Computer Systems Principles

Bits and Bytes

Contents

Overview

This project assignment will help you understand: how to compile C programs, information representation, and bitwise operators. You should complete all the exercises below using a text editor of your choice. Make sure you follow the instructions exactly. The actual code you write is fairly short; however, the details are quite precise. Programming errors often result from slight differences that are hard to detect. So be careful and understand exactly what the exercises are asking you to do.

Testing

We have provided tests that you can run to see if you have the correct solution. We will also be running additional private tests that will scrutinize your submissions even further. After we run the public and private tests, each submission will be assessed by others to check that you are not tricking the auto-grading facility. Make sure you review the academic honesty policy on the course website and what is included as part of this project's documentation toward the end.

Suggested Reading

You should do the reading for *this week*. It will also be useful to read up on printf to get a better understanding of how it is used and the different formatting options that are available. You can also look at the documentation directly from the command line using the `man` command like so:

```
$ man 3 printf
```

What does the `3` mean? It indicates which "manual section" you want to look in. Take a look at this to see which section `3` refers to as well as other manual sections that you have access to.

Part 0: Project Startup

Please download the **project startup tarball**. If you do not know what a "tarball" is you should read up on it. To do this, open a web browser by clicking on the left-most menu button followed by "Internet". We have installed Firefox and Chromium for you to use. You should do this within your virtual machine so you can download the tarball directly to your virtual machine disk. You should download the tarball to your home directory. By default, Chromium will download to the "Downloads" folder. You can change this by clicking on "student" in the save dialog box. Alternatively, you can download to the "Downloads" folder and then run the following Unix command from the terminal inside your home directory:

```
$ mv Downloads/bits-and-bytes-proj-student.tgz .
```

Remember, the `.` represents your current working directory, in this case your home directory. This command will move the tarball to your home directory. You can verify this using the `ls` command. Once you have the tarball in your home directory you can execute the following command:

```
$ tar xzvf bits-and-bytes-proj-student.tgz
```

This will *unarchive* the contents of the tarball and you will see the `bits-and-bytes-proj` directory. You can then go into that directory from the terminal using `cd` (change directory):

```
$ cd bits-and-bytes-proj-student
```

Part 1: Implement A Simple C program

The first part of this assignment is to write a simple C program, compile it, and run it. Type the following code into a file called `print-it.c` in your `bits-and-bytes-proj` directory:

```c
#include <stdio.h>

int main() {
  printf("Bit operations are fun!\n");
}
```

Now compile the C program like this:

```
$ gcc print-it.c -o print-it
```

And execute the resulting *executable object file* (binary) like this:

```
$ ./print-it
Bit operations are fun!
```

Run the test script `test/test01` to see how well you did:

```
$ source test/test01
```

Part 2: Print Integers Like a Champ!

In this part we will play around with the C `printf` function to get a handle on how it is used. Create a C file called `print-int.c` in your in your `bits-and-bytes-proj` directory by typing the following:

```c
#include <stdio.h>

int main() {
  int x = 10;
  int y = 13;

  printf("x = %d\n", x);
  printf("y = %d\n", y);
}
```

This C program declares two *signed* integers and prints out their corresponding base-10 (decimal) values. Go ahead, compile `print-int.c` with `gcc` and name the resulting binary file `print-int`. Run it to get the following output:

```
$ ./print-int
x = 10
y = 13
```

Do you know how many bits are contained in an `int`? Update your `print-int.c` file to use the `sizeof` operator to compute the size of an `int`. Do you forget how to use `sizeof`? If so, briefly read this introduction. Add an additional `printf` after the last statement to print out the size. The output of your program should look like this:

```
$ ./print-int
x = 10
y = 13
size of signed int in bytes is 4.
```

Note that the 4 is the number of bytes, not the number of bits. How would you get the number of bits? Modify your `print-int.c` file to output the number of bits. Compile and run to get the following output:

```
$ ./print-int
x = 10
y = 13
size of signed int in bytes is 4.
size of signed int in bits is 32.
```

Lastly, update your `print-int.c` program to add the two integers together and print the result. Compile and run to get the following output:

```
$ ./print-int
x = 10
y = 13
size of signed int in bytes is 4.
size of signed int in bits is 32.
10 + 13 = 23.
```

Run the test script `test/test02` to see how well you did:

```
$ source test/test02
```

Part 3: Print Floats Like a Champ!

In this part we will play with floating-point values. Create the C file `print-float.c` and write a program that does exactly the same as the program in Part 2. However, this program must use the `float` data type. Compile and run your program to get the following output (name your executable `print-float`):

```
$ ./print-float
x = 10.000000
y = 13.000000
size of single float in bytes is 4.
size of single float in bits is 32.
10.000000 + 13.000000 = 23.000000.
```

Make sure you use the correct `printf` format character to get *exactly* the same output as our program does!

