UNIVERSITY OF CALIFORNIA, MERCED

BERKELEY · DAVIS · IRVINE · LOS ANGELES · MERCED · RIVERSIDE · SAN DIEGO · SAN FRANCISCO

SANTA BARBARA · SANTA CRUZ

CSE30: Data Structures
Lab 2

Due Date: Sunday, September 20, 11:59 pm

1 Introduction

In this lab, we will be exploring some concepts related to pointers and the way things are stored in the memory of the computer.

2 Integer Representation in Memory

As we have seen in lectures, the memory of the computer is divided into chunks of 1 byte, and each such location has an address. In C++, the `int` type is 4 bytes (you can verify this by executing `sizeof(int)`), so it takes 4 consecutive memory locations to represent a single integer (That bit about consecutiveness is really important).

As we know, 1 byte is 8 bits, which basically means that we can write eight 0s and 1s inside a single memory address. With 8 bits at our disposal, we can represent $2^8 = 256$ different integers, so we can fit any integer value from $0 \ldots 255$ inside a single memory location. This is why a `char` is one byte big, because it needs to be able to represent any ASCII character, of which there are 265, labelled from 0 to 255.

What happens if we need to store an integer larger than 255. It does not fit in a single memory location, so we will need to use one of the other "boxes" to store part of the integer.

Assume we wish to store the integer 314. It is bigger than 255 so we know we need more than one block of memory (it will be two in this case), but what goes in each box.

The easiest way to illustrate that is to represent the number 314 as binary. There is a large number of converters available online, which a simple search engine query reveals. A random converter that I stumbled upon gave me 100111010 as the binary representation of 314. As you can see, the binary number we produced is 9 digits long (or 9 bits) so it does not fit in a single memory location. To fit it into two memory locations, we first add as many zeros to the front (that does not change the value) as are necessary to make it 16 bits. So we have 0000000100111010, and we take the first 8 bits and store them in a memory location, and then the other 8 bits in another memory location. In our example 00111010 will be stored in its own box, and 00000001 in a different box. You may remember that we have 4 boxes in total, and the number 314 only needed 2 boxes, so the remaining 2 unused boxes will get 00000000 (eight zeros) in them. In summary, the binary representation of 314 can be expressed as 4 groups of 8 bits, like this:

$$314 = 00000000\ 00000000\ 00000001\ 00111010$$

Now, you might say that this is a lot of digits to write just for that, and you would be correct. That is why in real life, computers do not store 0s and 1s in their memory. They actually store things represented in hexadecimal. Conveniently, it only takes 2 hexadecimal symbols to represent 8 bits. Recall, with 8 bits we can represent $2^8 = 256$ things. That is because we have 2 choices for the first symbol, 2 choices for the second symbol, and so on until the eight symbol, for which we have 2 choices. By the multiplicative counting principle (which is covered in numerous classes at UC Merced), we have $2^8 = 256$. Now, with hexadecimal notation we have 16 choices for each symbol, so if we have 2 symbols, that's $16^2 = 256$ choices. Long story short, 2 hexadecimal digits can represent any number from $0 \ldots 255$.

Let us use that "space saving technique" and represent our integer as 4 groups of 2 hexadecimal digits, instead of 4 groups of 8 binary digits.

You can use a converter you find online, or you can use the program in the scratchpad of this lab in order to convert between decimal and hexadecimal representations (the program in the scratchpad is described in great detail in the next section). The hexadecimal representation of 314 is 13A, which is 3 digits long. We need it to be 8 digits long (so that we can split it into 2 groups of 2 digits), so we add 5 zeros to the front, to get $314_{10} = 0000013A_{16}$. Splitting that up into groups as before, we get:

$$314 = 00\ 00\ 01\ 3A$$

Now that we have the number 314 represented as a group of 4 hexadecimal numbers, it is time to put it into our memory blocks. The only thing to remember is that, since our CPUs use the Little Endian representation, the groups displayed above are inserted into the memory in reverse order. The memory contents that represent the integer 314 are presented in Figure 1.

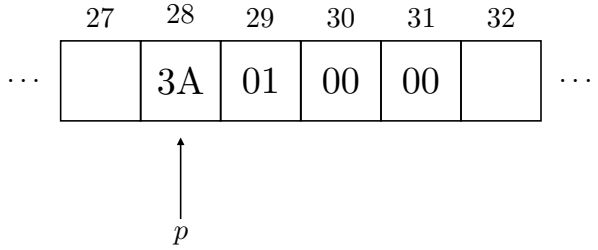

Figure 1: An example memory fragment containing the integer 314

3 Converter Utility in Scratchpad

The utility in the scratchpad is a simple C++ program that allows the user to convert between decimal and hexadecimal representations. These operations will be useful in completing this lab. A simple search with your favorite search engine will get you several lines of code that does conversion between decimal and hexadecimal representations. The program in the scratchpad does that too, but it is also a complete program that is ready to be used widely. The reason is that it incorporates input validation, as well as error checking and handling. That is what the `try` and `catch` statements are for. In practice, when a user is converting decimal to hexadecimal and vice-versa, the user might provide invalid input (for example the word `"HELLO"` instead of an integer). In such cases the pieces of code that you find online will crash (most of them anyway), as nobody has bothered to do input validation and error checking and handling. This is

basically one of the differences between real software and just some low-quality code someone posted on the internet, because we can not always count on the users to do the right thing. I encourage you to study the code in detail so you can get a sense of what is happening with the `try` and `catch` blocks.

