CSE 303
Lecture 12

structured data

reading: *Programming in C* Ch. 9

slides created by Marty Stepp

http://www.cs.washington.edu/303/

Lecture summary

- structured data
  - struct, typedef
  - structs as parameters/returns
  - arrays of structs

- linked data structures
  - stacks
  - linked lists

Structured data

```
struct typename {     // declaring a struct type
    type name;
    type name;
    ...
    type name;        // fields
};
```

- **struct**: A type that stores a collection of variables.
  - like a Java class, but with only fields (no methods or constructors)
  - instances can be allocated on the stack or on the heap

```
struct Point {    // defines a new structured
    int x, y;     // type named Point
};
```

Using structs

- a `struct` instance is declared by writing the type, name, and `;`
  - this allocates an instance of the structured type on the stack
  - refer to the fields of a struct using the `.` operator

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

int main(void) {
    struct Point p1;                    // on stack
    struct Point p2 = {42, 3};          // initialized
    p1.x = 15;
    p1.y = -2;
    printf("p1 is (%d, %d)\n", p1.x, p1.y);
    return 0;
}
```

typedef

`typedef` ***`type name`***`;`

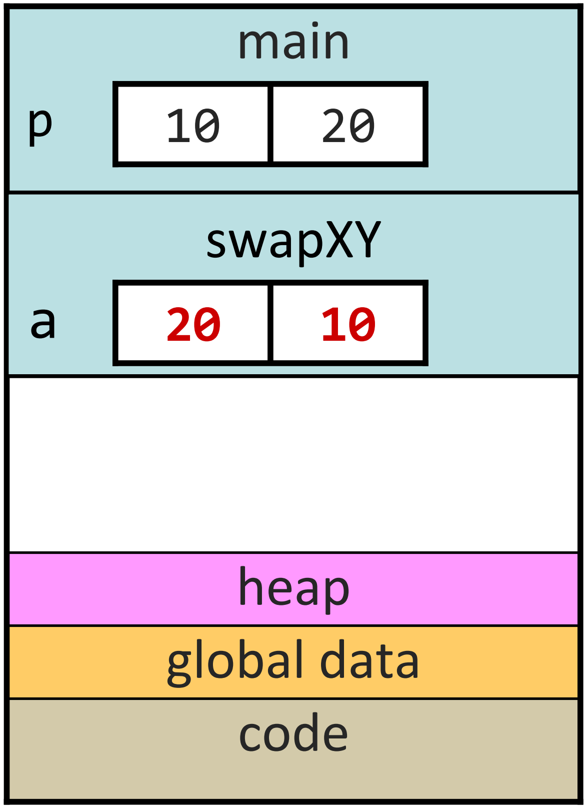
- tell C to acknowledge your struct type's name with `typedef`

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

int main(void) {
    Point p1;     // don't need to write 'struct'
    p1.x = 15;
    p1.y = -2;
    printf("p1 is (%d, %d)\n", p1.x, p1.y);
    return 0;
}
```

Structs as parameters

- when you pass a struct as a parameter, it is copied
  - not passed by reference as in Java

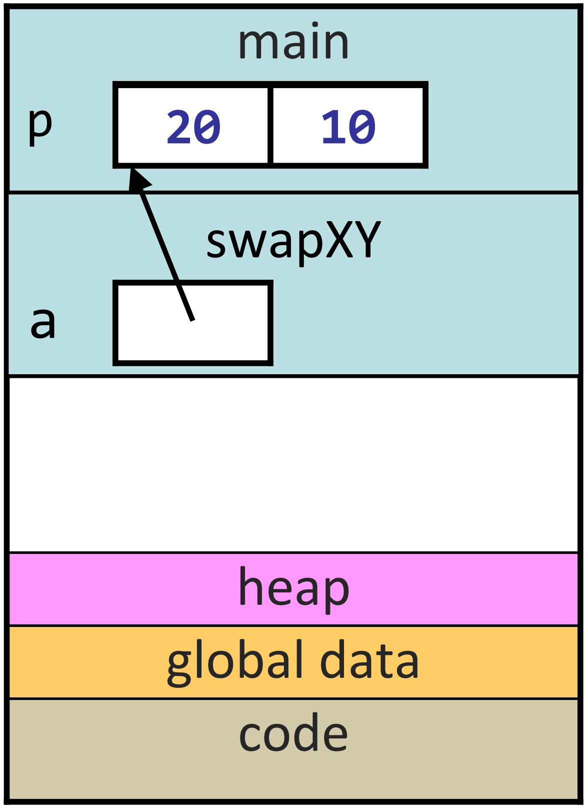
```
void swapXY(Point p1);

int main(void) {
    Point p = {10, 20};
    swapXY(p);
    printf("(%d, %d)\n", p.x, p.y);
    return 0;    // prints (10, 20)
}

void swapXY(Point a) {
    int temp = a.x;
    a.x = a.y;
    a.y = temp;  // does not work
}
```



