA  DC  07  CF  2B`

(Remember, a hex nybble is a 4-bit value, so there are two nybbles, and hence two hex digits, per byte.)

Let's consider a few details in the implementation of that function:

- We used **const** in specifying the second parameter. That means the function is not allowed to modify the contents of the array that's passed to it.
- We refer to the bytes in the memory block using the type **unit8_t**; that's so we can avoid any issues that might arise if the high bit of a byte happened to be 1 (remember 2's complement representation).
- The parameter `Index` is of type **uint16_t**; that limits the size of the memory block. The maximum value of a **uint16_t** variable is $2^{16} - 1$ or 65535. There's no good reason for that, really. It just gave me an excuse to add this to the discussion. A similar point could be made about the parameter `nToPrint`.
- The format string `"  %02X"` causes the variable to be displayed in two columns, with a leading 0 if necessary, in hexadecimal, preceded by two spaces.

But the memory display shown above is not very human-friendly. A more readable version would format the data so the individual bytes were separated, and indicate the offsets of those bytes as offsets relative to the beginning of the block:

```
      0  1  2  3  4  5  6  7      8  9  A  B  C  D  E  F
0    7F 10 22 EC 2B EA D1 F5     4E 92 62 89 7A 7E 39 03
1    9E F2 0A 22 F8 4A E7 F2     8A 40 24 1B CA 9E D0 49
2    AF F2 36 DA DC 07 CF 2B     99 32 DF 9D FD A2 D1 9B
3    94 DB BD 8D 26 A4 7F B0     E5 A4 CB AF 42 9B F8 F1
4    8E 2E CC 6A 35 9B 95 CE     CD 74 6B CA 16 3C 66 AB
5    18 23 38 3E C8 B7 EE AD     5B B9 5C 9E 0B D8 86 E2
6    83 5B 4D B8 F7 E2 87 C4     57 F2 8F 6D 2F F5 18 47
7    18 50 85 E0 08 73 8D 63     2C E9 01 80 3E 91 63 C1
8    EC B0 79 E3 92 00 A8 E9     F3 37 57 22 2C 6F 69 44
9    C0 EE 25 C8 61 B2 2B 8D     9C 2D 0D DA BE 70 9B AA
A    62 79 74 65 15 36 9C 08     6D F3 2A 99 63 93 DD 23
B    81 02 EB E2 B5 16 6F 89     7A 7C 2B 01 ED C6 AC 0D
C    DA 3A C0 EF 70 5D F7 DD     50 21 76 B3 B4 53 D6 35
D    56 C1 17 0B D8 86 5C 1B     03 87 1D F0 4D C9 FD 27
E    03 BE 17 73 1B 0E 50 6B     30 C6 1F E4 19 F5 1A 6F
F    B7 31 7A 8F B8 D6 AA BB     5D C7 AB AA 90 A8 D2 93
10   66 E9 06 81 F7 56 ED 27     1C 0C 0C 36 01 26 A5 B8
11   57 20 47 0F F6 F2 CA 54     B9 75 FE 4A 1E D0 DD 84
12   89 7A 7E 39 03 F3 D8 57     FF E4 8D 00 0A 32 B9 62
13   52 00 71 49 F2 3C 9D AC     B1 9B F6 CF 6C D3 54 25
14   B7 5A D6 F2 4D AF 49 4C     93 D6 4C 9E 08 05 00 5B
15   06 71 A4 F8 AD 41 A4 5F     DC 9A 2E 48 6E 82 6E 25
16   2C E9 01 80 3E 91 63 C1     81 02 EB E2 B5 16 6F 89
17   DC 44 0B D8 F3 60 75 87     36 C2 25 3F C7 25 9A CD
18   96 3E C6 44 7F 6A A3 5B     05 D1 A4 73 54 12 98 30
19   26 A4 7F B0 E5 A4 CB AF     39 03 47 54 68 2D 0D DA
1A   56 B0 5A 4A 10 CF D1 47     91 F6 0B D8 86 20 26 B1
1B   5E EC F5 DD 57 98 38 5C     6A DC CF BE EE 67 EE 45
1C   17 48 8F 28 18 60 6F A9     56 F5 02 71 15 29 22 73
1D   15 18 50 6C B0 88 C8 1A     65 97 D8 53 FF C7 98 16
1E   0F 27 3E 27 87 AD D1 DD     A2 D3 4E B7 FC 71 2A 12
1F   89 7A 7E 39 03 47 54 68     2D 0D DA B7 5A D6 FD A2
```

0x10 at offset 0x01 or $1_{10}$

0x4A at offset 0x15 or $21_{10}$

0xAA at offset 0xFB or $251_{10}$

0x39 at offset 0x1F3 or $499_{10}$

This is known as a *hexdump* view. The value in the first column shows the first two (hex) digits of the offset of the data displayed in that row. The column heading for a byte shows the last digit of the offset of that byte.