The program also works with command line arguments (because after all it is a command line utility). To use it, we first have to compile it on our projects. This is done with the `make scratch` command.

After successful compilation, you can run the program as many times as you want/need. When you run it in the terminal, you need to provide two arguments as well. The first one is the operation to perform and the second one is the value we are to perform it on. The two possible operations are `d2h` and `h2d`, which are Decimal-To-Hexadecimal, and Hexadecimal-To-Decimal, respectively. When the `d2h` operation is chosen, a valid integer is required as the second argument, and when the `h2d` operation is chosen, a valid hexadecimal number is required.

For example, to get the hexadecimal representation of the number 314, we run the command:

```
./bin/scratch d2h 314
```

The program produces the result, 13A.

Once again, you are encouraged to study the code in `scratchpad.cpp` very carefully so you can see an example of how to use `try-catch` statements, as well as writing programs that take in command line arguments.

4 Pointers in C++

The tutorial found at `http://www.cplusplus.com/doc/tutorial/pointers/` has all the information we covered in class, plus some additional examples. To recap some important stuff, we saw the `&` operator, which tells us the address of things in memory. If we have an `int`, we can find its address and store it in an `int` pointer. The code below does this.

```
int x = 7;

int* p = &x;
```

We now have the pointer `p` pointing to the first of the 4 boxes that the variable `x` occupies. Recall that with pointer arithmetic, we can add and subtract integer values from the pointer `p`. If we add 1 to `p` (which is `(p+1)`), it will point to the next integer over to the right, or it will move 4 boxes to the right. It skips that much because it is a pointer to `int`. A pointer to `long` moves by 8 spaces, and a pointer to `char` moves by 1 space.

So far we have seen how to make an `int` pointer point to the start of an integer variable, but can we make a different pointer type point there, for example a `char` pointer. It turns out that we can, by using typecasting. The only thing to point out is that we do not want to make a `char` pointer point to the beginning of an `int`. We want to use an `unsigned char` pointer. Why is that? Remember that every box of an integer can contain a value from $0 \ldots 255$, but a `char` can represent values from $-128 \ldots 127$, which does not match, but an `unsigned char` on the other hand is exactly what we need because it takes on values from $0 \ldots 255$. So to modify our code example above:

```
int x = 7;

int* p = &x; // So far it's the same

unsigned char* c = (unsigned char*) p;
// Typecasting p to an unsigned char pointer
```

Now we have a pointer `c` that points to the first box of the integer `x`.

The last thing we need to recall is the dereference operator, which lets us read the actual value stored in the memory location that a pointer is pointing to. For example, if we wish to read out the value stored in the location that `c` is pointing to, we need to use the dereference operator, like this:

```
cout << *c << endl;
```

The above code just prints the value stored at the location pointed to by `c`, but we can do other things with it, instead of just print it. It may also be necessary to typecast the value we are reading out because we may be expecting it to be an integer, but the compiler will read an unsigned char. To amend the code, we write:

```
cout << (int) *c << endl;
```

Now it prints the contents represented as an `int`. As mentioned earlier, you may want to store the contents in a variable, instead of just printing them with `cout`.

5 Tasks

There is a fair amount of code already written in this project. If you study the code in `scratchpad.cpp`, you will notice that it uses one of the functions defined in `MemoryStuff.h`, namely the `decToHex` function. This function has been unit tested in the `DecimalToHexTest` context in `test.cpp`. There are currently 2 test cases there.

Task 1 Do you think the two specs (test cases) in the `DecimalToHexTest` context are enough to convince us of the correctness of the `decToHex` function? If you believe they are enough, explain why you think so. Write your answer as a comment inside the scope of the `DecimalToHexTest` context. If you believe they are not enough, provide additional specs in the `DecimalToHexTest` context.

Task 2 Before attempting Task 3, complete the `TwoThousandTwenty_What` spec, which is found in the `MemoryContentFunctionTest` context. You have to provide the correct contents for each box of memory that is used to represent the number 2020. Use a colon (:) to separate the contents of each box.

Task 3 Implement the `std::string memoryContents(int)` function, which is found in `MemoryStuff.h`. The function basically verifies that our CPUs are indeed using Little Endian (remember the thing about the boxes being in reverse order). As such, your function should do the following:

1. Get a pointer to the location where the `int` value `x` is stored. The variable `x` occupies 4 boxes.
2. Get the value of the current box and save it as a hexadecimal representation. Basically a string that has the hex values.
3. Move your pointer to the next box, and repeat the process again.

When you have read the contents of all 4 boxes, display the result as a string, with the following format:

```
box1:box2:box3:box4
```

Basically, the contents of every box, separated by a colon symbol (:).

Task 4 Unit test your implementation from Task 3. There are 2 specs in the `MemoryContentFunctionTest` context, but this time we are telling you that that is not enough to be sure of the correctness of the function. Write additional specs in such a way that we can be assured of the correctness of a `memoryContents` function that passes all your tests.

Task 5 Create a command line application in your `app.cpp` where it will ask the user to enter an integer, and it will display the memory contents for the given integer. You should use your `memoryContents` function from `MemoryStuff.h`

6 Assessment

This lab is out of 100 points. They will be awarded as follows:

- 15 points for Task 1
- 15 points for Task 2
- 50 points for Task 3
- 15 points for Task 4
- 5 points for Task 5