CSCI 1730 Breakout Lab 09

Unions and IEEE 754

Learning Outcomes

- Implement a program that uses a union data structure.
- Demonstrate knowledge of a simple data structure (union) by writing out its memory map and tracing through the output of source code.
- Design and implement a program that uses bitwise operators on integers.
- Explain some key differences between Java, C++, and C.

Problem / Exercise

All things in our programs and computers are represented in binary. In lecture classes earlier in the semester, we discussed a binary encoding for unsigned integers and a binary encoding for signed integers. However, our C programs and computers also use a binary encoding for floats and a binary encoding for doubles. On odin and the version of the gcc compiler required for this course, a float is encoded using a 32 bit IEEE 754 representation and a double is encoded using a 64 bit IEEE 754 representation. More information about IEEE 754 can be found at `https://en.wikipedia.org/wiki/IEEE_754-1985` and `https://en.wikipedia.org/wiki/IEEE_754`. Read through the IEEE 754 representations in the aforementioned links.

For this lab assignment, we are going to use a union to convert a float to an int and another union to convert a double to a long. A union is a data structure that is part of the C and C++ programming languages, but it is not typically part of the Java programming language. In your source code, create one union that has two members: a float and an int. After that, create another union that has two members: a double and a long. Draw a memory map for each of these unions. For the first union, the float and int members both share how many bits of memory? For the second union, the double and the long members both share how many bits of memory?

Bitwise operators in C can be used to print out base-10 integer values in binary, but bitwise operators in C do not work on floats or doubles directly. However, we can use the first union to automatically convert a float to an int with the same binary encoding as its float since both members occupy the same block of memory, and afterwards in our program, use bitwise operators to print out the int member in binary. Similarly, we can repeat this process using the second union for doubles. We don't have to do the math (and we shouldn't program that math for this assignment) to convert a float to int or double to long if we use unions correctly. For this assignment, your I/O must match the Examples shown (spacing matters for clarity and for grading). You may assume two flags for this assignment: "-f" is for converting a float value to binary and "-d" is for converting a double value to binary as shown in the examples. You may also assume all float values and all double values inputted will have exact representations in binary (no roundoff errors). We are making this assumption since there are floating point values in base-10 that do not have an exact binary representation using a finite number of bits in our computers. We will use float values and double values with exact binary representations to grade your program for this assignment.

Examples

After you finish lab09.c, the I/O from running lab09.out should match exactly what is shown below. Each example is a single execution of a correctly working program for this assignment.

```
./lab09.out -f -18.5
-18.500000 encoded in binary using a 32 bit IEEE 754 encoding is below.
11000001100101000000000000000000
sign bit:          1
exponent (8 bit):  10000011
```

```
fraction (23 bit): 00101000000000000000000

./lab09.out -d 18.5
18.500000 encoded in binary using a 64 bit IEEE 754 encoding is below.
0100000000110010100000000000000000000000000000000000000000000000
sign bit:          0
exponent (11 bit): 10000000011
fraction (52 bit): 0010100000000000000000000000000000000000000000000000

./lab09.out -f 1315625e-5
13.156250 encoded in binary using a 32 bit IEEE 754 encoding is below.
01000001010100101000000000000000
sign bit:          0
exponent (8 bit):  10000010
fraction (23 bit): 10100101000000000000000

./lab09.out -d -1315625e-5
-13.156250 encoded in binary using a 64 bit IEEE 754 encoding is below.
1100000000101010010100000000000000000000000000000000000000000000
sign bit:          1
exponent (11 bit): 10000000010
fraction (52 bit): 1010010100000000000000000000000000000000000000000000

./lab09.out -f 687.1875
687.187500 encoded in binary using a 32 bit IEEE 754 encoding is below.
01000100001010111100110000000000
sign bit:          0
exponent (8 bit):  10001000
fraction (23 bit): 01010111100110000000000

./lab09.out -d -17826.515625
-17826.515625 encoded in binary using a 64 bit IEEE 754 encoding is below.
1100000011010001011010001010000100000000000000000000000000000000
sign bit:          1
exponent (11 bit): 10000001101
fraction (52 bit): 0001011010001010000100000000000000000000000000000000
```

