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
17488F2818606FA956F5027115292273151850 6CB088C81A6597D853FFC79816
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:

```
/**  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 **`uint8_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 creating that limitation, 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
```

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

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

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

value 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.

Redesigning to Use Pointer Arithmetic

The function given earlier uses array syntax to manage the data region. An experienced C programmer would probably choose to implement the logic of that function using pointers. Here's what such an implementation might look like:

```
void showBytesAtIndex(FILE* Out, const uint8_t* const pBase,          // 1
                                 uint16_t Index, uint8_t nToPrint) {  // 2

   const uint8_t* currentPos = pBase + Index;                         // 3

   for (uint8_t pos = 0; pos < nToPrint; pos++) {                     // 4

      fprintf(Out, "  %02X", *currentPos);                           // 5

      currentPos++;                                                  // 6
   }

   fprintf(Out, "\n");                                               // 7
}
```

In line **1**, we use an explicit pointer to the beginning of the data region. That pointer can be used exactly as an array name, but we won't do that.

In line **3**, we create a new pointer and set its target to be the position in the data region at which the first byte we are supposed to print is located:

```
const uint8_t* currentPos = pBase + Index;
```

Recall that adding `K` to a pointer `P` yields the address of the byte that is a distance of `K * sizeof(*P)` from the target of `P`. If that's not clear to you, read the Appendix on pointers at the end of this specification.

Adding the value `Index` to `pBase` yields the address of the byte that is exactly `Index` bytes (forward) from the target of the pointer `pBase`, because the target of `pBase` is one byte in size.

So, the first byte we want to print is now the target of the pointer `currentPos`.

And, if we increment `currentPos`, it will then point to the next byte in memory. So, we can use the for loop shown above to print the specified bytes.

Aside: Efficiency

Does using a pointer-based approach necessarily lead to greater efficiency at runtime?

In short, no. At runtime, machine code is executed, not C code. When C code is translated into assembly code (which is them translated to machine code), there is no longer any notion of most of the high-level programming language concepts you have learned about.

Ultimately, machine code will operate on the basis of addresses and data values… in other words pointer-based accesses.

The Functions

For this assignment, you'll implement four 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 memory; any use of array bracket notation will result in a score of 0.

The first function you will implement involves copying a specified set of bytes from one memory region into another:

```
/**  Uses pointer-based logic to access a specified portion of a region of
 *   memory and copy bytes at an interval of skipSize to a supplied array.
 *   For example, if skipSize == 0, then consecutive bytes are copied, and
 *   if skipSize == 2 then every third byte is copied.
 *
 *   Pre:   pDest points to the beginning of a region of memory large
 *             enough to hold maxBytes offsets;
 *          blockAddress points to the first byte of the memory region;
 *          blockLength is the number of bytes in the memory region;
 *          startOffset is the location, relative to blockAddr, of the first
 *             relevant byte of the memory region;
 *          skipSize is the number of bytes between bytes to be copied
 *             (so copies consecutive bytes if skipSize == 0)
 *          maxBytes is the maximum number of bytes that are to be copied
 *
 *   Post:  the specified bytes have been copied, in order, into the memory
 *             region that is pointed to by pBytes;
 *          no changes have been made to any other memory locations
 *
 *   Returns:
 *          number of bytes that were copied
 *
 *   Restrictions:
 *      You must use only pointer syntax in accessing memory.
 */
uint32_t getScatteredBytes(uint8_t* const pDest,
                           const uint8_t* const blockAddress, uint32_t blockLength,
                           uint32_t startOffset, uint8_t skipSize, uint32_t maxBytes);
```

The interface of the function deserves a short discussion. The second parameter, `blockAddress`, 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 pointer itself (which would make it point to a different location in memory).

In this case, we don't want 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 `*blockAddress` is the first byte:

```
getScatteredBytes(..., blockAddress,  512,   0, 0,  7):  7F 10 22 EC 2B EA D1
getScatteredBytes(..., blockAddress,  512,   0, 3,  5):  7F 2B 4E 7A 9E
getScatteredBytes(..., blockAddress,  512, 488, 4, 30):  A2 71 7E 68 5A
```

In the third example, only 5 bytes are copied because we reached the end of the data block.

