

# Intermediate Programming

## Day 14

# Outline

- Exercise 13
- Binary file I/O
- Bitwise operations
- Review questions

# Exercise 13

Declare the **structs**

*soccer.h*

```
...
typedef struct
{
    int num_of_goals , num_of_assists;
    float pass_accuracy;
    int min_played , num_of_shots;
    float shot_accuracy;
} Stat;

typedef struct
{
    int day , month , year;
} Date;

typedef struct
{
    int age , jersey_num;
    bool goalkeeper;
    Date *date;
    Stat *stat;
} Player;
...
```

# Exercise 13

Find the index of the **Player** with the latest start **Date**.

*main.c*

```
...
int cmp_dates( const Date *d1 , const Date *d2 )
{
    if( d1->year!=d2->year ) return d1->year - d2->year;
    if( d1->month!=d2->month ) return d1->month - d2->month;
    return d1->day - d2->day;
}

int main()
{
    ...
    int index = -1;
    for( int i=0 ; i<TEAMSIZ ; i++ )
        if( index==-1 || cmp_date( team[i].date , team[index].date )>0 )
            index = i;
    ...
}
```

# Exercise 13

Update the *Player* with the latest start *Date*

*main.c*

```
...
int cmp_dates( const Date *d1 , const Date *d2 )
{
    if( d1->year!=d2->year ) return d1->year - d2->year;
    if( d1->month!=d2->month ) return d1->month - d2->month;
    return d1->day - d2->day;
}

int main()
{
    ...
    int index = -1;
    for( int i=0 ; i<TEAMSIZ ; i++ )
        if( index==-1 || cmp_date( team[i].date , team[index].date )>0 )
            index = i;

    free( team[index].stat );
    team[index].stat = new_stat;
    ...
}
```

# Exercise 13

## Clean up

*main.c*

```
...
int cmp_dates( const Date *d1 , const Date *d2 )
{
    if( d1->year!=d2->year ) return d1->year - d2->year;
    if( d1->month!=d2->month ) return d1->month - d2->month;
    return d1->day - d2->day;
}

int main()
{
    ...
    int index = -1;
    for( int i=0 ; i<TEAMSIZ ; i++ )
        if( index==-1 || cmp_date( team[i].date , team[index].date )>0 )
            index = i;

    free( team[index].stat );
    team[index].stat = new_stat;

    ...
    for( int i=0 ; i<TEAMSIZ ; i++ )
    {
        free( team[i].date );
        free( team[i].stat );
    }
    ...
}
```

# Outline

- Exercise 13
- **Binary file I/O**
- Bitwise operations
- Review questions

# Binary File I/O

When working with file handles we:

1. Create a file handle
2. Access the file's contents
3. Close the handle

*main.c*

```
#include <stdio.h>
#include <stdlib.h>
int main( void )
{
    unsigned int values[100];
    for( int i=0 ; i<100 ; i++ ) values[i] = rand();
    FILE *fp = fopen( "foo.txt" , "w" );
    if( !fp ) return 1;
    for( int i=0 ; i<100 ; i++ ) fprintf( fp , "%u\n" , values[i] );
    fclose( fp );
}
```



# Binary File I/O

When working with file handles we:

1. Create a file handle

- `fopen` with the file-name and mode
  - "w" for (ASCII) write
  - "r" for (ASCII) read

*main.c*

```
#include <stdio.h>
#include <stdlib.h>
int main( void )
{
    unsigned int values[100];
    for( int i=0 ; i<100 ; i++ ) values[i] = rand();
    FILE *fp = fopen( "foo.txt" , "w" );
    if( !fp ) return 1;
    for( int i=0 ; i<100 ; i++ ) fprintf( fp , "%u\n" , values[i] );
    fclose( fp );
}
```

# Binary File I/O

When working with file handles we:

2. Access the file's contents

- `fprintf` with file handle, format string, and values for (ASCII) write
- `fscanf` with file handle, format string, and addresses for (ASCII read)

*main.c*

```
#include <stdio.h>
#include <stdlib.h>
int main( void )
{
    unsigned int values[100];
    for( int i=0 ; i<100 ; i++ ) values[i] = rand();
    FILE *fp = fopen( "foo.txt" , "w" );
    if( !fp ) return 1;
    for( int i=0 ; i<100 ; i++ ) fprintf( fp , "%u\n" , values[i] );
    fclose( fp );
}
```

# Binary File I/O

When working with file handles we:

3. Close the handle
  - `fclose` with file handle

*main.c*

```
#include <stdio.h>
#include <stdlib.h>
int main( void )
{
    unsigned int values[100];
    for( int i=0 ; i<100 ; i++ ) values[i] = rand();
    FILE *fp = fopen( "foo.txt" , "w" );
    if( !fp ) return 1;
    for( int i=0 ; i<100 ; i++ ) fprintf( fp , "%u\n" , values[i] );
    fclose( fp );
}
```

# Binary File I/O

If we write out a list of 100 (random) `ints` to a file

Q: How big would the file be?

