ECE Summer Workshop Software

APS105 Review & ECE244 Preparation

Motivation

- To provide summer activities for ECE students
- To review materials in APS105
- To prepare for ECE244 & ECE297
- To have fun!

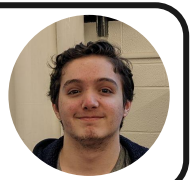
Team

Iris Wang

irisjiayi.wang@mail.utoronto.ca

Nicholas Mutlak

n.mutlak@mail.utoronto.ca



Ryan Seto

ryan.seto@mail.utoronto.ca

Simon Xu

xiaofengsimon.xu@mail.utoronto.ca

Shuntaro Wakamatsu

shuntaro.wakamatsu@mail.utoronto.ca

ECE Summer 2023 Workshop Teaching Team

Iris Wang

irisjiayi.wang@mail.utoronto.ca

Nicholas Mutlak

n.mutlak@mail.utoronto.ca

Ryan Seto

ryan.seto@mail.utoronto.ca

Simon Xu

xiaofengsimon.xu@mail.utoronto.ca

Shuntaro Wakamatsu

shuntaro.wakamatsu@mail.utoronto.ca

Overall Content

Day 1

APS105 Review

- Loops
- Arrays
- Pointers
- Functions

Activities

- Sudoku

Day 2

APS105 Review

- String I/O
- Dynamic Malloc
- Structs
- Linked Lists

Activities

- Dynamic 2D Array
- Pokemon

Day 3

APS105 Review

- Bubble
- Insertion
- Time C.
- Merge
- Quick

Activities

- Sorting

Day 4

APS105 Review

- Binary Search Trees
- Hash Tables
- Headers

Activities

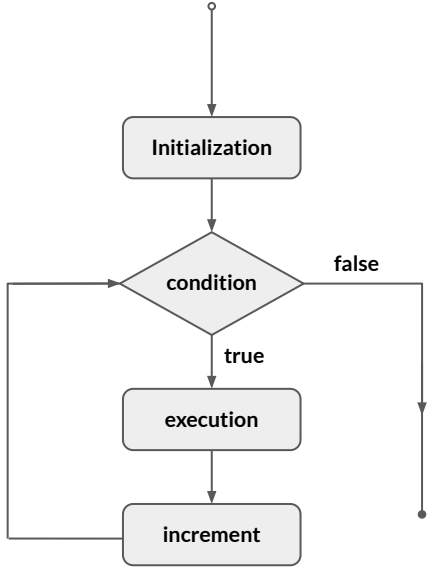
- LeetCode

Day 1 Content

- Loops
- Array
- Pointers
- Functions
- Sudoku Project

For Loops

```c
for(int i = 0; i < 10; i++){
    printf("%d, ", i);
}

//Nested for loops
for(int i = 0; i < 10; i++){
    for(int j = 0; j < 10; j++){
        printf("%d, ", i * 10 + j);
    }
    printf("\n");
}
```



- Basic syntax of **for loops** includes **Initialization, Condition, and Increment.**
- A for loop initializer is scoped to **the loop body**
- In this example, variable “i” is gone after the loop

For Loops (break, continue)

In C, we can also use “break” and “continue” to specify the condition for exiting the loop

```c
int i = 0;
for( ; ; ){     //Infinite loop (no condition for stopping)
    printf("%d, ", i);
    i++;        //Incrementing
    if(!(i < 10)){
        break;  //Exiting the loop
    }
}

for(int i = 0; i < 10; i++){
    if( (i < 3) || (7 < i) ){
        continue;   //Skips everything after this line
                    // and goes back to the beginning of the loop
    }
    printf("%d, ", i);
}
```

break: immediately exits the loop

continue: Ignores the rest of the lines and goes back to the top of the loop

While Loops

```c
#include <stdio.h>

int main(void) {
  //COUNTDOWN
  int countdown = 6;
  while (countdown > 0){
    countdown--;
    printf("%d\n",countdown);
  }
  printf("Blast Off!\n");
```

```
> ./main
5
4
3
2
1
0
Blast Off!
>
```

- Basic syntax of **while loops** still includes **Initialization, Condition, and Increment.**
- **Initialization** is done prior to the loop, **Condition** is checked as the loop begins, and **Increment** is done inside the loop.
- The variable “countdown” still exists after the while loop finishes

Do While Loops

```c
#include <stdio.h>

int main(void) {
    int input = '1';

    do{
        printf("Do you want to stop? (1/0)\n");
        scanf("%d", &input);

    } while(input != 1);
```

```
> ./math
Do you want to stop? (1/0)
0
Do you want to stop? (1/0)
0
Do you want to stop? (1/0)
0
Do you want to stop? (1/0)
0
Do you want to stop? (1/0)
1
>
```

- Same setup as while loops with one exception, The **Condition** is checked at the end of the loop, after an **Increment** occurs.
- The variable “input” also still exists after the do while loop finishes

1D Arrays

```
3  //Declaration
4      int array1[5];
5  //Initialization
6      int array2[5] = {0, 1, 2, 3, 4};
7      //Initialize each element individually
8      int array3[5] = {};
9      //Initialize all elements with 0
10     int array4[5] = {0};
11     //Initialize all elements with 0 (Only works with 0)
12     int array5[5] = {[2] = 7};
13     //Initialize all elements
14     char array6[] = {'H', 'e', 'l', 'l', 'o'};
15     //Initialize w/o specifying the # of elements
16     char array7[10] = "Hello";
17     //Initialize with a string
18 //Referencing & Assigning
19     int foo = array1[2];
20     //Referencing
21     array1[3] = 7;
22     //Assigning
```

The elements that you did not specify what to be initialized with will be initialized with 0

You can initialize an array without specifying the number of elements

2D Array

```cpp
//2D Array

//Declaration & Initialization
    int 2Darray1[3][5] = {
        {0, 1, 2, 3, 4},
        {5, 6, 7, 8, 9},
        {10, 11, 12, 13, 14}
    };
    //Initializing using a list

    char 2Darray2[][20] = {"Tokyo", "Kyoto", "Osaka"};
    //Initialize w/o specifying the # of rows
    //You must specify the # of columns !

//Referencing & Assigning
    int foo = 2Darray1[2][2];
    //Referencing

    2Darray1[2][2] = 7;
    //Assigning
```

```cpp
/*
        col1    col2    col3    ...
row1     0       1       2      ...

row2     5       6       7      ...

row3     10      11      12     ...

  .       .       .       .
  .       .       .       .
  .       .       .       .

*/
```

2D Array is a matrix-like data structure that organizes its elements in rows and columns

What a 2D Array looks like

```c
#include <stdio.h>

int main(){
    int TwoD[3][5] = {
        {3, 1, 4, 1, 5},
        {9, 2, 6, 5, 3},
        {5, 8, 9, 7, 9}
    };
    printf("%p", TwoD);
}
```

3fffff80	0000b2e8	0000805c	0000b6a4	00000000
3fffff90	0000b2e8	00006a34	0000b6a4	00008160
3fffffa0	0000b6a4	00000003	00000001	00000004
3fffffb0	00000001	00000005	00000009	00000002
3fffffc0	00000006	00000005	00000003	00000005
3fffffd0	00000008	00000009	00000007	00000009
3fffffe0	0000bc58	0000a748	00000000	000000d0
3ffffff0	0000b2d4	000002d4	aaaaaaaa	aaaaaaaa

```
0x3fffffa4
```

In the memory, a 2D array is, in fact, just a 1D array
We are just treating a 1D array as a 2D array
n-D arrays are the same

Strings

- Character array terminated by a NULL character
- Initializing a string in C language:
  - char greeting[6] = {'H', 'e', 'l', 'l', 'o', '\0'};
  - char greeting[] = "Hello";
-

Index	0	1	2	3	4	5
Variable	H	e	l	l	o	\0
Address	0x23451	0x23452	0x23453	0x23454	0x23455	0x23456

String Functions - 1

- strcpy(s1, s2);

Copies string s2 into string s1.

- strcat(s1, s2);

Concatenates string s2 onto the end of string s1.

- strlen(s1);

Returns the length of string s1.

- strcmp(s1, s2);

Returns 0 if s1 and s2 are the same; less than 0 if s1<s2; greater than 0 if s1>s2.

Case sensitive, meaning ‘A’ and ‘a’ are treated as different letters.

String Functions - 2

- strstr(const char *s1, const char *s2);

Returns a pointer to the first occurrence of string s2 in string s1.

- strchr(const char *s1, const char c);

Returns a pointer to the first occurrence of the character c in the string str

Or NULL if the string or character is not found.

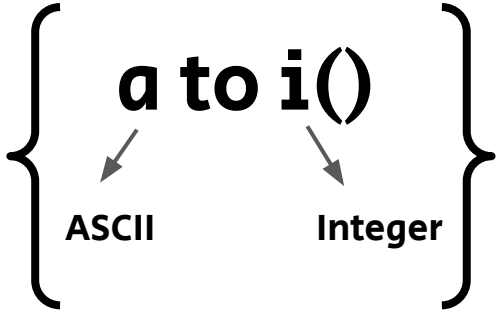
String Functions - 3

- int atoi(s1);

Converting a string to integer

- float atof(s1);

Converting a string to float




```c
#include<stdio.h>
#include<stdlib.h> // Include the header file for atoi()

int main() {
    char temp[] = "25";
    int i = atoi(temp);
    printf("%d\n", i); // Format specifier %d to print an integer

    char temp2[] = "send 25 cookies";
    int j = atoi(temp2);
    printf("%d\n", j);
    return 0;
}
```

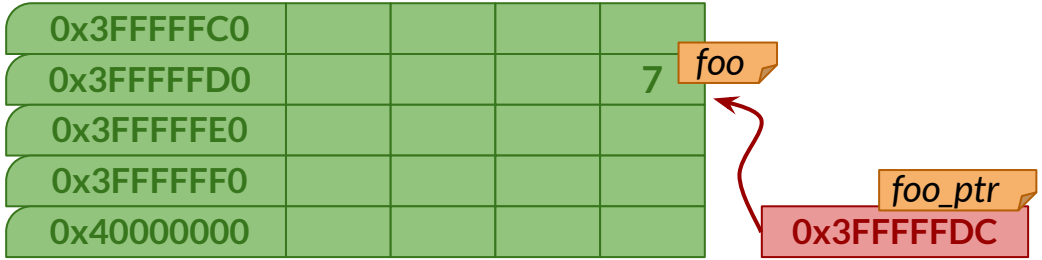
```
25
0
```

Pointer

```c
//Pointers
int foo = 7;          //(int) type variable
int *foo_ptr;         //(int *) type variable (= a pointer to an int)
foo_ptr = &foo;       //storing the address of foo into foo_ptr

printf( "Value: %d\n", *foo_ptr );
printf( "Address of \"foo\": %p\n", &foo );
printf( "Value inside \"foo_ptr\": %p\n", foo_ptr );
```

Output

```
Value: 7
Address of "foo": 0x3fffffdc
Value inside "foo_ptr": 0x3fffffdc
```



Pointer is a variable that holds **the memory address** of another variable.

You can navigate through the memory using **Pointer Arithmetic operations**

Referencing (*) operator gets the actual content at an address

Address-of (&) operator gets the address of a variable

Pointer & Array

```c
//Integer Array
int array[10] = {0, 2, 4, 6, 8, 10, 12, 14, 16, 18}; //integer array
int *a_ptr = array;          //(int *) type variable
                             //store the address of the HEAD of "array"
int bar = array[6];          //Accessing the 7th element using the Subscript operator []
int baz = *(array + 6);      //Accessing the 7th element using pointer arithmetic
int qux = *(a_ptr + 6);      //Proof that "array" is actually a (int *) variable
                             //that points to the head of the sequence of integers
int quux = *(&(array[6]));   //Referencing (*) the address (&) of the 7th element ([6])
                             // of "array"
for(int i = 0; i < 6; i++){
    a_ptr++;                 //Increment the pointer 6 times
                             //Now, it is pointing to the 7th
}
int corge = *a_ptr;

printf("bar: %d, baz: %d, qux: %d, quux:%d, corge:%d\n", bar,
printf("Address of \"array\": %p", array);
```

The name of an array is actually **a pointer variable** that points to **the head** of the array

You can use pointer arithmetic to iterate through the elements

Output

```
bar: 12, baz: 12, qux: 12, quux:12, corge:12
Address of "array": 0x3fffffb8
```

3fffff50	48460	45880	48680	32680
3fffff60	46732	45880	48480	2
3fffff70	45880	42968	48460	45880
3fffff80	45880	32892	46836	0
3fffff90	45880	27220	46836	33152
3fffffa0	46836	4294967295	48640	0
3fffffb0	48632	44708	0	2
3fffffc0	4	6	8	10
3fffffd0	12	14	16	18
3fffffe0	48296	42856	0	208
3ffffff0	45860	752	2863311530	2863311530
40000000	4294967295	4294967295	4294967295	4294967295
40000010	4294967295	4294967295	4294967295	4294967295

2D Array using Pointers

```c
//2D Array using Pointers
    //Useful when you don't know the size of the array
    int row, col;
    row = 3;
    col = 5;

    int **ptr1;
    //A double pointer
    ptr1 = malloc(sizeof(int *) * row);
    //Dynamically allocate a space for 3 (int *) type variables

    for(int i = 0; i < row; ++i){
        ptr1[i] = malloc(sizeof(int) * col);
    }
    //Dynamically allocate a space for 5 (int) type variables for each row


    //Freeing the memory
    for(int i = 0; i < row; ++i){
        free(ptr1[i]);
        //Freeing the columns. This must come first!
    }
    free(ptr1);
    //Freeing the rows.
```

When you don’t know the size of the array, you can dynamically allocate memory location using pointers.

If you dynamically allocate memory, you always need to free them before the program ends

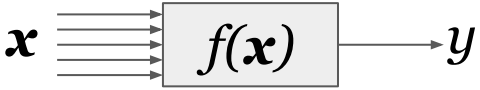
2D Array as a 1D Array

```c
//2D Array as a 1D Array (My favourite method)
    int row, col;
    row = 3;
    col = 5;

    int *ptr1;
    ptr1 = malloc(sizeof(int) * col * row);
    //Dynamically allocate a space for 3 x 5 integers

    for(int i = 0; i < row; ++i){
        for(int j = 0; j < col; ++j){
            ptr1[i * col + j] = 0;
            //Initialize all elements with 0

            printf("%d, ", ptr1[i * col + j]);
            //Referencing
        }
    }

    free(ptr1);
```

You can also use a 2D array using only one pointer.

This reflects the actual memory usage more accurately (I think)

Functions



A function performs a specific task/computation to **multiple inputs (arguments)** and returns **a single output (return value)**

You can also use a function prototype to declare the **name, return type, and argument types** to organize your code

You can use **void** for the inputs and the output, if you do not need to pass or return anything

```
//Functions
    /*
    return_type functionName( arg1_type arg1_name, arg2_type arg2_name, ...){
        return something;
    }
    */
    int sum(int foo, int bar, int baz){
        return sum + bar + baz;
    }

    //A function can return ONLY ONE value
    //Return type can be void, if you don't need to return anything
    //Arguments can be void, if you don't need to pass anything

    //Calling and Obtaining the returned value
        sum(1, 2, 3); //Just Calling
        int qux = sum(4, 5, 6) //Calling and Storing the returned value (qux = 15)

    //Prototype
        int sum(int foo, int bar, int baz); // <-- Prototype

        int main(){
            sum(1, 2, 3);
            return 0;
        }

        int sum(int foo, int bar, int baz){
            return sum + bar + baz;
        }

        //Allows you to write the function definition after the line where that
        //function is called. You don't need to worry about the order you define
        // the functions
```

Passing Arguments (Pass by Value)

```
160        //Pass by Value
161            int a = 1;
162            mul(a);
163            int mul(int foo){ return foo * 2; } // a is still 1
164
165            //This calls a constructor of (int) type, and a new object will be generated
166            // in the stack memory (In short, a copy of the argument will be generated)
167            //Advantages:   Any change made to the arguments inside the function do not
168            //              affect the original variables
169            //Disadvantages:It slows down the program when you pass a huge size variable
170            //              because a new copy has to be made every time that function
171            //              is called
```

3fffffa0	0	856	0	1073741756	
3fffffb0	0	592	44928	1073741772	
3fffffc0	8256	4294967295	44928	2	foo (int)
3fffffd0	0	668	0	1	a
3fffffe0	48200	42864	0	208	
3ffffff0	45764	760	2863311530	2863311530	
40000000	4294967295	4294967295	4294967295	4294967295	

Passing Arguments (Pass by Pointer)

```
173        //Pass by Pointer
174            int a = 1;
175            mul(&a);
176            bool mul(int *foo){
177                if(foo == NULL){ return false; }
178                *foo = (*foo) * 2;
179                return true;
180            } // a is now 2
181
182            //This passes an address of the (int) type variable, a, and in the mul
183            // function, the value of a will be updated by referening the pointer.
184            //Advantages:   You can change the original variables
185            //              Does not take much time regardless of the arguments' size
186            //Disadvantages:Hard to read.
187            //              Need for a NULL check. If a NULL pointer is passed,
188            //              referening that pointer results in undefined behaviour
189            //              , and the program might crash.
```

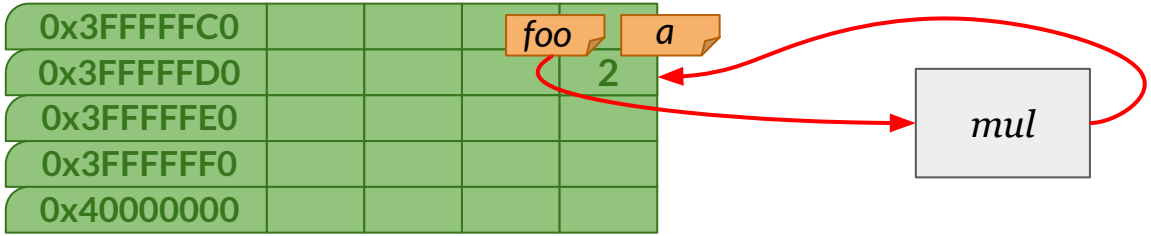
3fffffa0	00000000	00000360	00000000	3fffffbc
3fffffb0	00000000	00000260	0000af88	3fffffdc
3fffffc0	00002040	ffffffff	0000af88	3fffffdc
3fffffd0	3fffffdc	000002a4	00000000	00000002
3fffffe0	00000000	0000a778	00000000	000000d0
3ffffff0	0000b2cc	00000300	aaaaaaaa	aaaaaaaa
40000000	ffffffff	ffffffff	ffffffff	ffffffff

*foo (int *)*

a

Passing Arguments (Pass by Reference)

```
191        //Pass by Reference (ONLY in C++)
192            int a = 1;
193            mul(a);
194            void mul(int &foo){
195                foo = foo * 2;
196            } // a is 2!
197
198            //This passes an "alias" of the (int) type variable, a.
199            //Advantages:   You can chage the original variable
200            //              Easier to read. You can write the program as if you were
201            //              passing by value.
202            //Disadvantages:(reference) type does not have the cooncept of "NULL"
203            //              Therefore, the callee assumes that the passed reference is
204            //              valid, and the caller must gurantee that it never passes an
205            //              invalid value.
206            //              You cannot tell whether the passed arguments will be modified or
207            //              not in the function, without looking at the prototype.
208            //              (Cannot tell if it's a pass by value or a pass by reference)
```



Passing Arguments (Summary)

	Pros	Cons	Code

Pass by Value	Does not affect the original var	Takes time	`int a = 1;`
			`mul(a);`
			`int mul(int foo){`
			`  foo = foo* 2;`
			`  return foo;`
			`}//a is still 1`

Pass by Pointer	Can affect the original var
	Does not take time
		Hard to read
		Need to NULL check
			`int a = 1;`
			`mul(&a);`
			`bool mul(int *foo){`
			`  if(foo == NULL){ return false; }`
			`  *foo = (*foo) * 2;`
			`  return true;`
			`} // a is now 2`

Pass by Reference	Can affect the original var
	No need for a NULL check
	Easier to read
		Not allowed to pass an invalid value
		Cannot tell if the passed value will be changed
			`int a = 1;`
			`mul(a);`
			`void mul(int &foo){`
			`  foo = foo * 2;`
			`} // a is 2!`

Function-like Macros

```c
//Function-like Macros
    #define INT_MAX(foo, bar) ((foo > bar) ? (foo) : (bar))

    int main(){
        printf("%d" , INT_MAX(100, 200)); //Prints 200
        return 0;
    }
    //Acts like a function, but what it's doing is just replacing INT_MAX(100, 200)
    //  with ((foo > bar) ? (foo) : (bar)) at compile time.
```

- You can define a macro that resembles a function call
- At compile time, the macro used in the code are replaced by the defined code (In this example, comparing “foo” and ”bar” and returning the bigger one)

Passing Arrays to Functions (1D)

```c
#include <stdio.h>

double avg(int *array, int size){
    //Need to pass the address of the head of the array
    //as well as the size of the array
    int sum = 0;
    for(int i = 0; i < size; ++i){
        sum += array[i];        //We can use the subscript operator
    }
    return (double)sum / size;
}

int main()
{
    int array1[] = {3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5, 8, 9, 7, 9};
    double average = avg(array1, sizeof(array1) / sizeof(int) );
    //The size of the array is
    //      Total size (in bytes) / one integer size (in bytes)

    printf("%lf", average);
    return 0;
}
```

To navigate through all the elements of the array, you need **the head address** and **the size**

Safe way to obtain the size of an array is to use the **sizeof operator**

Passing Arrays to Functions (char 1D)

```c
#include <stdio.h>

void print(char *str){
    printf("%s\n", str);
}

int main()
{
    char *c_array = "Engineering";
    print(c_array);

    return 0;
}
```

In the case of a char array, we do not need to pass the size, since char array always end with '\0'

Char is **1 byte**

1 byte is 8 bits, meaning 1 byte can express 2^8 (=256), which is more than enough for ASCII code

```
0000b050  e12fff1e  0000b378  0000b378  0000b394  ••/• x••• x••• ••••
0000b060  0000b378  00000000  ffff5d4   00000001  x••• •••• •••• ••••
0000b070  69676e45  7265656e  00676e69  000a7025  Engi neer ing• %p••
0000b080  0000b3d8  0000000a  20202020  20202020  •••• ••••
```

Passing Arrays to Functions (2D) - 1

```c
double avg(int array[][5], int row){
    int sum = 0;
    for(int i = 0; i < row; ++i){
        for(int j = 0; j < 5; ++j){
            sum += array[i][j];
        }
    }
    return (double)(sum) / (5 * row);
}

int main()
{
    int i_array[][5] = {
        {3, 1, 4, 1, 5},
        {9, 2, 6, 5, 3},
        {5, 8, 9, 7, 9}
    };
    double average = avg(i_array, sizeof(i_array) / (sizeof(int) * 5));
    printf("%lf", average);
    return 0;
}
```

In this syntax, the number of column is fixed, so we only need to pass **the head address** and **the number of row**

array is (**int (*)[5]**) type

Passing Arrays to Functions (2D) - 2

```c
double avg(int *array, int row, int col){
    int sum = 0;
    for(int i = 0; i < row; ++i){
        for(int j = 0; j < col; ++j){
            sum += array[i * col + j];
        }
    }
    return (double)(sum) / (5 * row);
}

int main()
{
    int i_array[][5] = {
        {3, 1, 4, 1, 5},
        {9, 2, 6, 5, 3},
        {5, 8, 9, 7, 9}
    };
    int row = sizeof(i_array) / (sizeof(int) * 5);
    double average = avg(i_array, row, 5);
    printf("%lf", average);
    return 0;
}
```

In this syntax, we pass **the head address, the number of columns** and **the number of row**

The head address is a single pointer to (int) type

i_array is (**int (*)[5]**) type and needs to be typecast to (**int ***)

You can choose the number of rows and the number of columns freely

Passing Arrays to Functions (2D) - 3

```c
double avg(int **array, int row, int col){
    int sum = 0;
    for(int i = 0; i < row; ++i){
        for(int j = 0; j < col; ++j){
            sum += array[i][j];
        }
    }
    return (double)(sum) / (5 * row);
}

int main()
{
    int row = 3;
    int col = 5;
    int **i_array = (int **)malloc(row * sizeof(int *));
    for(int i = 0; i < row; ++i){
        i_array[i] = (int *)malloc(col * sizeof(int));
        for(int j = 0; j < col; ++j){
            i_array[i][j] = (row * col) - (i * col + j);
        }
    }
    double average = avg(i_array, row, col);
    printf("%lf", average);
    return 0;
}
```

You can pass a 2D array using a double pointer.

