Comp. Prog.

Lec 4

Previously

- Recursive programs
  - factorial | fibonacci | n choose k | permutations
- Caching (Memoization/Dynamic Programing)
- Typedef
- Representation/Memory management for Perm
  - Creation/Destruction
- GDB
  - breakpoints | watch expressions | call stack | step over/in/out
- Recursive Drawing
  - recursive circles

Sorting

- Input: an array of integers
- Ouptut: array sorted in increasing order

Pseudo Code: Merge Sort

- Base Case: If array of size 1 then return.
- Otherwise
  - Divide the array into two halfs
  - Copy the halfs into arrays L, R
  - Recursively sort L and R seperately
  - Merge L, R such that the result is sorted

Merge Function

- Input: two arrays L, R that are sorted (seperately)
- Output: an array with size = size(L) + size(R) containing their elements and is sorted
- Example: Merge({1,3,5}, {2,4,6}) should return {1,2,3,4,5,6}

Pseudo Code for Merge

```
t = size(L) + size(R)
A = array of size t
c = 0, l = 0, r = 0;
while(c < t):
    if l reached size(L):
        copy remaining elements from R to A
    if r reached size(R):
        copy remaining elements from L to A
    if L[l] < R[r]:
        A[c++] = L[l++]
    else:
        A[c++] = R[r++]
```

Code for Merge

```
void merge(int *L, int sL, int* R,
           int sR, int *A) {
    int l = 0, r = 0, c = 0;
    while(c <= sL + sR -1) {
        if (r == sR - 1 ) {
            A[c++] = L[l++];
            continue;
        }
        if (l == sL - 1) {
            A[c++] = R[r++];
            continue;
        }
        if (L[l] < R[r]) {
            A[c++] = L[l++];
        } else if (L[l] >= R[r]) {
            A[c++] = R[r++];
        }
    }
}
```

Code for Merge Sort

```
void copy_array(int *A, int start,
                int end, int *B) {
    for(int i = start; i <= end; i++) {
        B[i-start] = A[i];
    }
}
void sort(int *A, int len) {
    if (len == 1) {
        return;
    } else {
        int mid = len/2;
        int L [ mid], R [len - mid];
        copy_array(A, 0, mid, L);
        copy_array(A, mid, len, R);
        sort(L, mid);
        sort(R,len - mid);
        merge(L, mid, R, len-mid, A);
    }
}
```

Doesnt seem to work

Debug and fix!

Yet another recursive algo

For sorting an array A from `start` to `end`

- Base case: `start == end` then nothing to be done.
- Find the index `i` of the smallest element in A from `start` to `end`
- swap the values of `A[start]` and `A[i]`
- Sort `A` from `start + 1` to `end`

HW: Implement it.

Why should you pass pointers instead of arrays?

Why should you pass pointers instead of arrays?

- C passes arguments **by value**.
- Every time an array is passed, entire element is copied.
- Also modifications done to the Array will not be saved.

Passing pointers

- Passing pointers, only results in copying of the pointer(ie address of the first element of the array).
- Also modifications will remain, as we are changing the actual memory location.

Now Some Drawing

Allegro Library (simplified)

```
#include <allegro5/allegro5.h>
#include <allegro5/allegro_primitives.h>

ALLEGRO_EVENT_QUEUE* queue = NULL;
ALLEGRO_DISPLAY* disp = NULL;

// Checks a condition `test`.
// If false, prints out a `description` and exits
void must_init(bool test, const char *description)

// Initialization logic of allegro library
// creates queue or disp
void allegro_init()

// frees all the memory allocated by allegro
void allegro_close()
```

Allegro Library (simplified)

```
int main() {

    allegro_init();

    ALLEGRO_EVENT event;

    draw();

    // wait for some event to enter the queue.
    // In this case, it is only the event
    // for closing the window
    al_wait_for_event(queue, &event);
    allegro_close();

    return 0;
}
```

Multifile Code Organization

Problem:

- We want to write many program that use Allegro (1_learn_draw.c, 2_rec_drawing.c).
- Then the functions defined in slide 3 has to be repeated in both the files.