Q: How would we get the 7<sup>th</sup> value?

*main.c*

```
#include <stdio.h>
#include <stdlib.h>
int main( void )
{
    unsigned int values[100];
    for( int i=0 ; i<100 ; i++ ) values[i] = rand();
    FILE *fp = fopen( "foo.txt" , "w" );
    if( !fp ) return 1;
    for( int i=0 ; i<100 ; i++ ) fprintf( fp , "%u\n" , values[i] );
    fclose( fp );
}
```

# Binary File I/O

If we write out a list of 100 (random) `ints` to a file

Q: How big would the file be?

A: 1056 bytes

We have 32 bits per `int`:

⇒ range of  $[0, \sim 2 \times 10^9]$

⇒ 9-10 decimal places (average)  
+1 for the `"\n"`

⇒  $10 \times 100 - 11 \times 100$  bytes

⇒ Size is not fixed

But the values always require  
400 bytes in memory!!!

```
>> ./a.out
>> ls -l foo.txt
-rw-----. 1 misha users 1056 Mar 30 23:17 foo.txt
>>
```

*main.c*

```
#include <stdio.h>
#include <stdlib.h>
int main( void )
{
    unsigned int values[100];
    for( int i=0 ; i<100 ; i++ ) values[i] = rand();
    FILE *fp = fopen( "foo.txt" , "w" );
    if( !fp ) return 1;
    for( int i=0 ; i<100 ; i++ ) fprintf( fp , "%u\n" , values[i] );
    fclose( fp );
}
```

# Binary File I/O

If we write out a list of 100 (random) ints to a file

Q: How would we get the 7<sup>th</sup> value?

A: 64 characters in

We have 32 bits per int

⇒ 10 × 6 – 11 × 6 bytes

⇒ Offset is not fixed

✗ We cannot jump to the 7<sup>th</sup> value since we don't know the sizes of the values that come before

⇒ **fscanf** one int at a time until we get the 7<sup>th</sup> value

```
#include <stdio.h>
#include <stdlib.h>
int main( void )
{
```

```
    unsigned int values[100];
    for( int i=0 ; i<100 ; i++ ) values[i] = rand();
    FILE *fp = fopen( "foo.txt" , "w" );
    if( !fp ) return 1;
    for( int i=0 ; i<100 ; i++ ) fprintf( fp , "%u\n" , values[i] );
    fclose( fp );
}
```

```
>> ./a.out
>> more foo.txt
1804289383
846930886
1681692777
1714636915
1957747793
424238335
719885386
...
>>
```

# Binary File I/O

- Until now, all files we've accessed in C have been plain *text files*
  - Write: Convert everything to a string of characters that is written to the file
  - Read: Convert everything from a string of characters that is read from the file
- Non-text files are known as *binary files* in C
  - Write: perform a bit-by-bit copy from memory to the file
  - Read: perform a bit-by-bit copy from the file to memory

# Binary File I/O

As with text files, we:

1. Open the file
2. Access the file's contents
3. Close the file

*main.c*

```
#include <stdio.h>
#include <stdlib.h>
int main( void )
{
    unsigned int values[100];
    for( int i=0 ; i<100 ; i++ ) values[i] = rand();
    FILE *fp = fopen( "foo.dat" , "wb" );
    if( !fp ) return 1;
    fwrite( values , sizeof(int) , 100 , fp );
    fclose( fp );
}
```



# Binary File I/O

`FILE * fopen( const char *fileName , const char *mode );`

## 1. Open the file

- Use `fopen` to create a file handle:
  - `fileName`: name of the file
  - `mode`: mode of I/O
    - To open a file in binary mode, add the "b" flag in the string of `mode` characters\*
- Returns a file pointer (or NULL if the `fopen` failed)

```
main.c
#include <stdio.h>
#include <stdlib.h>
int main( void )
{
    unsigned int values[100];
    for( int i=0 ; i<100 ; i++ ) values[i] = rand();
    FILE *fp = fopen( "foo.dat" , "wb" );
    if( !fp ) return 1;
    fwrite( values , sizeof(int) , 100 , fp );
    fclose( fp );
}
```

\*The file extension does not affect how the file is opened.