1 C Program

1.1 General Requirements

1. Place the files for this assignment in a directory called `lab09`.
2. Your program's source file(s) must compile on odin or a vcf node using the provided Makefile and using the required compiler for this course (`gcc version 11.2.0`).
3. Place your C source code for this assignment in a file named `lab09.c`. See the Examples section to see what your C program should output. Compile your C program using the provided Makefile for this assignment. Do NOT modify any commands, targets, or dependencies that are associated with compiling source code in the provided Makefile for this assignment. If there are any warnings or errors from the compiler, then you should fix them before you submit this assignment to us for grading.
4. Your program cannot contain a memory leak when executed. You are responsible for using valgrind to test your program to ensure it does not contain a memory leak, and valgrind should be run for every test case for your program. A program that contains one or more memory leaks (definitely, indirectly, or possibly lost) identified by valgrind will lose points.
5. The only include directives that you are authorized to use in your source code for this assignment are listed below.

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <stdbool.h>
#include <unistd.h>
```

You CANNOT use any other include directives anywhere in your source code.

6. All written and verbal instructions stated by the teaching staff (lecture instructor, lab instructor(s), etc.) must be followed for this assignment. Failure to follow instructions may result in losing points.

1.2 Coding Style Requirements

1. All functions must be commented. Comments must include a brief description of what the function does, its input(s), its output, and any assumptions associated with calling it. If your function has a prototype and an implementation separated into different parts of your source code, then you only need to put the comments for that function above its prototype (there is no need to comment both the prototype and implementation; commenting the prototype is sufficient).
2. All structs, unions, and enums must be commented.
3. All global variables and static variables must be commented.
4. All identifiers must be named well to denote their functionality. Badly named identifiers are not allowed. For example, identifiers like a, aaa, b, bbb, bbbb are bad names for identifiers.
5. Every line of source code must be indented properly and consistently.

1.3 README.txt File Requirements

Make sure to include a `README.txt` file (use a .txt file extension) that includes the following information presented in a reasonably formatted way:

- Your First and Last Name (as they appear on eLC) and your 810/811#
- Instructions on how to compile and run your program. Since we are using a Makefile for this assignment, then these instructions should pretty simple based on the provided Makefile.

2 Submission

Submit your files before the due date and due time stated on eLC. Submit your files on eLC under the Assignment named Lab 09. Submit only the following files.

1. All source files required for this lab: lab09.c
2. A README.txt file filled out correctly

Do not submit any compiled code. Also, do not submit any zipped files or directories. Do not submit a Makefile. We will use our own copy of the provided Makefile to compile your program. We only need the files mentioned above with the file extensions aforementioned.

3 Breakout Lab Attendance: 3 points

Breakout lab attendance is required for this assignment, and it is worth three points. Students must physically attend the entire period of the breakout lab section that they registered for during the week of this assignment to earn these three points for attendance. If you complete the assignment before the end of your breakout lab period during the week of this assignment, then you are still required to attend the entire breakout lab period to earn attendance points, and you may use this time for independent study for the next exam in this course.

4 Grading: 7 points

If your program does not compile on odin or a vcf node using the provided Makefile and using the required compiler for this course (`gcc version 11.2.0`), then you'll receive a grade of zero on this part of the assignment. Otherwise, your program will be graded using the criteria below.

Program runs correctly with various test cases on odin	7 points
README.txt file missing or incorrect in some way	-1 point
Not submitting to the correct Assignment on eLC	-1 point
Late penalty for submitting 0 hours to 24 hours late	-2 points
One or more compiler warnings	-2 points
Not adhering to one or more coding style requirements	-2 points
Valgrind identifies a memory leak (definitely, indirectly, or possibly lost)	-2 points
Not submitting this assignment before its late period expires	-7 points
Penalty for not following instructions (invalid I/O, etc.)	Penalty decided by grader

You must test, test, and retest your code to make sure it compiles and runs correctly on odin with any given set of valid inputs. This means that you need to create many examples on your own (that are different than the aforementioned examples) to ensure you have a correctly working program. Your program's I/O must match the examples given in the Examples section. We will only test your program with valid sets of inputs.