MCS 2050

Week 3: Types and Memory Management

Enums

Enums

- Enums allow you to give more meaningful names to a limit series of values, rather than using “magic numbers”
- You can **enumerate** different options with enums

- In previous lectures, we mostly stuck with C/C++ that has been around for a long time
- We do this partly because it’s usually simpler, and newer features are generally more advanced
- C++11 introduces “enum classes” that are way better than the old way, so that’s what we’ll use

Enums

- Consider a function that takes in some parameter that could be one of many options.
- We can make those options be strings, for example:

```
void foo(const char* fruit) {
        if (strcmp(fruit, “banana”) == 0)
                doSomething();
        else if (strcmp(fruit, “apple”) == 0)
                doSomethingElse();
}
```

Enums

- Problem: string comparisons are kinda slow
- strcmp potentially checks the entire strings to see if there’s a difference
- We could instead do these comparisons very fast with integers:

```
void foo(int fruit) {
        if (fruit == 0) // banana
                doSomething();
        else if (fruit == 1) // apple
                doSomethingElse();
}
```

Enums

- Problem: what does 0 mean? What kind of fruit is a 1?
- We can know this by context in the program, or with things like comments
- But it’s not good that we need to remember what value is what fruit

- Enums give us a way to name things:

```
void foo(Fruit fruit) {
        if (fruit == Fruit::banana)
                doSomething();
        else if (fruit == Fruit::apple)
                doSomethingElse();
}
```

Enums

- Enums are the best of both worlds
- Readable like strings
- Fast like ints

- They’re also safer
- Spelling mistakes are caught by the compiler

```
void foo(const char* fruit) {
        if (strcmp(fruit, “bananananananana”) == 0)
                doSomething();
}
```

- Enums would catch this

Enums

- Defining an enum is simple:

```
enum class Fruit {
        banana,
        apple,
        orange
};
```

Enums

- You can use enums like they were ints
- Except its values are limited to what we’ve defined
- In the previous example, Fruit::banana, Fruit::apple, and Fruit::orange

```
Fruit fruit1 = Fruit::banana;
Fruit fruit2 = Fruit::apple;
Fruit fruit3 = Fruit::orange;

Fruit fruit4 = Fruit::pear; // ERROR, no pear enum value defined
```

Enums

- The reason we can use enums like ints is because, internally, they are ints
- The first enum entry is 0, then each one that follows increases by 1

```
printf("%d\n", Fruit::banana); // 0
printf("%d\n", Fruit::apple); // 1
printf("%d\n", Fruit::orange); // 2
```

Enums

- We can also set enums to use a specific value, if we want

```
enum class Fruit {
        banana = 10,        // 10
        apple,              // 11
        orange              // 12
};
```

Enums

- Practical example might be HTTP status codes

```
enum class Response {
    ok = 200,
    // ...
    notFound = 404,
    // ...
};
```

Structs

Structs

- So far all we've seen are primitive types and pointers to them (and therefore, strings)
- But any decent programming language needs to be able to build composite types out of these primitive types
- In C, that's done with a **struct**:

```
struct NameOfType {
        <list of member variables>
};
```

Structs

- In older compilers and in C-only compilers you might also see:

```
typedef struct NameOfType {
        <list of member variables>
} NameOfType;
```

- We’ll only use the former in this class
- Note: we’ll be covering this from the point of view of C
- C++ adds more capabilities to structs, which we’ll see next week

Structs

- Member variables in structs can be primitive types, or other structs:

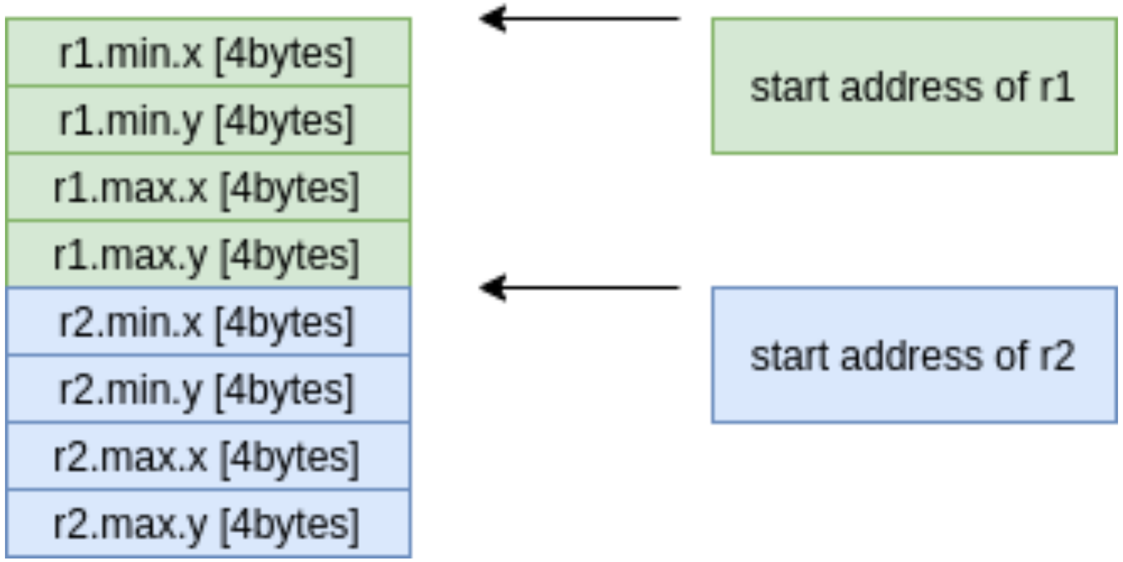
```
struct Point {
        float x;
        float y;
}

struct Rect {
        Point topleft;
        Point bottomright;
}
```

Structs

- Members are arranged in memory sequentially, in the same order as they were declared

```
Rect r1;
Rect r2;
```

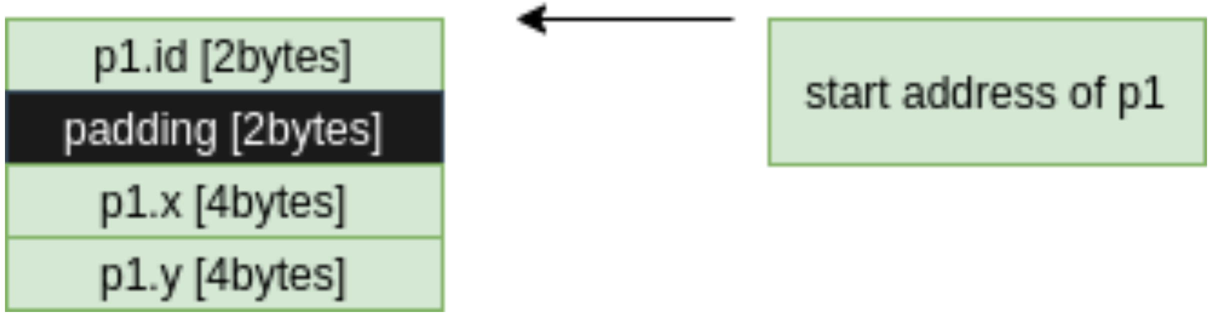


Structs

- Depending on platform, extra padding might get added to adhere to CPU restrictions on alignment

```
struct Point {
        short id;
        int x;
        int y;
};

Point p1;
```



Structs

- Padding doesn’t happen on Intel, but may happen on other platforms like ARM
- Need to be careful/aware of this if doing “bad” things like casting and pointer arithmetic
- Means that casting and assuming memory layout is bad code, since it can be platform dependent

```
Point p1;
char* p = (char*) &p1;

short id = *((short*)(p)); // this is technically fine
int x = *((int*)(p+2)); // this will be wrong if there’s ever platform-dependent padding
```

Struct

- As we saw in previous examples, you can create structs like any other type:

```
Point p1;
Point p2, p3;
```

Structs

- Accessing the members of a struct depends on whether the variable is an instance of a struct on the stack (as will be explained soon) or whether it's being accessed through a pointer
- Without a pointer:

```
Point p;
p.x = 10.0;
p.y = 5.0;

Rect r;
r.topleft.x = 10.0;
r.topleft.y = 10.0;
r.bottomright.x = 50.0f;
r.bottomright.y = 50.0f;
```

Structs

- If you have a pointer to a struct, it’s awkward to have to dereference and then use dot syntax.
- This can be done, but it’s annoying:

```
void foo(Point* p) {
        (*p).x = 10.0;
}
```

