

# CS 261 Spring 2024

Mike Lam, Professor



## Structs and I/O

# Typedefs

- A `typedef` is a way to create a new type name
  - Basically a synonym for another type
  - Useful for shortening long types or providing more meaningful names
  - Names are usually postfixed with "\_t"

```
typedef unsigned char byte_t;
```

```
byte_t b1, b2;
```

- Use the `size_t` typedef (defined to be the same as `long unsigned int` in the `stu` headers) for non-negative sizes and counts

```
const size_t STR_SIZE = 1024;
```

# Structs

- A **struct** contains a group of related sub-variables
  - New "kind" of type
  - Similar to classes from Java, but without methods and everything is "public"
  - Sub-variables are called **fields**
  - Struct variables are declared with **struct** keyword

```
struct vertex {  
    double x;  
    double y;  
    bool visited;  
};
```

```
int main()  
{  
    struct vertex p1;  
    p1.x = 4.2;  
    p1.y = 5.6;  
    p1.visited = false;  
}
```

```
double dist(struct vertex p1, struct vertex p2)  
{  
    return sqrt( (p1.x-p2.x)*(p1.x-p2.x) +  
                (p1.y-p2.y)*(p1.y-p2.y) );  
}
```

# Typedef structs

- Convention: create a typedef name for struct types
  - E.g., `struct vertex -> vertex_t`
  - More concise and readable
  - For projects, we'll provide structs and typedefs in headers

```
typedef struct vertex {  
    double x;  
    double y;  
    bool visited;  
} vertex_t;
```

```
double dist(vertex_t p1, vertex_t p2)  
{  
    return sqrt( (p1.x-p2.x)*(p1.x-p2.x) +  
                (p1.y-p2.y)*(p1.y-p2.y) );  
}
```

```
int main()  
{  
    vertex_t p1;  
    p1.x = 4.2;  
    p1.y = 5.6;  
    p1.visited = false;  
}
```

# Struct memory layout

- Fields are stored (mostly) contiguously in memory
  - Each field has a fixed **offset** from the beginning of the struct

offset 0



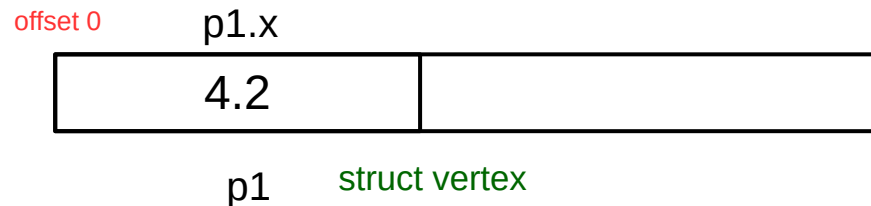
p1 struct vertex

```
typedef struct vertex {  
    double x;  
    double y;  
    bool visited;  
} vertex_t;
```

```
int main()  
{  
    vertex_t p1;  
    p1.x = 4.2;  
    p1.y = 5.6;  
    p1.visited = false;  
}
```

# Struct memory layout

- Fields are stored (mostly) contiguously in memory
  - Each field has a fixed **offset** from the beginning of the struct

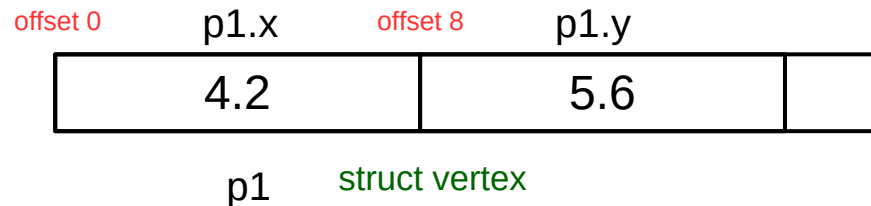


```
typedef struct vertex {  
    double x;  
    double y;  
    bool visited;  
} vertex_t;
```

```
int main()  
{  
    vertex_t p1;  
    p1.x = 4.2;  
    p1.y = 5.6;  
    p1.visited = false;  
}
```