You can choose the number of rows and the number of columns freely

This method allows you to use the 2D array in a natural way inside the callee function

Passing Arrays to Functions (2D) - 4

```c
double avg(int row, int col, int array[row][col]){
    int sum = 0;
    for(int i = 0; i < row; ++i){
        for(int j = 0; j < col; ++j){
            sum += array[i][j];
        }
    }
    return (double)(sum) / (5 * row);
}

int main()
{
    int i_array[][5] = {
        {3, 1, 4, 1, 5},
        {9, 2, 6, 5, 3},
        {5, 8, 9, 7, 9}
    };
    int row = sizeof(i_array) / (sizeof(int) * 5);
    double average = avg(row, 5, i_array);
    printf("%lf", average);
    return 0;
}
```

This method only works for C99

The array name has to be the third argument

Random Numbers

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main()
{
    srand((unsigned int)time(NULL));
    //Changes the seed of rand() function according to the time
    int foo = 10 + rand() % 100;
    //Generates a random 2-digit number (10 - 99)
    double bar = rand() / (double)RAND_MAX;

    return 0;
}
```

Include the library “stdlib.h”

To get random number of a given range use **“+”** and **“%”**

rand() calculates X(n+1) based on X(n) using the LCG equation

Need to **change X0 (seed)** to get different random numbers every time

LCG equation $$X_{n+1} = (aX_n + c) \bmod m$$

Challenge 1: Longest Palindromic String

Write a function that returns the longest palindromic string given a string

Palindromic: a sentence or a word that reads the same forward and backward.
Example:

Was it a cat I saw?

Function Prototype:

```
char * longestPalindrome(char * s);
```

Solution 1 (Simple)

- Check for all the possible combinations
- For a string of **n** letters, the number of iteration is **n(n + 1)/2**
- Example)

  banana → (banana), (banan, **anana**), (bana, anan, nana), (ban, **ana**, **nan**, **ana**), …

checkPalindrome(char *s)

```
38  bool checkPalindrome(char *s){
39      int size = (int)strlen(s);
40      for(int i = 0; i < size / 2; i++){
41          if(s[i] != s[size - 1 - i]){
42              return false;
43          }
44      }
45      return true;
46  }
```

Compare the **nth** element with the **length - nth** element

longestPalindrome(char *s)

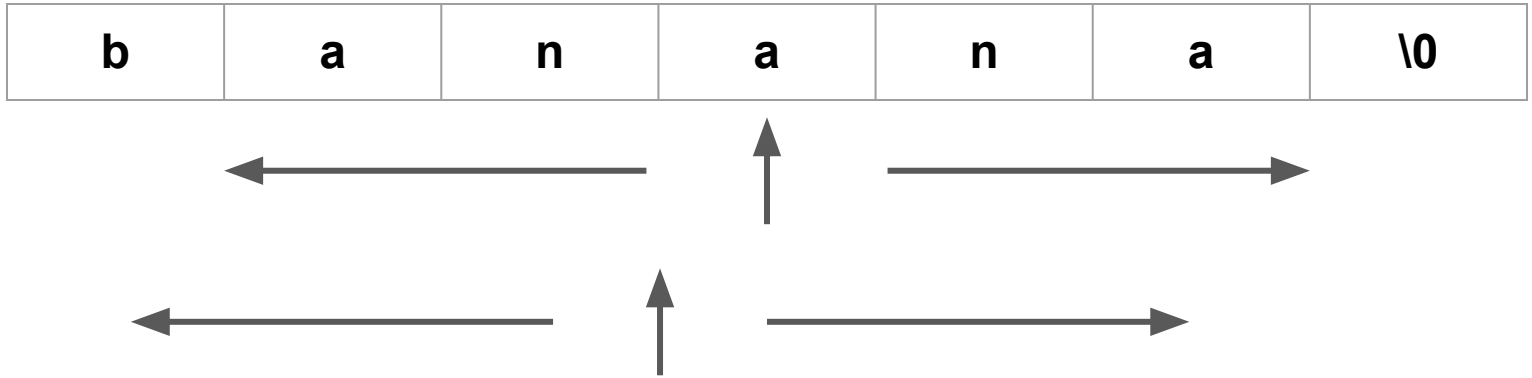
```
54      for(int i = 0; i < size; i++){
55          for(int j = 0; j < size - i; j++){
56              char substring[j + 2];
57              for(int k = 0; k < j + 1; k++){
58                  substring[k] = s[i + k];
59              }
60              substring[j + 1] = '\0';
61              if(checkPalindrome(substring)){
62                  cur_length = j + 1;
63                  if(cur_length > cur_longest){
64                      cur_longest = cur_length;
65                      start_idx = i;
66                  }
67              }
68          }
69      }
```

Making a copy substring

Update only if the string is a palindrome and longer

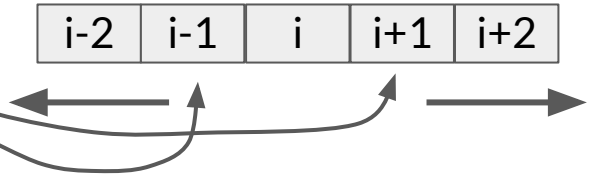
Solution 2 (Faster)

- Starting from a point, you expand outwards
- For a string of **n** letters, the worst case iteration is **n^2 / 4 + n(n-2)/4**, the best case iteration is **0**



longestPalindrome (starting from a char)

```
42      for(int i = 0; i < size; i++){
43          int right_idx = i + 1;
44          int left_idx = i - 1;
45          cur_length = 1;
46          while(0 <= left_idx &&
47                right_idx <= (size - 1) &&
48                s[left_idx] == s[right_idx]){
49              cur_length += 2;
50              right_idx++;
51              left_idx--;
52          }
53          if(cur_length > cur_longest){
54              cur_longest = cur_length;
55              start_idx = i - (cur_length - 1) / 2;
56          }
57      }
```



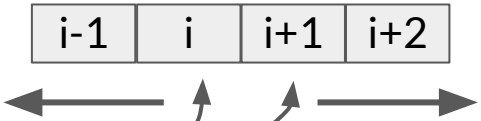
If not out of bounds
& left end = right end

Expand as much as you can

If the
expanded
str is longer,
update

longestPalindrome (starting between chars)

```
59      for(int i = 0; i < size - 1; i++){
60          int right_idx = i + 1;
61          int left_idx = i;
62          cur_length = 0;
63          while(0 <= left_idx &&
64                right_idx <= (size - 1) &&
65                s[left_idx] == s[right_idx]){
66              cur_length += 2;
67              right_idx++;
68              left_idx--;
69          }
70          if(cur_length > cur_longest){
71              cur_longest = cur_length;
72              start_idx = i - (cur_length - 2) / 2;
73          }
74      }
```



If not out of bounds
& left end = right end

Expand as much as you can

If the
expanded
str is longer,
update

Recursion

```
void recurse()
{
    ... .. ...
    recurse();
    ... .. ...
}

int main()
{
    ... .. ...
    recurse();
    ... .. ...
}
```

recursive call

The idea of approaching recursion to code in C

- Considering the edge case(s)
- Perform one step of the action
- Assuming the result to the rest of the action is given by a magic person
- Combine everything together

Recursion Example: finding factorial

The idea of approaching recursion to code in C

- Considering the edge case(s)
- Perform one step of the action
- Assuming the result to the rest of the action is given by a magic person
- Combine everything together

```c
#include <stdio.h>

int fact (int);
int main()
{
    int n,f;
    printf("Enter the number whose factorial you want to calculate?");
    scanf("%d",&n);
    f = fact(n);
    printf("factorial = %d",f);
}

int fact(int n)
{
    ...
}
```

Your code goes here

1. Considering the edge case(s)

- If the user inputs 0?

We will simply get 1 for factorial of 0!

```
int fact(int n)
{
    if(n == 0)
    {
        return 1;
    }

}
```

2. Perform one step of the action

As we know:

factorial of n

= n * (n-1) * (n-2) * … * 2 * 1

- We are only coding one step of the entire multiplication

```
int fact(int n)
{
    if(n == 0)
    {
        return 1;
    }
    return n * ...
}
```

Assuming the result to the rest of the action is given by a magic person

As we know:

factorial of n

= n * (n-1) * (n-2) * … * 2 * 1

- The rest of the multiplication is the factorial n-1
- We assume that a magic person kindly provide us the answer

```
int fact(int n)
{
    if(n == 0)
    {
        return 1;
    }
   return n * fact(n - 1);
}
```

Recursion Example: finding factorial

The idea of approaching recursion to code in C

- Considering the edge case(s)
- Preform one step of the action
- Assuming the result to the rest of the action is given by a magic person
- Combine everything together

```c
#include <stdio.h>

int fact (int);
int main()
{
    int n,f;
    printf("Enter the number whose factorial you want to calculate?");
    scanf("%d",&n);
    f = fact(n);
    printf("factorial = %d",f);
int fact(int n)
{
    if(n == 0)
    {
        return 1;
    }
   return n * fact(n - 1);
}
    else if ( n == 1)
    {
```

Understanding how recursion functions

```
return 5 * factorial(4) = 120
    return 4 * factorial(3) = 24
        return 3 * factorial(2) = 6
            return 2 * factorial(1) = 2
```

```
int fact(int n)
{
    if(n == 0)
    {
        return 1;
    }
   return n * fact(n - 1);
}
```

Challenge 2: Permutation

- Permutation is a number made by arranging numbers so that each number only appears once.
- Example) permutations of {1, 2, 3}

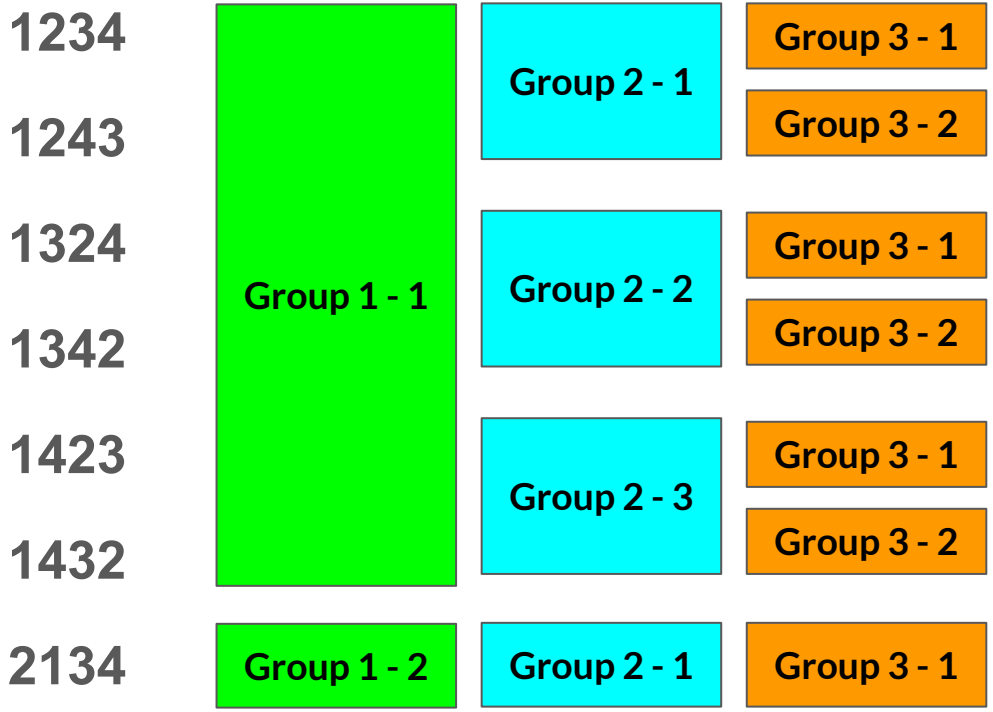
(123), (132), (213), (231), (312), (321)

Find the kth permutation when listing all the n! permutations in ascending order.

Function Prototype:

```
char * getPermutation(int n, int k)
```

Solution



5th permutation?

5 / 3! = **0** ... 5 = Group 1-1
5 / 2! = **2** ... 1 = Group 2-3
1 / 1! = **1** ... **0** = Group 3-1

1. First unused from [**1**, 2, 3, 4]
2. Third unused from [2, 3, **4**]
3. First unused from [**2**, 3]
4. The last one [**4**]
5. The 5th permutation is:

1423

Example: $n = 5$, $k = 89$ ($1 \leqq k \leqq 5!$)

Step 1: What is the most significant digit? In which group does it belong to?

Question 1: How many different permutations are there

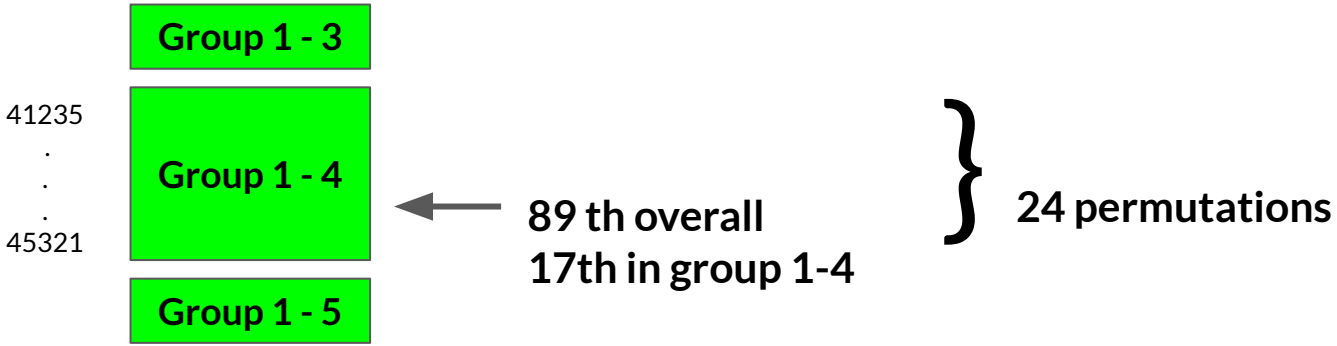
Question 2: How many groups are there when you classify all the permutations based on their most significant digit? How many permutations in each group?

Question 3: In which group does 89th permutation belong to

Example: n = 5, k = 89 ($1 \leqq k \leqq 5!$)

Question 4: What is the position of 89th permutation in the group 4?

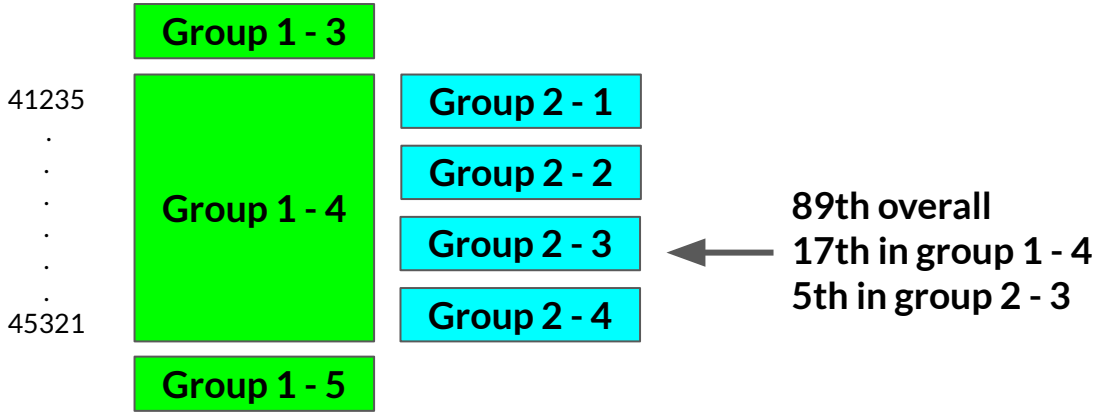
The 89th permutation belongs to the group 4, and is the 17th permutation in the group 4



Example: n = 5, k = 89 ($1 \leqq k \leqq 5!$)

Likewise,

- in group 1-4, there are **4** subgroups each containing **3! (= 6)** permutations
- 17 / 6 = 2 (89th permutation belongs to the **group 2-3**)
- 17 % 6 = 5 (**5th** permutation in the group 2-3)



Example: n = 5, k = 89 ($1 \leqq k \leqq 5!$)

Likewise,

- in group 2-3, there are **3** subgroups each containing **2! (= 2)** permutations
- 5 / 2 = 2 (89th permutation belongs to the **group 3-3**)
- 5 % 2 = 1 (**1st** permutation in the group 3-3)

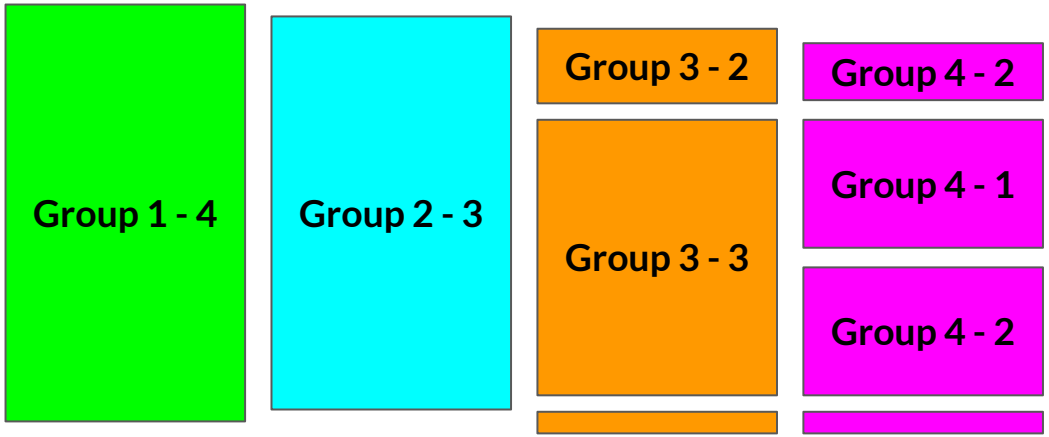
Example: n = 5, k = 89 ($1 \leqq k \leqq 5!$)

Likewise,

- in group 3-3, there are **2** subgroups each containing **1! (= 1)** permutations
- 1 / 1 = 1 (89th permutation belongs to the **group 4-1** (**remainder = 0**))
- 1 % 1 = 0 (**1st and only** permutation in the group 4-1)

43512

43521



89th overall
17th in group 1 - 4
5th in group 2 - 3
1st in group 3 - 3
1st in group 4 - 1

Example: n = 5, k = 89 ($1 \leqq k \leqq 5!$)

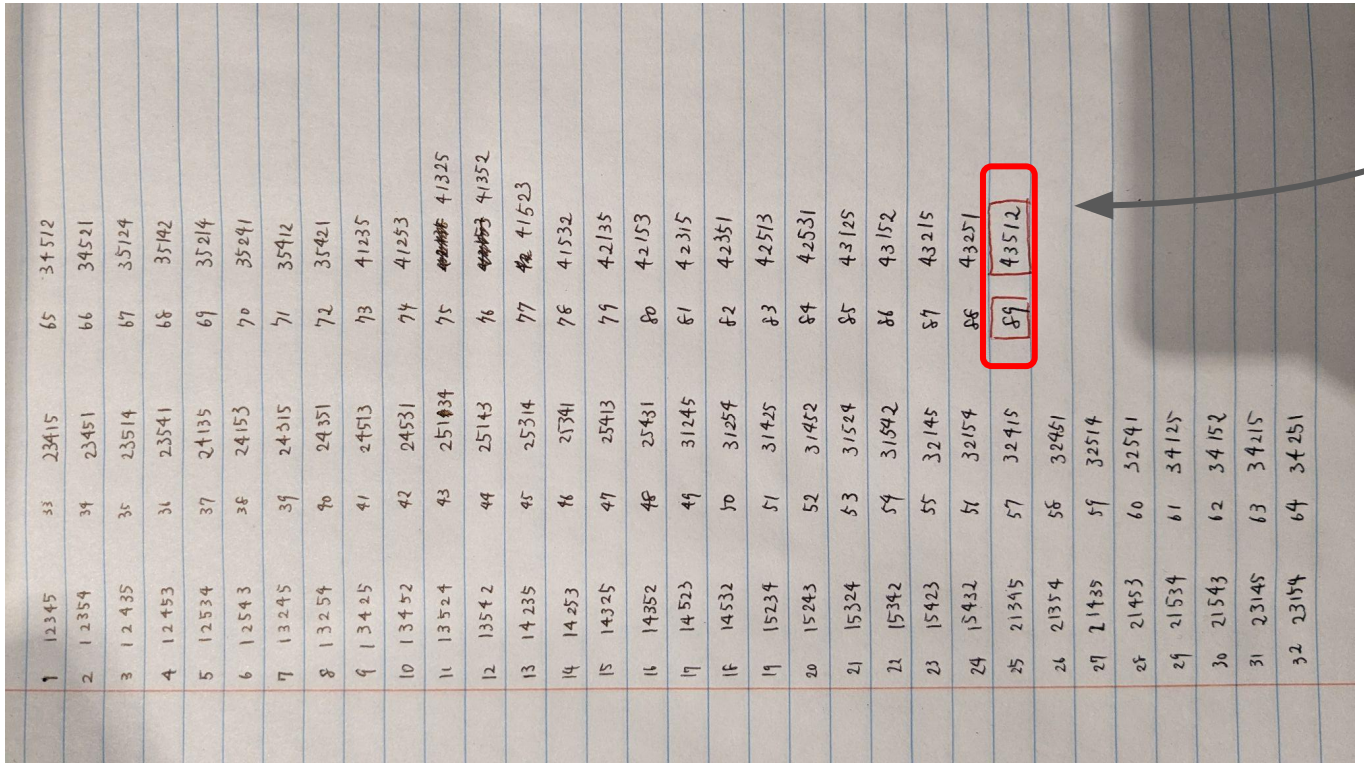
89th permutation belongs to

- Group 1 - **4**
- Group 2 - **3**
- Group 3 - **3**
- Group 4 - **1**

0	**2**	**0**	**0**	**0**

43512

Still skeptical?



int * recursion(int n, int k)

```
23      if(n == 1){ //Edge case
24          int *base_str = (int *)malloc(sizeof(int) * 1);
25          base_str[0] = 0 + k;
26          return base_str;
27      }
```

Always 0 (Why ?)

n: the number of “numbers” left unselected
k: position starting from the smallest number in the current group

```
31      int fac = fact(n - 1);
32      rank = k % fac;
33      group = k / fac;
34      group = (rank == 0) ? group : group + 1;
35      rank = (rank == 0) ? fac : rank;
```

When the permutation you are looking for is the last member of a group

rank: position in the next smaller group
group: the current group

int * recursion(int n, int k)

