Laboratory Exercise 11

Multi-dimensional arrays of structures

Purpose of the exercise

This exercise will help you do the following:

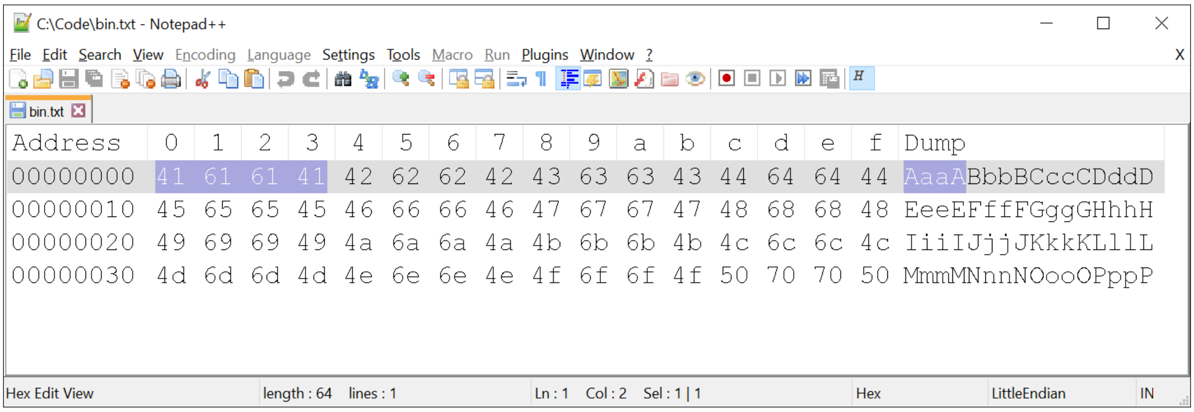
- Practice working with 2D arrays of structures.
- Strengthen your understanding of objects' memory layout.

Overview

Binary data

When we talk about data in a binary form, we generally mean data as a sequence of binary digits, bits, and forgo its abstraction as data types. As it is more convenient to manipulate bits in groups rather than as long strings of individual bits, we usually shorten their notation to *octal* or - more commonly - *hexadecimal* digits that group, respectively, every consecutive 3 or 4 bits into a digit symbol.

A hex editor is a program that allows direct manipulation of binary data in files. Most hex editors display binary data in hexadecimal form and their ASCII counterparts, side-by-side:



When we see a sequence of binary data, we are able to meaningfully interpret it only if we understand what is its in-memory data type representation and ordering. This is the focus of this laboratory exercise.

Your task is to implement a generic function `print_data()` that takes in the address of an object of any type, its size, with the number of hexadecimal digits per line of output, and prints out to the console with a layout similar to that of a hex editor interface.

The main driver program will pass to this function an array object of array objects (`Data`) of structures (`Datum`). Your task is twofold:

- Firstly, write the code that displays the object in the right format to the output stream.
- Secondly, study the output and use it to analyze 5 commented out statements that test your understanding of memory addressing.

You only have to submit the code from the first part, but the knowledge from the second step will be vital to your work as a programmer, and to your success in the final test of the course.

Expected output of the program:

```
C:\Code>main
41 61 61 41   42 62 62 42   43 63 63 43   44 64 64 44   |   A a a A   B b b B   C c c C   D d d D
45 65 65 45   46 66 66 46   47 67 67 47   48 68 68 48   |   E e e E   F f f F   G g g G   H h h H
49 69 69 49   4a 6a 6a 4a   4b 6b 6b 4b   4c 6c 6c 4c   |   I i i I   J j j J   K k k K   L l l L
4d 6d 6d 4d   4e 6e 6e 4e   4f 6f 6f 4f   50 70 70 50   |   M m m M   N n n N   O o o O   P p p P

41 61 61 41   42 62 62 42   |   A a a A   B b b B
43 63 63 43   44 64 64 44   |   C c c C   D d d D
45 65 65 45   46 66 66 46   |   E e e E   F f f F
47 67 67 47   48 68 68 48   |   G g g G   H h h H
49 69 69 49   4a 6a 6a 4a   |   I i i I   J j j J
4b 6b 6b 4b   4c 6c 6c 4c   |   K k k K   L l l L
4d 6d 6d 4d   4e 6e 6e 4e   |   M m m M   N n n N
4f 6f 6f 4f   50 70 70 50   |   O o o O   P p p P

41 61 61 41   |   A a a A
42 62 62 42   |   B b b B
43 63 63 43   |   C c c C
44 64 64 44   |   D d d D
45 65 65 45   |   E e e E
46 66 66 46   |   F f f F
47 67 67 47   |   G g g G
48 68 68 48   |   H h h H
49 69 69 49   |   I i i I
4a 6a 6a 4a   |   J j j J
4b 6b 6b 4b   |   K k k K
4c 6c 6c 4c   |   L l l L
4d 6d 6d 4d   |   M m m M
4e 6e 6e 4e   |   N n n N
4f 6f 6f 4f   |   O o o O
50 70 70 50   |   P p p P

C:\Code>_
```

`const`-ness and `const`-correctness

Programmers of C and C++ languages pay a lot of attention to making sure each object is clearly indicated as *mutable* (that means it can be changed; it is modifiable) or *immutable* (that means it is read-only). Immutability offers a lot of benefits, from clearer communication about the code with other programmers, through making implicit promises about object responsibility (i.e. to indicate if a function parameter passed by pointer is an immutable input parameter or mutable output parameter), to giving a compiler opportunities for code optimization. The most important keyword in this aspect is `const`.