Instead, there is a shorthand via the -> syntax:

```
void foo(Point* p) {
        p->x = 10.0;
}
```

Structs

- If you pass a struct to a function as a parameter, you’re making a copy of that struct, and all its members (and their members and so on).

```
struct Matrix {
	float m[16];
};

void foo(Matrix matrix) {
	matrix.m[0] = 1.0;
	…
}
```

Struct

- In that last example, with the function signature “void foo(Matrix matrix)” a copy of matrix will be created at the call to foo
- This copy would only be used within foo
- Same behavior as primitive types

- This can be bad if structures are big, because it needs to copy all that data
- And if we want a function to modify an existing struct, this won’t work

Structs

- We can instead pass pointers to structs in C (or references in C++, as we'll explain soon)
- The copy is only a pointer – 4 or 8 bytes (32 or 64 bit architecture)
- Remember const correctness!

```
void setIdentity(Matrix* pMatrix) {
        pMatrix->m[0] = 1.0;
        …
}
void printMatrix(const Matrix* pMatrix) {
        printf("%.2f ", pMatrix->m[0]);
        …
}
Matrix matrix;
setIdentity(&matrix);
printMatrix(&matrix);
```

Memory Allocation: The Stack

The Stack

- The stack is a region of memory given to the program to manage all its temporary data.

- C/C++ is a stack-based language (like almost all languages).
- Local variables are allocated on the stack, and the stack grows and shrinks as functions are called and return

The Stack

- We'll use the notation [a,b,c] to show the stack, where a is the value lowest on the stack and c is the most recent
- This is not quite accurate, as a / b / c would all likely be different sizes, taking up multiple bytes of memory
- We'll use that stack notation to indicate the state of the stack after the CPU executes that line of code

```
int main() {
        int a = 4; // [4]
        int b = 5; // [4, 5]
}
```

The Stack

- Copies of parameters are pushed to the stack prior to the CPU jumping to the function's code
- Local variables of the function being called continue to add to the stack

```
int foo(int param1, int param2) {
        int foo1 = 10;          // [4,5,4,5,10]
        int foo2 = 20;          // [4,5,4,5,10,20]
}
int main() {
        int a = 4;              // [4]
        int b = 5;              // [4,5]
                                // [4,5,4,5], copying parameters
        foo(a,b);
                                // [4,5], popping off the stack as function has returned
        int c = 6;              // [4,5,6]
}
```

The Stack

- Notice how the data is only on the stack while it is “in scope”

```
int main() {
        int a = 4;                // [4]
        int b = 5;                // [4,5]

        if (b > a)
                int c1 = 6;       // [4,5,6]
        else
                int c2 = -1;      // never ends up on stack
                                  // c leaves scope, popped off the stack
        int d = 7;                // [4,5,7]
}
```

The Stack

- return values are placed on the stack if assigned to a local variable

```
int add(int a, int b) {
        return a + b;
}

int main() {
        int a = 4;                  // [4]
        int b = 5;                  // [4,5]
                                    // [4,5,4,5]  prior to jumping to add
        int c = add(4,5);           // [4,5,9]
}
```

The Stack

- What is wrong with the following code?

```
int* getAnswer() {
        int answer = do_some_calculation();
        return &answer;
}
int add(int a, int b) {
        return a + b;
}
int main() {
        int *pAnswer = getAnswer();
        int x = add(10, 20);
        …
}
```

The Stack

- Most platforms have a fixed size for the stack, or protections to stop it growing too large
- If the stack ends up growing past that maximum size, it will crash the program
- This is called a **Stack Overflow**
- One of the common ways to get this problem is with recursion: calling recursively too many times (especially infinite recursion!) and/or creating too large local variables during recursion:

```
void recurseMe(int times) {
        int bigLocalBuffer[1024];
        do_something(bigLocalBuffer);
        if (times > 0)
                recurseMe(times – 1);
}
recurseMe(9999999);
```

Memory Allocation: The Heap

The Heap

- Most memory an application will use will live on the **heap**.
- The heap is a much larger area of memory than the stack, and it can grow much larger than the stack.
- The heap is needed whenever big allocations of memory need to be made, or when objects have more complicated lifetimes than the stack allows.
- The application can, at any time, ask to be given (allocate) a bunch of memory of a specific size.
- So long as the system has memory available, it will update its bookkeeping to know that memory is now taken, and return a pointer to the start of the allocation.