# Struct memory layout

- Fields are stored (mostly) contiguously in memory
  - Each field has a fixed **offset** from the beginning of the struct

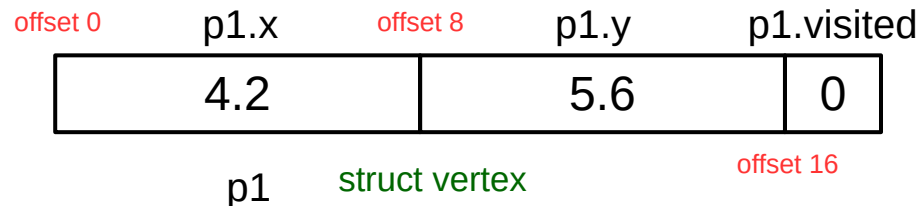


```
typedef struct vertex {  
    double x;  
    double y;  
    bool visited;  
} vertex_t;
```

```
int main()  
{  
    vertex_t p1;  
    p1.x = 4.2;  
    p1.y = 5.6;  
    p1.visited = false;  
}
```

# Struct memory layout

- Fields are stored (mostly) contiguously in memory
  - Each field has a fixed **offset** from the beginning of the struct



```
typedef struct vertex {  
    double x;  
    double y;  
    bool visited;  
} vertex_t;
```

```
int main()  
{  
    vertex_t p1;  
    p1.x = 4.2;  
    p1.y = 5.6;  
    p1.visited = false;  
}
```



# Struct memory layout

- Given the following code, how much space will be allocated for the "data" variable? Assume chars are one byte each and ints are four bytes each.

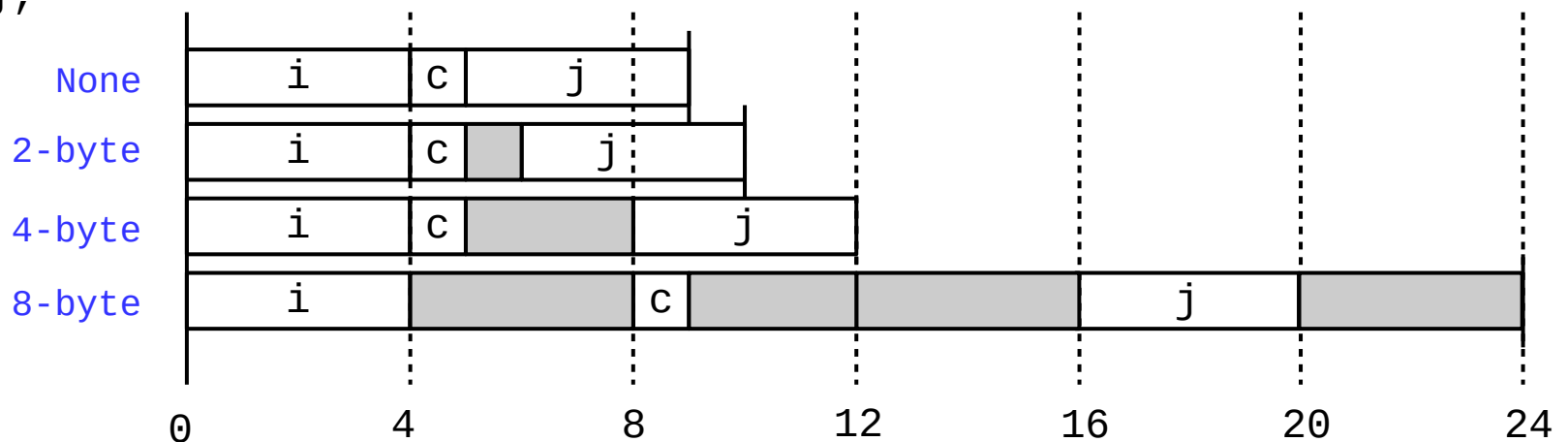
```
struct stuff {  
    char a;  
    char b;  
    char c;  
    int x;  
} data;
```

- A) 4 bytes
- B) 7 bytes
- C) 8 bytes
- D) 16 bytes
- E) There is not enough information to know.