The term `const`-ness refers to declaring objects as immutable, constant. An object marked as `const` cannot be changed after its initialization. This explanation may be straightforward for primitive object types, but when working with pointers, it gets more complex as there are two levels of `const`-ness:

- **Top-level `const`-ness** means a pointer is constant and cannot point at a different place in memory. However, through this pointer you can still modify the object at that address. The following declaration shows a top-level constant pointer `pi`:

```
int i = 10;
int* const pi = &i;
*pi = 20;    // This is a legal operation.
pi = NULL;   // NC! This is NOT a legal operation.
```

- **Low-level `const`-ness** means a pointer is pointing at constant objects; it can be changed to point at another constant object in memory, but when it is dereferenced it returns a constant object. The following declaration shows a low-level constant pointer `pi`:

```
const int i = 10;
const int j = 20;
const int* pi = &i;
pi = &j;     // This is a legal operation.
*pi = 30;    // NC! This is NOT a legal operation.
```

A pointer can use top-level `const`-ness, low-level `const`-ness, or both. The following declaration shows a pointer that is both top-level and low-level constant:

```
const int i = 10;
const int* const pi = &i;
```

It may be convenient to read pointer declarations from right-to-left:

> A variable `pi` is a constant (top-level) pointer to `int` constant (low-level).

The term `const`-correctness is a stricter application of `const`-ness used when new objects are initialized with values from existing objects: if in a context where the new object is used it can be declared immutable, it should be marked as `const` even if the existing object allows for modification. For example, if a function only reads data from its formal parameters, they should be marked constant even if in a call mutable objects are being passed. Likewise, if a function returns a mutable object that gets assigned to a variable that will only be accessed for reading, this variable should be `const`. In other words, the code should be as strictly `const` as possible.

Tasks

In this exercise you will implement the following single function inside *q.c* file:

```
void print_data(
    const void* ptr,     // Address of the first byte.
    size_t size,         // Total count of bytes.
    size_t span          // Count of bytes printed per line.
);
```

While implementing this exercise you have to follow these constraints in your implementation:

- It must be **generic** - it must be able to print any object: any block of memory that is fully accessible up to its given length.
- It must strictly observe **`const`-ness and `const`-correctness** for all parameters and local variables; no `static`, dynamic or file-scoped variables are allowed.
- It must print hexadecimal data byte-by-byte always with 2 hexadecimal digits per byte; it must print ASCII data byte-by-byte always with 1 character per byte.
- It must not include any other headers but `<stdio.h>` and `"q.h"`.

Step 1. Prepare your environment

Open your WSL Linux environment, prepare an empty *sandbox* directory where the development will take place, save the provided text files into this directory. Then create a new file *q.c* and open it for editing.

Step 2. Review expected output file

Open the provided output file and make sure you understand exactly how it is formatted. Also, study the main driver *main.c* used to generate this file to make sure that you know what objects' memory is represented by the file, what is its layout.

Consider the following aspects:

- How does the endianness of a machine impact the byte sequence (the code must be tested with a little-endian 64-bit x86 processor)?

- Which array is the inner array that contains `struct` objects, and which array is the outer array with array elements?
- Will there be any padding included in `struct` objects' layout?
- What is a storage sequence of data members of a `struct` and elements of an array?

Step 3. Implement *q.c*

Provide a definition of the function `print_data()` as declared in the header *q.h*. Take note of the constraints listed above that **must** be observed.

Step 4. Compile the code

Now it is the time to check the program. Compile and test the code:

```
gcc -Wall -Werror -Wextra -Wconversion -Wstrict-prototypes -pedantic-errors -std=c11 -o main main.c q.c
./main > actual-output.txt
diff actual-output.txt expected-output.txt --strip-trailing-cr
```

Work on the code until there are no differences between the actual and the expected outputs. Make sure that the files match **exactly**.

Step 5. Add file-level documentation

Add the file-level documentation to *q.c*. Remember that every source code file you submit for grading must start with an updated file-level documentation header.

Step 6. Add function-level documentation

Add the function-level documentation to the function in *q.c*. Remember that every function in every source code file that you submit for grading must be preceded by a function-level documentation header that explains its purpose, inputs, outputs, side effects, and considerations.

Step 7. Clean-up the code

Clean up the formatting, make sure it is consistent and easy to read. Break long lines of code; a common guideline is that no line should be longer than 80 characters to fit within the printable width of a page.

Step 8. Test the code

Once again before submission test the code. Your actual output must exactly match the contents of the expected output. Use the *diff* command to compare the files; no differences in the output should be reported.

Step 9. Analyze code examples

In the main driver *main.c* there are 5 commented-out lines testing your understanding of memory layout, arrays, structures and pointer arithmetic. Study these examples, use the generated output to figure out the expected output from each statement.

Only then enable the code and check if your answers are correct.

Submission

Once your implementation of *q.c* is complete, again ensure that the program works and that it contains updated file-level and function-level documentation comments.

Then upload the files to the laboratory submission page in *Moodle*. There are 31 lines of expected output and 31 corresponding automated test cases. To get the maximum grade, make sure that all test case output matches.

If you're done early and are bored...

Try to implement a *makefile* build script for this program. Do not submit this *makefile*!