```
36      int *str = recursion(n - 1, rank);
37      int *new_str = (int *)malloc(sizeof(int) * n);
38      for(int i = 0; i < n - 1; i++){
39          new_str[i] = str[i];
40      }
41      free(str);
42      new_str[n - 1] = 0 + group;
43
44      return new_str;
```

recursion function

- Calls itself after selecting one number (**group**) and updating the position in the next group (**rank**)
- Stores the returned int array in a pointer **str**
- Copies the returned int array in a **bigger** int array **new_str**
- Adds one value (**current group**) at the end of **new_str**

char * getPermutation(int n, int k)

```c
int *orders = recursion(n, k);
char *permutation = (char *)malloc(sizeof(char) * (n + 1));
```

Calling the recursion function

```c
for(int i = 0; i < n; i++){
    int counter = orders[n - (i + 1)];
    int idx = 0;
    while(counter > 0){
        if(numbers[idx] != 0){
            counter--;
        }
        idx++;
    }
    idx--;
    permutation[i] = '0' + numbers[idx];
    numbers[idx] = 0;
}
```

orders tells you in which group the **n**th **least** significant digit belongs to

Increment idx until you reach the counter-th unused number

Store the corresponding number

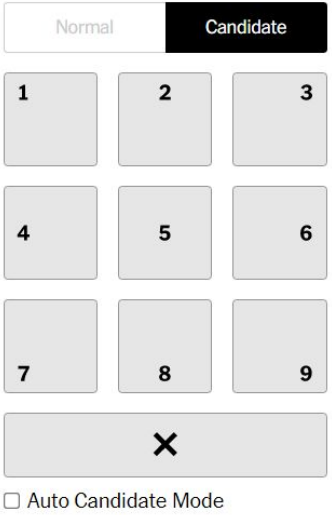
Mark as used

numbers is an int array that contains all the number from 1 to n (size n)

Scan for Sudoku Skeleton

Sudoku

1		6		4		9		
2				8			5	
5	8		7	2	1	6	3	
	3	1		5	8			6
6	5		4					8
			1	3		2		5
		7	5		3		4	
4	1	8	9			5	6	
					4	7	2	

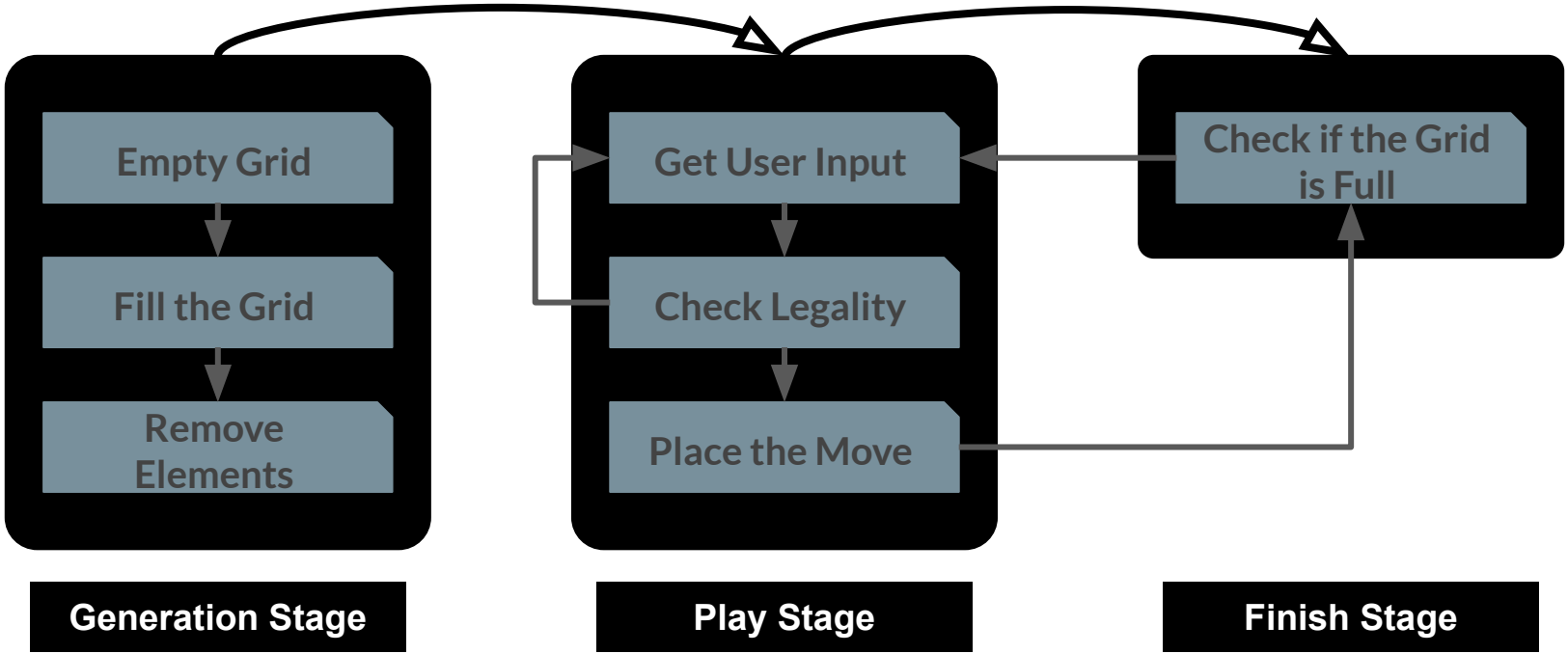


Each **row**, **column**, and **3x3 subgrids** must contain all the numbers from 1 to 9 without any repetition

Based on the numbers that are already there, you fill in the whole grid

New York Times

Flow


Empty Grid
Fill the Grid
Remove Elements
Get User Input
Check Legality
Place the Move
Check if the Grid is Full
Generation Stage
Play Stage
Finish Stage

Header and Implementation

```
void function(int a, int b);
```

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

```
#ifndef Header_h
#define Header_h

//Function Prototypes

#endif // Header_h
```

Header (.h)

Impleme ntation (.c)

Header files usually use this format (↑) to prevent multiple inclusion

#ifndef means "**Define only if it has not defined yet**"

Hides implementation but still shows the functionality

Can be included in multiple files

int generatedGrid[9][9];

main
.c

	0	1	2	3	4	5	6	7	8
0									
1									
2									
3									
4									
5									
6									
7									
8									

- A 2D array
- First, we will fill this 2D array with random numbers following the sudoku rules (→ **linearFill**)
- Then, we remove some of the numbers to create a puzzle (→ **removeKDigits**)
- We will print this 2D array on the console (→ **printSudokuGrid**)
- Then, we get the user input, check if the user input is valid, and update the 2D array (→ **playing**)

void linearFill(int grid[9][9])

sudoku Generator .c

```
> make -s
> ./main

3 |1 |7 |  9 |2 |4 |  5 |8 |6 |
4 |5 |9 |  7 |6 |8 |  1 |2 |3 |
6 |8 |2 |  5 |3 |1 |  7 |9 |4 |

5 |9 |6 |  2 |1 |3 |  8 |4 |7 |
7 |3 |1 |  8 |4 |9 |  6 |5 |2 |
8 |2 |4 |  6 |5 |7 |  3 |1 |9 |

2 |7 |3 |  1 |9 |5 |  4 |6 |8 |
1 |6 |8 |  4 |7 |2 |  9 |3 |5 |
9 |4 |5 |  3 |8 |6 |  2 |7 |1 |
Select a digit from 1-9:
```

- We will use a 2D array **int availableNum[81][9]** to keep track of which number works and which number does not
- Each row represents a cell in the grid
- Starting from **top left,** for each cell, we select a random number and see if it is valid to place that number in that cell
- If it is, update **int grid[9][9]**, mark that number as used in **availableNum**, and go to the next cell
- If it is NOT, try another number until you find a number that it works
- If none of the number works, go back to the previous cell, select other available number, and restart with that number

int availableNum[81][9]

sudoku Generator .c

9	0	2	3	4	5	6	7	8	9
10	1	2	0	0	5	6	7	8	9
11	1	0	0	4	5	6	7	8	0
12	1	2	0	4	0	6	7	8	0
13	1	2	0	4	0	0	0	8	0
14	1	2	0	4	0	6	0	0	0
15	0	2	0	4	0	6	0	0	0
16	0	0	0	4	0	6	0	0	0
17	0	0	0	4	0	0	0	0	0
18	0	2	0	4	5	6	7	8	9
19	1	2	3	0	5	0	7	8	0
20	1	0	3	4	0	0	7	0	9
21	1	2	3	4	5	0	0	0	0

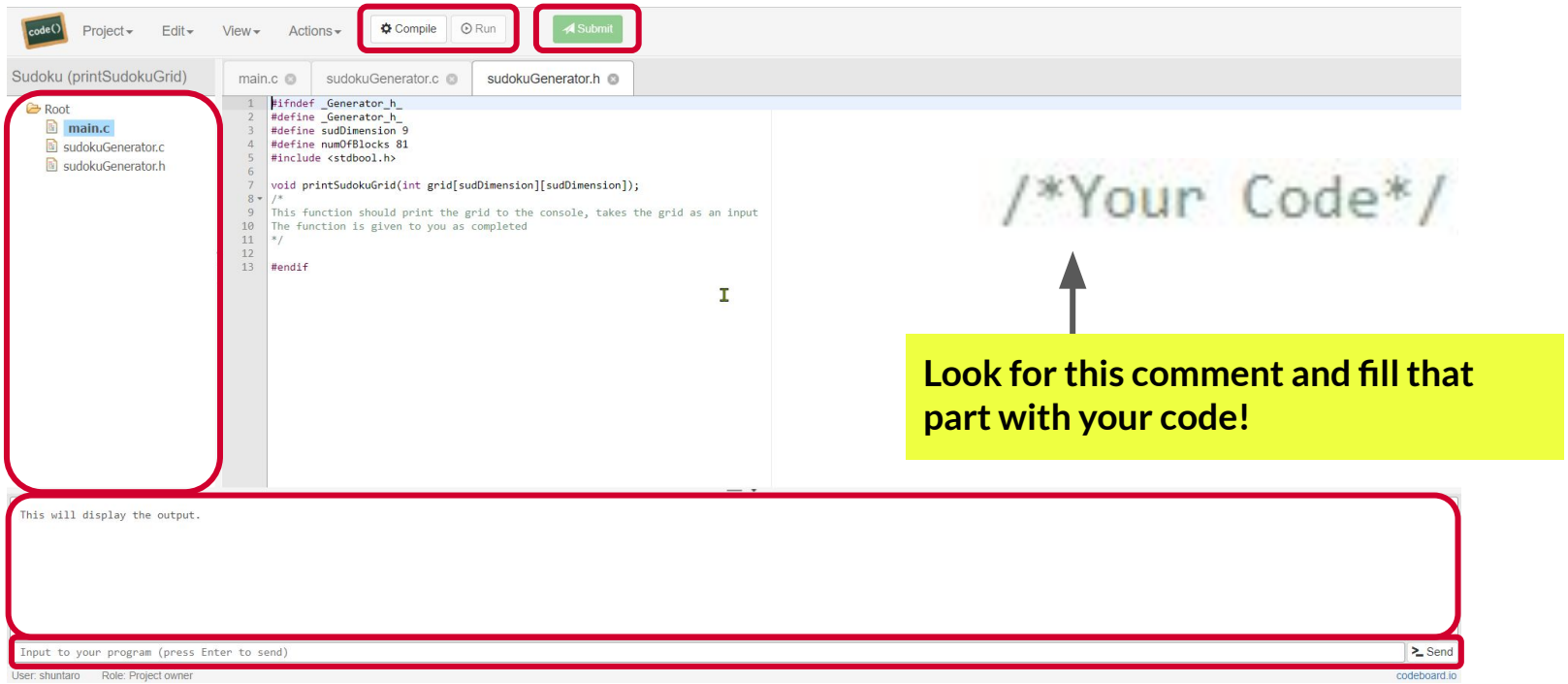
Row 1

Row 2

random → 2

	0	1	2	3	4	5	6	7	8
0	1	4	2	9	6	3	7	8	5
1	3	9	5	7	8	1	2	6	4
2	6	8	7	2	4	5	3	1	9
3									
4									
5									
6									
7									
8									

Code Board https://codeboard.io/signup



Activity 1: printing a board

sudoku Generator .c

https://codeboard.io/projects/390517

Expected Output:

```
3 |6 |7 |  5 |3 |5 |  6 |2 |9 |
1 |2 |7 |    |9 |3 |  6 |  |6 |
2 |6 |1 |  8 |7 |9 |  2 |  |2 |

3 |7 |5 |  9 |2 |2 |  8 |9 |7 |
3 |6 |1 |  2 |9 |3 |  1 |9 |4 |
7 |8 |4 |  5 |  |3 |  6 |1 |  |

6 |3 |2 |    |6 |1 |  5 |5 |4 |
7 |6 |5 |  6 |9 |3 |  7 |4 |5 |
2 |5 |4 |  7 |4 |4 |  3 |  |7 |
>
```

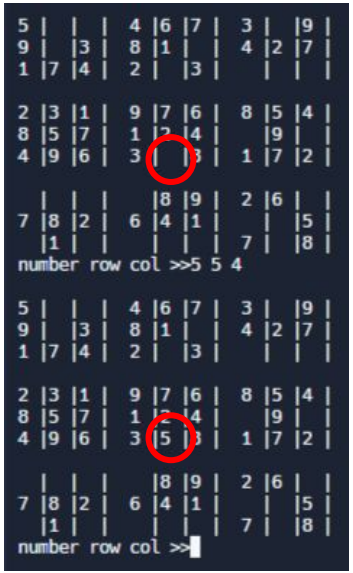
- The numbers on the board does not matter
- Your goal is to make the board printed on the console
- After successfully finishing this activity, you can modify your code to make the board even cooler than ours!!! (adding colors, more bars etc)

Don't Forget to Submit ;-)

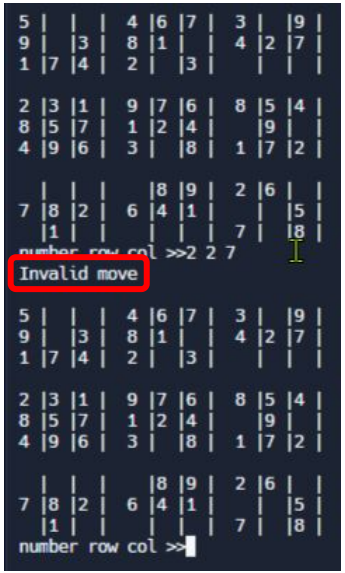
Activity 1: checking the legality of moves

sudoku Generator .c

https://codeboard.io/projects/390533



Valid Moves



Invalid Moves

- Your goal is, given a user input (this part is already done), to check if that input move is legal
- If the move is legal, the board will be updated(this part is also done)
- If the move is not legal, it asks for another user input (this part is also done)
- This is not necessary, but you can also take a look at main.c to see how **getting user input** works

Don’t Forget to Submit ;-)

Activity 2: out of number & replenish

sudoku Generator .c

https://codeboard.io/projects/390530

Initial Output:

```
   |  |  |  |  |  |  |  |  |
3 |6 |7 |5 |3 |5 |6 |2 |9 |
1 |2 |7 |  |9 |3 |6 |  |6 |
2 |6 |1 |8 |7 |9 |2 |  |2 |
3 |7 |5 |9 |2 |2 |8 |9 |7 |
3 |6 |1 |2 |9 |3 |1 |9 |4 |
   |  |  |  |  |  |  |  |
7 |8 |4 |5 |  |3 |6 |1 |  |
6 |3 |2 |  |6 |1 |5 |5 |4 |
7 |6 |5 |6 |9 |3 |7 |4 |5 |
2 |5 |4 |7 |4 |4 |3 |  |7 |
8 |6 |8 |8 |4 |3 |1 |4 |9 |
2 |  |6 |8 |9 |2 |6 |6 |4 |
9 |5 |  |4 |8 |7 |1 |7 |2 |
7 |2 |2 |6 |1 |  |6 |1 |5 |
9 |4 |9 |  |9 |1 |7 |7 |1 |
1 |5 |9 |7 |7 |6 |7 |3 |6 |
   |  |  |  |  |  |  |  |
5 |6 |3 |9 |4 |8 |1 |2 |9 |
3 |9 |  |8 |8 |5 |  |9 |6 |
3 |8 |5 |6 |1 |1 |5 |9 |8 |
4 |8 |1 |  |3 |  |4 |4 |4 |
4 |7 |6 |3 |1 |7 |5 |9 |6 |
```

Expected Output:

```
1 |2 |3 |4 |5 |6 |7 |8 |9 |
3 |6 |7 |5 |3 |5 |6 |2 |9 |
1 |2 |7 |  |9 |3 |6 |  |6 |
2 |6 |1 |8 |7 |9 |2 |  |2 |
3 |7 |5 |9 |2 |2 |8 |9 |7 |
3 |6 |1 |2 |9 |3 |1 |9 |4 |
1 |2 |3 |4 |5 |6 |7 |8 |9 |
7 |8 |4 |5 |  |3 |6 |1 |  |
6 |3 |2 |  |6 |1 |5 |5 |4 |
7 |6 |5 |6 |9 |3 |7 |4 |5 |
2 |5 |4 |7 |4 |4 |3 |  |7 |
8 |6 |8 |8 |4 |3 |1 |4 |9 |
2 |  |6 |8 |9 |2 |6 |6 |4 |
9 |5 |  |4 |8 |7 |1 |7 |2 |
7 |2 |2 |6 |1 |  |6 |1 |5 |
9 |4 |9 |  |9 |1 |7 |7 |1 |
1 |5 |9 |7 |7 |6 |7 |3 |6 |
1 |2 |3 |4 |5 |6 |7 |8 |9 |
5 |6 |3 |9 |4 |8 |1 |2 |9 |
3 |9 |  |8 |8 |5 |  |9 |6 |
3 |8 |5 |6 |1 |1 |5 |9 |8 |
4 |8 |1 |  |3 |  |4 |4 |4 |
4 |7 |6 |3 |1 |7 |5 |9 |6 |
```

- The numbers on the board does not matter
- Your goal is to detect the rows that is out of available numbers and replenish them with numbers 1 - 9
- These functions are for availableNum[81][9]

Don't Forget to Submit ;-)

Activity 3: filling the board

sudoku Generator .c

https://codeboard.io/projects/390529

```
7 |9 |8 |  3 |6 |5 |  2 |1 |4 |
5 |1 |2 |  7 |4 |9 |  8 |6 |3 |
4 |6 |3 |  1 |8 |2 |  5 |7 |9 |

8 |4 |5 |  2 |9 |1 |  7 |3 |6 |
9 |2 |7 |  6 |5 |3 |  1 |4 |8 |
6 |3 |1 |  8 |7 |4 |  9 |5 |2 |

2 |5 |9 |  4 |3 |7 |  6 |8 |1 |
3 |7 |6 |  9 |1 |8 |  4 |2 |5 |
1 |8 |4 |  5 |2 |6 |  3 |9 |7 |
>
```

1st Run

```
3 |5 |8 |  4 |9 |2 |  1 |6 |7 |
1 |4 |9 |  7 |8 |6 |  3 |2 |5 |
6 |7 |2 |  5 |3 |1 |  9 |8 |4 |

2 |8 |7 |  9 |4 |5 |  6 |3 |1 |
9 |3 |5 |  6 |1 |7 |  8 |4 |2 |
4 |6 |1 |  8 |2 |3 |  7 |5 |9 |

7 |1 |6 |  2 |5 |8 |  4 |9 |3 |
5 |9 |3 |  1 |6 |4 |  2 |7 |8 |
8 |2 |4 |  3 |7 |9 |  5 |1 |6 |
>
```

2nd Run

- Your goal is, given an empty 9x9 2D array, to generate a complete sudoku board that does not violate the rules
- You should use a rand() function in a way that every time you run this function, you get a different sudoku board
- This function needs to call the functions you implemented in activity 2 & activity 3

Don’t Forget to Submit ;-)

Activity 4: removing some numbers

sudoku Generator .c

https://codeboard.io/projects/390535

```
5 |2 |8 |  4 |6 |7 |  3 |1 |9 |
9 |6 |3 |  8 |1 |5 |  4 |2 |7 |
1 |7 |4 |  2 |9 |3 |  5 |8 |6 |

2 |3 |1 |  9 |7 |6 |  8 |5 |4 |
8 |5 |7 |  1 |2 |4 |  6 |9 |3 |
4 |9 |6 |  3 |5 |8 |  1 |7 |2 |

3 |4 |5 |  7 |8 |9 |  2 |6 |1 |
7 |8 |2 |  6 |4 |1 |  9 |3 |5 |
6 |1 |9 |  5 |3 |2 |  7 |4 |8 |
```

Filled

```
  |2 |8 |    |  |7 |  3 |  |9 |
9 |  |3 |    |1 |5 |    |  |  |
1 |7 |4 |    |9 |3 |    |  |6 |

2 |3 |1 |  9 |7 |6 |  8 |5 |4 |
8 |  |7 |  1 |2 |4 |  6 |9 |3 |
4 |9 |  |  3 |5 |8 |    |7 |  |

3 |4 |5 |  7 |8 |  |  2 |  |  |
7 |8 |2 |    |  |1 |  9 |  |5 |
6 |1 |9 |    |  |2 |    |4 |8 |
>
```

Removed

- Your goal is, given a complete sudoku board, to remove some numbers to make it a puzzle
- You should use a rand() function in a way that every time you run this function, you remove different cells
- This activity is independent from the last 4 activities, but we will combine all of them later
- We can adjust the number of cells we remove to change the difficulty of the puzzle

Don’t Forget to Submit ;-)

Activity 5: let the user play with the game

play Mode .c

https://codeboard.io/projects/390506

```
  |4 |2 |  6 |  |5 |  3 |8 |7 |
  |  |  |    |  |  |  4 |  |9 |
6 |7 |3 |  4 |  |  |    |2 |5 |

  |  |1 |    |4 |2 |  9 |  |8 |
  |  |5 |    |  |6 |  7 |  |2 |
  |2 |  |    |  |  |  6 |3 |  |

  |  |7 |    |  |  |    |1 |  |
  |3 |6 |    |5 |1 |  8 |9 |  |
5 |1 |9 |    |  |4 |  2 |  |3 |
Select a digit from 1-9: 6
Select the coordinates for row and column between 0-8:
1 7
Valid Move

  |4 |2 |  6 |  |5 |  3 |8 |7 |
  |  |  |    |  |  |  4 |6 |9 |
6 |7 |3 |  4 |  |  |    |2 |5 |

  |  |1 |    |4 |2 |  9 |  |8 |
  |  |5 |    |  |6 |  7 |  |2 |
  |2 |  |    |  |  |  6 |3 |  |

  |  |7 |    |  |  |    |1 |  |
  |3 |6 |    |5 |1 |  8 |9 |  |
5 |1 |9 |    |  |4 |  2 |  |3 |
Select a digit from 1-9:
```

- Your goal is to complete the fully functional sudoku game
- **playing** function should ask for the user's input, check its legality, and decide whether that move should be played or not
- You should also check for unexpected user inputs to prevent the program from crashing
- After finishing this part, you can work on further to improve the user interface!

Don't Forget to Submit ;-)

Our Solution

https://replit.com/@ShuntaroWakamat/Sudoku

```
4 |8 |  |  6 |  |  |  9 |2 |5 |
1 |5 |6 |    |2 |9 |  7 |8 |  |
3 |  |9 |  8 |5 |  |  6 |4 |1 |

  |  |8 |    |9 |6 |  3 |  |  |
7 |6 |1 |  2 |4 |  |  8 |5 |9 |
9 |3 |5 |  7 |  |8 |  2 |  |  |

  |  |4 |  1 |  |2 |    |3 |8 |
5 |7 |  |    |8 |4 |  1 |9 |  |
8 |  |  |  9 |  |5 |  4 |7 |2 |
Select a digit from 1-9:
```

Debugging & Testing: Simon, Nick
Commenting & Organizing Code: Ryan
Adding Color: Iris
Activities: Me

Give It a Try! “Power of Three”

Given an integer n, return *true if it is a power of three. Otherwise, return false*.

An integer n is a power of three, if there exists an integer x such that n == 3x.

Example 1:

Input: n = 27
Output: true
Explanation: 27 = 33

Example 2:

Input: n = 0
Output: false
Explanation: There is no x where 3x = 0.

Sample solution of Power of Three

```
long powerOfThree(int m){
    if(m == 0)
    {
        return 1;
    }
    return 3 * powerOfThree(m - 1);
}

bool isPowerOfThree(int n){

    for (int i = 0; powerOfThree(i) <= n; i++)
    {
        if(powerOfThree(i) == n)
        {
            return true;
        }
    }
    return false;
}
```

ECE Summer Workshop Software

APS105 Review & ECE244 Preparation
Day 2

Today we will be covering:

- Dynamic Memory Allocation
- Struct
- Linked Lists
- Sorting Linked Lists

String I/O - 1

- gets(s1);
- puts(s1);

printf(), scanf() for strings.

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

int main(){

    char temp[20];
    printf("Enter your Name: ");
    gets(temp);

    printf("My Name is: ");
    puts(temp);
    return 0;
```

```
Enter your Name: Iris
My Name is: Iris
```

String I/O - 2

- Difference between gets() and scanf()
- In scanf() function white-space is not considered as input character and also it stops reading input from external source if any white-space is encountered in between
- On other hand gets() function is used to receive input from the keyboard till it encounters a newline or EOF. The whitespace is considered as a part of the input.