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 offset 488 would be `0x1E8`. 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 large enough to hold the results;
 *          blockAddress points to the first byte of the memory region;
 *          blockLength is the number of bytes in the memory region;
 *          Byte is the value to be searched for
 *          maxBytes is the maximum number of bytes that are to be copied
 *
 *   Post:  the file offsets of the matching bytes are copied, in order,
 *          into the array pointed to by pOffsets
 *
 *   Returns:  the number of occurrences of Byte found in the memory region
 *
 *   Restrictions:
 *      You must use only pointer syntax in accessing memory.
 */
uint32_t findOccurrencesOfByte(uint32_t* const pOffsets,
                               const uint8_t* const blockAddress, uint32_t blockLength,
                               uint8_t Byte, uint32_t maxBytes);
```

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

```
findOccurrencesOfByte(..., blockAddress, 512, 0x7F, 10):    0   36  184  192
findOccurrencesOfByte(..., blockAddress, 512, 0x00,  3):   85  12B  131
findOccurrencesOfByte(..., blockAddress, 512, 0xE9, 10):   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 extends the requirements for the second one:

```
/**  Uses pointer-based logic to search a specified portion of a region of
 *   memory for occurrences of a specified pattern (sequence of bytes).
 *
 *   Pre:  pOffsets points to an array (hopefully of sufficient dimension)
 *         blockAddress points to the first byte of the memory region
 *         blockLength is number of bytes in the memory region
 *         pPattern points to a copy of the pattern that is to be found
 *         patternLength is the number of bytes in the target pattern
 *
 *   Post: The offsets of occurrences of the given pattern have been stored
 *            in pOffsets, in ascending order, starting at index 0.
 *
 *   Returns:  the number of occurrences of the pattern found in the memory region
 *
 *   Restrictions:
 *      You must use only pointer syntax in accessing memory.
 */
uint32_t findOccurrencesOfPattern(uint32_t* const pOffsets,
                                  const uint8_t* const blockAddress, uint32_t blockLength,
                                  const uint8_t* const pPattern, uint8_t patternLength);
```

The interface of this function (deliberately) contains a serious flaw; the function has no information about the size of the array that `pOffsets` points to. Therefore, the function can only trust the caller to have provided a sufficiently large array to hold the results; if not, the caller is at fault and the function has no way to avoid any resulting error.

This "feature" is included simply to illustrate the perils of working with caller-supplied arrays. Of course, if the caller specifies the wrong dimension for the array when calling either of the earlier functions, the same issue arises.

As for functionality, this extends the idea behind the second function by searching in memory for a specific pattern of bytes, rather than just a single byte. Viewed properly, this merely requires making one modification to the logic of the second function (perhaps with the use of a helper function).

Here are some sample results, with a different data block:

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	33	CF	AB	79	5C	4E	DB	26	DA	4E	31	E4	97	84	AA	20
1	85	89	43	E1	65	5A	F4	B8	FD	D2	CF	8D	3B	98	69	6E
2	67	14	E7	C4	62	C3	EA	3C	11	1B	20	A8	9F	CA	9F	CA
3	9F	CA	06	B9	66	FA	71	63	CC	41	F1	07	D9	5A	75	41
4	6E	5D	05	D1	20	EF	0D	31	0A	2E	DA	A9	F8	A2	CD	4C
5	AE	D3	05	14	CD	77	77	9A	B8	68	A1	91	C2	17	D2	31
6	74	D7	02	94	C6	0F	C5	D0	3D	9F	79	36	42	47	82	F0
7	1A	88	04	E8	FF	7B	82	B7	E4	23	48	A6	3A	1B	D7	AE
8	F2	D9	42	4A	DF	D9	64	67	DD	4C	66	E9	4A	DF	D9	64
9	67	DD	4C	66	59	CE	1D	3D	F2	65	E3	2C	80	BB	DB	CA
A	C8	24	54	7D	25	B5	A4	CD	B8	00	B6	02	DF	90	67	46
B	6D	B3	AC	C6	82	C9	03	74	2F	E7	A0	C3	EA	66	FA	71
C	63	4E	B4	BE	04	58	8B	BC	58	42	BF	38	D2	26	7E	3F
D	D9	2B	06	5B	F4	09	CF	23	F0	70	D3	92	EB	F5	C9	84
E	44	7D	43	48	D5	CE	04	2D	10	C3	65	E2	E9	E4	22	C3
F	0F	28	1E	03	31	EE	27	22	5E	FA	B4	49	EF	7D	CE	33

C3 EA: 25 BB

4A DF D9 64 67 DD 4C 66: 83 8C

22 5E FA B4 49 EF 7D CE 33 FF FF: no occurrences

Be warned:

- the pattern may or may not occur in the given data block
- the pattern may occur more than once in the data block
- occurrences of the pattern might overlap (not illustrated in the previous example, but look there for occurrences of the pattern `9F CA 9F CA`)
- searches for the pattern must not go outside the boundaries of the given data block

The given test code is randomizing the test data, so some cases may occur less frequently than others. Be sure to test thoroughly.