# Struct data alignment

- **Alignment restrictions** require addresses be  $n$ -divisible
  - E.g., 4-byte alignment means field offsets must be divisible by 4
  - Chosen by compiler based on hardware
  - Improves memory performance
  - Can be avoided in C using “attribute (packed)” (as in `elf.h`)

```
struct {  
    int i;  
    char c;  
    int j;  
} rec;
```



# Function parameters

- In C, parameters are passed **by value**
  - Values are copied to a function-local variable at call time
  - Local changes are not visible to the caller unless returned
- It is expensive to pass large structs by value
  - Must copy the entire struct even if it is not all needed
  - Alternative: pass variables **by reference** using a pointer
  - Local changes **through the pointer** are visible to the caller
  - Local changes **to the pointer** are **not** visible to the caller
- Parameters can be passed as `const`
  - Shouldn't be changed by the function (checked by compiler)
  - Useful for ensuring you don't accidentally overwrite a by-reference parameter pointer

# Struct pointers

- New "->" (arrow) operator dereferences a pointer to a struct and accesses a field in that struct

```
vertex_t v;  
vertex_t *vp = &v;  
(*vp).x = 1.0;           // set field "x"  
vp->y = 2.0;           // set field "y"
```

```
typedef struct vertex {  
    double x;  
    double y;  
    bool visited;  
} vertex_t;
```

Faster!  
(copy 8-byte pointer instead of 17-byte struct)

```
double dist(vertex_t *p1, vertex_t *p2)  
{  
    return sqrt( (p1->x - p2->x) * (p1->x - p2->x) +  
                (p1->y - p2->y) * (p1->y - p2->y) );  
}
```

# Aside: Enums

- An **enumeration** is a type where all values are listed
  - Declared in C using enum keyword
  - In C, the actual values are stored as integers
  - Can assign integer values if desired
  - Primary advantage: named constants

```
typedef enum {  
    MON = 1, TUE, WED, THU, FRI, SAT, SUN  
} day_t;
```

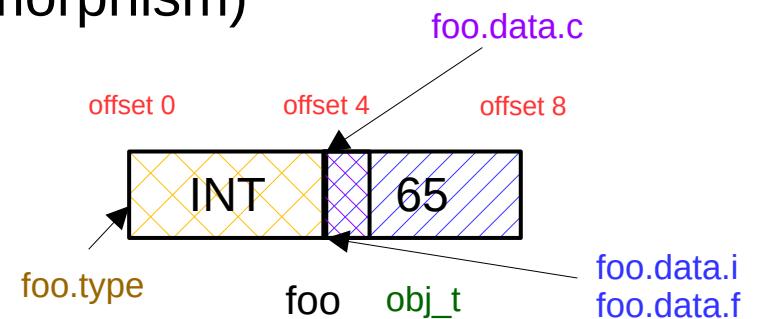
```
// essentially the same as: int midterm_day = 3;  
day_t midterm_day = WED;
```

# Aside: Unions

- A **union** is also a variable that can store data of different types
  - One “thing”, but it could be multiple sizes depending on what kind of “thing” it currently is (so context is even more important!)
  - All “fields” start at offset zero
  - Generally a bad idea! (circumvents the type system in C)
  - Can be used to do OOP in C (i.e., polymorphism)

```
typedef enum { CHAR, INT, FLOAT } objtype_t;
```

```
typedef struct {  
    objtype_t type;  
    union {  
        char c;  
        int i;  
        float f;  
    } data;  
} obj_t;
```



```
obj_t foo;
```

```
foo.type = INT;  
foo.data.i = 65;
```

```
printf("%c", foo.data.c); ← VALID!
```

# File I/O

- C standard library provides **opaque** file stream handles: `FILE*`
  - Internal representation is implementation-dependent
- File manipulation functions:
  - Open a file: `fopen`
    - Mode: read ('r'), write ('w'), append ('a')
  - Read a character: `fgetc`
  - Read a line of text: `fgets`
  - Read binary data: `fread`
  - Set current file position: `fseek`
  - Write formatted text: `fprintf`
  - Write binary data: `fwrite`
  - Close a file: `fclose`