**The Functions**

For this assignment, you'll implement three C functions to perform different kinds of accesses to a block of memory. Be sure to pay attention to the restrictions that are imposed on your implementation. In particular, each function is restricted to using pointer notation to manage all accesses to the data; use of array bracket notation will result in a score of 0.

The first function you will implement is functionally equivalent to the example function given earlier, but it will use pointer syntax instead of array syntax:

```
/**  Uses pointer-based logic to access a specified portion of a region of
 *   memory and copy the corresponding bytes to a supplied array.
 *
 *   Pre:   pDest points to an array of dimension nBytes + 1, or greater
 *          baseAddr points to the first byte of the memory region
 *          Offset is the location, relative to baseAddr, of the first
 *              relevant byte of the memory region
 *          nBytes is the number of bytes to be copied
 *   Restrictions:
 *      You must use only pointer syntax in accessing the data block.
 */
void getBytesAtOffset(uint8_t* const pDest, const uint8_t* const baseAddr,
                                     uint16_t Offset, uint8_t nBytes);
```

The interface of the function deserves a short discussion. The second parameter, baseAddr, illustrates the use of **const** in two different ways. For a pointer used as a parameter, placing **const** before the pointer type means the function is not permitted to modify the value of the target of the pointer, and placing **const** after the pointer type means the function is not permitted to modify the value of the parameter (which would make it point to a different target).

In this case, we don't what this function to modify the contents of the memory block; the first use of **const** enforces that. And, we don't see any reason for the function to work with a pointer that has a different target than we have decided on, so the second use of **const** enforces that.

It's good practice to use **const** appropriately when designing function interfaces, especially when a parameter is a pointer. Even so, you should also understand that the C language does make it possible for a programmer to create a local pointer, initialize it from the parameter, and use that local pointer in ways that violate the **const** restrictions you may have imposed. That doesn't make **const** useless, and a principled C programmer will avoid sidestepping **const**.

Here are some example results, based on the memory block shown earlier, assuming *baseAddr is the first byte:

```
showBytesAtOffset(..., baseAddr,  23,  4):  F2 8A 40 24
showBytesAtOffset(..., baseAddr, 219,  2):  F0 4D
showBytesAtOffset(..., baseAddr, 338, 12):  A4 F8 AD 41 A4 5F DC 9A 2E 48 6E 82
```

By the way, the parameters in the examples above are given in base 10; if you want to check the results against the hexdump, you may want to convert the offsets to hex as well. The first one would be `0x17`.

We guarantee that testing will only be done with logically valid parameters.

The second function requires traversing the given memory region and looking for matches to a given one-byte value:

```
/**   Uses pointer-based logic to search a specified portion of a region of
 *    memory for occurrences of a specified one-byte value.
 *
 *    Pre:  pOffsets points to an array of dimension at least 256
 *          baseAddr points to the first byte of the memory region
 *          Length is number of bytes in the memory region
 *          Byte is the value to be found
 *    Post: The offsets of up to 256 occurrences of Byte have been stored
 *             in pOffsets, in ascending order, starting at index 0.
 *    Returns:  the number of occurrences of Byte found in the memory region
 *              (which may be less than the actual number of occurrences)
 *    Restrictions:
 *       You must use only pointer syntax in accessing the data block.
 */
uint8_t findOccurrencesOfByte(uint16_t* const pOffsets,
                              const uint8_t* const baseAddr,
                              uint32_t Length, uint8_t Byte);
```

