

ECE 220

Lecture x000F

Recap/reminders

- Last time
 - Streams & buffers
 - File I/O
 - Formatted I/O
 - Examples
- Reminders
 - This lecture concludes the material for MT2
 - Class drop-deadline is tomorrow
 - Conflict sign-up link is live

Exercise

- Write a function to transpose a given TSV file and write the output to `transposed.tsv`

```
4 3
Zariski 99 Monday
Newton 43 Sunday
Russel 72 Saturday
Maxwell 32 Wednesday
```

- The number of rows and columns will be present as the first line of the input file:

```
records.tsv
```

- TSV stands for Tab-Separated-Values.

↓

```
3 4
Zariski Newton Russel Maxwell
99 43 72 32
Monday Sunday Saturday Wednesday
```

Exercise

- How about comma-separated values? Let us transpose a matrix stored on disk and write it back to disk.
- The input matrix is in file `mat.csv` with the first line specifying the number of rows and columns in the matrix.
- Write output to file `t_mat.csv`.

Lesson Objectives

- Understand and use structs in C.
 - Definition, initialization, direct access, pointer-based access, and passing to and from functions.
- Understand how structs are laid out linearly in memory (using the LC3 memory map).
- Understand the concept of type definition.
- Write and read structs as binary files to disk.

Introduction to structs

- Often useful to the programmer to combine pieces of information into a single abstract unit
- Example(s)
 - A student could have a name (`char[80]`), UIN (`unsigned long int`), year (`unsigned int`) and GPA (`float`)
 - A flight could have an altitude (`unsigned int`), latitude (`float`), longitude (`float`), airspeed (`float`) and airline code (`char[20]`)

Introduction to `structs`

- Achieved by letting the programmer create their own *data type* using the `struct` keyword.
- Examples (definition):

```
struct student{
    char name[80];
    unsigned long UIN;
    unsigned int year;
    float GPA;
};
```

```
struct flightType{
    char flightCode[20];
    unsigned int altitude;
    float longitude;
    float latitude;
    float airSpeed;
};
```

Defining structs

```
struct flightType{  
    char flightCode[20];  
    unsigned int altitude;  
    float longitude;  
    float latitude;  
    unsigned float airSpeed;  
};
```

However ... no memory
allocated yet!

- A `struct` allows the user to define a **new data type** that groups together items of types that are *already* defined.
- *Defining* a struct tells the compiler
 - How big the struct is ...
 - How to lay items out in memory ...

```
struct flightType{
    char flightCode[20];
    unsigned int altitude;
    float longitude;
    float latitude;
    unsigned float airSpeed;
};
```

```
struct student{
    char name[80];
    unsigned long UIN;
    unsigned int year;
    float GPA;
};
```

Using structs

- Memory is only allocated when variables are created using the newly defined type.

```
struct flightType plane;
struct student s1;
```

- Elements of a struct are called its *members*. Members can be accessed using the “dot” notation.

```
plane.altitude = 1000;
plane.airspeed = 800.0;
```

- struct variables can also be initialized at declaration.

```
struct student s1 = {"Garfield",
123456, 6, 3.5};
```

- Also possible to create arrays of structs

```
struct student BL3[2] = {s1,
{"Scooby", 234578164, 2, 4.0}};
printf("Name is %s", BL3[1].name);
```

Memory mapping

- How many bytes of memory should one *instance* of `student` take?

```
struct student{
    char name[80];
    unsigned long UIN;
    unsigned int year;
    float GPA;
};

struct student s1 =
{"Garfield", 123456, 6, 3.5}
```

	...
G	s1.name[0]
a	s1.name[1]
...	...
...	s1.name[78]
...	s1.name[79]
123456	s1.UIN
6	s1.year
3.5	s1.gpa

$$80 + 8 + 4 + 4$$

Memory mapping

- What if we change the definition to this one?

```
struct student{  
    char name[74];  
    unsigned long UIN;  
    unsigned int year;  
    float GPA;  
};
```

$$80 \ 74 + 8 + 4 + 4 = ?$$

Let us check using
`sizeof` function.

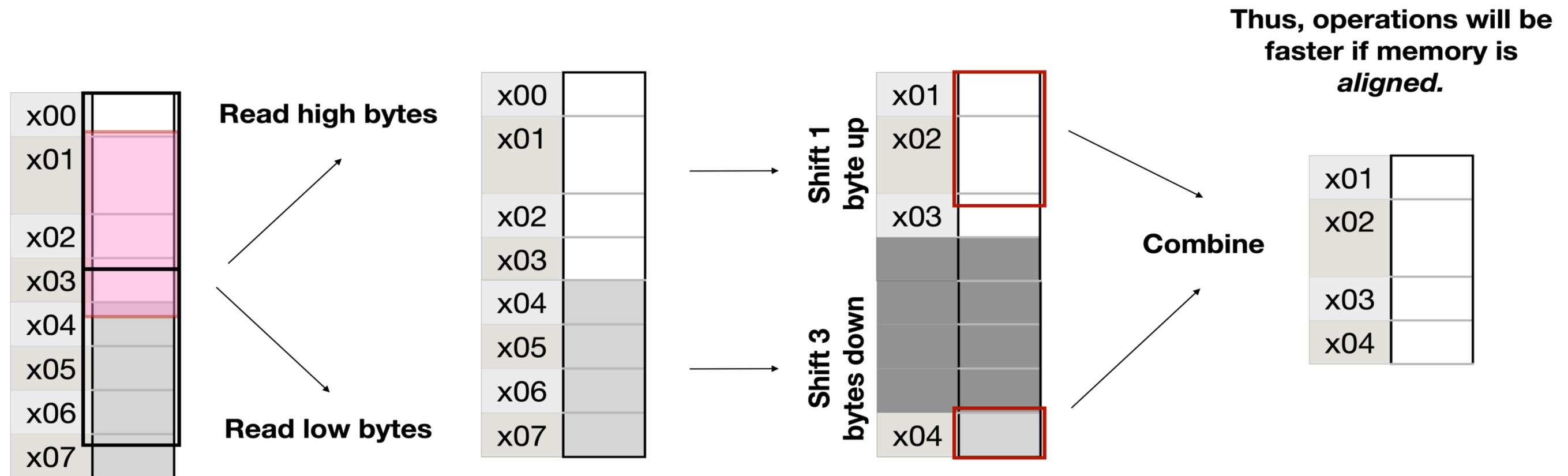
What happened?