**These are all documented  
in the function reference  
(on website)**

# Standard I/O

- Standard "file" streams: `stdin`, `stdout`, `stderr` (type is `FILE*`)
  - Like `System.in`, `System.out`, and `System.err` in Java
  - Available to all programs; no need to open or close
  - Flushed when newline (`'\n'`) encountered (included by `fgets!`)
  - Use CTRL-D to indicate end-of-file when typing input from the terminal
- Formatted input/output (`scanf` / `printf`)
  - Variable number of arguments (`varargs`)
  - Format string and type specifiers:
    - `%d` for signed int, `%u` for unsigned int
    - `%c` for chars, `%s` for C strings (`char *`, passing NULL is undefined behavior)
    - `%f` or `%e` for float, `%x` for hex, `%p` for pointer
    - Prepend `'l'` for long or `'ll'` for long long (e.g., `%lx` = long hex)
    - Include number for fixed-width field (e.g., `%20s` for a 20-character field)
    - Many more useful options; see documentation for details



# Standard I/O

- What is wrong with the following code?

```
char buffer[20];
```

```
fgets(buffer, 30, stdin);
```

- A) The buffer is not initialized before calling fgets.
- B) The buffer is the wrong size.
- C) The buffer size parameter is wrong.
- D) The call to fgets has too few parameters.
- E) There is nothing wrong with this code.

# Security issues

- Input: beware of buffer overruns
  - Like carelessly copying strings, reading input improperly is a common source of security vulnerabilities
  - Best practice: declare a fixed-size buffer and use “safe” input functions (e.g., `fgets`)
  - You may NOT use unsafe functions in this course! (e.g., `gets`)
  - Here is a partial list of unsafe functions; see function reference on website for complete list

## UNSAFE

`atoi`  
`atof`  
`gets`  
`strcat`  
`strcpy`

## Safer alternative

`strtol`  
`strtod`  
`fgets`  
`strncat`  
`snprintf`

**Be careful with code that you find online—never use code that you don't fully understand or that you haven't verified to be safe.**

# Projects

- You are now a C programmer!
  - We have now covered all topics necessary for P0 and P1
  - There is certainly more to learn about C, but we have covered all the necessary topics for this course
  - References and resources on our website
  - Next time, we'll cover a few more useful things and some technicalities that we've glossed over
  - Now all you need is practice :)

# Exercise

- Let's write a simple version of the 'cat' utility
  - Copy all text from standard in to standard out
    - No need to open/close a “real” file
  - Handle a line at a time
    - To reduce memory requirements
  - What is the basic form of our code?
    - What variable(s) will we need?

# Simple “cat” program

```
#include <stdio.h>

int main (int argc, char **argv)
{
    const int BUF_SIZE = 1024;
    char buffer[BUF_SIZE];

    while ( ) {
        printf("%s", buffer);
    }

    return 0;
}
```

**CS 261 C function reference:**

[w3.cs.jmu.edu/lam2mo/cs261/c\\_funcs.html](http://w3.cs.jmu.edu/lam2mo/cs261/c_funcs.html)

## File I/O

- `FILE* fopen` (`char *filename, char *mode`)  
*Open a file (modes: 'r', 'w', 'a')*
- `int fgetc` (`FILE *stream`)  
*Read a single character from a file*
- `char* fgets` (`char *str, int count, FILE *stream`)  
*Read a line of text from a file*
- `int fscanf` (`FILE *stream, char *format, ...`)  
*Read formatted data from a file (scanf assumes stdin)*
- `size_t fread` (`void *buffer, size_t size, size_t count, FILE *stream`)  
*Read (size x count) bytes from a file*
- `int fseek` (`FILE *stream, long offset, int origin`)  
*Set the current file position (origin: 'SEEK\_SET', 'SEEK\_CUR')*
- `int fprintf` (`FILE *stream, char *format, ...`)  
*Write formatted text to a file (printf assumes stdout)*
- `size_t fwrite` (`void *buffer, size_t size, size_t count, FILE *stream`)  
*Write (size x count) bytes to a file*
- `int fclose` (`FILE *stream`)  
*Close a file*