The Heap

- After allocation, that memory stays with the application until it returns it back to the system
- If an application continues to allocation but never returns memory, this is known as a memory leak, and can eventually lead to the program crashing

- Note that allocating and returning memory to/from the heap is a relatively costly operation (compared to the stack)
- Process involves searching, coalescing, and bookkeeping

The Heap

- Two main ways to allocate memory on the heap, the C way and the C++ way
- C way uses **malloc** and gets a void* in return

```
void* pBuffer = malloc(1024*1024); // allocates 1MB buffer
```

The Heap

- Since malloc returns a void*, it doesn't care what lives there
- You're responsible for allocating enough space for your needs
- You will have to cast it to another type to use it

```
void* pBuffer = malloc(1024*1024); // Allocate 1MB buffer
int* pAsInts = (int*)pBuffer;
for (int i = 0; i < (1024*1024)/sizeof(int); ++i)
        pAsInts[i] = 0;
```

The Heap

- The C++ way can be more complex, but offers more type safety
- We make use of the **new** keyword in C++ to allocate memory
- When using new, we allocate 1 or more “objects” in a row, where an “object” could be a primitive type (int, char, etc.) or a user type (struct)
- For example, to allocate a buffer of 100 ints, we can write:

```
int* pBuffer = new int[100];
```

- Note how, unlike C’s malloc, we’re now specifying the type of data that will live at the address
- This means we get the appropriately typed pointer back, not a void*

The Heap

- If you wanted to allocate a “generic buffer” like we did with malloc, we can allocate a char buffer of the exact byte size we need:

```
char* pBuffer = new char[bufferSize];
```

- We could then cast that to whatever we need (at the normal risks of doing so)

The Heap

- Allocating space for a series of structs is identical
- C++ will calculate how many bytes are needed from the type and quantity

```
struct Vec3 {
	float x, y, z;
};

Vec3 pPoints = new Vec3[getNumVerts()];
```

The Heap

- To allocate a single primitive, we omit the [] brackets:

```
int* pI = new int;
float* pF = new float;
```

- The same can be done for structs, but for reasons we'll see next week, it's better to also add ():

```
Vec3* p = new Vec3();
```

The Heap

- Each way of allocating memory also has a way to return memory (deallocate memory) back to the heap
- Generally speaking, you cannot mix and match these functions
- If you use malloc, you need to also return the memory the C++ way

The Heap

- In C we use **free()** to return memory previously allocated from malloc:

```
void* p = malloc(1024);
free(p);
```

- If you pass a pointer to free that was not previously returned from malloc, your program might crash. The same is true if you free too many times:

```
void* p = malloc(1024);
free(p);
free(p);
```

- Probably won’t look like this, but it’s easier to accidentally do this than it looks

The Heap

- C++ gives us a **delete** keyword

```
Vec3* p = new Vec3();
delete p;
```

- While it won’t always be an error, you should not use delete in this way for arrays you allocated

The Heap

- If you used [] when allocating, you should use [] when deleting too
- The reason why won’t make sense until next week’s material, so trust me for now

```
Vec3* p = new Vec3[100];
float* pFloats = new float[1000];
delete[] p;
delete[] pFloats;
```

- If you don’t include the [], behaviour is undefined (your program could crash, depending on the runtime)

References

References

- C++ introduced a different take on pointers that aimed to both simplify and improve safety (make it harder to do the wrong thing): **references**
- References are basically pointers, but you can't do pointer arithmetic or other footguns that can get you into trouble with pointers.
- A reference **refers** to another object (or primitive value), much like a pointer points to one
- They come with different syntax but, confusingly, reuse the & operator but in a different context

- When an & comes after stating a type, you are saying it is a **reference** to a value of that type

```
void foo(int& a, int& b) { }
```

- This is unrelated to "int* p = &i" syntax before a variable

References

- Assigning references to variables is simpler than the pointer equivalent

```
int i = 10;
int& r = i;
int* p = &i;
i = 40;
printf("%d %d %d\n", i, r, *p); // 40 40 40
```

- Note that getting the address of the value, and dereferencing the underlying pointer, is all hidden from the programmer