String I/O - 3

- Difference between gets() and scanf()
- Scanf() function takes the format string and list of addresses of variables. e.g. scanf(“%d”, &number);
- It can also read multiple values of different data types.
- On other hand get() function takes the name of the variable to store the received value. e.g. gets(name);
- get() function will only get character string data.

Moving on…

Dynamic Memory Allocation

- Why use Dynamic Memory Allocation in C?
  - An array is a collection of a fixed number of values.
  - Declaring the size of the array which is unknown might cause insufficiency.
- To solve this issue, you can allocate memory manually during run-time

Dynamic Memory Allocation

- All functions are defined in the <stdlib.h> header file
  - malloc()
    - ptr = (castType*) malloc(sizeof(castType));
    - Memory is uninitialized
  - calloc()
    - Memory is initialized with all bits to zero
  - free()
    - Frees used memory to prevent memory leaks

```c
// Program to calculate the sum of n numbers entered by the user

#include <stdio.h>
#include <stdlib.h>

int main() {
  int n, i, *ptr, sum = 0;

  printf("Enter number of elements: ");
  scanf("%d", &n);

  ptr = (int*) malloc(n * sizeof(int));

  // if memory cannot be allocated
  if(ptr == NULL) {
    printf("Error! memory not allocated.");
    exit(0);
  }

  printf("Enter elements: ");
  for(i = 0; i < n; ++i) {
    scanf("%d", ptr + i);
    sum += *(ptr + i);
  }
  printf("Sum = %d", sum);

  // deallocating the memory
  free(ptr);

  return 0;
}
```

```
Enter number of elements: 5
Enter elements: 1
2
3
4
5
Sum = 15
```

Variable Length Arrays + Scanf(%ms)

Whereas, malloc gives you more manual control, you can let the computer handle memory allocation:

1. You can let the computer handle assigning memory for the array the same way as when you assign arrays a length, but use a variable instead (Variable Length Arrays)
   - “array[n]”
   - Need to be careful of variable scope
2. You can also take in strings dynamically using %ms (malloc string), so that you don’t have to pre-assign memory for a string
   - “scanf(%ms, char* variable)”

Let’s Build It: Pokemon Activity

Create a dynamically changing array database of length n, received from the user which stores the names (strings) of pokemon, that are also received from the user.

As a template:

1. Get the user to enter in the number of pokemon they want to add
2. Get the user to type in the names of ‘n’ pokemon
3. Build the dynamic 2D array database of pokemon names

Struct

- You can create a structure by using the struct keyword and declare each of its members inside curly braces:

```
struct MyStructure {    // Structure declaration
  int myNum;            // Member (int variable)
  char myLetter;        // Member (char variable)
}; // End the structure with a semicolon
```

Struct

- Create a struct variable with the name “s1”
- Defining our own "syntactic sugar" using the typedef keyword.

```
struct myStructure {
  int myNum;
  char myLetter;
};

int main() {
  struct myStructure s1;
  return 0;
}
```

```
typedef struct myStructure {
        int myNum;
        char myLetter;
    } myStructure;
```

Struct

- To access members of a structure, use the dot syntax (.):

```c
// Create a structure called myStructure
struct myStructure {
  int myNum;
  char myLetter;
};

int main() {
  // Create a structure variable of myStructure called s1
  struct myStructure s1;

  // Assign values to members of s1
  s1.myNum = 13;
  s1.myLetter = 'B';

  // Print values
  printf("My number: %d\n", s1.myNum);
  printf("My letter: %c\n", s1.myLetter);

  return 0;
}
```

Struct

- You can also assign values to members of a structure variable at declaration time, in a single line.

```c
// Create a structure
struct myStructure {
  int myNum;
  char myLetter;
  char myString[30];
};

int main() {
  // Create a structure variable and assign values to it
  struct myStructure s1 = {13, 'B', "Some text"};

  // Print values
  printf("%d %c %s", s1.myNum, s1.myLetter, s1.myString);

  return 0;
}
```

Linked Lists

- Collection of nodes/structs linked together by pointers

Linked Lists: Nodes

```
typedef struct Node {
  int key;
  struct Node* next;
} Node;

typedef struct LinkedList {
  Node* head;
} LinkedList;
```

Code For Activity:
https://codeboard.io/projects/390588

Linked List: Insert

```
void insert(LinkedList* list, Node* n) {

}
```

Linked Lists: Print

```
void print(LinkedList* list) {

}
```

Linked Lists: Find

```
Node* find(LinkedList* list, int val) {

}
```

Linked Lists: Delete

```
void delete(LinkedList* list, int val) {

}
```

Mini Activity: Using LinkedList

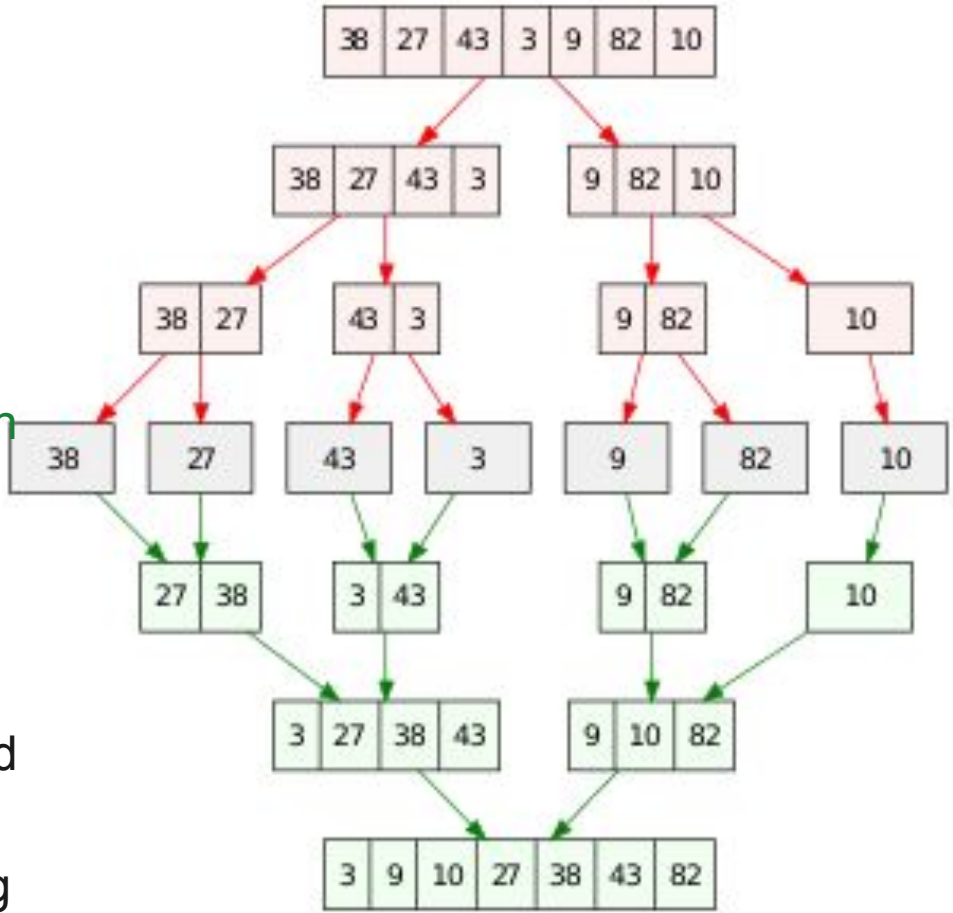
- Play around with the four operators for the LinkedList
- Example Challenges:
  - Insert Nodes from 0-99 in random order and print them
  - Delete odd numbered nodes
- **Challenge:** build a doubly-linked list

Activity: Pokemon Database

- Build a LinkedList database that keeps track of Pokemon
  - Store at least the name and pokedex number (but you can add more!)
  - You can either add more data to the *Node* struct and rename it or replace *int key* with a new struct called *Pokemon*
- Build a console user interface to insert, print, find, and delete pokemon from the database
  - Pseudo-code given in main.c

Linked Lists: Sorting

- General Idea: Divide and Conquer
- Recursively make splits
- Merge them back together in sorted order



*Special considerations for linked lists:
Swapping Numbers -> Swapping Nodes

ECE Summer Workshop Software

APS105 Review & ECE244 Preparation
Day 3

Sorting Algorithms

Iterative

Bubble Sort

Insertion Sort

Recursive

Merge Sort

Quick Sort

What makes a sorting algorithm fast?

1. **Comparisons:** How many comparisons, before we know we're done?
2. **Data Movement:** How often does it have to shift data around?
3. **Setup Time:** How quickly can it make those comparisons?

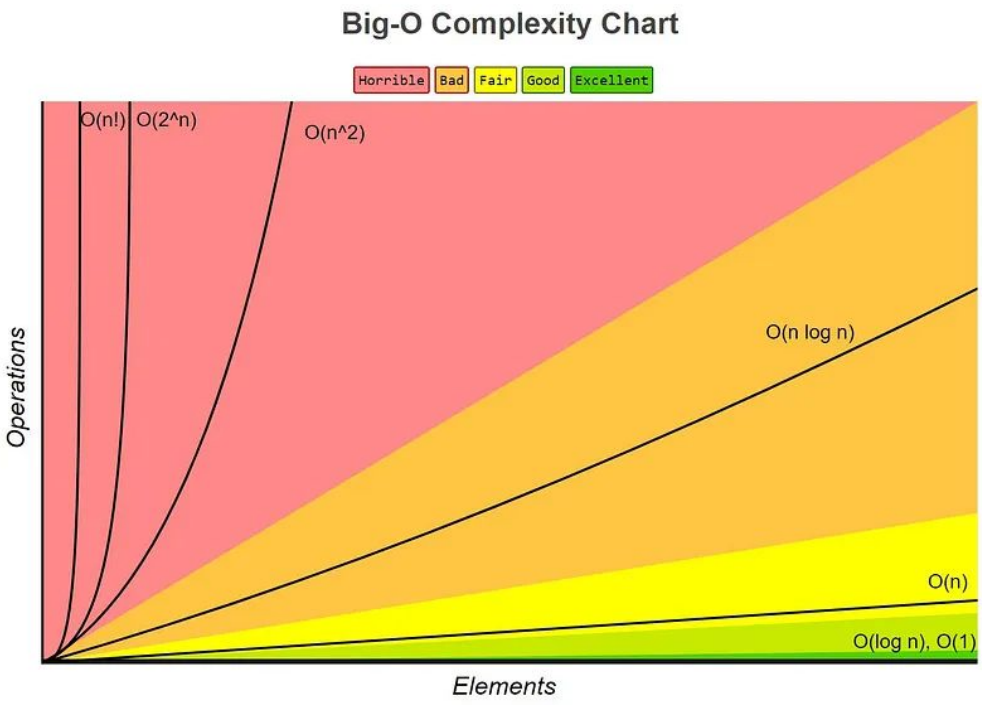
Randomness: The data we're given, might already be partially sorted, or sorted already in a unique way than what we desire

- ie. sorted from highest to lowest, when we desire lowest to highest

Time Complexity

Big-O Complexities:

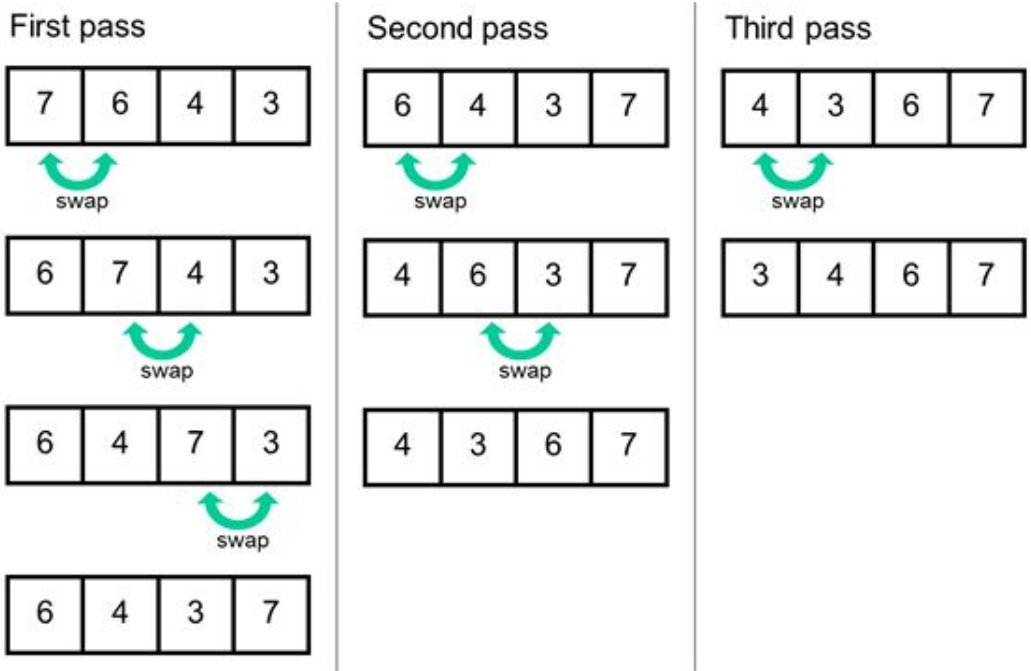
1. O(1)
2. O(logn)
3. O(n)
4. O(nlogn)
5. O(n^k)
6. O(k^n)
7. O(n!)



Bubble Sort

Time Complexity
Best: O(n^2)
Worse: O(n^2)

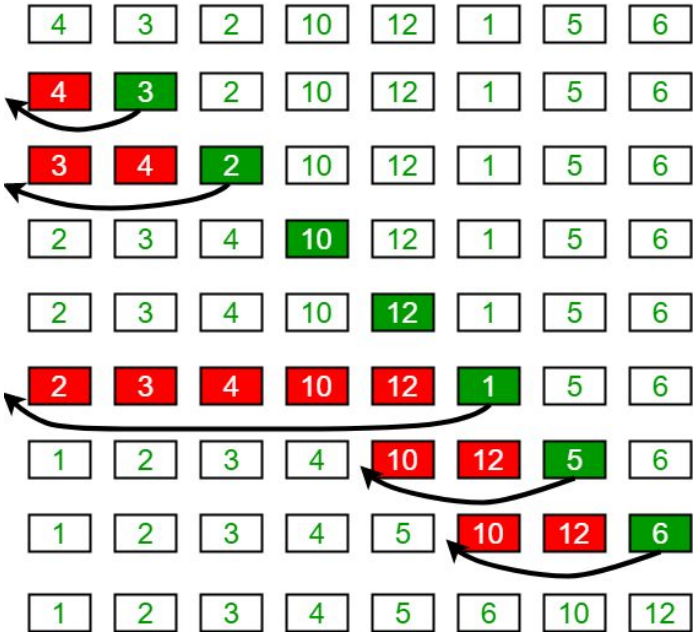
Sort array through shifting biggest numbers to the end



Insertion Sort

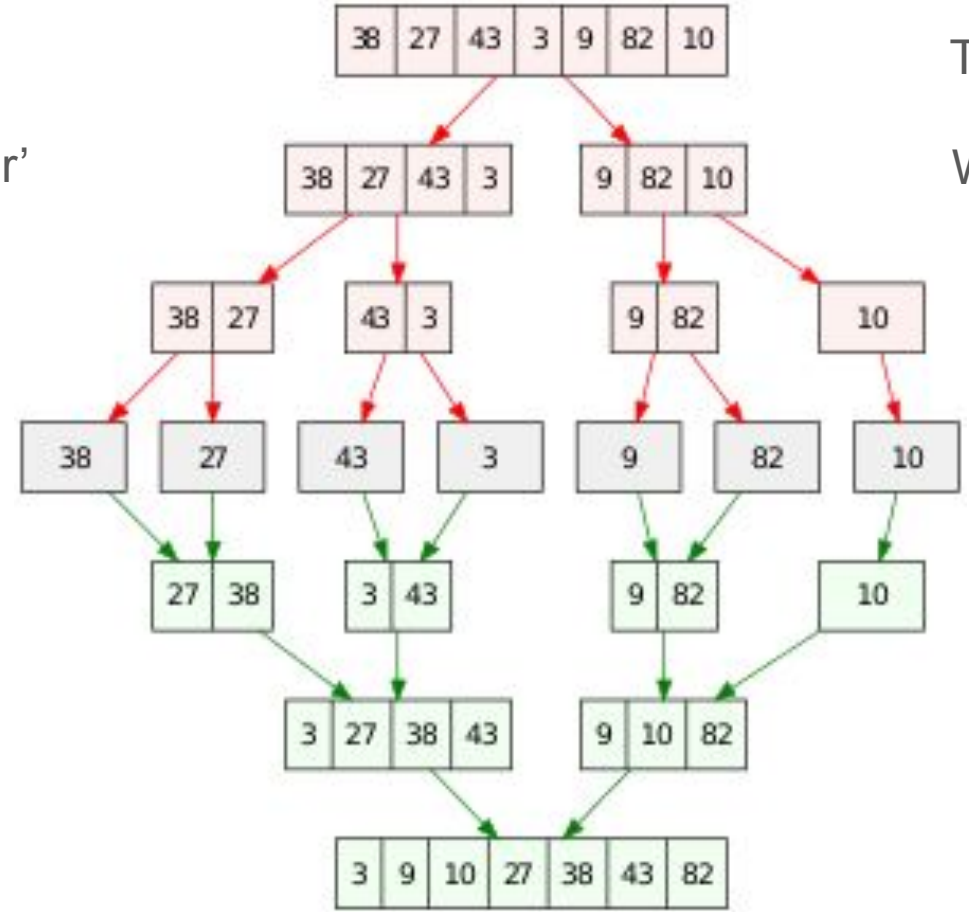
Time Complexity
Best: O(n)
Worse: O(n^2)

Sort array by shifting numbers backwards into their correct spots



Merge Sort

‘Divide and conquer’

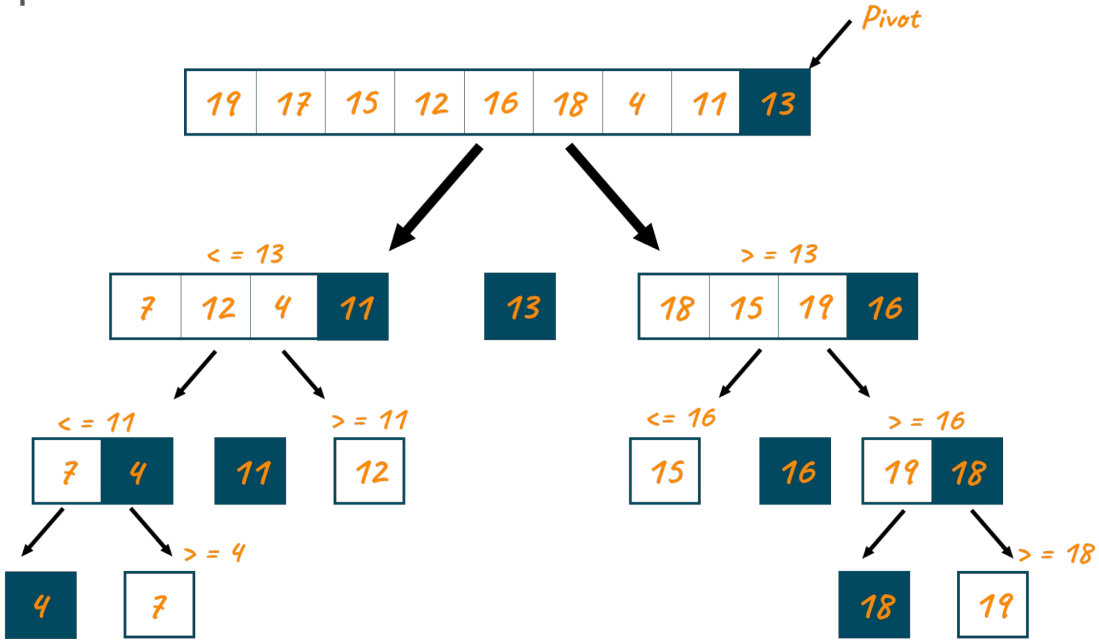


Time Complexity
Best: O(nlogn)
Worse: O(nlogn)

Quick Sort

Time Complexity
Best: O(nlogn)
Worse: O(n^2)

‘Divide and conquer’ with more risk



Quick Sort - Median of 3 Improvement

Time Complexity
Best: O(nlogn)
Worse: O(n^2)

Pivot – Median of Three

2	5	6	4	13	3	12	19	6

A[left] = 2, A[center] = 13, A[right] = 6

2	5	6	4	6	3	12	19	13

Swap A[center] and A[right]

2	5	6	4	6	3	12	19	13

pivot

Choose A[center] as pivot

Can this work with Linked Lists?

Adjustments we would have to make:

1. How do we traverse through nodes?
2. How do we compare nodes?
3. How do we swap nodes?

ECE Summer Workshop Software

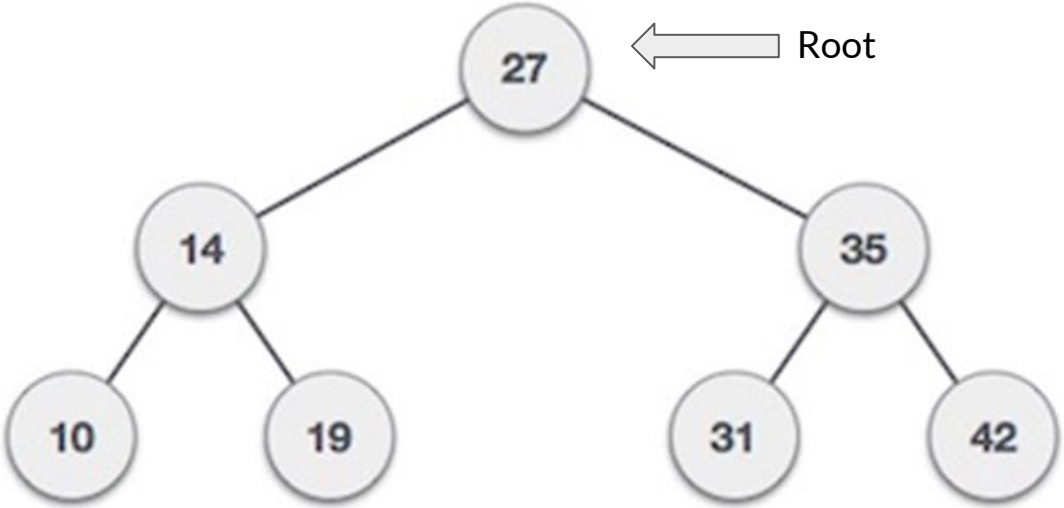
APS105 Review & ECE244 Preparation
Day 4

Binary Tree

- A binary tree is a tree structure, with a parent node having at most two children nodes.
- It is represented using a pointer to the root node of the binary tree.
- There is no restriction of how the nodes of a binary tree should be structured, except for at most 2 children nodes per parent node.