# Binary File I/O

```
size_t fwrite( const void *ptr , size_t sz , size_t count , FILE *fp );
```

## 2. Access the file's contents

- Use `fwrite` to write to a binary file:
  - `ptr`: starting address of data to write out
  - `sz`: size of a single data element
  - `count`: number of data elements
  - `fp`: file handle to write to
- Returns the number of elements written

```
main.c
#include <stdio.h>
#include <stdlib.h>
int main( void )
{
    unsigned int values[100];
    for( int i=0 ; i<100 ; i++ ) values[i] = rand();
    FILE *fp = fopen( "foo.dat" , "wb" );
    if( !fp ) return 1;
    fwrite( values , sizeof(int) , 100 , fp );
    fclose( fp );
}
```

# Binary File I/O

```
size_t fread( void *ptr , size_t sz , size_t count , FILE *fp );
```

## 2. Access the file's contents

- Use `fread` to read from a binary file:
  - `ptr`: starting address of data to read into
  - `sz`: size of a single data element
  - `count`: number of data elements
  - `fp`: file handle to read from
- Returns the number of elements read

```
main.c
#include <stdio.h>
#include <stdlib.h>
int main( void )
{
    unsigned int values[100];
    for( int i=0 ; i<100 ; i++ ) values[i] = rand();
    FILE *fp = fopen( "foo.dat" , "wb" );
    if( !fp ) return 1;
    fwrite( values , sizeof(int) , 100 , fp );
    fclose( fp );
}
```

# Binary File I/O

```
int fclose( FILE *fp );
```

## 3. Close the file

- Use `fclose` to close the file handle
  - `fp`: file handle
- Returns 0 if the stream was closed

```
main.c  
#include <stdio.h>  
#include <stdlib.h>  
int main( void )  
{  
    unsigned int values[100];  
    for( int i=0 ; i<100 ; i++ ) values[i] = rand();  
    FILE *fp = fopen( "foo.dat" , "wb" );  
    if( !fp ) return 1;  
    fwrite( values , sizeof(int) , 100 , fp );  
    fclose( fp );  
}
```

# Binary File I/O

If we write out a list of 100 `ints` to a file

Q: How big would the file be?

A: 400 bytes. Always!

```
>> ./a.out
>> ls -l foo.dat
-rw-----. 1 misha users 400 Mar 30 23:17 foo.dat
>>
```

```
main.c
#include <stdio.h>
#include <stdlib.h>
int main( void )
{
    unsigned int values[100];
    for( int i=0 ; i<100 ; i++ ) values[i] = rand();
    FILE* fp = fopen( "foo.dat" , "wb" );
    if( !fp ) return 1;
    fwrite( values , sizeof(int) , 100 , fp );
    fclose( fp );
}
```

# Binary File I/O

As we read/write data, the **FILE** pointer tracks our position in the file:

- In some cases, we would like to change our position in the file:
  - Writing: To over-write something that was previously written
  - Reading: To jump to where the data we are interested in resides

# Binary File I/O

```
int fseek( FILE *fp , long int offset , int whence );
```

- Use `fseek` to change the position of the file pointer
  - `fp`: file pointer
  - `offset`: number of bytes to move
    - Could be positive or negative, depending on whether we move forward or back
  - `whence`: where we move from:
    - `SEEK_SET`: beginning of the file
    - `SEEK_CUR`: current position
    - `SEEK_END`: end of the file
- Returns zero if the change succeeded

# Binary File I/O

*main.c (part 1)*

```
#include <stdio.h>
#include <stdlib.h>
unsigned int getValue( FILE *fp , size_t idx )
{
    if( fseek( fp , sizeof(unsigned int)*idx, SEEK_SET ) )
    {
        fprintf( stderr , "Failed to seek\n" );
        return -1;
    }
    unsigned int v;
    if( fread( &v , sizeof( int ) , 1 , fp )!=1 )
    {
        fprintf( stderr , "Failed to read\n" );
        return -1;
    }
    return v;
}
```

*main.c (part 2)*

```
int main( void )
{
    FILE *fp = fopen( "foo.dat" , "rb" );
    if( !fp )
    {
        fprintf( stderr , "Failed to open\n" );
        return -1;
    }
    printf( "%u\n" , getValue( fp , 7 ) );
    fclose( fp );
}
```



# Binary File I/O

```
size_t fwrite( const void *ptr , size_t sz , size_t count , FILE *fp );  
size_t fread( void *ptr , size_t sz , size_t count , FILE *fp );
```