# Documentation

## fgets

the 'restrict' keyword means "this is the only active pointer to this variable"

Defined in header `<stdio.h>`

```
char *fgets( char *str, int count, FILE *stream ); (until C99)
char *fgets( char *restrict str, int count, FILE *restrict stream ); (since C99)
```

Reads at most `count - 1` characters from the given file stream and stores them in the character array pointed to by `str`. Parsing stops if end-of-file occurs or a newline character is found, in which case `str` will contain that newline character. If no errors occur, writes a null character at the position immediately after the last character written to `str`.

The behavior is undefined if `count` is less than 1.

### Parameters

- `str` - pointer to an element of a char array
- `count` - maximum number of characters to write (typically the length of `str`)
- `stream` - file stream to read the data from

### Return value

`str` on success, null pointer on failure.

If the failure has been caused by end-of-file condition, additionally sets the `eof` indicator (see `feof()`) on `stream`. The contents of the array pointed to by `str` are not altered in this case.

If the failure has been caused by some other error, sets the `error` indicator (see `ferror()`) on `stream`. The contents of the array pointed to by `str` are indeterminate (it may not even be null-terminated).

# Simple “cat” program

```
char *fgets( char *restrict str, int count, FILE *restrict stream );    (since C99)
```

## Return value

str on success, null pointer on failure.

```
#include <stdio.h>
```

```
int main (int argc, char **argv)
{
    const int BUF_SIZE = 1024;
    char buffer[BUF_SIZE];

    while ( ) {
        printf("%s", buffer);
    }

    return 0;
}
```



# Simple “cat” program

```
char *fgets( char *restrict str, int count, FILE *restrict stream ); (since C99)
```

```
#include <stdio.h>
```

```
int main (int argc, char **argv)  
{
```

```
    const int BUF_SIZE = 1024;  
    char buffer[BUF_SIZE];
```

```
    while (fgets( , , ) != NULL) {  
        printf("%s", buffer);  
    }
```

```
    return 0;
```

```
}
```

# Simple “cat” program

```
char *fgets( char *restrict str, int count, FILE *restrict stream ); (since C99)
```

```
#include <stdio.h>
```

```
int main (int argc, char **argv)
{
    const int BUF_SIZE = 1024;
    char buffer[BUF_SIZE];

    while (fgets(buffer,           ,           ) != NULL) {
        printf("%s", buffer);
    }

    return 0;
}
```

# Simple “cat” program

```
char *fgets( char *restrict str, int count, FILE *restrict stream ); (since C99)
```

```
#include <stdio.h>
```

```
int main (int argc, char **argv)  
{
```

```
    const int BUF_SIZE = 1024;  
    char buffer[BUF_SIZE];
```

```
    while (fgets(buffer, BUF_SIZE,           ) != NULL) {  
        printf("%s", buffer);  
    }
```

```
    return 0;
```

```
}
```

# Simple “cat” program

```
#include <stdio.h>

int main (int argc, char **argv)
{
    const int BUF_SIZE = 1024;
    char buffer[BUF_SIZE];

    while (fgets(buffer, BUF_SIZE, stdin) != NULL) {
        printf("%s", buffer);
    }

    return 0;
}
```

# Exercise

- Write a program that reverses every line from standard in (`stdin`)
  - Reminder: to compile your program (after creating `rev.c`):

```
gcc -o rev rev.c
```
  - To test your program (after creating `input.txt`):

```
./rev <input.txt
```

 (or just `./rev` and type text followed by CTRL-D)

*Hint: use `fgets()` to read the input a line at a time into a char array, printing the characters in reverse after reading each line*

`char* fgets (char *str, int count, FILE *stream)`  
*Read a line of text input from a file (returns str, count is max chars)*

`size_t strlen (char *str)`  
*Calculate the length of a null-terminated string*

## Sample input:

```
Hello, world!  
My name is Bob.
```

ENOD

## Sample output:

```
!dlrow ,olleH  
.boB si eman yM
```

DONE