# CSCE 312 Lab 1

Kevin Lei

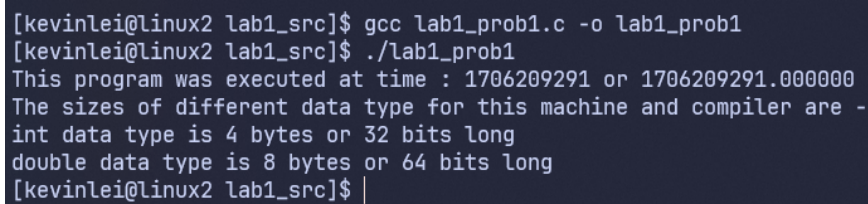January 31, 2024

## 1 Problem 1

### 1.1 Part A

**Tag 1:** The purpose of this code is to make sure that the file is open, or the point to the file is not null. If the file is not open, then the program will exit.

**Tag 2:** This code calls the `gettimeofday()` function from the `time.h` header. It then assigns the value to the `this_instant` variable which is passed by reference.

**Tag 3:** This line of code writes the int data type size in bits and bytes to the output file.

**Tag 4:** This line does the same as Tag 3 except it writes to the console instead of a file.

### 1.2 Part B

```
[kevinlei@linux2 lab1_src]$ gcc lab1_prob1.c -o lab1_prob1
[kevinlei@linux2 lab1_src]$ ./lab1_prob1
This program was executed at time : 1706209291 or 1706209291.000000
The sizes of different data type for this machine and compiler are -
int data type is 4 bytes or 32 bits long
double data type is 8 bytes or 64 bits long
[kevinlei@linux2 lab1_src]$
```

Figure 1: Compiling and running `lab1_prob1.c`

### 1.3 Part C

The `timeval` struct contains the `tv_sec` variable, which is a numeric value like int or double.

# 2 Problem 2

## 2.1 Part A

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>

int main() {
    FILE* my_file_pointer;
    if ( (my_file_pointer = fopen("lab1_prob2_out.txt", "w")) == NULL) {
        printf("Error opening the file, so exiting\n");
        exit(1);
    }

    fprintf(my_file_pointer, "int data type is %lu bytes or %lu bits long\n",sizeof(int), sizeof(int)*8);
    fprintf(my_file_pointer, "unsigned int data type is %lu bytes or %lu bits long\n",sizeof(unsigned int), sizeof(unsigned int)*8);
    fprintf(my_file_pointer, "double data type is %lu bytes or %lu bits long\n",sizeof(double), sizeof(double)*8);
    fprintf(my_file_pointer, "long data type is %lu bytes or %lu bits long\n",sizeof(long), sizeof(long)*8);
    fprintf(my_file_pointer, "long long data type is %lu bytes or %lu bits long\n",sizeof(long long), sizeof(long long)*8);
    fprintf(my_file_pointer, "char data type is %lu bytes or %lu bits long\n",sizeof(char), sizeof(char)*8);
    fprintf(my_file_pointer, "float data type is %lu bytes or %lu bits long\n",sizeof(float), sizeof(float)*8);
    fprintf(my_file_pointer, "struct timeval data type is %lu bytes or %lu bits long\n",sizeof(struct timeval), sizeof(struct timeval)*8);
    fprintf(my_file_pointer, "short data type is %lu bytes or %lu bits long\n",sizeof(short), sizeof(short)*8);
    fprintf(my_file_pointer, "FILE* data type is %lu bytes or %lu bits long\n",sizeof(FILE*), sizeof(FILE*)*8);

    return 0;
}
```

Figure 2: Source code for `lab1_prob2.c`

```
[kevinlei@linux2 lab1_src]$ gcc lab1_prob2.c -o lab1_prob2
[kevinlei@linux2 lab1_src]$ ./lab1_prob2
[kevinlei@linux2 lab1_src]$ cat lab1_prob2_out.txt
int data type is 4 bytes or 32 bits long
unsigned int data type is 4 bytes or 32 bits long
double data type is 8 bytes or 64 bits long
long data type is 8 bytes or 64 bits long
long long data type is 8 bytes or 64 bits long
char data type is 1 bytes or 8 bits long
float data type is 4 bytes or 32 bits long
struct timeval data type is 16 bytes or 128 bits long
short data type is 2 bytes or 16 bits long
FILE* data type is 8 bytes or 64 bits long
[kevinlei@linux2 lab1_src]$
```

Figure 3: Compiling and running `lab1_prob2.c`, including the output.

## 2.2 Part B

The structs `employee1` and `employee2` have the same bit and byte sizes of 448 bits and 56 bytes respectively. In theory, the structs should use different amounts of memory. Although they both have an int field, the `employee2` struct has an array of characters of maximum length 52, while the `employee1` struct has an array of characters of maximum length 50. The code and output are shown below.