Here are some example results, based on the memory block shown earlier, assuming *baseAddr is the first byte:

```
findOccurrencesOfByte(..., baseAddr, 512, 0x7F):    0   36  184  192
findOccurrencesOfByte(..., baseAddr, 512, 0):          85  12B  131  14E
findOccurrencesOfByte(..., baseAddr, 512, 0xE9):    79   87  101  161
```

```
         0  1  2  3  4  5  6  7      8  9  A  B  C  D  E  F
0       7F 10 22 EC 2B EA D1 F5     4E 92 62 89 7A 7E 39 03
1       9E F2 0A 22 F8 4A E7 F2     8A 40 24 1B CA 9E D0 49
2       AF F2 36 DA DC 07 CF 2B     99 32 DF 9D FD A2 D1 9B
3       94 DB BD 8D 26 A4 7F B0     E5 A4 CB AF 42 9B F8 F1
4       8E 2E CC 6A 35 9B 95 CE     CD 74 6B CA 16 3C 66 AB
5       18 23 38 3E C8 B7 EE AD     5B B9 5C 9E 0B D8 86 E2
6       83 5B 4D B8 F7 E2 87 C4     57 F2 8F 6D 2F F5 18 47
7       18 50 85 E0 08 73 8D 63     2C E9 01 80 3E 91 63 C1
8       EC B0 79 E3 92 00 A8 E9     F3 37 57 22 2C 6F 69 44
9       C0 EE 25 C8 61 B2 2B 8D     9C 2D 0D DA BE 70 9B AA
A       62 79 74 65 15 36 9C 08     6D F3 2A 99 63 93 DD 23
B       81 02 EB E2 B5 16 6F 89     7A 7C 2B 01 ED C6 AC 0D
C       DA 3A C0 EF 70 5D F7 DD     50 21 76 B3 B4 53 D6 35
D       56 C1 17 0B D8 86 5C 1B     03 87 1D F0 4D C9 FD 27
E       03 BE 17 73 1B 0E 50 6B     30 C6 1F E4 19 F5 1A 6F
F       B7 31 7A 8F B8 D6 AA BB     5D C7 AB AA 90 A8 D2 93
10      66 E9 06 81 F7 56 ED 27     1C 0C 0C 36 01 26 A5 B8
11      57 20 47 0F F6 F2 CA 54     B9 75 FE 4A 1E D0 DD 84
12      89 7A 7E 39 03 F3 D8 57     FF E4 8D 00 0A 32 B9 62
13      52 00 71 49 F2 3C 9D AC     B1 9B F6 CF 6C D3 54 25
14      B7 5A D6 F2 4D AF 49 4C     93 D6 4C 9E 08 05 00 5B
15      06 71 A4 F8 AD 41 A4 5F     DC 9A 2E 48 6E 82 6E 25
16      2C E9 01 80 3E 91 63 C1     81 02 EB E2 B5 16 6F 89
17      DC 44 0B D8 F3 60 75 87     36 C2 25 3F C7 25 9A CD
18      96 3E C6 44 7F 6A A3 5B     05 D1 A4 73 54 12 98 30
19      26 A4 7F B0 E5 A4 CB AF     39 03 47 54 68 2D 0D DA
1A      56 B0 5A 4A 10 CF D1 47     91 F6 0B D8 86 20 26 B1
1B      5E EC F5 DD 57 98 38 5C     6A DC CF BE EE 67 EE 45
1C      17 48 8F 28 18 60 6F A9     56 F5 02 71 15 29 22 73
1D      15 18 50 6C B0 88 C8 1A     65 97 D8 53 FF C7 98 16
1E      0F 27 3E 27 87 AD D1 DD     A2 D3 4E B7 FC 71 2A 12
1F      89 7A 7E 39 03 47 54 68     2D 0D DA B7 5A D6 FD A2
```

The third function requires exploiting pointer typecasts in order to impose a specified interpretation on a segment of data in memory. You should be sure you review and understand the notes on that topic; reading the short tutorial on the Resources page of the course website will help.

Consider the four bytes at the address `0xA0` in the hexdump: `62 79 74 65`

We could interpret those four bytes in a number of different, conflicting ways. They could be interpreted as four extended ASCII codes ('b', 'y', 't', 'e'). They could be interpreted as a 32-bit floating-point number (a **float** in C or Java). They could be interpreted as a 32-bit signed integer or a 32-bit unsigned integer. Let's consider that in a bit more detail.

As ASCII codes, the four bytes represent the characters 'b', 'y', 't', and 'e', respectively. Numeric values are stored in little-endian order on x86 systems, so as an integer value, those four bytes will be interpreted as 0x65747962. As a signed integer (2's complement representation), that yields the value $1702132066_{10}$. As an unsigned integer (base-2 representation), that yields the same value, since the high bit is 0

We will also use an enumerated type to indicate to the function whether it should interpret the specified segment of data as a signed integer or an unsigned integer:

```
enum _Sign {SIGNED, UNSIGNED};
typedef enum _Sign Sign;
```