Compilers will often perform “padding” to *align* memory.

Use the `sizeof` operator to get accurate results!

Why padding is done?

- Compilers prefer to *align* memory to make operations *faster*.
- Memory typically has an *access granularity*.
- Suppose we have **4 byte** memory access granularity.
 - Task: Read 4 bytes from address x01



The typedef keyword

- Note how we declared a struct variable:

```
struct flightType plane;  
struct student s1;
```

- *Annoying* to keep having to say `struct xyz, struct abc` - more so in the context of function calls
- C provides a mechanism to avoid this verbosity.

```
typedef struct flightType{  
    char flightCode[20];  
    unsigned int altitude;  
    float longitude;  
    float latitude;  
    unsigned float airSpeed;  
} Flight;
```

```
Flight f1 = {"AA 4324",  
            33000,  
            87.6,  
            41.8,  
            700};
```

Pointers to structs

- One can define pointers to structs the usual way.
- To access struct elements via pointers you can

```
Flight planes[100];  
Flight *ptr1;  
ptr1 = &planes[10];  
Flight *ptr2;  
ptr2 = planes;
```

- Dereference and dot

```
printf("I am %f feet high",  
      (*ptr1).altitude);
```

- Arrow

Special syntax!

```
← printf("I am %f feet high",  
        ptr1->altitude);
```

Passing structs as arguments

- One can write function definitions involving using structs in either way:

```
void print_student(struct student s){  
    printf("Student %s is associated with UIN: %lu\n", s.name, s.UIN);  
    printf("%s is in Year %d with GPA %f\n", s.name, s.year, s.GPA);  
}
```

```
void print_flight(Flight f){  
    printf("Flight #%s is at altitude %u\n", f.flightCode, f.altitude);  
    printf("%s has speed %f\n", f.flightCode, f.airSpeed);  
}
```

Passing structs as arguments

- We could also pass the struct via reference:

```
void print_flight_loc(Flight *f) {  
    printf("Flight #%s is at altitude %u\n", f->flightCode, f->altitude);  
    printf("%s has latitude: %f\n", f->flightCode, f->latitude);  
    printf("%s has longitude: %f\n", f->flightCode, f->longitude);  
}
```

- Which is cheaper in terms of memory/run-time stack?
 - What if we had an array of structs?

Structs within structs

- Nothing stops us from creating a struct composed of structs.

Suppose we have:

```
struct geoloc{
    float lattitude;
    float longitude;
};
```

- Then we can do:

```
typedef struct flight{
    char code[8];
    unsigned int arrival_time;
    unsigned int depart_time;
    struct geoloc origin;
    struct geoloc destination;
} Flight;
```

Example: Airport management

- Writing a struct to a file:

```
fwrite(void *ptr, size, n_memb, FILE *stream)
```

- `ptr` is pointer to instance of the struct to **write**
- `size` is the size in bytes of each element to be **written** (use `sizeof`)
- `n_memb` is the number of items to **write**, each with size of `size` bytes
- `stream` is the pointer to FILE object in ***binary write mode***.

Example: Airport management

- Writing a struct to a file:

```
fread(void *ptr, size, nmemb, FILE *stream)
```

- `ptr` is pointer to instance of the struct to **hold data**
- `size` is the size in bytes of each element to be **read** (use `sizeof`)
- `nmemb` is the number of items to **read**, each with size of `size` bytes
- `stream` is the pointer to FILE object in ***binary read mode***.

Exercise

- In a C file, use a loop and have the user input three records of the **Flight** struct.
 - Write this data to disk using **fwrite**.
- In another C file, read the data back to an array of **Flight** using **fread**.

```
struct geoloc{
    float lattitude;
    float longitude;
};
```

```
typedef struct flight{
    char code[8];
    unsigned int arrival_time;
    unsigned int depart_time;
    struct geoloc origin;
    struct geoloc destination;
} Flight;
```

Other user defined types: enums

- Enum is short for *enumeration*. Idea is to assign meaningful names to integers for code readability.

- **Syntax:** `enum [tag] {enumerator list};`

```
enum weekday {SUN, MON, TUE, WED, THR, FRI, SAT};
```

```
int is_workday(enum weekday day) {  
    if (day>SUN && day<SAT)  
        return 1;  
    else  
        return 0;  
}
```

Find out: Can you override default values assigned to enums?

Other user defined types: enums

```
int main(void) {  
  
    enum weekday today=THR;  
    enum weekday day_after_next = today+2;  
  
    printf("Today is day #%d of the week.\n", today);  
    printf("Today is %s\n", is_workday(today) ? "a workday" : "not a workday");  
  
    printf("\n");  
    printf("Day after tomorrow is day #%d of the week.\n", day_after_next);  
    printf("That day is %s\n",  
           is_workday(day_after_next) ? "a workday" : "not a workday");  
}
```