Solution:

Seperate the common code into `allegro.h` and `allegro.c`

Step 1: Declare in `allegro.h`

Write a `allegro.h` that

- includes the allegro, stdio, math header files
- declares variables `disp`, `queue` (as extern; but do not initialize).
- declares the functions (also called prototypes)

```
#include <allegro5/allegro5.h>
#include <allegro5/allegro_primitives.h>
#include <math.h>
#include <stdio.h>
// These are initialized in allegro.c
extern ALLEGRO_EVENT_QUEUE* queue;
extern ALLEGRO_DISPLAY* disp;
// Functions bellow are defined in allegro.c
void must_init(bool test, const char *description);
void allegro_init();
void allegro_close();
```

Step 2: Define in `allegro.c`

```
#include "allegro.h"
ALLEGRO_EVENT_QUEUE* queue = NULL;
ALLEGRO_DISPLAY* disp = NULL;

void must_init(bool test, const char *description) {
    ...
}
void allegro_init() {
    ...
}

void allegro_close() {
    ...
}
```

Step 3: Include `allegro.h`

- Include `allegro.h` in both `1_learn_draw.c` and `2_rec_drawing.c`.

```
#include "allegro.h"
void draw() {
...
}

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

Step 4: Compile together

```
gcc 1_learn_draw.c allegro.c -o 1.out \\
     -lm -lallegro_primitives -lallegro


gcc 2_rec_drawing.c allegro.c -o 2.out \\
     -lm -lallegro_primitives -lallegro
```

Recursive Tree Drawing

Recursive Tree Drawing

- `3_tree_drawing.c`

```c
#include "allegro.h"

// draws a line starting at (x, y), at an angle, with length len
// recursion goes to 11 level
// at the last two levels, line is drawn in green.
void drawTree(int x, int y, float angle, float len, int level) {
    // the other end point of the line, is obtained
    // by moving len distance at an angle
    int x2  = x + len*cos(2*3.14*angle/360.0);
    int y2  = y - len*sin(2*3.14*angle/360.0);

    // if level < 10 give brown color otherwise green
    ALLEGRO_COLOR c =  al_map_rgb_f(1,1,1) ;

    // set line width to 5 if level <= 2
    al_draw_line(x,y,x2 , y2, c, 1);

    if (level <= 10) {
        drawTree(x2,y2, angle - 20, len, level+1);
        drawTree(x2,y2, angle + 20, len, level +1);
    }
}
```

Structs

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

int main() {
    Point p = {.x = 320, .y = 100};
    printf("X coordinate of p is %d", p.x);
    printf("Y coordinate of p is %d", p.y);
}
```

Struct

```
struct Point {
    int x;
    int y;
};
void drawTree(struct Point p, float angle, float len, int level) {
    struct Point p2 = {
        .x = p.x + len*cos(2*3.14*angle/360.0),
        .y = p.y - len*sin(2*3.14*angle/360.0)
    };
    ...
    if (level <= 10) {
        drawTree(p2, angle - 20, len, level+1);
        drawTree(p2, angle + 20, len, level +1);
    }
}
```

Struct (simplified with typedef)

```
typedef struct Point {
    int x;
    int y;
} Point;

void drawTree(Point p, float angle, float len, int level) {
    Point p2 = {
        .x = p.x + len*cos(2*3.14*angle/360.0),
        .y = p.y - len*sin(2*3.14*angle/360.0)
    };
    ...
    if (level <= 10) {
        drawTree(p2, angle - 20, len, level+1);
        drawTree(p2, angle + 20, len, level +1);
    }
}
```

Struct (another way)

```
typedef struct Point {
    int x;
    int y;
} Point;

void drawTree(Point p, float angle, float len, int level) {
    ...
    if (level <= 10) {
        drawTree((Point){
            .x = p.x + len*cos(2*3.14*angle/360.0),
            .y = p.y - len*sin(2*3.14*angle/360.0)
        }, angle - 20, len, level+1);
        drawTree((Point){
        .x = p.x + len*cos(2*3.14*angle/360.0),
        .y = p.y - len*sin(2*3.14*angle/360.0)
        }, angle + 20, len, level +1);
    }
}
```

Building Structs from struct

```
typedef struct Circle {
    Point center;
    int radius;
} Circle;
void drawCircle(Circle c) {
     al_draw_circle(c.center.x, c.center.y, c.radius);
    if(c.radius > 32) {
    Circle
    c1 = {.center = {c.center.x + c.radius/2, c.center.y},
    .radius = c.radius/2},
    c2 = {.center = {c.center.x - c.radius/2, c.center.y},
    .radius = c.radius/2},
    c3 ={.center = {c.center.x, c.center.y + c.radius/2},
    .radius = c.radius/2},
    c4 ={.center = {c.center.x, c.center.y - c.radius/2},
    .radius = c.radius/2};
    drawCircle(c1);
    drawCircle(c2);
    drawCircle(c3);
    drawCircle(c4);
    }
}
```