Binary Search Tree

- A node-based binary tree, with the nodes arranged in an order
- Value of the left child < value of the parent node
- Value of the right child > value of the parent node



Define a BST

```
typedef struct node
{
    int data;
    char character;
    char* string;
    // Add whatever data you want the node to hold here
    //make sure you are arranging the BST based on only 1 data
    BSTNode* left;
    BSTNode* right;
}BSTNode;
```

Create a new node for a BST

- Default constructor
- Returns a **pointer** to the BSTNode

```
BSTNode *newNode(int dataIn, char characterIn, char* stringIn){
    BSTNode *temp = (BSTNode *)malloc(sizeof(BSTNode));
    temp->data = dataIn;
    temp->character = characterIn;
    temp->string = stringIn;
    temp->left = NULL;
    temp->right = NULL;
    return temp;
}
```

Insertion in BST

- Returns the root of the new BST after being modified
- Determine the order you want the BST to be based on!
- In this case, “string” is the one being evaluated on to determine the order of the BST.
- How to determine the order of the strings? How to actually insert all the nodes based on alphabetical order?
- Start with the most basic case…

Insertion of new node

- Since it is a BST structure, the root will never change when we are inserting the new node after the first node is selected as the root, so we will start from here.
- The “node” here is the root of the BST to be inserted into.
- Also important for the recursion process coming later.

```
BSTNode *insertNewNode(BSTNode *node, int dataIn, char chatacterIn,  char* stringIn){
    if(node == NULL){
        //If there is no node at the beginning, create a node and use it as the root
        return newNode(dataIn, chatacterIn, stringIn);
    }
}
```

Insertion of new node

- Compare the strings using strcmp(), to determine the order.
- Recursion until find the correct location to insert the node!

```
BSTNode *insertNewNode(BSTNode *node, int dataIn, char chatacterIn,  char* stringIn){
    if(node == NULL){
        //If there is no node at the beginning, create a node and use it as the root
        //In recursion, if there is no specified child of a node, insert the new node as the child
        return newNode(dataIn, chatacterIn, stringIn);
    }
    if (strcmp(stringIn, node->string) > 0) {
        //If the string of the new node is larger than the root string
        //Insert it into the right branch of the tree and continue the recursion
        return insertNewNode(node->right, dataIn, chatacterIn, stringIn);
    }
    else if (strcmp(stringIn, node->string) < 0) {
        //If the string of the new node is smaller than the root string
        //Insert it into the left branch of the tree and continue the recursion
        return insertNewNode(node->left, dataIn, chatacterIn, stringIn);
    }
}
```

Avoid duplicate values

- By definition, BST cannot have duplicate values for the order key.
- Either use another key to determine the “secondary order”, or do not insert the value at all.
- If you decide to use another key to determine the secondary order, make sure to account for it in your deletion algorithm as well.

```
else if (strcmp(stringIn, node->string) == 0) {
    //if dupilicate value, either use another key to determine the "second order of searching"
    //or just abandon the value to be inserted, like how it is done here
    return NULL;
}
```

Searching in BST

- Use strcmp to compare the key and the root's string, if no match, then based on the value, go into either the right or left subtree and do recursion.

```
BSTNode *searchNode(BSTNode *root, char* target){

    if (root == NULL){
        return root;
    }
    else if (strcmp(target, root->string) > 0){
        //going into the right subtree
        return searchNode(root->right, target);
    }
    else if (strcmp(target, root->string) < 0){
        //going into the left subtree
        return searchNode(root->left, target);
    }
    else{
        //The target is the same as the current root's string
        return root;
    }
}
```

Deletion in BST

- Deletion is a bit complex in BST, just because after deletion, the BST has to still remain sorted.
- You have to search for the matching BSTNode first, and then delete it following the rules of the BST. You don’t have to search for the things that sorts the list (in our case, the string), but you have to use the thing that sorts the list to arrange the BST after deletion.
- Again, let’s use divide and conquer approach, starting from the most basic case…

Deletion in BST

- Basic case 1: The matching node has no children. In this case, we have to delete and free the node, go back to the parent node and make it points to NULL.
- Basic case 2: The matching node has only one children. In this case, we only have to make the parent node of the deleted node pointing to its only children, and delete and free the node.

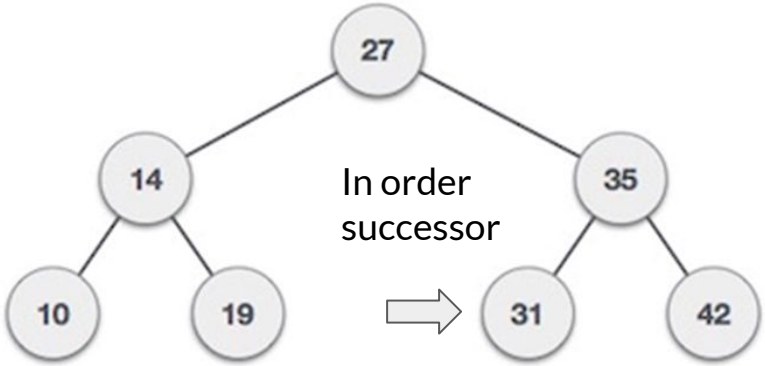
Deletion in BST - Basic case

```
BSTNode *deleteNode(BSTNode *root, char* key){
    if(root == NULL){
        return NULL;
    }
    if(strcmp(root->string, key) == 0){
        //found
        //case of only one child, treated it basically just like a linked list.
        if(root->left == NULL){
            BSTNode *temp = root->right;
            free(root);
            //temp here is the new root
            return temp;
        }else if(root->right == NULL){
            BSTNode *temp = root->left;
            free(root);
            //temp here is the new root
            return temp;
        }
    }
}
```

Deletion in BST - 2 children

- So, what if there are 2 children for the BSTNode of the matching key?
- Get the in order successor (in this case, the smallest key in this BST that is larger than the matching key), “replace” the deleted node with it.
- We need a helper function that finds the in order successor, which is the left most node of the right subtree of the original BST.

```
BSTNode *minValue(BSTNode *node){
    BSTNode *current = node;
    while (current != NULL && current->left != NULL){
        current = current->left;
    }
    return current;
}
```



Deletion in BST: 2 children

- It is like a chain reaction, you are replacing the deleted node with its inorder successor, but that will also cause the original inorder successor node to be “deleted”, so you have to go delete that node and update the root of the subtree.

```
//case of two children
//get the inorder successor
BSTNode *temp = minValue(root->right);
//copy the inorder successor's content to the current node
//"replace the deleted node"
root->string = temp->string;
//go through the same process of the inorder successor
//because it is like the root of the subtree got deleted when making the replacement
root->right = deleteNode(root->right, temp->string);
return root;
```

Deletion in BST: recursion

- Now we add the last part of the deletion: recursion, which if the string is larger than the root's string value, we move to the right subtree of the BST, and vice versa.
- You can find the sample completed solution in your materials.

```
if(strcmp(key, root->string) < 0){
    return deleteNode(root->left, key);
}
else if(strcmp(key, root->string) > 0){
    return deleteNode(root->right, key);
}
```

Traversal in BST

- In order traversal: left -> node -> right
- Pre order traversal: node -> left -> right
- Past order traversal: left -> right -> node
- For BST, the in order traversal will get all the elements traversed in the ascending order (and hence the in order successor in the deletion part).

```
void in_order_traversal(BSTNode *root){
    if(root != NULL){
        in_order_traversal(root->left);
        printf("%s\n", root->string);
        in_order_traversal(root->right);
    }
}
```

Activity on BST

- Go to the following website for example LeetCode questions and complete them to test your understandings.
- If you don't have a LeetCode account, you can register one quickly.
- If there are any questions, feel free to reach out to us!
  - https://leetcode.com/problems/search-in-a-binary-search-tree/
  - https://leetcode.com/problems/convert-sorted-array-to-binary-search-tree/
  - https://leetcode.com/problems/validate-binary-search-tree/
  - https://leetcode.com/problems/insert-into-a-binary-search-tree/

Header files, modularity and project organization

- Makes no difference to the compiler, but makes a lot of difference to you and your co-workers!
- Imagine working on a project that involves million lines of codes…
- Split the codes into different functions, and put different functions into different files.
- Potential benefits:
  - Division of work between you and your co-workers
  - Easier to debug and maintain the code
  - Easier for others to add more things to your code without causing conflicts or problems!
- You will understand the importance of it in ECE297…

Header files

- Each header file (.h) has its own matching source files (.c), where usually header files includes the function declaration, and the source file includes the function definition.
- By including the header files, you are able to use the functions defined in its source files.
- Example: One of the files in our ECE297 mapper project

```
//Libraries
#include <iostream>
#include <math.h>
#include <map>
#include <algorithm>
#include <vector>
#include <chrono>

//APIs
#include "StreetsDatabaseAPI.h"
#include "OSMDatabaseAPI.h"

//Utils
#include "map_initializers.hpp"

//Map structures
#include "map_database.hpp"
#include "interface.hpp"

//Other
#include "m1.h"
```

Header Files

- It is a good practice to write comment in the header files about what the function does, so without knowing what the actual code is in the source file, others will know how to use those functions.
- Connection to Black Box Method and business collaboration…

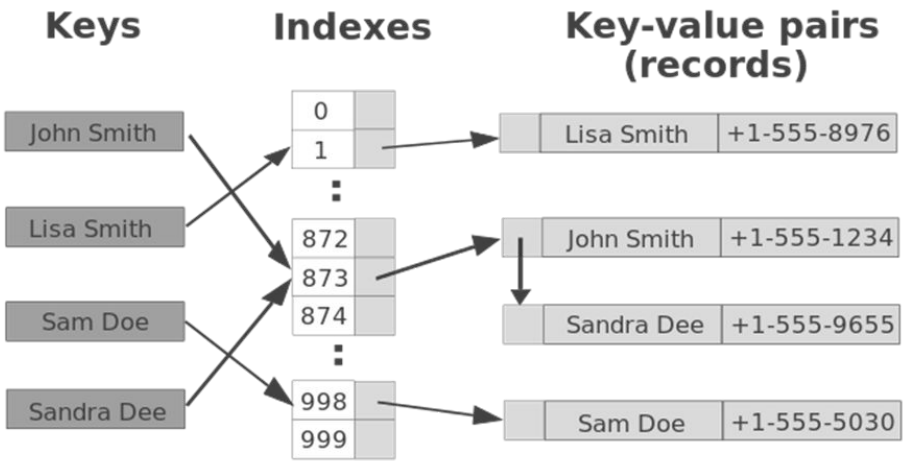
Same header files being included multiple times

- Causes compilation errors!
- Equivalent to multiple declaration of the same functions
- Solution: guard the header files!

```
#ifndef guardedheader
#define guardedheader
/*code*/
#endif
```

Hash Table

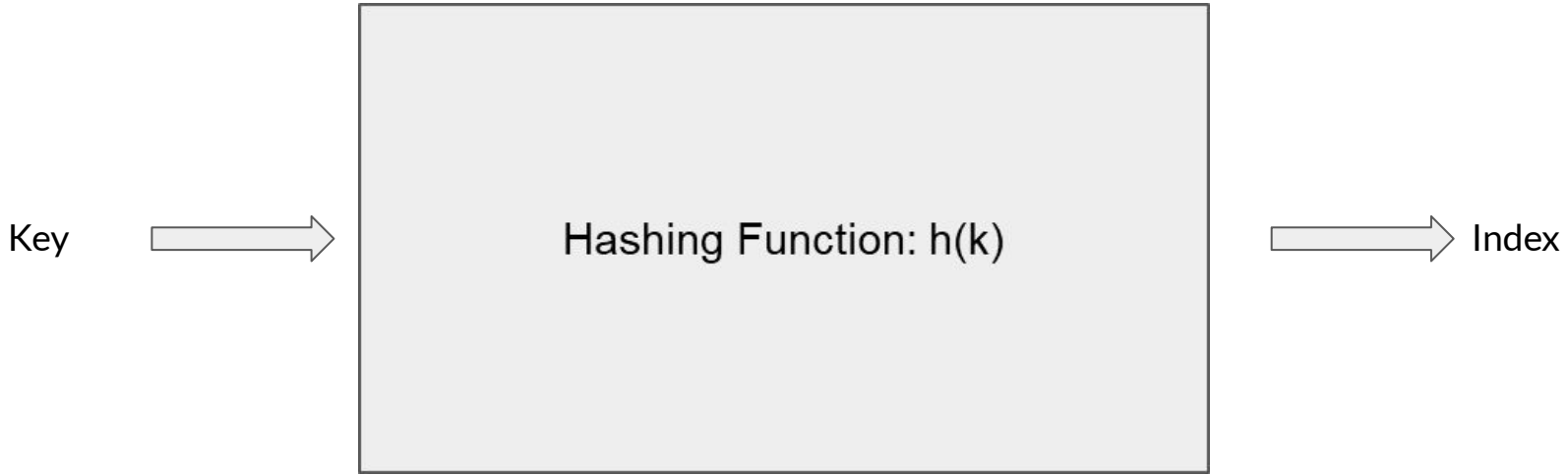
- Key, data and index.
- Use a function to convert the key to the index in the hash table (usually a big array), which stores a pointer to a struct with all the data in there.
- It gets much easier to use in C++, which you have access to:
  - Vector
  - Class and objects
  - Operator overloading



Hashing functions

Example hashing function:

Index by division: h(k) = k % m, which m is the size of the table

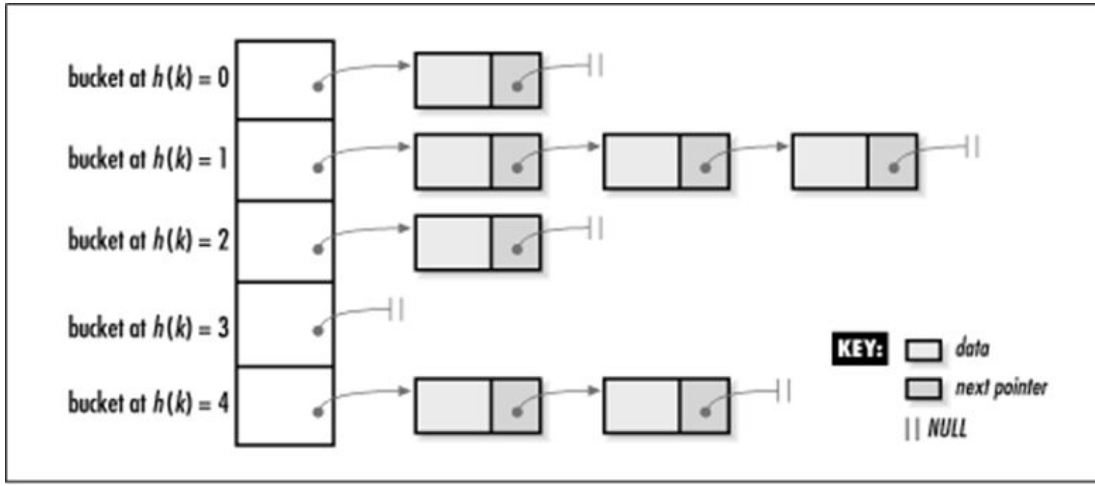


Collision resolution

- Sometimes the hashing function will get the same index for two different keys, which will then cause collision
- Commonly, there will be 2 resolution methods:
  - Chaining
  - Open addressing/linear probing

Chaining

- Similar to linked list!
- Link the conflict struct as linked list after the first one
- Construct the struct with a pointer to the next struct, set to NULL by default (just like how we did in linked list).



Open addressing/linear probing

- Change the original hashing function into a recursive one.
- If the index is not available, keep doing recursion until an available slot is found.
- Depends on the purpose of the hash table, it may increase the time complexity, of accessing the data stored.

Open addressing/linear probing

```
void insertElement(int key) {
    //in this example I will just use the key as the data.
    int size = 10000;
    int hashTable [10000] = {0};
    int index;
    //This is our hashing function
    index = key % size;
    while (hashTable[index] != 0){
        //repeat until an available spot is found.
        index = (index + 1) % size;
    }
    //store the data into the hash table.
    hashTable[index] = key;
}
```

Q&A Session

- Ask us anything!

ECE Summer Workshop Software

ECE244 Preparation
Day 3

Today we will be covering:

- Comparison of C and C++
- I/O and Strings
- Object Oriented Programming and Classes
- Accessors and Mutators
- Constructors and Destructors

Difference between C and C++

- Although C is completely compatible in C++, the main idea is different. C is a procedural language and C++ is object oriented. Because of that, C++ can be used to build more complex applications in a better and more efficient way.
- And now we are going to take a look at the things in C++ and compare it with how we originally do it in C.
- Remember: you can always just write C in C++, but now you have a better tool so why bother…

Difference between C and C++

- Procedural Language
  - Code executes top to bottom
- Lower level language
  - Faster to compile
  - Better for hardware

- Object Oriented Language
  - Code is organized into objects and namespaces
- Higher level language
  - Takes longer to compile but it's easier to code/read
  - Better for software

Important because:

1. As programs get bigger, upfront planning becomes especially important and the structure from objects help with this
2. Allows programmers to join projects faster (smoother onboarding)

Difference between C and C++ Cont’d

We’ll go through:

1. Syntax difference
2. Underlying functionality difference (where necessary)

User I/O Syntax

- In C, you have to specify the type of input and output, and consider the spaces in input and output

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

```
scanf("%d %d", input1, input2);
printf("%d %d\n", input1, input2);
```

- In C++, you do not have to specify the type, and you do not need to take consideration of the spaces when user inputs
- Both cin and cout are objects of the class iostream

```
#include <iostream>
using namespace std;
```

```
cin >> input1 >> input2;
cout << input1 << " " << input2 << endl;
```

I/O Underlying Functionality

- The library is called <iostream> because the input and output flow in and out of the program
  a. The input flows from the *cin* stream into program
  b. The output flows from the program into the *cout* stream

```
cin >> input1 >> input2;
cout << input1 << " " << input2 << endl;
```

Strings Syntax

- In C, a string is represented as an array of characters, with a pointer to the start of the array.
- You use different independent string handling functions to do actions to strings like duplication and comparison.

```
char string1 [] = "something";
char* string2 = "somethingElse";
if (strcmp(string2, string1) > 0) {
    string2 = strdup(string1);
}
```

- In C++, with the <string> library, string can be handled just like other variable types like int or char.
- C++ strings do not need independent functions like in C

```
string string1 = "something";
string string2 = "somethingElse";
if (string2 > string1) {
    string2 = string1;
}
```

Activity

- In this activity, you will follow the instances that we have given to you, input the required information and output them. The skeleton code is given to you.

https://codeboard.io/projects/390663

Input string with spaces

- Sometimes we want to have spaces be part of the input.
- We then use a getline(), function defined in <string>
- It takes in two parameters, the cin object and the string you want to be input in.
- This will result th ntil the end of the line.

```
int main(void) {
    string input;
    getline(cin, input);
}
```

StringStream

- Sometimes we use the getline to get all the input, but then we might want extract inputs from the string, which is where we use stringStream.
- Similar to cin, you also do not have to specify the type of the variable, but you do need to initialize the stream from the string that you want to parse.

```
int main(void) {
    int age;
    char gradeForAPS100;
    string firstName;
    string lastName;

    string information = "20 H Xiaofeng Xu";

    stringstream sin(information);

    sin >> age >> gradeForAPS100 >> firstName >> lastName;
}
```

StringStream

- What if the stream fails?
- Either due to the type mismatch, reach the end of the string, etc.
- You can check the reason and decide the action by its member functions.

```
sin >> test;
if (sin.fail()) {
    if (sin.eof()) {
        cout << "End of file reached." << endl;
    }
    else {
        cout << "Bad input." << endl;
        //Handle the bad input, you don't have to
        //clear the flag or it will continue to fail
        sin.clear();
        //ignore the bad input until space occurs and continue to receive inputs
        sin.ignore(10000, ' ');
    }
}
sin >> test;
```

Activity

- In this activity, you will help a student to only output the numeric grades while you have a string of his grades with numeric and letter grades. You are not allowed to create new variables to stream in the letter grades. The skeleton code is giving to you.
- https://codeboard.io/projects/390664

Objected Oriented vs Procedural

	Objected Oriented	Procedural
Pros	Reusable, customizable Encapsulation, Inheritance, Polymorphism	Easy-to-learn Fast & Efficient Suitable for small-scale projects
Cons	Hard-to-read Representing actual world	Not reusable, customizable Not intuitive
Components	Classes, Objects	Modulus, Functions
Languages	C++, Java, Ruby, Python	C, BASIC, FORTRAN

Objected Oriented vs Procedural

The most important difference between Object Oriented and Procedural is **where the data is stored**

In procedural languages, the data can be stored only as a **local variable** (inside the function) or as a **global variable** (outside of the function). The **caller function** (esp main) has to manage all the variables and properly pass those to the functions .

In object oriented languages, each instance can hold some data that is a member of the class. Using those data, each instance can execute tasks more independently than simple function calls, leading to more organization through **modularity**.

Modularity and Establishing Terminology

- Because we are modularizing our code, we now have data in different places/objects within the code
  - We will be referring to things that are within an object as **members**
- Opens a discussion of:
  - Which parts of the code can access the members of a class, are they **public** or **private**?
  - How can the code **access** or **mutate** their data members?
  - How are these objects **constructed/instantiated**? How are they **destructed**?