Enumerated types (also available in Java) allow you to define a custom type whose values are descriptive labels, which can make your code much easier to understand. Without the **typedef** statement, we'd have to declare variables of this type as **enum** _Sign; the **typedef** statement allows us to declare them as **Sign**, which is more convenient.

```
/**  Uses pointer-based logic to compute the sum of two values taken from a
 *   region of memory, using pointer typecasts to control the number of bytes
 *   that are summed, and the way those bytes are interpreted.
 *
 *   Pre:   pDest points to a memory region large enough to hold the
 *             requested value
 *          baseAddr points to the first byte of the memory region
 *          Offset1 is the location, relative to baseAddr, of the first
 *             byte of the first operand
 *          Offset2 is the location, relative to baseAddr, of the first
 *             byte of the second operand
 *          Sign indicates whether the bytes are to be interpreted as
 *             representing signed or unsigned integers
 *          nByte is the number of bytes to be considered (1, 2, 4, or 8 only)
 *   Post:  *pDest holds the sum of the two specified values
 *   Returns:  false if either of the requested values could not be obtained;
 *             true otherwise
 *   Restrictions:
 *      You must use only pointer syntax in accessing the data.
 */
bool sumValuesAtOffsets(void* pDest, const uint8_t* const baseAddr,
                        uint32_t Offset1, uint32_t Offset2,
                        Sign Sgn, uint8_t nBytes);
```

For example, suppose we have the data block shown earlier, and the function is called with `baseAddr` pointing to the beginning of the block, `Offset1 == 0x148`, `Offset2 == 0xA4`, `Sgn == SIGNED`, and `nBytes == 2`.

Now, `Offset1` points to the bytes `93 D6`, which represent the signed value 0xD693 (which is $-10605_{10}$).

And, `Offset2` points to the bytes `15 36`, which represent the signed value 0x3615 (which is $13845_{10}$).

Therefore, the sum of the two values is 0x0CA8 (which is $3240_{10}$).

Here's another example, based on the memory block shown earlier, assuming *baseAddr is the first byte:

    sumValuesAtOffsets(..., baseAddr,  0x74, 0x8C, UNSIGNED,  4):  $2817974836_{10}$

In this case:

```
Offset      Bytes              Value₁₀          Sum₁₀
------------------------------------------------------
0x74        08 73 8D 63        1670214408
0x8C        2C 6F 69 44        1147760428       2817974836
```

As a signed value, this would have yielded $-1476992460_{10}$.

It's also worth discussing the use of **void**\* as the type of the first parameter.  Since Sgn can be either SIGNED or UNSIGNED, and nBytes can be any of 1, 2, 4 or 8, the value we are requesting could be any of 8 different C types. Rather than supply 8 different parameters, only one of which would be used in any particular call, we take a generic approach.  A **void**\* can point to data of any type, so the caller can create an appropriate variable, pass a generic pointer to it, and the function can use that generic pointer (by casting) to set the caller's variable to the correct value.

The positive aspect of this use of **void**\* is that it simplifies the function interface, at the expense of some pointer casts on the part of the caller and the function.  The negative aspect is that there is a complete loss of type-checking, so if the caller does not prepare a "receptacle" of the type indicated by the parameters, there will be no guarantee of a useful error message, from the compiler, or at runtime.


**Testing**

Download C05Files.tar from the course website and unpack in a CentOS 7 directory; you will find these files:

    driver.c          driver for testing/grading code
    PtrFuncs.h        declarations for functions specified in this assignment
    PtrFuncs.c        shell file for implementing specified functions
    TestCode.h        declarations for public test/grading functions
    TestCode.o        64-bit Linux binary for testing/grading code
    Generator.h       declarations for data block generation and output
    Generator.o       64-bit Linux binary for data block generation and output code

You can compile the program with the following command:

```
  CentOS> gcc -o driver -std=c99 -Wall driver.c PtrFuncs.c TestCode.o Generator.o
```

The test driver can be executed with the command:

                          CentOS> ./driver [-all | -get | -find | -sum]

The various optional switches determine which functions are tested (-all is the default).  By default, each execution of driver will produce a different set of test data, unless you use the -repeat switch:

                     CentOS> ./driver [-all | -get | -find | -sum] [-repeat]

In that case, the test data file created by a previous run will be reused.  That allows you to focus on a fixed set of test cases while you are debugging.  In fact, you could even edit an existing test case file and create test cases as you like, and then use the -repeat switch to use those test cases.

The driver will also create a file, DataBlock.txt, which contains a hex dump of the specific memory region generated for the test run.  You may find that useful when debugging your results.

You should test your solution thoroughly; the given testing code generates random test data, and there is no guarantee that it will cover all cases unless you run it a number of times.

**What to Submit**

You will submit your modified version of the file `PtrFuncs.c` to the Curator, this time via the collection point C05. That file must include any helper functions you have written and called from your implementations of the various specified functions; any such functions must be declared (as **static**) in the file you submit. You must not include any extraneous code (such as an implementation of `main()` in that file).