The fourth function requires summing values in a sequence of bytes, with a bit of a twist:

```
/**  Interprets the bytes of a given sequence as int16_t values and sums them.
 *
 *   Pre:  blockAddr points to the first byte of the memory region
 *         blockLength is number of bytes in the memory region
 *         sequenceOffset is the offset of the first byte of the sequence that
 *           is to be summed
*         sequenceLength is the number of bytes in the sequence, and
 *           sequenceLength > 0
 *
 *   Returns:  the sum of the int16_t values obtained from the given sequence;
 *             if the sequence contains an odd number of bytes, the final byte
 *             is ignored; return zero if there are no bytes to sum
 *
 *   Restrictions:
 *      You must use only pointer syntax in accessing memory.
 */
int16_t sumSequence16(const uint8_t* const blockAddress, uint32_t blockLength,
                      uint32_t sequenceOffset, uint8_t sequenceLength);
```

Parameters `blockAddress` and `sequenceOffset` specify the first byte in a sequence of bytes; the parameter `sequenceLength` specifies how many bytes are in that sequence. Starting with the first by the sequence, you must interpret each successive pair of bytes as a signed 16-bit integer (**`int16_t`**), and compute the sum of those integers.

While the earlier functions required you to make use of pointer arithmetic, this one is best accomplished if you also make use of pointer typecasts. See the Appendix for examples.

Consider the data segment below:

```
     0  1  2  3  4  5  6  7     8  9  A  B  C  D  E  F
0   56 11 7B 8C EF A7 4A 0B    05 C8 44 E9 A2 9C A4 C6
1   44 A5 A0 94 8E B3 BC F7    95 40 CB 61 FE 01 D7 54
2   13 53 E0 02 FA 2B 0E 00    F3 52 E9 95 EE 8E 5B 32
3   33 FB C6 C1 AF 82 B8 44    C3 84 A5 C1 85 7C 16 98
4   CF F6 9B CA 21 A9 CA 14    FB B3 AA E9 41 05 1B 75
5   01 E1 36 B0 63 EF F4 26    73 99 E8 F8 15 FE 91 E5
6   F4 2C AF 16 D5 79 2A D0    2C D4 B9 6E DA D4 E3 DB
7   B5 19 8B 18 08 7F 3F 7B    18 27 74 2D 25 05 12 19
8   31 C1 2F 06 3A 5A D6 67    2E 8F D5 08 63 B8 E3 18
9   D1 6E 30 DA ED 6F 55 05    96 C9 13 53 E0 02 D5 FF
A   07 04 05 41 5E DB A8 8D    6A 7D 95 CD FB C6 E5 D1
B   6E 30 DA ED 6F 55 05 1C    00 0D D7 CE 53 AC AF 82
C   B8 44 9B 0F AF 44 9C 19    C1 32 E7 F7 AB CC FE 92
D   E2 DF 67 68 15 42 84 15    4F 5B E4 A2 08 B2 FC B9
E   85 98 C8 34 DC 65 4E 9D    97 35 94 42 01 92 D4 E4
F   71 3C 4C 87 7E D0 9C CD    2B 80 70 33 32 6C EC B8
```

If we were asked to sum the 17 bytes starting at offset 63_{10}, we'd discover that 63_{10} is 0x3F, and the relevant bytes are:

```
98  CF  F6  9B  CA  21  A9  CA  14  FB  B3  AA  E9  41  05  1B  75
```

From there, the successive pairs of bytes yield the following **`int16_t`** values (in base 10):

```
-12392  -25610    8650  -13655   -1260  -21837   16873    6917
```

Remember that multi-byte integer values are stored in little-endian format. Here, the final byte is ignored, and the sum would be 23222.

Testing

Download `c05Files.tar` from the course website and unpack in a CentOS 7 directory; you will find these files:

`c05driver.c`	driver for testing/grading code
`PtrFuncs.h`	declarations for functions specified in this assignment
`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
`README`	instructions for compilation and testing

You must create a C source file `PtrFuncs.c` to hold your implementations of the specified functions.

You can compile the program with the following command:

```
CentOS> gcc -o c05driver -std=c11 -Wall -W c05driver.c PtrFuncs.c TestCode.o Generator.o
```

Executing the test driver with the `-help` switch provides some instructions:

```
centos> c05 -help
Driver for testing PtrFuncs.c

Invocation:  c05 [-all | -copybytes | -findbyte | -findpattern] [-sum16] [-repeat]
Examples:    c05 -scatteredbytes
             c05 -findbyte -repeat
             c05 -all

Switches:
   -help          display invocation syntax
   (none)         run all tests, randomize testing
   -all           run tests for all three functions
   -copybytes     run test only for getScatteredBytes()
   -findbyte      run test only for findOccurrencesOfByte()
   -findpattern   run test only for findOccurrencesOfPattern()
   -sum16         run test only for sumSequence16()
   -repeat        repeat with previous test data

Note: -repeat is only guaranteed to repeat the previous test data
         if the other switches are kept the same
```

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. 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.

Each function test produces a separate output file showing testing details. The test driver also produces a summary file, `c05Results.txt`, that shows overall score information.