Classes

- They are a blueprint of some object you're trying to model in the real world
  - What **attributes** does this object have?
  - What **behavior** does this object have?
- They offer more design functionality that structs, which means as programmers we need to put more effort into planning/designing classes

Struct in C vs Class in C++

- In C, if you want a variable that represents a collection of multiple variables, you will need to use struct
- Access to these variables are by default public (anyone can access)
- Any functions that involve the struct must be written independently of the struct

- In C++, you can use classes to represent a collection of multiple variables
- Access to these member variables are by default private (only the class can access them)
- You can also write **member functions** of a class (similar to helper functions)

Struct

```c
typedef struct structure {
    int integer;
    int anotherInteger;
    char character;
}structure;
//constructor
structure *newStructure(int value1, int value2, char character1) {
    structure *temp = (structure*)malloc(sizeof(structure));
    temp->integer = value1;
    temp->anotherInteger = value2;
    temp->character = character1;
    return temp;
}
```

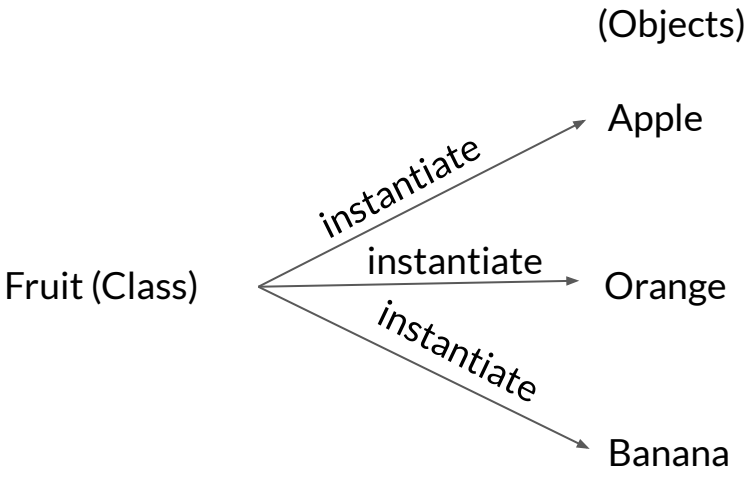
```c
int main(void) {
    structure actualStructure = {3, 4, 'c'};
    actualStructure.integer = 1;
    actualStructure.anotherInteger = 2;
    int i = actualStructure.integer;
}
```

Class

```cpp
class classDemo{
private:
    int integer;
    int anotherInteger;
    char character;
public:
    //constructor
    classDemo(int first, int second, char third) {
        integer = first;
        anotherInteger = second;
        character = third;
    }
    //accessor
    int integerAccessor () {
        return integer;
    }
    int anotherIntegerAccessor () {
        return anotherInteger;
    }
    //mutator
    void integerMutator (int value) {
        integer = value;
        return;
    }
    void anotherIntegerMutator (int value) {
        anotherInteger = value;
        return;
    }
};
```

```cpp
int main(void) {
    classDemo demo(3, 4, 'c');
    demo.integerMutator(1);
    demo.anotherIntegerMutator(2);
    int i = demo.integerAccessor();
}
```

Understanding of Object and Class



- Objects are instances of their class/blueprint
- Each member has concrete value
  - Student is a “Class” as each of you is an “Object”

Public and Private Data Members

- Classes can have both private and public members; private can only be accessed by the class and public can be accessed freely by the rest of the program.
- If we want to access private variables in the rest of our program, we can do so by giving our class **public member functions** called “Accessors” and “Mutato

```
class classDemo{
private:
    string myPassword; // "definitely_not_my_phone_num";
    string myPhoneNumber; // "647-909-9090";
public:
    string myName; // "Nick";

int main(void) {
    classDemo demo("definitely_not_my_phone_num", "647-909-9090", "Nick");
    demo.
        myName                    std::string classDemo::myName
```

Classes: Accessors

```
class classDemo{
private:
    string myPassword; // "definitely_not_my_phone_num";
    string myPhoneNumber; // "647-909-9090";
public:
    string myName; // "Nick";
    string getMyPassword(); //public member function
};

string classDemo::getMyPassword(){ //define our accessor and what it does
    return myPassword;
}

int main(void) {
    classDemo demo("definitely_not_my_phone_num", "647-909-9090", "Nick");
    demo.getMyPassword();
```

Classes: Mutators

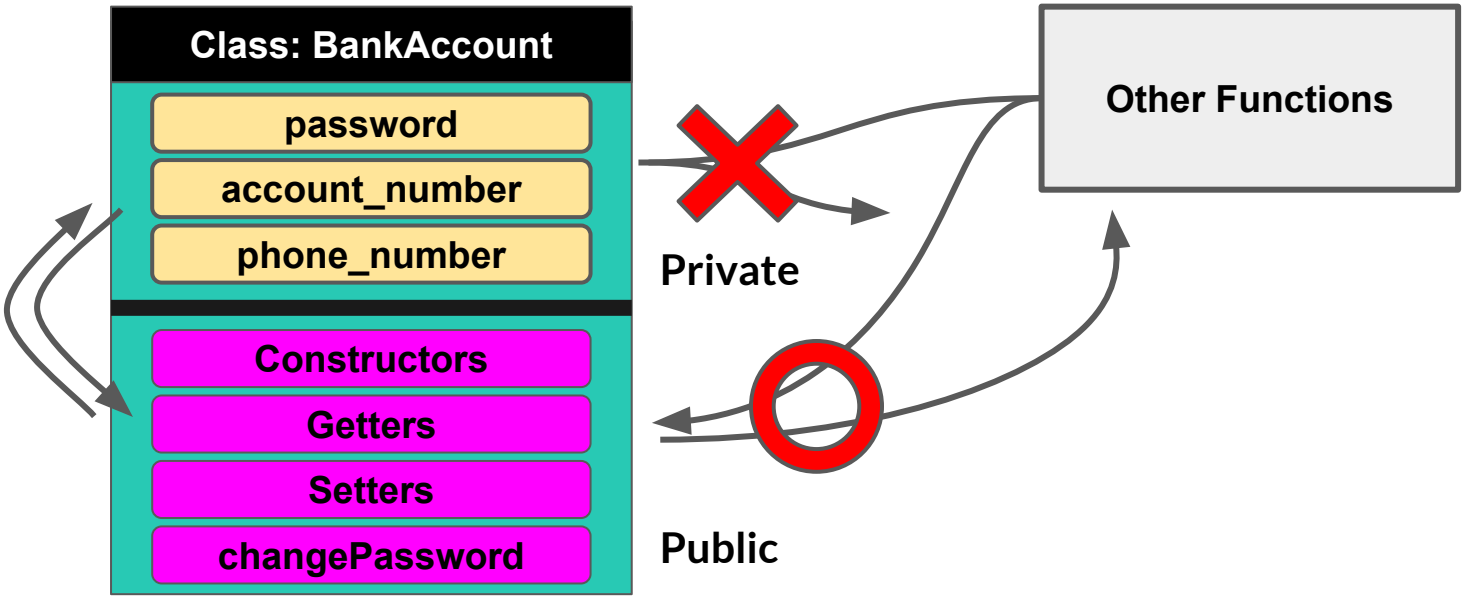
```
class classDemo{
private:
    string myPassword; // "definitely_not_my_phone_num";
    string myPhoneNumber; // "647-909-9090";
public:
    string myName; // "Nick";
    void makeNewPassword(string newPassword); //public member function
};

void classDemo::makeNewPassword(string newPassword){ //define our mutator and what it does
    myPassword = newPassword;
}

int main(void) {
    classDemo demo("definitely_not_my_phone_num", "647-909-9090", "Nick");
    demo.makeNewPassword("definitely_not_professor's_phone_num");
```

Activity - Class Encapsulation

Bank Account .cpp

main .cpp

https://codeboard.io/projects/390702



Constructors

- Special member functions whose job is to instantiate an object
- You can pass values to populate data members during instantiation (ie. declaration)

How to Identify a Constructor?

- A constructor has to:
  - Have the same name as the class
  - Have NO return value - not even void
  - Be public
- Invoked when an object is instantiated
  - Which line will cause the constructor to be invoked?

```
Time t;
Time *p;
p = new Time;
```

```
//Student.h
class Student {
    private:
        int height;
        int age;
    public:
        Student();
};
```

Constructor Code

```
//Student.h
class Student {
        private:
                int height;
                int age;
        public:
                Student();
};
```

```
//Student.c
Student::Student()
{
        height = age = 0;
        cout << “Student data initialized!” << endl;
}
```

Multiple Constructors

- Initialize objects in different ways?
  - Multiple constructors needed

- Use function overloading (supported in C++)

Function Overloading: A function with the same name, but with a different parameter list:

1. Different amount of parameters
2. Different types of parameters

*Different variable names WILL NOT WORK

Multiple Constructors

```
//Student.h
class Student {
    private:
        int height;
        int age;
    public:
        Student( ); //default constructor
        Student( int, int );
};
```

```
//Student.c
Student::Student()
{
    height = age = 0;
}
Student::Student(int _h, int _a)
{
height = _h;
age = _a;
}
```

Comparing with default constructor

Calling a default constructor

```
Time t;
Time *p;
p = new Time;
```

Calling a customized constructor

```
Time t1(10,0);
Time t2 = Time(12,30);
Time *p = new Time(12,30);
```

Destructor

- Complement of constructor
- Automatically called when an object is destroyed
- Not required to have a destructor (more on this later)

When is a destructor needed

In .h file

```
class Student{
        private:
                char *name;
        public:
                Student();
                Student(char*);
};
```

In .c file

```
Student::Student()
{
        name = NULL;
}

Student::Student (char *_name)
{
        name = new char[100];
        strcpy(name, _name);
}
```

Do you see any problem with this?

When is a destructor needed

In .h file

```
class Student{
        private:
                char *name;
        public:
                Student();
                Student(char*);
};
```

Memory leak !

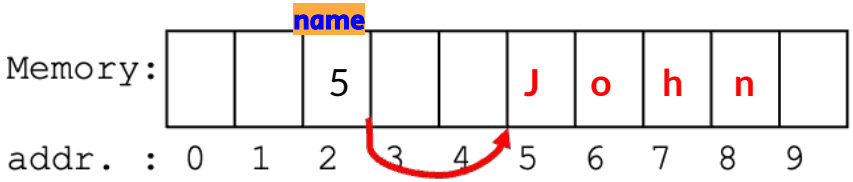
In .c file

```
Student::Student()
{
        name = NULL;
}

Student::Student (char *_name)
{
        name = new char[100];
        strcpy(name, _name);
}
```

When is a destructor needed

```
In .c file
Student::Student()
{
        name = NULL;
}

Student::Student (char *_name)
{
        name = new char[100];
        strcpy(name, _name);
}
```

- If an object is instantiated with the class student having a name “John”; then a char array is dynamically allocated. When the object is destroyed, the allocated array remains on the heap, with nothing referencing it!
- Memory leak!



How would a destructor fix the issue:

```
class Student
{
        private:
                char *name;
        public:
                Student();
                Student(char*);
                ~Student();
};
```

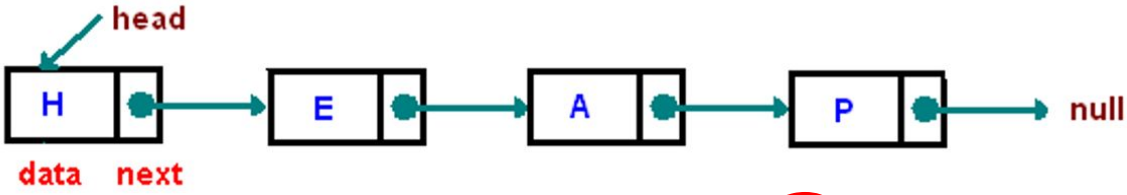
- Destructor: `~' followed by class name
- Cannot take any parameter!
- Cannot return any type!
- Must be public

How would a destructor fix the issue:

```
Student::Student (char *_name)
{
        name = new char[100];
        strcpy(name, _name);
}

Student::~Student()
{
        if (name != NULL) delete [ ] name;
}
```

- If “new” or malloc is used anywhere in your class, you likely need a destructor with a “delete”

Test your knowledge: where is destructor needed?




```
class Node {
        private:
                int data;
                Node *next;
        public:
                Node(int d, Node *n) {
                        data = d;
                        next = n;
                }
}
```

```
class List {
        private:
                Node *head;
        public:
                void insert (int d) {
                        head=new Node (d, head);
                }
}
```

```
class Node {
        private:
                int data;
                Node *next;
        public:
                Node(int d, Node *n) {
                        data = d;
                        next = n;
                }
                ~Node() {
                        if (next != NULL)
                       delete next;
                  }
}
```

```
class List {
        private:
                Node *head;
        public:
                void insert (int d) {
                        head=new Node (d, head);
                }
                ~List(){
                        if (head!=NULL)
                        delete head;
                }
}
```

Activity The Office

https://codeboard.io/projects/390733

- Define Constructors and Destructors so that 4 office employees are initialized properly

Object Oriented vs Procedural

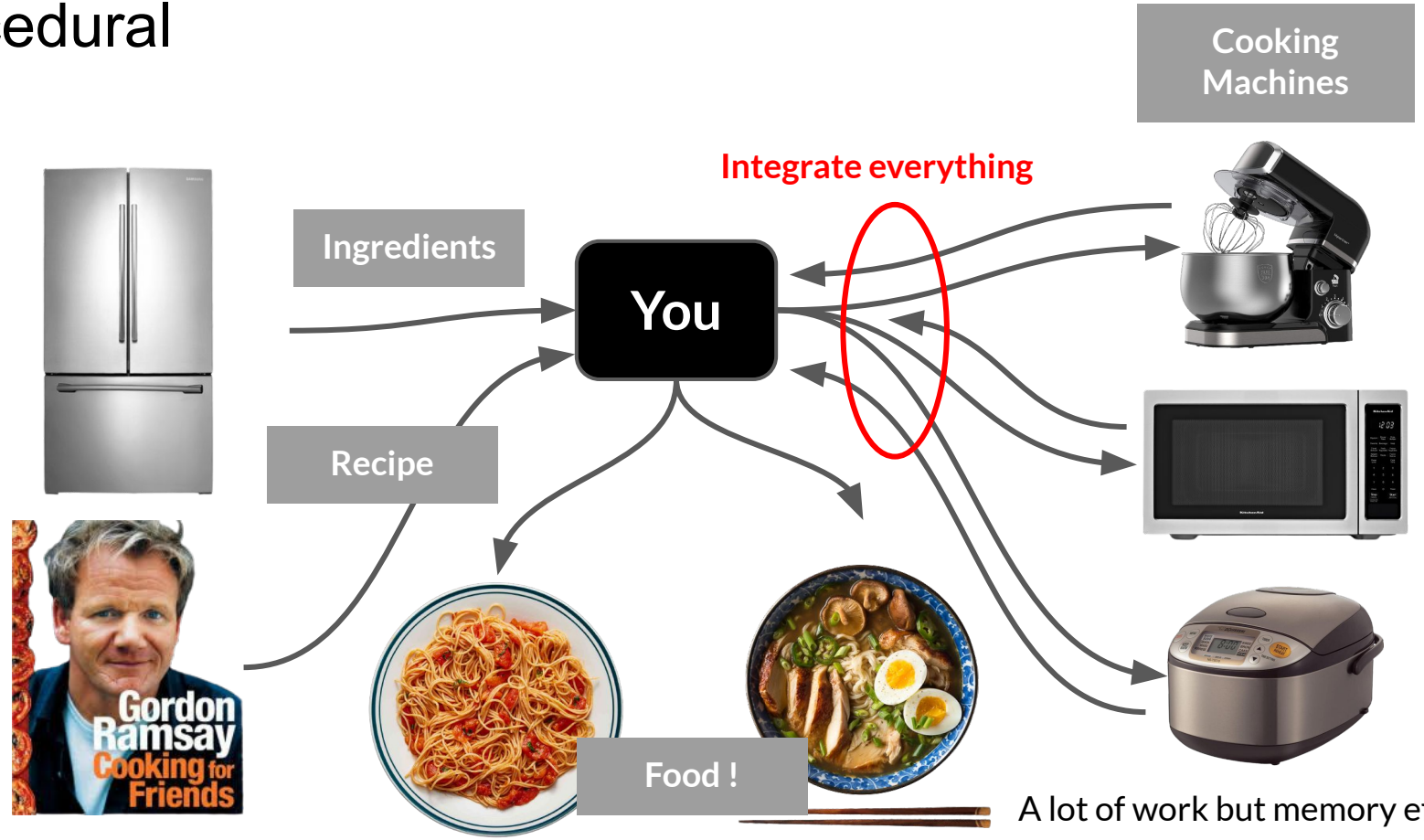
In procedural programming…

- The caller function (main.c) has to manage all the variables that are passed to the functions (It has to know the type, where it is, in what order it's supposed to pass, need to update them if necessary) and tell them what to do exactly. Like "Go to SF building at 12:00 using TTC"
- This is too much work

In object oriented programming…

- The variables are managed by the instances themselves. Everything the caller function needs to do is tell them what to do (in an abstract way, no need to specify the details). Like "Go to school".
- This is much better

