THE UNIVERSITY OF SYDNEY

COMP2017 / COMP9017 Tutorial 4

Structs, Unions, Bitfields and Files

Structs and Unions

Within C you can declare a new 'type' using structs. A struct is a collection of existing types with a constant size in memory defined at the time of compilation, and with a packing that is consistent within the architecture of the system and the version of the compiler.

Each field within the struct has its own name and can be called using the dot operator "."

```c
struct coordinate {
        int x;
        int y;
};

struct coordinate point;
point.x = 0;
point.y = 0;
printf("%d %d\n", point.x, point.y);
```

Similarly to arrays, structs can be initialised using curly braces.

```c
struct coordinate {
        int x;
        int y;
};

struct coordinate point = { 0, 0 };
//or, to specify fields
struct coordinate point = { .x = 0, .y = 0 };
```

As with all other types in C, we can have pointers to struct types.

```c
struct coordinate* point_ptr = &point;
```

Naively accessing fields of a pointer to a struct is somewhat arduous, but C has the arrow operator '->' to simplify this. The following statements are equivalent.

```
(*point).x;
point->x;
```

You may also be wondering about why we brought up that structs need to have a constant size in memory at compile time; this limits the objects that can be stored within a struct. For instance a struct cannot contain an array of non-constant size, though it can contain a pointer to an array located elsewhere in memory. Similarly a struct cannot contain another struct that hasn't been previously declared, though it can contain a pointer to a struct that has yet to be declared. This is possible as the size of a pointer is constant and does not depend on the type of the object it is pointing to.

Unions are a 'special' type of struct that can be used as one of multiple types listed within it. The size of the union is the size of the largest possible type it can take on.

```
union number
{
        int i;
        float f;
        double d;
};

union number n;
n.i = 10;
n.f = 10.05;
n.d = 12.02;
```

Note that unlike a struct, each of these are stored in exactly the same region of memory. When you change the value using the double representation and then attempt to read it as an integer, it will simply interpret the binary data within the first four bytes of this region of memory as an integer. This is **not** the same as casting and will result in garbage.

Pre-tutorial Work

Question 1: Sizeof

What is the size in bytes of each of the following objects? Do any of these change depending on the architecture of the computer they are used on?

```
int a;
int* b = &a;
int* c = NULL:

unsigned d;
short e;
long f;
size_t g;
long long h;

uint8_t i;
uint32_t j;

struct quoll
{
        char name[20];
        uint8_t age;
};

struct quokka
{
        char* name;
        struct quokka* quokka_father;
        struct quokka* quokka_mother;
};

union mammal
{
        struct quoll l;
        struct quokka a;
};
```

- What is the size of `int`, `short`, `long`?
- What is the size of `union mammel`, `struct quokka` and `struct quoll`
- What is the size of `struct quoll*` and `struct quokka*`?
- Compile with the `-m32` flag and report what the differences in sizes.

- Does the size of `uint8_t` and `uint32_t` change due to the `-m32` flag? What can be said about the portability of `stdint.h` types?

Question 2: Structs Properties

The following questions are based on the following code snippet:

```c
enum TYPE { FIRE, WATER, FLYING, ROCK, ELECTRIC };
struct pokemon {
    const char* name;
    enum TYPE type;
};
```

1. Which of the following code snippets compile?

   (a) `pokemon pikachu = { "Pikachu", ELECTRIC };`

   (b) `struct pokemon pikachu = { ELECTRIC, "Pikachu" };`

   (c) `struct pokemon pikachu = { "Pikachu", ELECTRIC };`

   (d) `struct pokemon pikachu = { .type = ELECTRIC, .name = "Pikachu" };`

   (e) `struct pokemon blank = { 0 };`

2. What assumptions can you make about `sizeof(struct pokemon)`?

3. What does the following code do?

```c
struct pokemon pikachu = { "Pikachu", ELECTRIC };
struct pokemon *ptr = &pikachu;
ptr->name = "Raichu";
ptr->type = ELECTRIC;
```

4. What would the following code do?

```c
void evolve(struct pokemon mon) {
    mon.name = "Raichu";
    mon.type = ELECTRIC;
}

int main() {
    struct pokemon pikachu = { "Pikachu", ELECTRIC };
    evolve(pikachu);
}
```

5. Based on the outcome from the previous code segment, what changes could you to the `evolve` function to ensure they modify the object.