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
5.00	Oct 15		Base document.
5.10	Oct 18	8	Corrected hex representation of the offset, and the base-10 representations of the values being added.

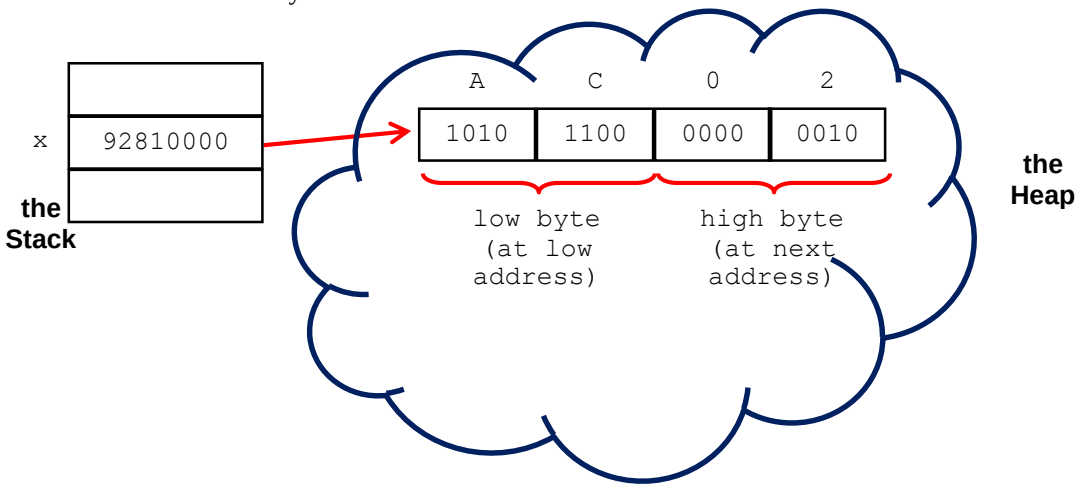
Appendix I: Brief tutorial on using pointer arithmetic and pointer typecasts in C

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:

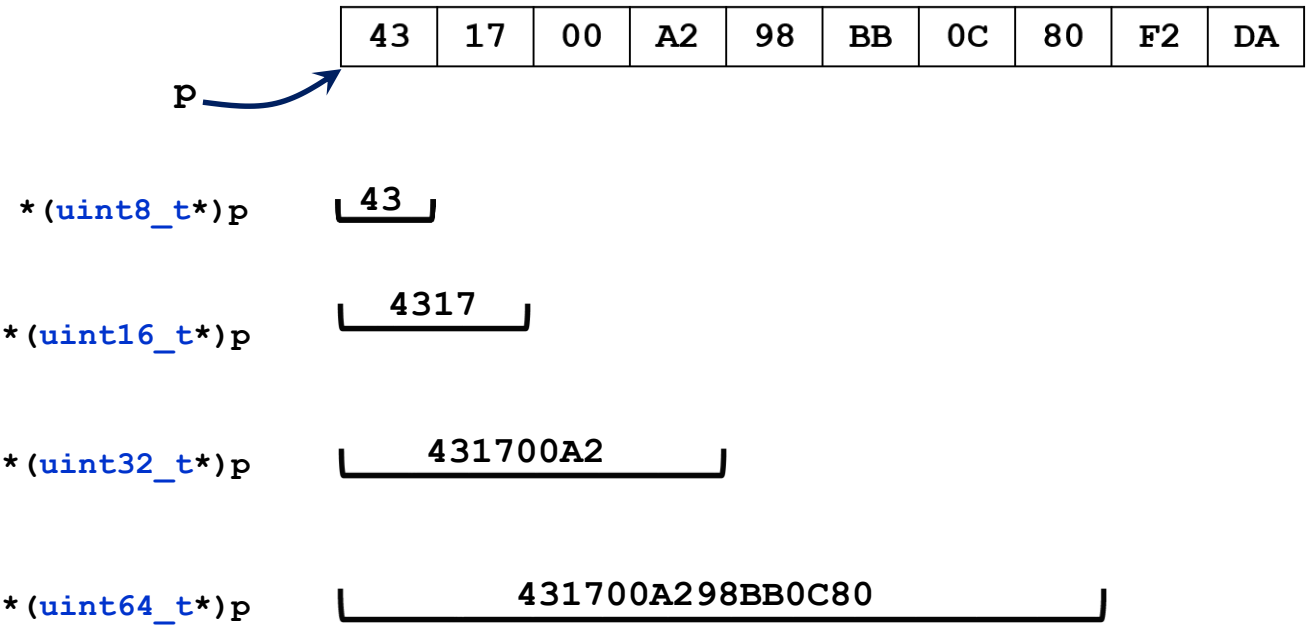
```
uint8_t *y = NULL;                  // 3:  y == 0x00000000
y = (uint8_t*) x;                   // 4:  y == 0x00008192

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.