## Note:

- The `fread/fwrite` functions only need to be able to read the bits/bytes from memory, they don't need to know the data-type stored.
- ⇒ We are not limited to reading/writing integers and numbers

```
main.c  
#include <stdio.h>  
typedef struct{ ... } MyStruct;  
int main( void )  
{  
    unsigned MyStruct values[100];  
    FILE *fp = fopen( "foo.dat" , "rb" );  
    if( !fp ) return 1;  
    fread( values , sizeof(MyStruct) , 100 , fp );  
    fclose( fp );  
}
```

# Binary File I/O

```
size_t fwrite( const void *ptr , size_t sz , size_t count , FILE *fp );  
size_t fread( void *ptr , size_t sz , size_t count , FILE *fp );
```

## Note:

If the `struct` contains pointers, the address is written out, not the contents at the address!!!

- The `fread/fwrite` functions only need to be able to read the bits/bytes from memory, they don't need to know the data-type stored.
- ⇒ We are not limited to reading/writing integers and numbers

```
main.c  
#include <stdio.h>  
typedef struct{ ... } MyStruct;  
int main( void )  
{  
    unsigned MyStruct values[100];  
    FILE *fp = fopen( "foo.dat" , "rb" );  
    if( !fp ) return 1;  
    fread( values , sizeof(MyStruct) , 100 , fp );  
    fclose( fp );  
}
```

# Outline

- Exercise 13
- Binary file I/O
- Bitwise operations
- Review questions

# Integer representation

- In C every variable is ultimately represented by some number of bytes.
- Each byte is represented by 8 bits.
- A bit can only have one of two values, 0 or 1.

# Integer representation

Every (non-negative) integer can be represented as a sum of a subset of\*:  
 $\{\dots, 2^k, \dots, 16, 8, 4, 2, 1\}$

⇒ We can represent an integer by denoting which of these summands it contains:

$$\begin{aligned} 117 &= 64 + 32 + 16 + 4 + 1 \\ &= \dots + 0 \cdot 128 + 1 \cdot 64 + 1 \cdot 32 + 1 \cdot 16 + 0 \cdot 8 + 1 \cdot 4 + 0 \cdot 2 + 1 \cdot 1 \end{aligned}$$

$(01110101)_2$

\*More on this next lecture.

# Bit-wise operations: Integer types only

- $n \ll k$ : shifts  $n$  to the left by  $k$  positions
  - This is equivalent to multiplying by  $2^k$
  - Note that the new, right-most, bits are set to 0
  - Once shifted out, the left-most bits are lost

```
#include <stdio.h>
int main( void )
{
    char a = 5;          // (00000101)_2
    char b = a << 2;      // (00010100)_2
    printf( "%d\n" , b );
    return 0;
}
```

```
>> ./a.out
20
>>
```

# Bit-wise operations: Integer types only

- $n \ll k$ : shifts  $n$  to the left by  $k$  positions
- $n \gg k$ : shifts  $n$  to the right by  $k$  positions
  - This is equivalent to dividing by  $2^k$
  - Note that the new, left-most, bits are set to 0
  - Once shifted out, the right-most bits are lost

```
#include <stdio.h>
int main( void )
{
    char a = 5;          // (00000101)_2
    char b = a >> 2;     // (00000001)_2
    printf( "%d\n" , b );
    return 0;
}
```

```
>> ./a.out
1
>>
```

# Bit-wise operations: Integer types only

- $n \ll k$ : shifts  $n$  to the left by  $k$  positions
- $n \gg k$ : shifts  $n$  to the right by  $k$  positions
- $n \& m$ : compute the bit-wise *and* of  $n$  and  $m$ 
  - The corresponding bit in the output is 1 if both bits are 1 in the input

```
#include <stdio.h>
int main( void )
{
    char a = 5;          // (00000101)_2
    char b = 14;         // (00001110)_2
    char c = a & b;      // (00000100)_2
    printf( "%d\n" , a & b );
    return 0;
}
```

```
>> ./a.out
4
>>
```



# Bit-wise operations: Integer types only

- $n \ll k$ : shifts  $n$  to the left by  $k$  positions
- $n \gg k$ : shifts  $n$  to the right by  $k$  positions
- $n \& m$ : compute the bit-wise *and* of  $n$  and  $m$
- $n | m$ : compute the bit-wise *or* of  $n$  and  $m$ 
  - The corresponding bit in the output is 1 if either (or both) bits are 1 in the input

```
#include <stdio.h>
int main( void )
{
    char a = 5;      // (00000101)_2
    char b = 14;     // (00001110)_2
    char c = a | b;   // (00001111)_2
    printf( "%d\n" , a | b );
    return 0;
}
```

```
>> ./a.out
15
>>
```