```c
#include <stdio.h>

int main() {
    struct employee1{
        int id;
        char name[50];
    };

    struct employee2{
        int id;
        char name[52];
    };

    printf("Size of struct employee1 (bits): %d\n", sizeof(struct employee1)*8);
    printf("Size of struct employee1 (bytes): %d\n", sizeof(struct employee1));
    printf("Size of struct employee2 (bits): %d\n", sizeof(struct employee2)*8);
    printf("Size of struct employee2 (bytes): %d\n", sizeof(struct employee2));

    printf("Size of int (bits): %d\n", sizeof(int)*8);
    printf("Size of char (bits): %d\n", sizeof(char)*8);
    printf("Size of char[50] (bits): %d\n", sizeof(char[50])*8);
    printf("Size of char[52] (bits): %d\n", sizeof(char[52])*8);

    return 0;
}
```

Figure 4: Source code for `lab1_prob2b.c`

```
[kevinlei@linux2 lab1_src]$ gcc lab1_prob2b.c -o lab1_prob2b
[kevinlei@linux2 lab1_src]$ ./lab1_prob2b
Size of struct employee1 (bits): 448
Size of struct employee1 (bytes): 56
Size of struct employee2 (bits): 448
Size of struct employee2 (bytes): 56
Size of int (bits): 32
Size of char (bits): 8
Size of char[50] (bits): 400
Size of char[52] (bits): 416
[kevinlei@linux2 lab1_src]$
```

Figure 5: Compiling and running `lab1_prob2b.c`, including the output.

As we can see in the screenshots, the structs have the same bit and byte sizes, However, `char name[50]` and `char name[52]` have difference bit sizes, 400 and 416 respectively. Adding 416 to 32 bits for the int field gives us 448 bits, which is the same as the bit size of the struct. This makes sense for the `employee2` struct, but not for the `employee1` struct, since $400 + 32 = 432$, not 448. The reason for this is called memory alignment, which is where memory chunks are organized into multiples of certain sizes. These sizes are typically the largest data type that can be handled by the processor. In this case, since the largest data type is `employee2` which uses 448 bits, the memory is aligned to 448 bits, and 448 bits are also allocated for `employee1`.

# 3 Problem 3

## 3.1 Part A

**Bell actuator truth table:**

| DSBF | ER | DC | DLC | DOS | KIC | BP | CM | BELL |
|------|----|----|-----|-----|-----|----|----|------|
| 0 | 0 | 0 | X | X | X | X | X | 0 |
| 1 | 0 | 0 | X | X | X | X | X | 0 |
| 0 | 0 | 1 | X | X | X | X | X | 0 |
| 1 | 0 | 1 | X | X | X | X | X | 0 |
| 0 | 1 | 0 | X | X | X | X | X | 1 |
| 1 | 1 | 0 | X | X | X | X | X | 1 |
| 0 | 1 | 1 | X | X | X | X | X | 1 |
| 1 | 1 | 1 | X | X | X | X | X | 0 |

**Door Lock actuator truth table:**

| DSBF | ER | DC | DLC | DOS | KIC | BP | CM | DLA |
|------|----|----|-----|-----|-----|----|----|-----|
| X | X | X | 1 | 0 | 0 | X | X | 1 |
| X | X | X | 1 | 1 | 0 | X | X | 1 |
| X | X | X | 1 | 0 | 1 | X | X | 0 |
| X | X | X | 1 | 1 | 1 | X | X | 1 |
| X | X | X | 0 | 0 | 0 | X | X | 0 |
| X | X | X | 0 | 1 | 0 | X | X | 0 |
| X | X | X | 0 | 0 | 1 | X | X | 0 |
| X | X | X | 0 | 1 | 1 | X | X | 0 |

**Brake actuator truth table:**

| DSBF | ER | DC | DLC | DOS | KIC | BP | CM | BA |
|------|----|----|-----|-----|-----|----|----|-----|
| X | X | X | X | X | X | 0 | 0 | 0 |
| X | X | X | X | X | X | 1 | 0 | 0 |
| X | X | X | X | X | X | 0 | 1 | 0 |
| X | X | X | X | X | X | 1 | 1 | 1 |

## 3.2 Part B

The boolean expressions for the Bell actuator, Door Lock actuator, and Brake actuator are shown below.

$$BELL = ER * (DSBF' + DC')$$
$$DLA = DLC * (KIC' + DOS)$$
$$BA = BP * CM$$

## 3.3 Part C

```
void control_action(){
    if (engine_running && (!doors_closed || !driver_seat_belt_fastened)) bell = 1;
    if (door_lock_lever && (!key_in_car || driver_on_seat)) door_lock_actu = 1;
    if (brake_pedal && car_moving) brake_actu = 1;
}
```

Figure 6: Source code for control system.

## 3.4 Part D



Figure 7: Compiling and running `lab1_prob3.c`, including the output.

# 4 Problem 4



Figure 8: Bitmasking reimplementation of `lab1_prob3.c`



Figure 9: Compiling and running `lab1_prob4.c`, including the output.

# 5 Problem 5



Figure 10: Compilation and output of `lab1_prob5.c` using the code from part 3.



Figure 11: Compilation and output of `lab1_prob5b.c` using the code from part 4.