Question 3: Greeter

You are to write a program that will play the role as a shop greeter, the program will record all customers that show up by asking them a few questions.

The greeter will ask them their name, age and what they are looking for, afterwards the greeter will allow them to continue into the store.

At the end of the day, the greeter will output all user data to the screen and close, ready for another adventure the following day.

```
$ ./greeter
Welcome to ShopaMocha,
Could you please tell me your name, age and what you looking for?
Lionel 25 Bees

Hrmm, I think you should talk to a ShopaMocha assistant to find "Bees"
Have a good day!
^D
Customer 0, Name: Lionel, Age: 25, Looking for: Bees
```

To make testing your code easier, create a few files that contain your keyboard input and use bash redirection when running your program.

Question 4: Files

Although piping and redirection on the Unix command line is very useful, this is a feature of Unix operating systems rather than C. The C standard library defines a simple way of interacting with files as shown below.

There are four steps to reading from and writing to text files in C.

1. Open the file using `FILE* fopen(const char* path, const char* mode);`
2. Read text from the file using `int fscanf(FILE* stream, const char* format, ...);`
3. Write text to the file using `int fprintf(FILE* stream, const char* format, ...);`
4. Close the file using `int fclose(FILE* fp);`

The `mode` specifies what operations are allowed as well as the behaviour of `fopen` if the file does not exist.

MODE	STREAM POSITION	RESULT
`r`	beginning of file	open file for reading
`r+`	beginning of file	open file for reading and writing
`w`	beginning of file	open for writing, creates an empty file if none exist truncates file to zero length if file exists
`w+`	beginning of file	open for reading and writing, creates an empty file if none exist truncates file to zero length if file exists
`a`	end of file	open for appending, creates an empty file if none exist
`a+`	beginning of file	open for reading and appending, creates an empty file if none exist output is always appended to the end of the file

The C11 standard introduces the `x` specifier that can be appended to `w`, for example: `wx` and `w+x` which causes `fopen` to fail if the file already exists. Now, use your knowledge of file I/O in C to extend the following code.

```c
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char** argv) {
    // attempt to open a file that does not exist
    FILE* file = fopen("missingno", "r");
    if (file == NULL) {
        perror("unable to open file");
        return 1;
    }
    fclose(file);
    // TODO: write to a file of your choice using fopen and fprintf
    // TODO: read from a file of your choice using fopen and fscanf
    // TODO: write to stdout and stderr using fprintf
    return 0;
}
```

Tutorial Work

Question 5: Discussion

Discuss with your tutor the answer to questions 1 and 2. Consider the struct alignment and if fields have been padded.

Question 6: `wc`

Implement the `wc` program in C. Your program should behave in a similar way to the Unix `wc` program. If no arguments were passed to your `wc`, read from `stdin` and print the number of lines, words and characters. For every argument that was passed to your `wc`, it should read the arguments as files and output the number of lines, words and bytes for each file with the respective file name, and also output the total counts for the files.

Use the `isspace` function to determine if a character is whitespace. A contiguous string of non whitespace characters count as a word. We recommend that you use the `fread` and `fwrite` functions for this exercise.