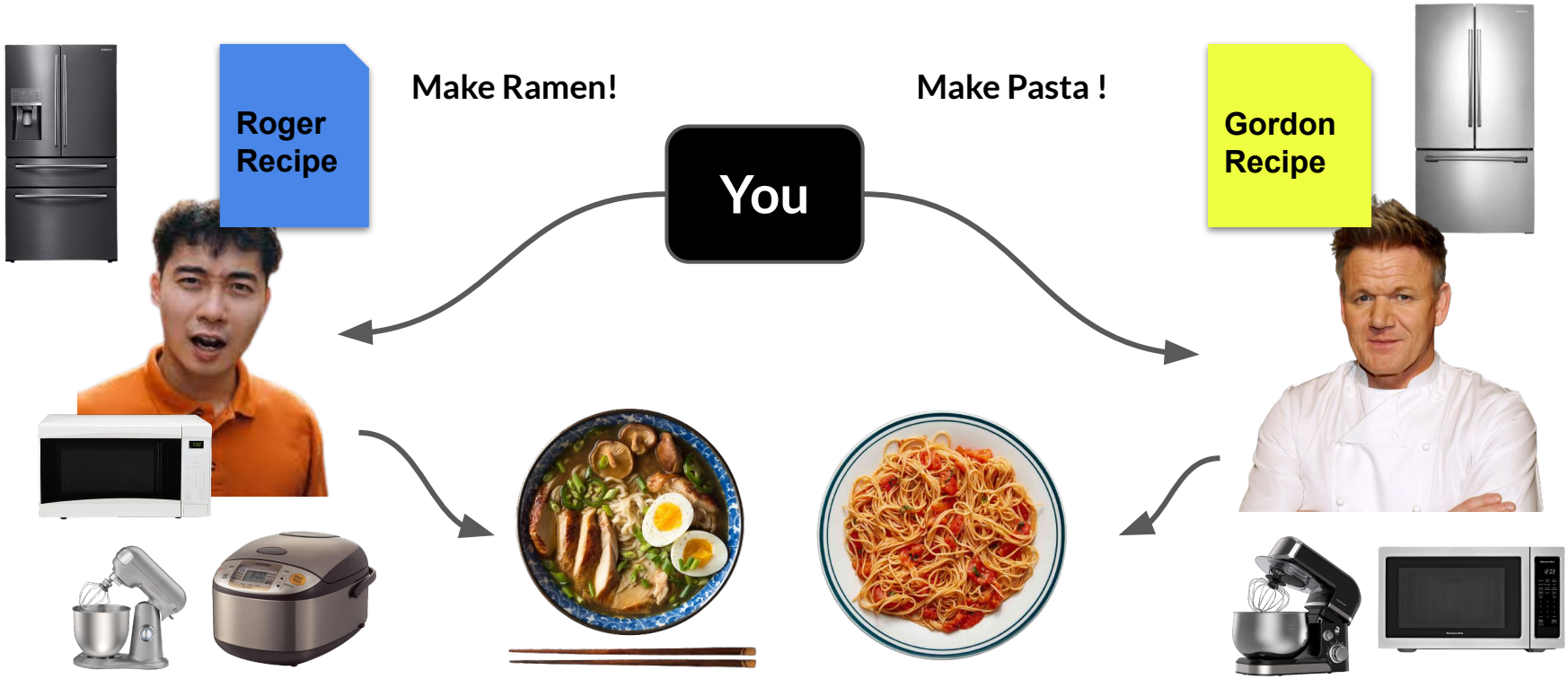
Procedural



A lot of work but memory efficient

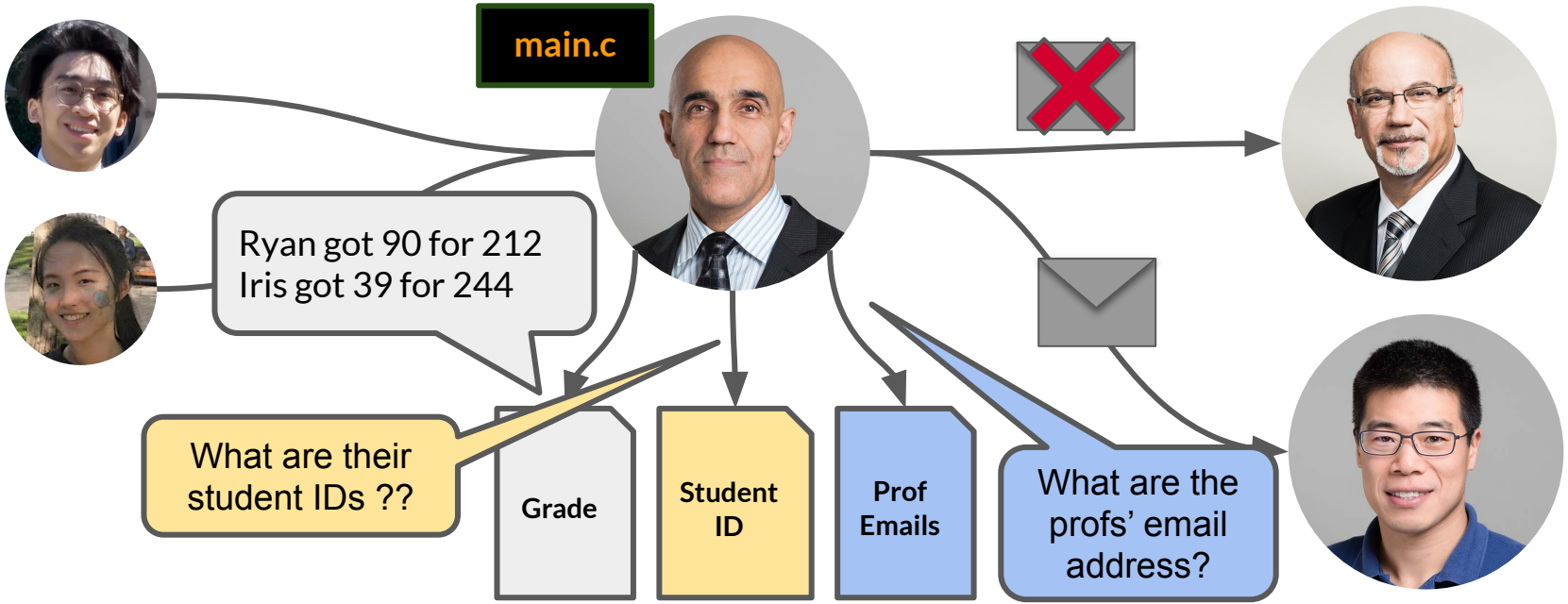
Object Oriented

Easy but uses more memory



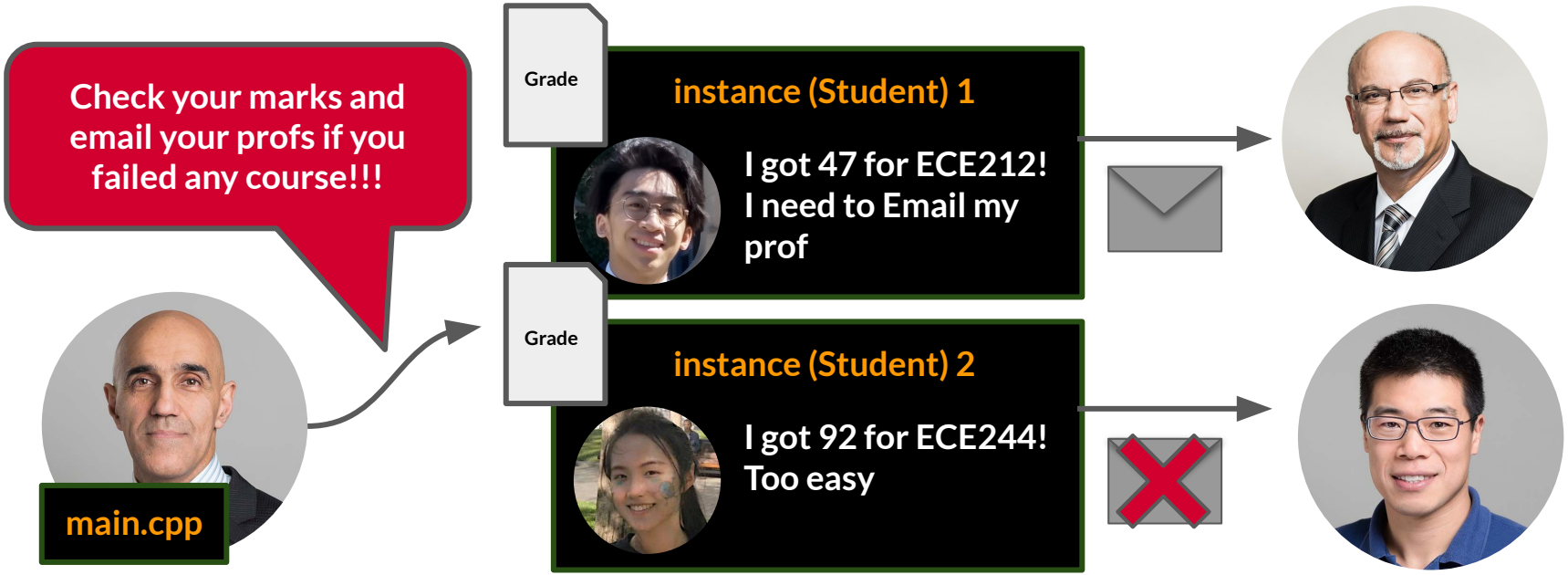
Procedural Programming Activity

https://codeboard.io/projects/390662



Object Oriented Programming Activity

https://codeboard.io/projects/390679



ECE Summer Workshop Software

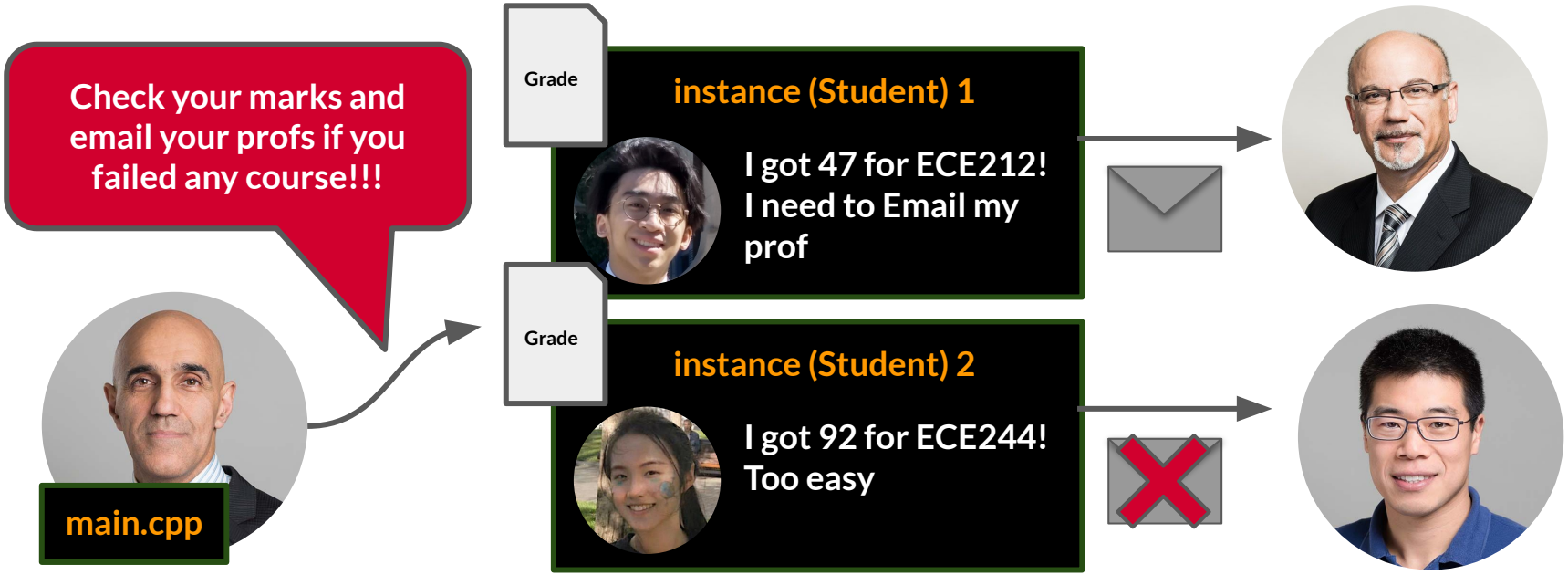
APS105 Review & ECE244 Preparation
Day 4

Day 4 Content

- Finish OOP Activity from Yesterday
- Overloaded Operators and Copy Constructors
- STL Library
- Inheritance

Object Oriented Programming Activity

https://codeboard.io/projects/390679



Review of Overloaded Functions

- Functions can have the same name if they have different parameter lists, which means:

  a. Different parameter names ✗

  b. Different parameter types ✓

  c. Different number of parameters ✓

Operators

- Recall operators include anything such as:
  - Equals (=), Less than (<), Greater than (>)
  - Add (+), Subtract (-), Multiply (*), Divide (/), Modulus (%)
  - Arithmetic Assignment (+=, -=, *=, /=, %=)
- **Are these operators functions?**

Recall For Strings…

```
string string1 = "something";
string string2 = "somethingElse";
if (string2 > string1) {
    string2 = string1;
}
```

Classes: Copy Constructor

- There's one more type of constructor in C++ called a copy constructor
- Used to copy the data members of one object to another
- **Note:** this is equivalent to overloading the (=) operator

```
int main() {
  demoClass test1(5);
  demoClass test2(10);
  test2 = test1;
```

Does this work?

Classes: Copy Constructors

- Usually when you are copying an object of the class, a default copying constructor will be generated and used.
- But that does not always give you the result you wanted!
- What happens if there is a pointer in one of the variables of the classes?
    - Recall: Deep copy and shallow copy!

Classes: Copy Constructors

- Imagine you are copying your friend Joe's work for your Quiz (you shouldn't do that at the first place!).
- Your friend's answer vs your answer

Author: Joe
In conclusion, Simon will be kicked out of the school for being a failure. For detailed explanations, refer to page 5 of Joe's paper.

Author: you
In conclusion, Simon will be kicked out of the school for being a failure. For detailed explanations, refer to page 5 of Joe's paper.

- If you do that, you will join me and get kicked out of the school.

Classes: Copy Constructors

- Therefore, you will need to write you own copy constructor, manually create a deep copy for the pointer.
- Syntax:

```
class myclass {
public:
    myclass(const myclass & randomName) {
```

Example

```
class student {
private:
    int gradeForECE221;
    int* pointerToGradeForECE231;
public:
    student(const student &source) {
        gradeForECE221 = source.gradeForECE221;
        int* deepCopyPointer = new int;
        *deepCopyPointer = *pointerToGradeForECE231;
        pointerToGradeForECE231 = deepCopyPointer;
    }
};
```

- Problems?
- You have allocated new memory on the heap…
- You also need to write you own destructor to ensure there is no memory leaks!

Example

```
class student {
private:
    int gradeForECE221;
    int* pointerToGradeForECE231;
public:
    student(const student &source) {
        gradeForECE221 = source.gradeForECE221;
        int* deepCopyPointer = new int;
        *deepCopyPointer = *pointerToGradeForECE231;
        pointerToGradeForECE231 = deepCopyPointer;
    }
    ~student(){
        delete pointerToGradeForECE231;
    }
};
```

- Any more problems?
- Copy constructor is only used when you declare and initiate at the same time (student b = a;)
- It won't be called if you do something like:

  student b;
  b = a;

Example

```
void operator=(const student &source){
    gradeForECE221 = source.gradeForECE221;
    int* deepCopyPointer = new int;
    *deepCopyPointer = *pointerToGradeForECE231;
    pointerToGradeForECE231 = deepCopyPointer;
}
```

- Just the same as the copy constructor that we just declared and defined.
- We are returning void, is there a problem with that?
- What if we are doing c = b = a?

Utilize Operator Overloading on ‘=’

- To account for the situation mentioned previously, we have to overload the operator “=” in order for it to work the way it is intended to.
- Although we are only covering overloading the “=” operator, you can overload most of the operators for the class you want, but you can do it only within the class.
- It could also be helpful with some of the algorithm in the STL (covered later), for example, if you have overloaded the “<” operator, you can use the function “sort” to sort a vector of objects of the class with the “<” operator overloaded, in the way you define it to be.

Example

- Simple, easy, efficient.

```
//Utility
bool Street::operator < (const Street& rhs) { //Overloaded < operator
    return streetNameRevised < rhs.streetNameRevised;
}
```

```
std::sort(cityStreetsSorted.begin(),cityStreetsSorted.end());
```

Operator Overloading: this

- “this” is a keyword that specifically refers to a pointer to the object that uses this pointer.
- It is used in member functions to refer to the object itself, which has its use under a lot of circumstances.
- In the previous example, if we want c to actually copy b, we will need the the operator “=” to return b.
- How do you address b itself in a member function of b?

Example

```
student& operator=(const student &source){
    gradeForECE221 = source.gradeForECE221;
    int* deepCopyPointer = new int;
    *deepCopyPointer = *pointerToGradeForECE231;
    pointerToGradeForECE231 = deepCopyPointer;
    return *this;
}
```

- Returning by reference
- Dereferencing “this” pointer for it to return to the actual object it points to.
- What happens if it does not return by reference?

Rule of Three

If a class requires one of the following three:

- User-defined destructor
- User-defined copy constructor
- User-defined assignment operator

It almost certainly requires all three.

Example Code

https://codeboard.io/projects/390783

- **This is not an activity**, just a link to code for overloaded operators and copy constructors for you to play around

Activity

- You are having a good time coding in SF1013 at Saturday 4am, haven’t left the building in the past 12 hours. Now you are about to finish, with only one class definition left to do.
- You suddenly realize that you need to manually define your copy constructor for this class.
- What else is there that you need to do before you can go home and have a nice and long sleep?

Standard Template Library (STL)

STL is a library that contains **template classes** and **template functions**

It includes

- **Containers**: vector, list, queue, map
- **Algorithms**: find, sort, copy, transform
- **Iterators**

Using the **template parameter,** STL provides a single generic function that can be used with many different data types

Template

A feature of C++ that allows functions and classes to work with **any given data types.**

Example: You want to implement the sum function below for both (int) inputs and (double) inputs. What would you do?

```
int sum(int x, int y){
    return x + y;
}

double sum(double x, double y){
    return x + y;
}
```

you until yesterday

```
template<typename T>
T sum (T a, T b){
    return a + b;
}
```

you starting from today

Quick Activity: https://codeboard.io/projects/390754

Template Containers

1. **Vector:** Dynamic Array
2. **Set:** Dynamic Array with Sorted Numbers
3. **Forward List:** Singly Linked List
4. **List:** Doubly Linked List
5. **Queue:** Linked List with FIFO
6. **Stack:** Linked List with LIFO
7. **Priority Queue:** Linked List with Sorted Numbers
8. **Maps:** Binary Search Tree
9. **Unordered Maps:** Hash Table

Template Algorithms

1. Find
2. Sort
3. Copy
4. Transform

```
> sh -c make -s
> ./main
10 9 8 7 6 5 4 3 2 1
1 2 3 4 5 6 7 8 9 10
>
```

```
int main() {
  std::vector<int> myVector;
  makeVector(myVector); // helper function to give vector 10 -> 1 values
  printVector(myVector); // helper function to print vector

  sort(myVector.begin(), myVector.end()); // STL sort algorithm

  printVector(myVector); // helper function to print vector

}
```

Vectors & Iterators

Vector [std::vector]

- Vectors are sequence containers whose **size can change dynamically**
- To use vectors you need to `#include <vector>`

Iterator

- Any object that is pointing to an element of a container and can be **incremented**
- A pointer is a form of an iterator (a random access iterator)
  - **Forward iterator**: can be LEH and RHS of an assignment (can be **read** & **written**)
  - **Bidirectional iterator**: can be **decremented** as well
  - **Random Access iterator**: supports **arithmetic** and **offset dereference**

Vector

- Declaration

```
std::vector<int> my_vec;
```

- Initialization

```
std::vector<int> my_vec(5, 20); //Initialize with 5 200s
std::vector<int> my_vec(5); //Initialize with 5 0s
```

- Traversing

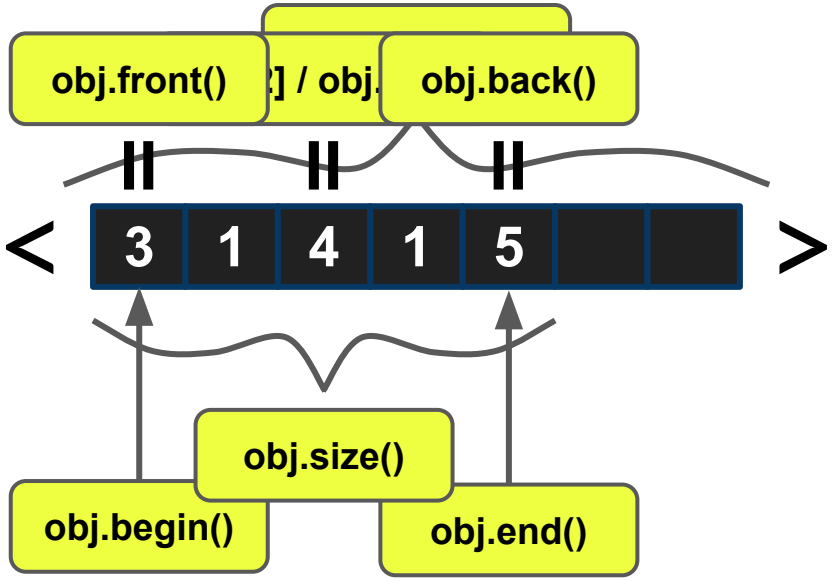
```
for(std::vector<int>::iterator itr = my_vec.begin(); itr != my_vec.end(); ++itr){
    cout << *itr << endl;
}
for(auto itr = my_vec.begin(); itr != my_vec.end(); ++itr){
    cout << *itr << endl;
}
```

auto: type will be automatically deducted from the initializer

std::vector - member functions -

Some of the important member functions of **std::vector** class are…

- obj.**begin()**: iterator to beginning
- obj.**end()**: iterator to end
- obj.**size()**: number of elements
- obj.**capacity()**: allocated storage
- obj**[i]:** i-th element
- obj.**at(i)**: i-th element
- obj.**front()**: the first element
- obj.**back()**: the last element



std::vector<int> obj

std::vector - member functions -

- obj.**push_back():** adds a new element at the end
- obj.**emplace_back():** adds a new element at the end (faster)
- obj.**resize():** changes the size (initializes with the 2nd argument)
- obj.**reserve():** changes the capacity
- obj.**shrink_to_fit():** reduces its capacity to fit its size
- obj.**clear():** removes all the elements(size = 0). Capacity might not be 0

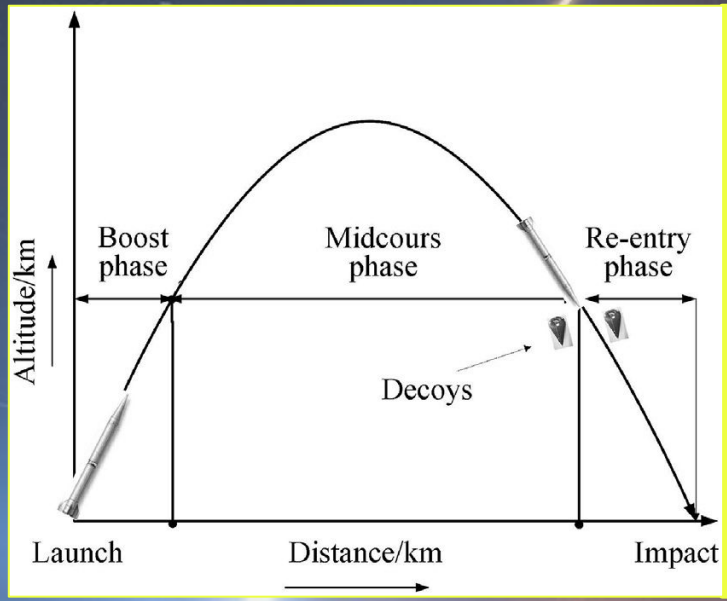
```
obj.reserve(6); t();
```

std::vector<int> obj

5

Activity -Rocket Trajectory Calculation-

https://codeboard.io/projects/390769



Equation of Motion (Constant acce

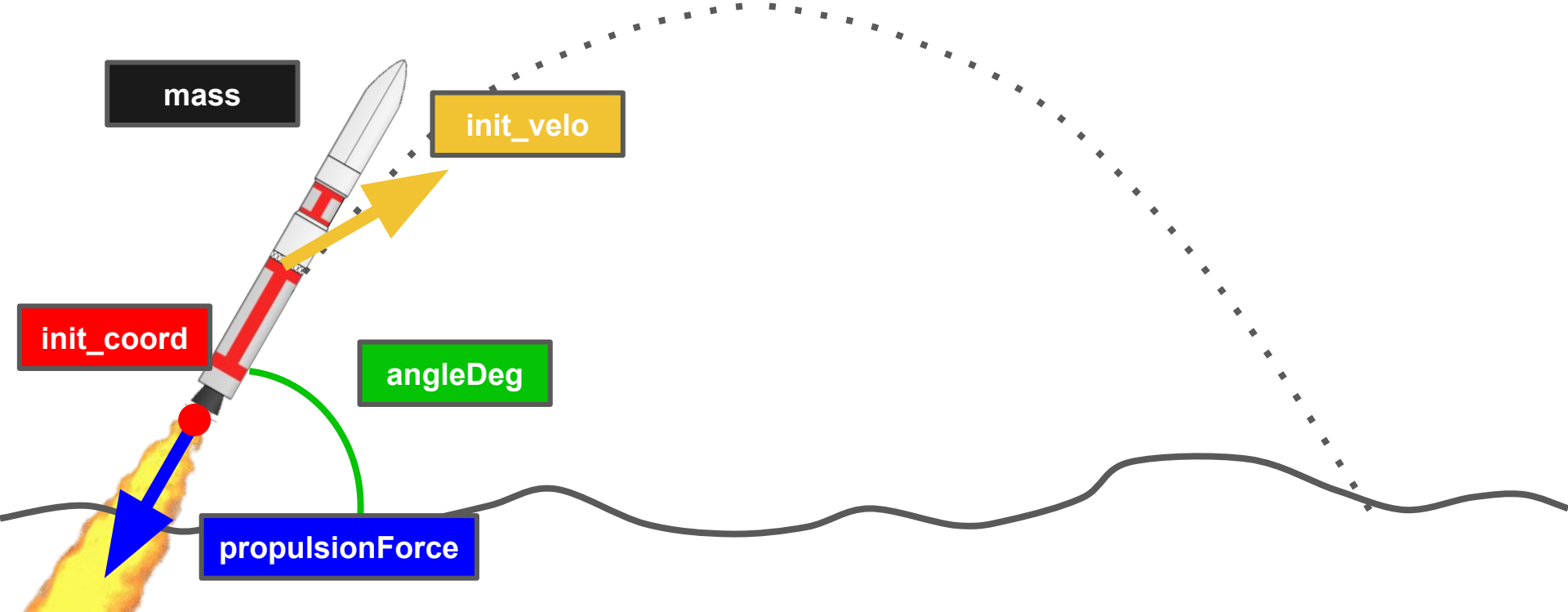
$$x(t + \Delta t) = x(t) + v_x(t)\Delta t + \frac{1}{2}a_x(\Delta t)^2$$

$$y(t + \Delta t) = y(t) + v_y(t)\Delta t + \frac{1}{2}(a_y - g)(\Delta t)^2$$

$$v_x(t + \Delta t) = v_x(t) + a_x\Delta t$$

$$v_y(t + \Delta t) = v_y(t) + (a_y - g)\Delta t$$

Activity -Rocket Trajectory Calculation-



inline function

Similar to **Function-like macros** in C

It looks identical to normal functions except for the keyword “**inline**”

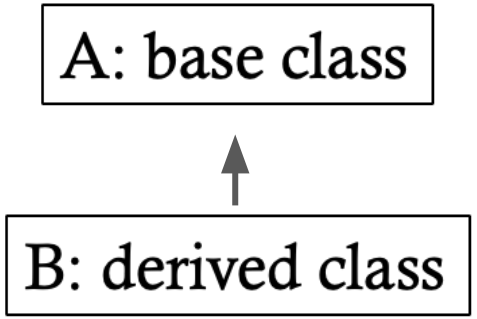
When an inline function is called, the definition of it will be “copy-pasted” there

```c
inline int sum(int a, int b){
    a = a * 2;
    b = b * 3;
    return a + b;
}

int main() {
    int answer = sum(2, 3);
}
```

answer = 13

What is Inheritance?

- A powerful mechanism to allow a class to “inherit” the content of another class
- There are two types of classes:
  - Base class
  - Derived class
- They share a lot of properties, yet the derived class has its own properties



What is Inheritance?

- Edible

Food

- Edible
- Supplies vitamins

Fruit

- Edible
- Supplies vitamins
- Sweet
- Crunchy

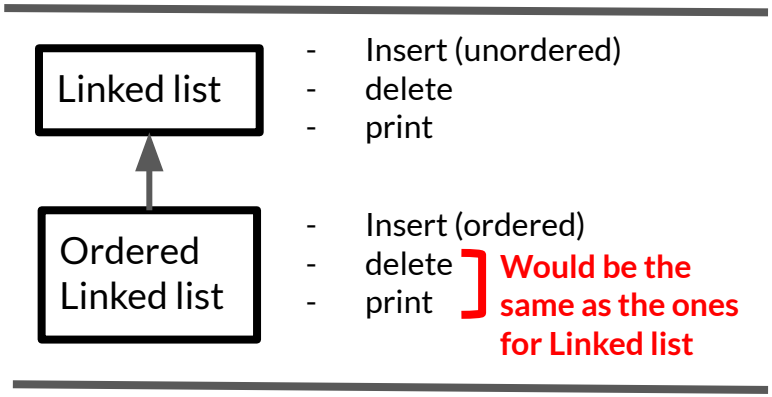
Apple

What is Inheritance?

- A derived class is always larger than the base class (meaning having more function members)
- Derived class receives all of the member variables and functions of the base class (usually)
- Therefore in derived class, we can directly use the members of the base class
- Can also extend/customize/override as needed

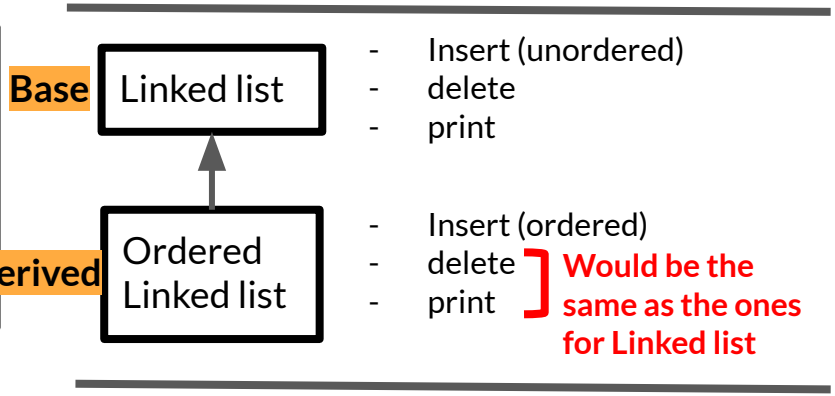
Key benefit and motivation of using inheritance

- You want to reuse code (or features) from other classes
- Example:
  - If you already implemented linked list, now you want to implement another class: ordered linked list
  - Can we reuse code?