Next, modify your program to cast the result of the floating-point addition to an integer. Compiler and run your program to get the following output:

```
$ ./print-float
x = 10.000000
y = 13.000000
size of single float in bytes is 4.
size of single float in bits is 32.
10.000000 + 13.000000 = 23.000000.
10.000000 + 13.000000 = 23.
```

Run the test script `test/test03` to see how well you did:

```
$ source test/test03
```

Part 4: Print Characters Like a Champ!

Now we will have some fun with characters. Create a new file called `print-char.c` and add the following C code:

```c
#include <stdio.h>

int main() {
  char c = 'C';
  char a = 65;

  printf("c = %c\n", c);
  printf("a = %c\n", a);
}
```

Compile and run this program to get the following output (name your executable `print-char`):

```
$ ./print-char
c = C
a = A
```

Next, modify your program by adding additional variables for other characters. These characters should allow you to produce the following output after you compile and execute your program:

```
$ ./print-char
c = C
a = A
CAFEBABE
```

So, where does the `CAFEBABE` reference come from? It turns out not to be as "random" as it might be - or is it? Take a look at the following to understand the historical context of this name:

- Why CAFEBABE?
- Java Class File Magic Number
- Hexspeak
- Cafe Babe? OR, what's in a name?

Now modify your program to print the number of bytes representing the character string `"CAFEBABE"`. Compile and run your program to produce the following output:

```
$ ./print-char
c = C
a = A
CAFEBABE
number of bytes: ?.
```

Where `?` is the number of bytes you computed in your program.

Run the test script `test/test04` to see how well you did:

```
$ source test/test04
```

Part 5: Packing Up The Bytes!

We learned that a character is represented by 8 bits, also known as a byte. We also learned that integers have potentially different sizes on different machines. We are programming in a 32-bit Linux/x86 environment, so the integers on this machine are represented in 32 bits or 4 bytes. Consider the start of the following C program:

```c
#include <stdio.h>

int main() {
  unsigned char b3 = 202;
  unsigned char b2 = 254;
  unsigned char b1 = 186;
  unsigned char b0 = 190;
}
```

Note that we are using an *unsigned* representation of a character. An unsigned character can represent the values 0-255. Your job is to use bitwise operators to copy the bytes b3-b0 into a corresponding unsigned integer called `u`. Your unsigned integer should place the bytes in the following spot in the integer `u`:

```
  [      b3     ][      b2     ][      b1     ][      b0     ]
 31             24             16              8              0
```

The numbers 31, 24, 16, 8, and 0 indicate the starting bit of each byte in the integer. Note that we begin on the right and count bits toward the left. This is known as little-endian representation and corresponds to the bit positions used by the powers of 2 to translate a number represented in bits to its decimal equivalent. In particular, little-endian means that the low-order byte of a value is stored at the lowest byte in memory. In this case, the "memory" is an integer represented as 4 bytes starting with the first byte (b0), starting at bit 0, and ending with the last byte (b3), starting a bit 24.

Extend the above C program in a new C file called `packing-bytes.c`. This new program must assign the bits in each of the unsigned characters b3-b0 into their corresponding place in the unsigned integer `u`. You should use only the bitwise operators for *left shift* (`<<`) and *or* (`|`). Print the resulting value stored in `u` as a hexidecimal value using `printf` (you can use the `%X` formatting option - see the man page for details). If you do this correctly your program should output the following after it is compiled and executed (name the executable file `packing-bytes`):

```
$ ./packing-bytes
????????
```

Where each of the `?` characters corresponds to a hexadecimal digit. Your output must be exactly 8 hexadecimal digits and must be exact. Note that the code to do this is rather short.

Run the test script `test/test05` to see how well you did:

```
$ source test/test05
```

Part 6: Unpacking The Bytes!

In the last part we were "packing" bytes into an integer. In this part you will do the reverse. That is, you need to unpack or extract the 8 bytes contained within two integers. The 8 bytes "packed" in the integer are encoded in ASCII. If you forget the ASCII encoding you should take a quick look at the ascii chart. After you unpack the bytes you will print out the character interpretation of those bytes using `printf`. The two integers you will unpack are:

```c
unsigned int i1 = 1835098984u;
unsigned int i2 = 1768842611u;
```

Note that the integer literals on the right-hand side of the assignment have a `u` on the end. This indicates that the underlying representation of that value must use *unsigned* bit format (not signed two's complement).

Create a new C file called `unpacking-bytes.c`. Create the main function. Do not forget to add the `#include <stdio.h>` to the top of this file - this will let you use the `printf` function. Add the two integer declarations from above inside the main function. At this point you should try to compile this file to make sure that you have not made any syntactic mistakes. You should name the resulting binary executable as `unpacking-bytes`.