Compare your output to the Unix `wc`.

```
$ echo hello | ./wc
    1    1    6

$ ./wc file1 file2 file3
    <lines>    <words>    <bytes>    file1
    <lines>    <words>    <bytes>    file2
    <lines>    <words>    <bytes>    file3
    <lines>    <words>    <bytes>    total
```

Question 7: Replace words

Write a program that will search for words within a file and replace it with a word from command line arguments.

```
$ cat rick.txt
Always gonna give you up
Always gonna let you down
Always gonna run around and desert you
Always gonna make you cry
Always gonna say goodbye
Always gonna tell a lie and hurt you

$ ./replace_word rick.txt astley.txt Always Never
$ cat astley.txt
Never gonna give you up
Never gonna let you down
Never gonna run around and desert you
Never gonna make you cry
Never gonna say goodbye
Never gonna tell a lie and hurt you
```

Question 8: Scores to CSV

Using the `struct batsman` definition, output the fields of an array of `struct batsman` to a comma-separated value file (CSV). You can use your solution to Question 4 from last week's tutorial to help input data and test your program.

```
struct batsman {
        char first_name[20];
        char last_name[20];
        int score[20];
};
int output_scores(struct batsman* batter, const char* filename);
```

CSV Output Example:

```
Cameron,Bancroft,40
Mitchell,Marsh,67
David,Warner,59
Ben,McDermott,Duck
Cameron,White,78
Usman,Khawja,54
...
```

Check the different modes that a file can be opened with.

Extension: Create a program that will convert a tab-separated value file to a comma-separated value file and vice versa.

Question 9: Saving Pokemon

The following code reads the names and levels of four Pokemon from `stdin` and then dumps the contents of the `pokemons` array to a binary file using `fwrite`.

Examine the contents of binary file with the `xxd` hex dump program. Does it contain what you expect? Is `sizeof(pokemon)` larger than its components, if so, why? How are the data types represented in memory?

```c
#include <stdio.h>

#define SIZE 4

typedef struct pokemon {
        char name[100];
        unsigned level;
} pokemon;

int main(void) {

        pokemon pokemons[SIZE];

        for (size_t i = 0; i < SIZE; i++) {
                if (scanf("%99s %u",
                                pokemons[i].name,
                                &pokemons[i].level) != 2) {
                        fprintf(stderr, "unable to read input\n");
                        return 1;
                }
        }

        printf("sizeof(size_t) = %zu\n", sizeof(size_t));
        printf("sizeof(unsigned) = %zu\n", sizeof(unsigned));
        printf("sizeof(char[100]) = %zu\n", sizeof(char[100]));

        printf("sizeof(pokemon) = %zu\n", sizeof(pokemon));
        printf("sizeof(pokemons) = %zu\n", sizeof(pokemons));
        // attempt to save to file
        FILE* file = fopen("pokemon.dat", "w");
        if (file == NULL) {
                perror("unable to open file for writing");
                return 1;
        }
        fwrite(pokemons, sizeof(pokemon), SIZE, file);
        fclose(file);
        return 0;
}
```

Question 10: Working with a stream

You will need to write a program that will read from a stream. This stream emulates data provided by a gamepad. You can access the program from the resources section. This program will create a file that the process and send data to that file. Your program will need to receive those packets and decode them.

Use the data layout below to help with decoding the data.

```
id: unsigned byte
locked: unsigned byte
buttons: 2 bytes
  x: unsigned bit
  y: unsigned bit
  z: unsigned bit
  w: unsigned bit
  a: unsigned bit
  b: unsigned bit
  c: unsigned bit
  d: unsigned bit
  l: unsigned bit
  r: unsigned bit
  st: unsigned bit
  sel: unsigned bit
analog: 8 bytes
  left: float
  right: float
```

After receiving the packet, the reading process needs to output what buttons have been pressed and the current state of the analog sticks. If a button has been released it will need to show this state information between one read and another. If a button has maintained its current value for more than 3 packets, it is considered to be a `Hold`.

Example:

```
PKT0: Analog Left: 0.0000, Analog Right: 0.0000
PKT1: X: Pressed, Analog Left: 0.0000, Analog Right: 0.0000
PKT2: X: Released, Analog Left: 0.0000, Analog Right: 0.0000
PKT3: X: Pressed, Analog Left: 0.0000, Analog Right: 0.0000
PKT4: X: Pressed, Analog Left: 0.2400, Analog Right: 0.0000
PKT5: X: Pressed, Analog Left: 0.2400, Analog Right: 0.0000
PKT6: X: Hold, Analog Left: 0.0000, Analog Right: 0.0000
...
```