References

- If you have a reference to a struct (or class, for later) you use the dot notation to access its members, much like you would with a stack copy of a struct

```
void printEntity(const Entity& entity) {
        printf("Entity's health is %d\n", entity.health);
}
```

- Compared to the pointer approach:

```
void onEntityDamaged(const Entity* entity) {
        printf("Entity's health is %d\n", entity->health);
}
```

References

- Const correctness still applies – unless you have good reason not to, you should always use a const reference over a normal reference
- This means you have a reference to another variable, but can only look at it (read from it), not change it (write to it):

```
void onEntityDamaged(const Entity& entity) {
	entity.health -= 10; // ERROR: cannot change const entity
}
void onEntityDamaged(Entity& entity) {
	entity.health -= 10;
}
```

Bitwise Operations

Bitwise Operations

- This section is brought to you in the spirit of “there was nowhere else I could fit it in!”

- Games are used a lot in games, embedded systems, high performance computing, etc. where low level access to the hardware is sought
- When working at that level, it’s important to be comfortable with manipulating individual bits

Bitwise Operations

- Why? Isn't memory/storage cheap?
- It is cheap, but we sometimes operate directly on bits for many reasons, such as:

1. Setting individual bits on I/O registers on consoles
2. Compressing data
3. Storing multiple flags in one byte/short/int
4. Some problems can be solved very efficiently with bitwise approaches

Bitwise Operations

- You can set individual bits on a number (int, char, short, etc.) by bitwise-OR'ing the number with another number where that bit is set, using a single |.

```
unsigned char i = 3;
i |= 0x80; // set bit 7
printf("%d\n", i); // 131
```

Bit	7	6	5	4	3	2	1	0
Bit Value	128	64	32	16	8	4	2	1
i = 3	0	0	0	0	0	0	1	1
0x80	1	0	0	0	0	0	0	0
i | 0x80	1	0	0	0	0	0	1	1

Bitwise Operations

- You can check if a particular bit is set on a number by bitwise-AND'ing the number with another number where that bit is set, using a single &

```
unsigned char i = 3;
if ((i & 0x2) != 0)
        printf("Second bit is set.\n");
```

Bit	7	6	5	4	3	2	1	0
Bit Value	128	64	32	16	8	4	2	1
i = 3	0	0	0	0	0	0	1	1
0x2	0	0	0	0	0	0	1	0
i & 0x2	0	0	0	0	0	0	1	0

Bitwise Operations

- Depending on what you compare to, you can check if ANY of a series of bits are set, or check if all are set:

```
if ((i & 0xF0) != 0)
        printf("At least one of the top 4 bits are set\n");
if ((i & 0xF0) == 0xF0)
        printf("All top 4 bits are set\n");
```

Bitwise Operations

- You can flip all the bits in a number with the use of the ~ operator

```
unsigned char i = ~3;
unsigned char j = ~0xF0;
```

Bit	7	6	5	4	3	2	1	0
3	0	0	0	0	0	0	1	1
~3	1	1	1	1	1	1	0	0
0xF0	1	1	1	1	0	0	0	0
~0xF0	0	0	0	0	1	1	1	1

Bitwise Operations

- One reason you'd want to invert bits is to clear a bit
- You can combine bitwise-AND'ing and inverting

```
unsigned char i = 0xF
i = i & ~0x3;
```

- Or, more succinctly (and commonly):

```
unsigned char i = 0xF;
i &= ~0x3;
```

Bitwise Operations

- We can shift bits left or right with bitshift operators (like double-arrows):

```
int i = 1 << 0;     // i = 1 (0001b)
i = i << 1;         // i = 2 (0010b)
i = i << 1;         // i = 4 (0100b)
i = i << 1;         // i = 8 (1000b)
i = i >> 2;         // i = 2 (0010b)
```

- Notice that shifting left is multiplying by 2 (or powers of 2), and shifting right is dividing by 2 (or powers of 2)

Bitwise Operations

- In embedded systems (and video game consoles) there are often “I/O registers” which are a bit pattern at a specified memory address
- Various bits do different things to the hardware, for example the documentation might state:

```
#define DISP_REGISTER (*(unsigned short*)0x80004000)
#define DISPLAY_BIT_SCREEN_ENABLE 0x8000
#define DISPLAY_BIT_SCREEN_VBLANK 0x4000
```