Next, you will need to add the right bitwise operations to extract each of the characters that are packed in the integer `i1` and `i2`. The only operators you are allowed to use are `<<` and `>>`, left shift and right shift respectively. The characters are packed in the following format:

```
i1 = [      c1      ][      c2      ][      c3      ][      c4      ]
    31             24               16               8               0
i2 = [      c5      ][      c6      ][      c7      ][      c8      ]
    31             24               16               8               0
```

After you unpack the bytes you must print out the corresponding ASCII encoded characters using `printf`. Compile and run your program to get the following output (name your binary executable `unpacking-bytes`):

```
$ ./unpacking-bytes
c1c2c3c4c5c6c7c8
```

Where c1 - c8 are replaced with the ASCII characters.

Run the test script `test/test06` to see how well you did:

```
$ source test/test06
```

Part 7: Printing The Bits!

Create a new C file called `print-bits.c`. Add the `main` function and the necessary details to have access to `printf` (as we have done before). Write a program that will print the bits of the `unsigned char 181` and the `signed char -75`. Remember, in C the `char` data type represents a byte (the name `char` can be misleading) so it is possible to assign integers to them. The idea here is to use any of the bitwise operators `<<`, `>>`, or `&` to extract each bit from the values `181` and `-75` (you need not use all of the operators). You must print the bits from left to right starting with the most significant bit to the least significant bit. More specifically, consider the bit representation of a byte:

```
  MSB                                         LSB
 [ b7 ][ b6 ][ b5 ][ b4 ][ b3 ][ b2 ][ b1 ][ b0 ]
   7     6     5     4     3     2     1     0
```

Where bit `b7` is in bit position `7` and is the most significant bit, and bit `b0` is in bit position `0` and is the least significant bit. You must print the bits starting from bit 7 to bit 0, from left to right. You should compile and run your program (save the resulting binary executable as `print-bits`) to produce the following output:

```
$ ./print-bits
b7b6b5b4b3b2b1b0
b7b6b5b4b3b2b1b0
```

Here the first line of output gives the bits from the unsigned byte (char) `181` and the second line of output gives the bits from the signed byte `-75`. Do you notice anything peculiar with the output? This should be pretty obvious - if it is then you know you have done this part successfully.

Run the test script `test/test07` to see how well you did:

```
$ source test/test07
```

Part 8: Extracting bit fields

Just as you can unpack same sized quantities such as bytes from a larger integer type such as `int`, you can extract a number of fields of *different* sizes from an integer type. In this part you are to extract 10 fields of given widths, from left to right (most significant to least significant) from a 32-bit `int` of a given value.

The field widths, in bits, are:

3, 4, 4, 3, 3, 4, 4, 3, 2, and 2.

If we use lower case letters `a` through `j` for the bits of the fields, in order, then the 32-bit value looks like this:

```
aaabbbbccccdddeeeffffgggghhhiijj
```

The specific value from which you are to extract the 10 fields is our old friend 0xCAFEBABE. You are to extract each field using only the `<<` and `>>` operators of C. Print the (unsigned) values of the 10 fields, from left to right, separated by spaces (e.g., by using a `printf` format such as `"%d %d ... %d\n"`.

Create a C file called `extracting-fields.c` and put in the `#include <stdio.h>` line and a `main` function as in the other parts. Initialize a new variable to the value `0xCAFEBABE` and then extract the fields into 10 separate variables. Finally, print the values of those variables, left ot right.

Run the test script `test/test08` to see how well you did:

```
$ source test/test08
```

Part 9: Updating bit fields

In this part you are to start with a given integer value, the update two specified ranges of bits, and then print the updated value. Create a C file called `updating-fields.c` with the `#include <stdio.h>` line and a `main` function, as usual. Initialize a new variable to the value `17512807u`.

Assume we number the bits as usual from 0 as least significant (on the right) to 31 (most significant, on the left). Update bits 18 through 21 with the integer value 8 and bits 10 through 14 with value 17 (decimal). Print the resulting value as an *eight digit* hexadecimal number to show all of the digits. (Hint: look up what a `printf` format of `%08x` does.) You should see an interesting pattern.

You may use the C operators `&`, `|`, `~`, and `<<` to form the updated value. You will probably want to form and use a suitable mask for each field.

Run the test script `test/test09` to see how well you did:

```
$ source test/test09
```

Part 10: Testing it all!

The last part of this assignment is easy. Run all of the tests to make sure you have covered everything. We have provided a script to do this:

```
$ source test/test-all
```

If you do everything correctly you should see as your final output:

```
test01 points: 8
test02 points: 8
test03 points: 8
test04 points: 8
test05 points: 8
test06 points: 8
test07 points: 8
test08 points: 8
test09 points: 8
```

Submission Instructions

You must submit your assignment as a *tarball*. After you complete the assignment you need to run the following command inside your project directory:

```
$ ./submit.sh
```

This will create the file `bits-and-bytes-proj-student-submit.tgz` which you need to upload to the assignment activity in Moodle. Please submit your assignment to Moodle by the assigned due date. Please make sure you have followed all the instructions described in this assignment. Failure to follow these instructions precisely will likely lead to considerable point deductions and possibly failure for the assignment.