Your submission will be graded by running the supplied test/grading code on it. A TA will also check to see if your solution violates any of the restrictions given in the header comment for the function; if so, your submission will be assigned a score of zero (0), regardless of how many tests it passes.

**Pledge:**

Each of your program submissions must be pledged to conform to the Honor Code requirements for this course. Specifically, you **must** include the following pledge statement in the submitted file:

```
//  On my honor:
//
//  - I have not discussed the C language code in my program with
//    anyone other than my instructor or the teaching assistants
//    assigned to this course.
//
//  - I have not used C language code obtained from another student,
//    the Internet, or any other unauthorized source, either modified
//    or unmodified.
//
//  - If any C language code or documentation used in my program
//    was obtained from an authorized source, such as a text book or
//    course notes, that has been clearly noted with a proper citation
//    in the comments of my program.
//
//  - I have not designed this program in such a way as to defeat or
//    interfere with the normal operation of the Curator System.
//
//
//    <Student Name>
//    <Student's VT email PID>
```

**We reserve the option of assigning a score of zero to any submission that is undocumented or does not contain this statement.**

**Change Log**

| Version | Posted | Pg | Change |
|---------|--------|----|--------|
| 3.00 | Oct 6 | | Base document. |

## Appendix I:           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:
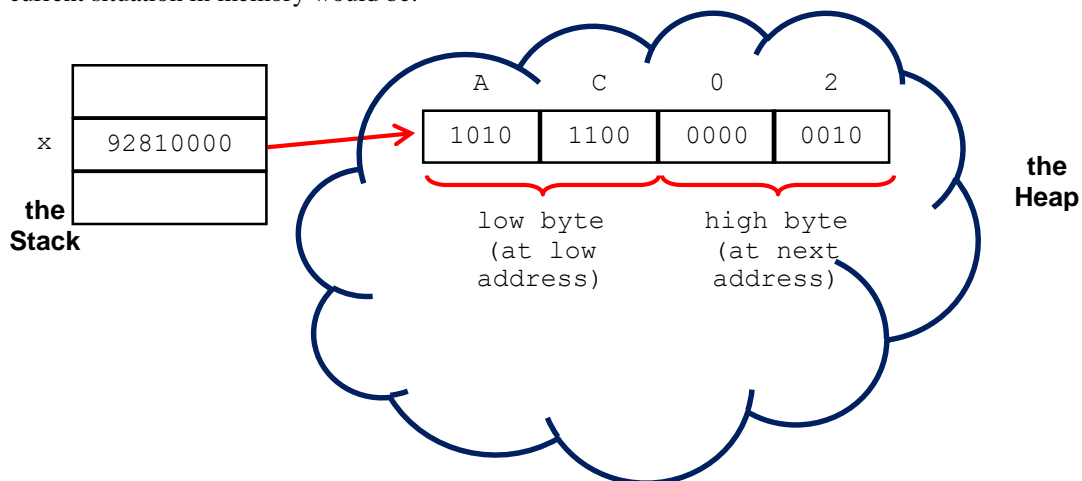


(The bytes are stored in little-endian order, just as they would be on any Intel-compatible system.)  If you dereference the pointer x, you'll obtain the contents of the two bytes beginning at the address stored in x.  That's because x was declared as a pointer to something of type uint16_t, and sizeof(uint16_t)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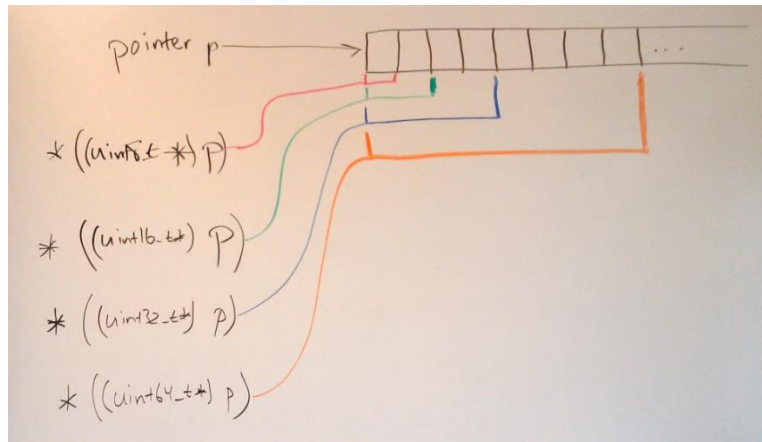


Note how we can use the type of a pointer to determine how many bytes of memory we obtain when we dereference that pointer, as well as how those bytes are interpreted. This can be really useful.

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.