Building Structs from struct

```
typedef struct Circle {
    Point center;
    int radius;
} Circle;

void drawCircle(Circle c) {
     al_draw_circle(c.center.x, c.center.y, c.radius);

    if(c.radius > 32) {
    Circle
    c1 = {.center = {c.center.x + c.radius/2, c.center.y}, .radius = c.radius/2},
    c2 = {.center = {c.center.x - c.radius/2, c.center.y}, .radius = c.radius/2},
    c3 ={.center = {c.center.x, c.center.y + c.radius/2}, .radius = c.radius/2},
    c4 ={.center = {c.center.x, c.center.y - c.radius/2}, .radius = c.radius/2};

    drawCircle((Circle){.center = (Point){c.center.x + c.radius/2, c.center.y}, .radius = c.radius/2});
    drawCircle(c2);
    drawCircle(c3);
    drawCircle(c4);
    }
}
```

Array of Structs

```
typedef struct Circle {
    Point center;
    int radius;
} Circle;

void drawCircle(Circle c) {
     al_draw_circle(c.center.x, c.center.y, c.radius,);

    if(c.radius > 32) {
        circle circles[4] = {
        {.center = {c.center.x + c.radius/2, c.center.y}, .radius = c.radius/2},
        {.center = {c.center.x - c.radius/2, c.center.y}, .radius = c.radius/2},
        {.center = {c.center.x, c.center.y + c.radius/2}, .radius = c.radius/2},
        {.center = {c.center.x, c.center.y - c.radius/2}, .radius = c.radius/2}
        };
        for(int i = 0; i < 4; i++) {
            drawCircle(circles[i]);
        }
    }
}
```

Pointer to struct

```
typedef struct Circle {
    Point center;
    int radius;
} Circle;

void drawCircle(Circle c*) {
     al_draw_circle(c->center.x, c->center.y, c->radius,);

    if(c->radius > 32) {
        circle circles[4] = {
        {.center = {c->center.x + c->radius/2, c->center.y}, .radius = c->radius/2},
        {.center = {c->center.x - c->radius/2, c->center.y}, .radius = c->radius/2},
        {.center = {c->center.x, c->center.y + c->radius/2}, .radius = c->radius/2},
        {.center = {c->center.x, c->center.y - c->radius/2}, .radius = c->radius/2}
        };
        for(int i = 0; i < 4; i++) {
            drawCircle(&(circles[i]));
        }
    }
}
```

Problem: Compiling is complicated

```
gcc 1_learn_draw.c allegro.c -o 1.out \
     -lm -lallegro_primitives -lallegro
```

Problem with Compiling

large projects

- Even if small number of files are changed, will compile everything!

Solution: Make files

Recompile only files that are needed.

Make files specify **dependencies** between files and recompiles only what is needed.

Makefile

```
LDLIBS = -lm -lallegro_primitives -lallegro

1.out : 1_learn_draw.o allegro.o
    cc -o 1.out 1_learn_draw.o allegro.o $(LDLIBS)

2.out : 2_rec_drawing.o allegro.o
    cc -o 2.out 2_rec_drawing.o allegro.o $(LDLIBS)

1_learn_draw.o : 1_learn_draw.c allegro.h
    cc -o 1_learn_draw.o 1_learn_draw.c $(LDLIBS)

2_rec_drawing.o : 2_rec_drawing.c allegro.h
    cc -o 2_rec_drawing.o 2_rec_drawing.c $(LDLIBS)

allegro.o : allegro.c allegro.h
    cc -o allegro.o allegro.c $(LDLIBS)

clean:
        rm -f *.out *.o
```

Makefile simplified

```
LDLIBS = -lm -lallegro_primitives -lallegro

1.out : 1_learn_draw.o allegro.o
    cc -o 1.out 1_learn_draw.o allegro.o $(LDLIBS)

2.out : 2_rec_drawing.o allegro.o
    cc -o 2.out 2_rec_drawing.o allegro.o $(LDLIBS)

prog_objs = 1_learn_draw.o 2_rec_drawing.o allegro.o

$(prog_objs) : allegro.h

clean:
        rm -f *.out *.o
```