DISP_REGISTER: 0x80004000															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SE	VB							DISPHEIGHT				DISPWIDTH			

SE: Screen Enable (0 turns screen off, 1 turns screen on)
VB: VBlank (0 not in vblank, 1 in vblank)
DISPWIDTH: width of screen in pixels / 256
DISPHEIGHT: height of screen in pixels / 256

Bitwise Operations

```
void turnOnDisplay() {
	DISP_REGISTER |= DISPLAY_BIT_SCREEN_ENABLE;
}

void turnOffDisplay() {
	DISP_REGISTER &= ~DISPLAY_BIT_SCREEN_ENABLE;
}

bool inVBlank() {
	return (DISP_REGISTER & DISPLAY_BIT_SCREEN_VBLANK) != 0;
}
```

Bitwise Operations

- Remember those 4-bit sections for screen dimensions?

```
int getDisplayWidth() {
    return (DISP_REGISTER & 0x000F) * 256; // or << 7
}
int getDisplayHeight() {
    return ((DISP_REGISTER & 0x00F0) >> 4) * 256;
}
void setDisplayResolution(int width, int height) {
    int w = width / 256;
    int h = height / 256;
    unsigned short res = w | (h << 4);
    DISP_REGISTER = (DISP_REGISTER & 0xFF00) | res;
}
```

Bitwise Operations

- There are other ways (better ways!) to do this example, but this is often seen in codebases:

```
#define FLAG_VISIBLE (1 << 0)
#define FLAG_INVINCIBLE (1 << 1)
#define FLAG_LOCKED (1 << 2)
#define TEST_FLAG(bitmap, flag) (((bitmap) & (flag)) == (flag))
#define SET_FLAG(bitmap, flag) (bitmap) |= (flag)
#define CLEAR_FLAG(bitmap, flag) (bitmap) &= ~(flag)

unsigned int playerFlags = 0;

if (TEST_FLAG(playerFlags, FLAG_VISIBLE))
        renderPlayer();
if (eatenPowerup())
        SET_FLAG(playerFlags, FLAG_INVINCIBLE);
```

Bitwise Operations

- There are other bitwise operations too, and much more we can do with them
- We haven’t mentioned where bitwise operations fit in precedence rules either
- We won’t go over these, but they can be useful to know

- ^ (XOR, exclusive or)
- >>= and <<= (bitshift assignment)

- and all sorts of fun things you can do with them:
- https://graphics.stanford.edu/~seander/bithacks.html

Exercise

Exercise

- Weekly assignment worth 15% of your grade
- You will work from the provided project to demonstrate an understanding of structs and memory management
- You are required to:

Exercise

- Create a new file, MyTypes.h, which has definitions for the following types:
- **Vec2D**, containing x and y coordinates (floats)
- **Vec3D**, containing x, y, and z coordinates (floats)
- **EntityType**, an enum which can be one of [Player, NPC]
- **Entity**, containing: position (2D vector), health, type (EntityType)
- **Player**, containing: score, name, xp, entity to represent player (Entity)

Exercise

- You will also need to write some functions
- Create a “Player” module (.cpp and .h) containing the following, using the types we created:

- initPlayer(player-pointer) – gives the player struct default values for its members

- getPlayerPosition(player-pointer)
- getPlayerHealth(player-pointer)
- getPlayerScore(player-pointer)
- getPlayerName(player-pointer)

- setPlayerName(player-pointer, name)
- setPlayerPosition(player-pointer, x, y)
- setPlayerHealth(player-pointer, health)
- setPlayerScore(player-pointer, score)

Exercise

- in main.cpp look for the function “memoryExercise()”
- in this function, using the C++ style of memory allocation (new/delete), you should:

- allocate an array of 1000 Vec3Ds on the heap, then initialize them all to 0,0,0
- Allocate a new player object on the heap, then set its xp to 1000
- Delete both allocations before the function returns

Exercise

- Look in debugMe.cpp.
- This code has a few (deliberate) problems.
- Identify all the problems and explain the issues in the code as comments, at the location of the problem
- You don't need to fix the problems, just identify them and explain what the issue is

- Tag your comments with "PROB:", for example:

```
void foo() {
	int i = 10 / 0; // PROB: You can't divide by 0. You exploded the universe.
}
```

Exercise

- The repo for this is https://gitlab.com/techlab2050/week3