# Bit-wise operations: Integer types only

- $n \ll k$ : shifts  $n$  to the left by  $k$  positions
- $n \gg k$ : shifts  $n$  to the right by  $k$  positions
- $n \& m$ : compute the bit-wise *and* of  $n$  and  $m$
- $n | m$ : compute the bit-wise *or* of  $n$  and  $m$
- $n \wedge m$ : compute the bit-wise *exclusive or* of  $n$  and  $m$

- The corresponding bit in the output is 1 if an odd number of bit are 1 in the input

```
#include <stdio.h>
int main( void )
{
    char a = 5;        // (00000101)_2
    char b = 14;       // (00001110)_2
    char c = a ^ b;    // (00001011)_2
    printf( "%d\n" , c );
    return 0;
}
```

```
>> ./a.out
11
>>
```

# Bit-wise operations: Integer types only

- $n \ll k$ : shifts  $n$  to the left by  $k$  positions
- $n \gg k$ : shifts  $n$  to the right by  $k$  positions
- $n \& m$ : compute the bit-wise *and* of  $n$  and  $m$
- $n | m$ : compute the bit-wise *or* of  $n$  and  $m$
- $n \wedge m$ : compute the bit-wise *exclusive or* of  $n$  and  $m$
- $\sim n$ : flip the bits of  $n$ 
  - The corresponding bit in the output is 1 if it is 0 in the input

```
#include <stdio.h>
int main( void )
{
    char a = 5;          // (00000101)_2
    char b = ~a;          // (11111010)_2
    char c = b + 1;       // (11111011)_2
    printf( "%d\n" , c );
    return 0;
}
```

```
>> ./a.out
-5
>>
```

# Bit-wise operations: Integer types only

- There are also variants of these that evaluate-and-set
  - $n \ll= k$
  - $n \gg= k$
  - $n \&= m$
  - $n |= m$
  - $n \hat{=} m$

```
#include <stdio.h>
int main( void )
{
    char a = 5;          // (00000101)_2
    a <<= 3;              // (00101000)_2
    printf( "%d\n" , a );
    return 0;
}
```

```
>> ./a.out
40
>>
```

# Bit-wise operations: Integer types only

- Masking
  - We can determine if a bit is on or off using << and &

```
#include <stdio.h>
int main( void )
{
    char a = 5;          // (00000101)_2
    char mask = 1<<2;    // (00000100)_2
    char c = a & mask;
    printf( "%d %d\n" , c , c!=0 );
    return 0;
}
```

# Bit-wise operations: Integer types only

- Masking
  - We can determine if a bit is on or off using << and &

```
#include <stdio.h>
int main( void )
{
    char a = 5;          // (00000101)_2
    char mask = 1<<2;    // (00000100)_2
    char c = a & mask;
    printf( "%d %d\n" , c , c!
    return 0;
}
```

```
>> ./a.out
4 1
>>
```

# Bit-wise operations: Integer types only

- Masking
  - We can determine if a bit is on or off using << and &
  - Or we can use >> and &

```
#include <stdio.h>
int main( void )
{
    char a = 5;          // (00000101)_2
    char b = a>>2;        // (00000001)_2
    char c = b & 1;
    printf( "%d %d\n" , c , c!=0 );
    return 0;
}
```

# Bit-wise operations: Integer types only

- Masking
  - We can determine if a bit is on or off using << and &
  - Or we can use >> and &

```
#include <stdio.h>
int main( void )
{
    char a = 5;          // (00000101)_2
    char b = a>>2;       // (00000001)_2
    char c = b & 1;
    printf( "%d %d\n" , c , c);
    return 0;
}
```

```
>> ./a.out
1 1
>>
```



# Outline

- Exercise 13
- Binary file I/O
- Bitwise operations
- Review questions

# Review questions

1. How do we read/write binary files in C?

`fread / fwrite` with a “b” option

# Review questions

2. What character represents the bitwise XOR operation? How does it differ from the OR operation?

Bitwise XOR:  $\wedge$  -- at each position checks if just one of the bits is on

OR operation:  $\mid$  -- at each position checks if any of the bits are on

# Review questions

3. What happens if you apply the bitwise OR operation on an integer value? (extra: what if we apply to `floats`)

It returns an integer where each bit is “on” if it is on in one of the two integers.

[WARNING] Do not use bitwise operations for `floats`.

# Review questions

4. What is the result of  $(15 \gg 2) \mid 7$ ?

1

# Review questions

5. What is the result of  $(15 \gg 2) \mid 7$ ?

7

# Exercise 14

- Website -> Course Materials -> Exercise 14