Syntax of Inheritance

```
class Derived_Class : public Base_Class
{
    Only list the inherited members if you wish to
    change them, plus new members you're
    adding.
}
```




```
class Ordered_LinkedList :  public LinkedList
{
        void insert (int v);
}
```

- Redefine insert()
- value, next, delete(), print()… are all inherited, and you can use them as if they were defined in Ordered_LinkedList

```
class Ordered_LinkedList : public LinkedList
{
        void insert (int v);
}
```

- Once derived class is declared, you can use it just like any other class
- For example:
  - void Ordered_LinkedList::insert(int v) { … }
  - Ordered_LinkedList *list_1 = new Ordered_LinkedList;

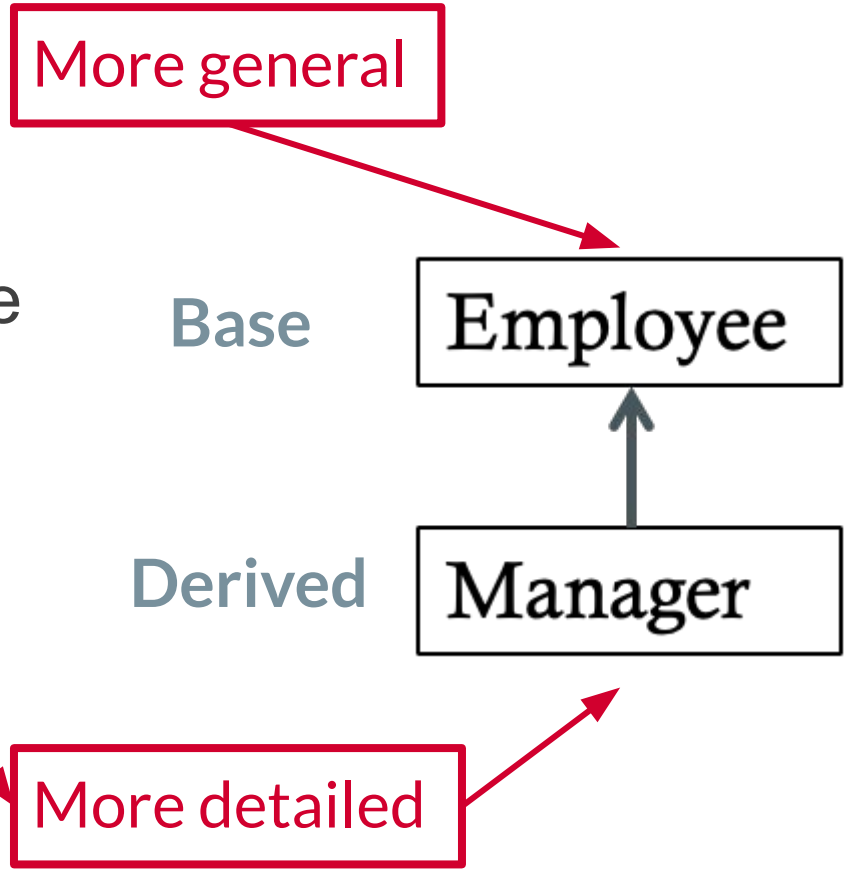
Note! A Manager is an Employee, but an Employee is not a Manager.

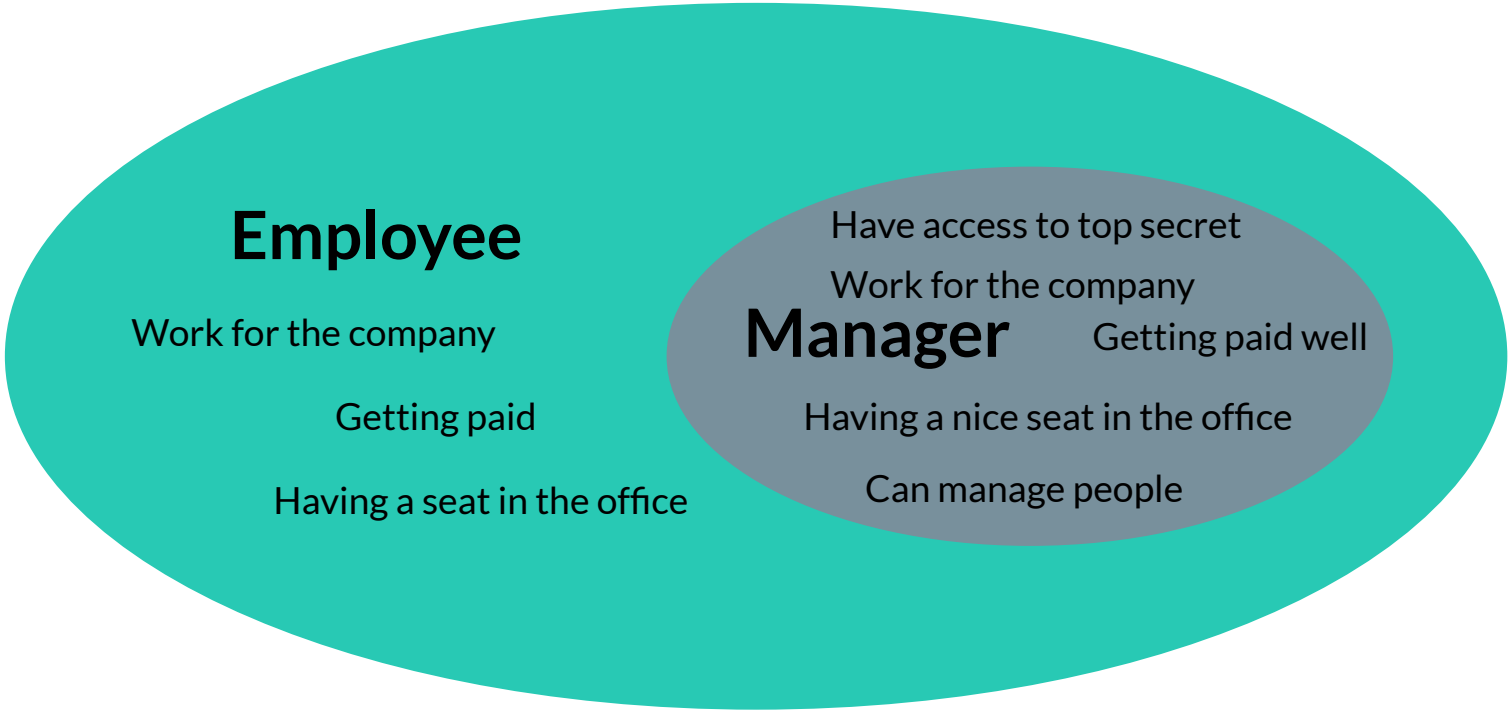
```
void g (Manager m1, Employee e1)
{
    Employee *Emp = &m1; // OK
    Manager *Man = &e1; // Not OK!
}
```

WHY?

Recall:

- A derived class is always larger than the base class




Employee
Work for the company
Getting paid
Having a seat in the office
Have access to top secret
Work for the company
Manager
Getting paid well
Having a nice seat in the office
Can manage people

A Manager is an Employee, but an Employee is not a Manager.

- This can be very useful!
- For example, you can implement a linked list of Employees, and insert a manager – no problem at all

```
void companyList(Manager m1, Employee e1)
{
  vector<Employee> vec;
  vec.push_back(e1);
  vec.push_back(m1);
  cout << “Size: “ << vec.size();
}
```

Data Protection

- Derived classes CANNOT access private members of base class
  - But these private members are inherited!
  - i.e., in this example, when you instantiate an object of Derived class, there will be memory allocated for “a”

```
class Base {
private:
  int a;
  .. ..
};
class Derived : public Base {
private:
..
};
```

Protected Members

- What if we want a member variable/function to be accessible to derived class, but inaccessible to outside?
- C++ provides another access specifier: **protected**, to replace private

Protected Member

```
class Base {
private:
protected:
  int a;
  .. ..
};
class Derived : public Base {
  // Can access “a” as a member variable.
};
```

Access	public	protected	private
members of the same class	yes	yes	yes
members of derived class	yes	yes	no
not members	yes	no	no

Test your knowledge:

- ✓ Derived class inherits all member variables from base class
- ✗ Derived class can access private members of base class
- ✓ Derived class can access protected members of base class
- ✓ Derived class can add member vars and functions
- ✗ Derived class can remove member vars and functions

Inheritance example: Accessing parent members

```
 5 class Base {
 6 private:
 7   int a;
 8 public:
 9   void set(int _a) { a = _a; }
10   void print() { cout << "Base.a = " << a << endl; }
11 };
12
13 class Derived : public Base {
14 private:
15   int b;
16 public:
17   void set(int _a, int _b) {
18     Base::set(_a);
19     b = _b;
20   }
21   void print() {
22     Base::print();
23     cout << "Derived.b = " << b << endl;
24   }
25 };
```

```
27 int main() {
28   Derived b;
29   b.set(1, 2);
30   b.print();
31   return 0;
32 }
```

Constructor in inheritance

- Customized constructors are NOT inherited
- For example, “ Base::Base(int) “ won’t be invoked by “ Derived D1(3); ” if “ Derived::Derived(int) ” is not defined.
- If none of the base constructors (customized) is explicitly called, then the default base constructor would be used when initializing a derived class object

What do we do we need a base customized/non-default constructor?

- You can explicitly call (non-default) base constructor using the following syntax:

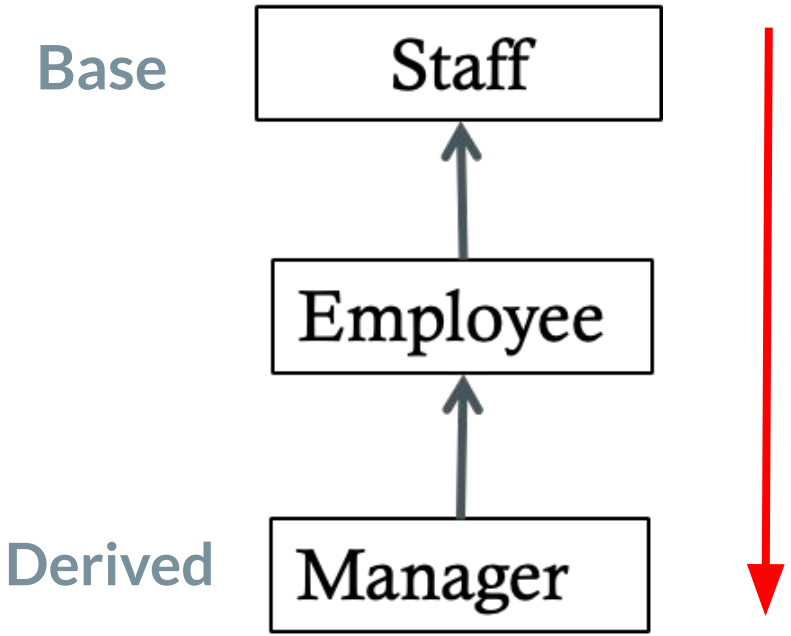
Customized constructor of the derived class

Customized constructor of the base class

```
Derived::Derived(..) : Base(..) {
        …
}
```

Order of Constructor Invocations in Inheritance

- In this case, Staff() being invoked first, then Employee(), then Manager()

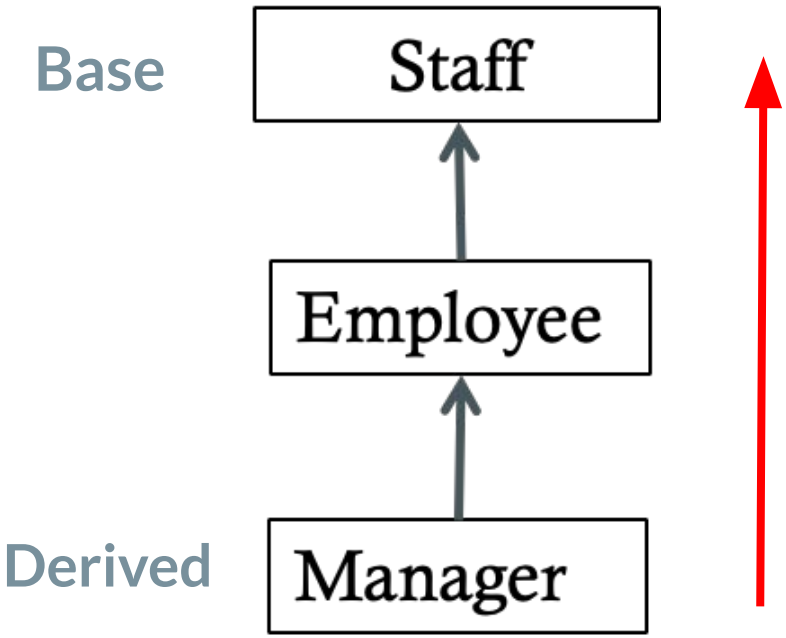


Note! For Constructors

- When there is a user declared constructor of any type, there is no compiler-generated default constructor

Destructor in Inheritance

- Destructor of the base class is called automatically
- In this case, the order of destructor invocation: ~Manager(), ~Employee(), ~Staff()



Test Your Knowledge:

```
class Base {
public:
  Base (int) { .. }
};

class Derived {

}

Derived p (10);
```

Is this correct?

Test Your Knowledge:

```
class Base {
public:
  Base (int) { .. }
};

class Derived {
/* No Derived(int) */
}

Derived p (10);
```

Compile Error!

Correction:

Derived p;

Activity

- Now you are the ECE216 professor and you have encountered 4 students, and you want to determine whether they are geniuses or failures, and you are going to have the failures be the slaves for you. You want to do it through the use of inheritance, because you don't want others say you are discriminating against those who do bad in ECE216, so you will try to pretend to treat them all as students.
- https://codeboard.io/projects/390747
- Solution: https://codeboard.io/projects/390748/

Culminating Demo

- Let’s say we want to build a software that can model 3D shapes to interact with
- App Functionality:
  - We can construct shapes
  - Move shapes around

- Learning Objectives:
  - See the value of classes in building software
  - See how quickly an application’s infrastructure can grow

Representing 3D Shapes in Code

1. What **objects** (ie. 3D shapes) do we want to model?

Types of Objects

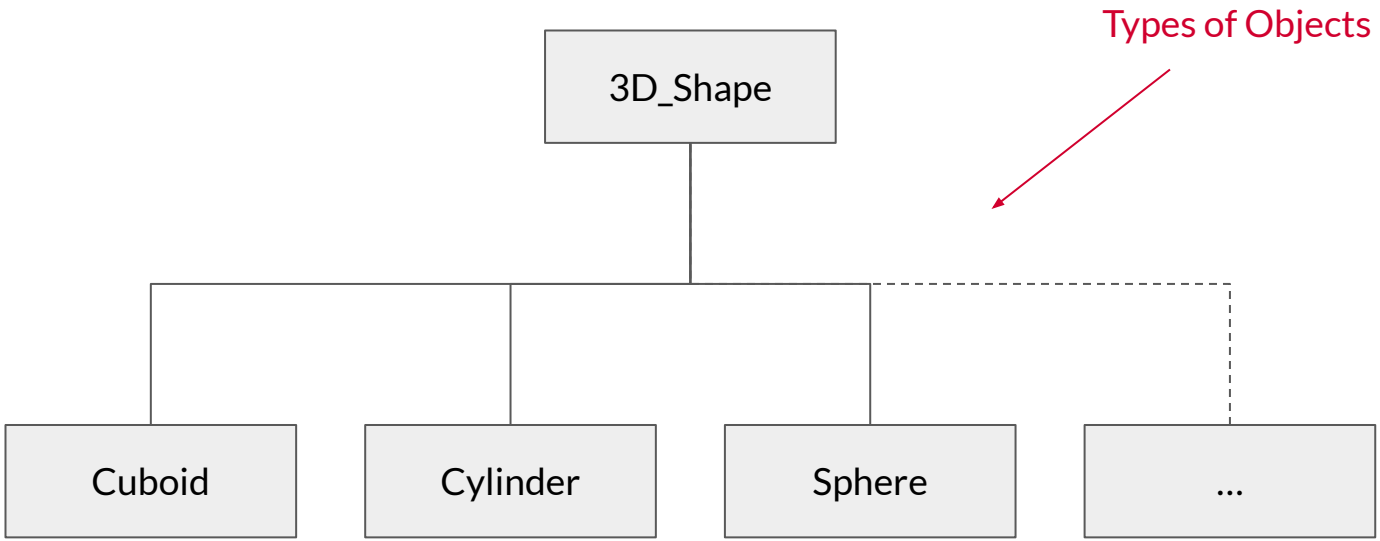
2. What **attributes** do these volumes have?

Types of Data Members

3. What **behaviors** do these volumes have?

Types of Member Functions

Modelling 3D Shapes



Representing 3D Shapes in Code

1. What **objects** (ie. 3D shapes) do we want to model?
   a. Cubes/Rectangular Prisms (AKA cuboid)
   b. Cylinders
   c. Spheres
   d. Etc.

   Types of Objects

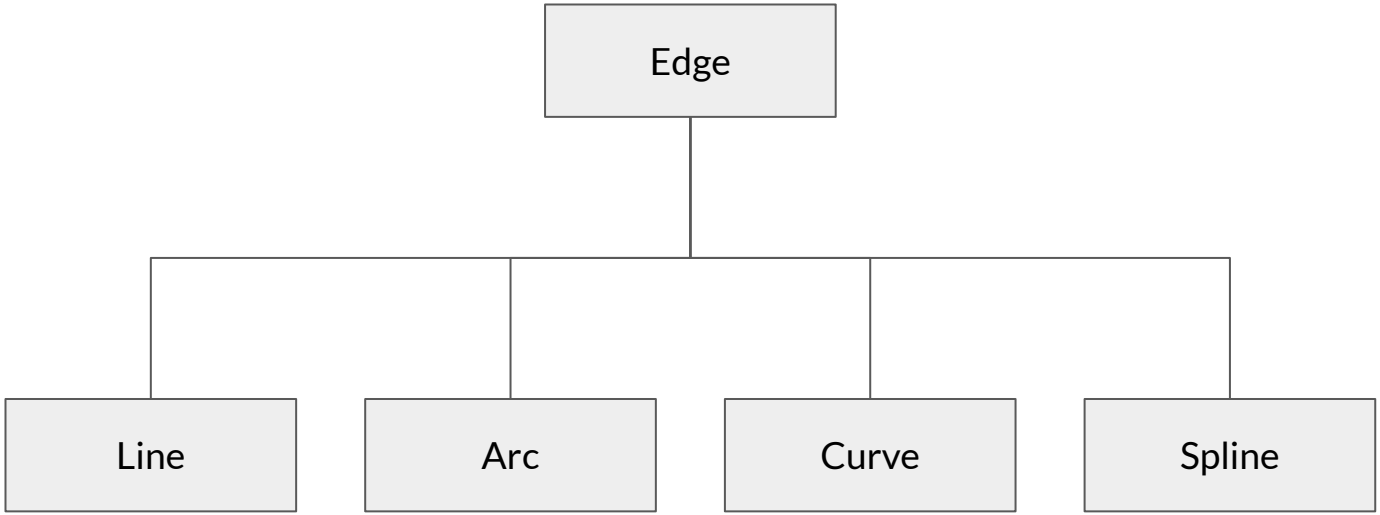
2. What **attributes** do these volumes have?
   a. Edges → ?
   b. Faces
   c. Volume → double
   d. Etc.

   Types of Data Members

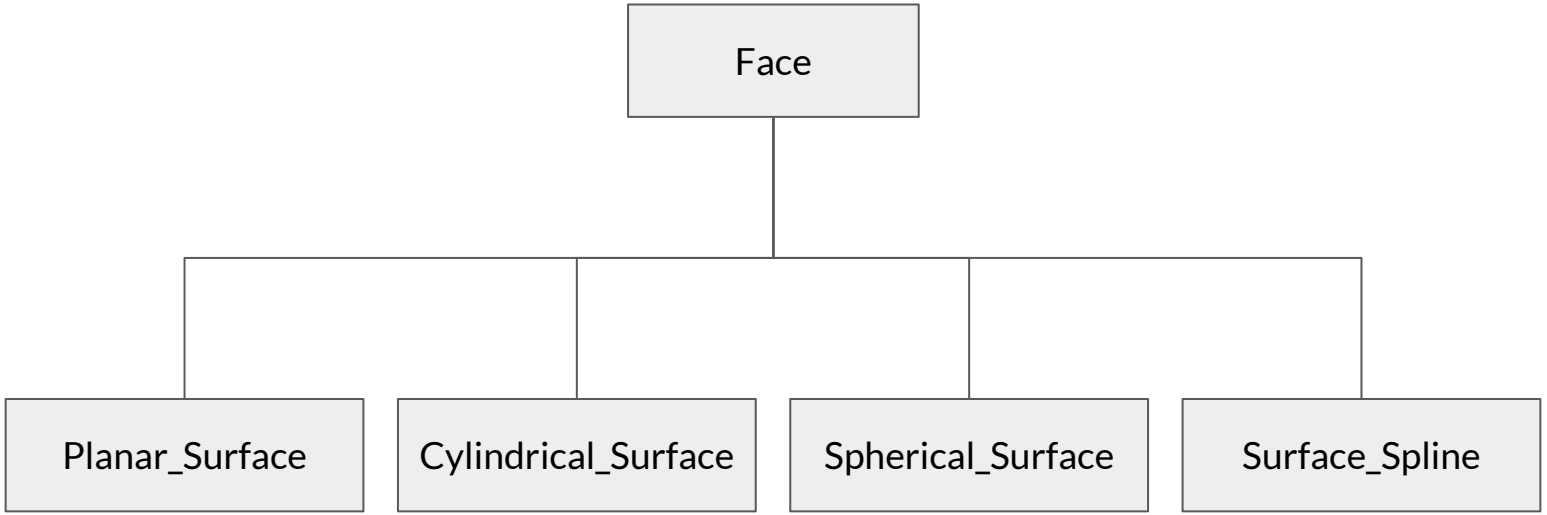
3. What **behaviors** do these volumes have?
   a. Translate
   b. Rotate
   c. Scale
   d. Etc.

   Types of Member Functions

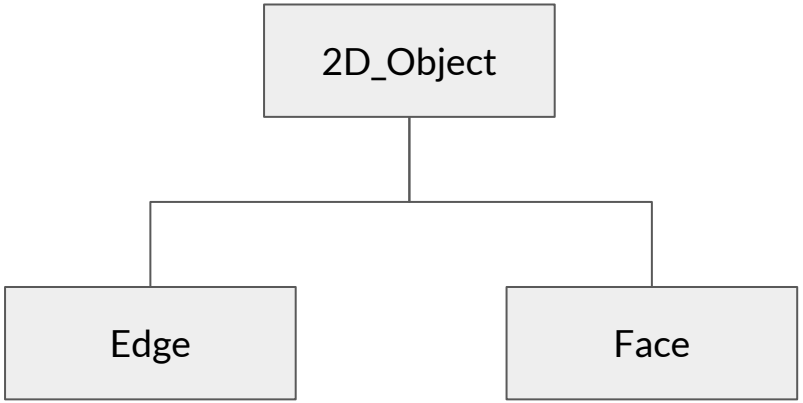
Modelling Edges



Modelling Faces



Modelling 2D Shapes



Is All Of This Necessary?

Short Answer → YES

Long Answer:

- Isolated Testing
  - The **modularity** allows each class to be tested individually and help debugging
- Organizes Files
  - All types of *Edge* classes can be in one file, all the *Face* classes can be in another, etc.

Conclusion

- Object oriented program allows you to effectively design the infrastructure of an application
- **Note:** This is just to model 3D objects in code, we haven't even discussed how to display these objects or how the user can interact with them
  - Even though the back-end model is so large, the app just gets larger

- This is what differentiates software engineering from computer science
  - Software engineering involves design
  - Computer science involves implementing said design

Workshop feedback form

No stepping out of the room until this is completed