Pointers to structs

- structs can be passed by reference using pointers
  - must use parentheses when dereferencing a *struct** (precedence)

```
void swapXY(Point* p1);

int main(void) {
    Point p = {10, 20};
    swapXY(&p);
    printf("(%d, %d)\n", p.x, p.y);
    return 0;    // prints (20, 10)
}

void swapXY(Point* a) {
    int temp = (*a).x;
    (*a).x = (*a).y;
    (*a).y = temp;
}
```



The -> operator

- more often, we allocate `structs` on the heap and pass pointers

  `pointer``->`***`field`*** is equivalent to `(*`***`pointer`***`).`***`field`***

```
void swapXY(Point* p1);

int main(void) {
    Point* p = (Point*) malloc(sizeof(Point));
    p->x = 10;
    p->y = 20;
    swapXY(p);
    printf("(%d, %d)\n", p->x, p->y);    // (20, 10)
    return 0;
}

void swapXY(Point* a) {
    int temp = a->x;
    a->x = a->y;
    a->y = temp;
}
```

Copy by assignment

- one structure's entire contents can be copied to another with =

```
struct2 = struct1;    // copies the memory
```

```
int main(void) {
    Point p1 = {10, 20},  p2 = {30, 40};
    p1 = p2;
    printf("(%d, %d)\n", p1.x, p1.y);    // (30, 40)

    // is this the same as p1 = p2; above?
    Point* p3 = (Point*) malloc(sizeof(Point));
    Point* p4 = (Point*) malloc(sizeof(Point));
    p3->x = 70;
    p3->y = 80;
    p3 = p4;
    printf("(%d, %d)\n", p3->x, p3->y);
    return 0;
}
```

Struct literals

- a structure can be assigned a state later using a `struct` literal:

```
name = (type) {value, ..., value};
```

```
int main(void) {
    Point p1 = {10, 20},  p2 = {30, 40};
    p1 = p2;
    printf("(%d, %d)\n", p1.x, p1.y);    // (30, 40)

    // is this the same as p1 = p2; above?
    Point* p3 = (Point*) malloc(sizeof(Point));
    Point* p4 = (Point*) malloc(sizeof(Point));
    *p3 = (Point) {70, 80};
    p3 = p4;
    printf("(%d, %d)\n", p3->x, p3->y);

    return 0;
}
```

Struct as return value

- we generally pass/return `struct`s as pointers
  - faster; takes less memory than copying the `struct` onto the stack
  - if a `struct` is `malloc`ed and returned as a pointer, caller must free it

```
int main(void) {
    Point* p1 = new_Point(10, 20);
    ...
    free(p1);
    return 0;
}

// creates/returns a Point; sort of a constructor
Point* new_Point(int x, int y) {
    Point* p = (Point*) malloc(sizeof(Point));
    p->x = x;
    p->y = y;
    return p;   // caller must free p later
}
```

Comparing structs

- relational operators (==, !=, <, >, <=, >=) don't work with structs

```
Point p1 = {10, 20};
Point p2 = {10, 20};
if (p1 == p2) { ...          // error
```

- what about this?

```
Point* p1 = new_Point(10, 20);
Point* p2 = new_Point(10, 20);
if (p1 == p2) { ...          // true or false?
```

Comparing structs, cont'd

- the right way to compare two `structs`: write your own

```
#include <stdbool.h>

bool point_equals(Point* a, Point* b) {
    if (a->x == b->x && a->y == b->y) {
        return true;
    } else {
        return false;
    }
}

int main(void) {
    Point p1 = {10, 20};
    Point p2 = {10, 20};
    if (point_equals(&p1, &p2)) { ...
```

Structs and input

- you can create a pointer to a field of a `struct`
  - structs' members can be used as the target of a `scanf` read, etc.

```
int main(void) {
    Point p;
    printf("Please type your x/y position: ");
    scanf("%d %d", &p.x, &p.y);
    return 0;
}

int main(void) {
    Point* p = (Point*) malloc(sizeof(Point));
    printf("Please type your x/y position: ");
    scanf("%d %d", &p->x, &p->y);
    return 0;
}
```

Arrays of structs

- **parallel arrays**: ≥ 2 arrays conceptually linked by index.
  - parallel arrays are bad design; isn't clear that they are related
  - you should often replace such arrays with an array of `structs`

```
int id[50];                 // parallel arrays to store
int year[50];               // student data (bad)
double gpa[50];


typedef struct Student {    // one array of structs
    int id, year;
    double gpa;
} Student;
...
Student students[50];
```

Structs with pointers

- What if we want a Student to store a significant other?

```
typedef struct Student {        // incorrect
    int id, year;
    double gpa;
    struct Student sigother;
} Student;
```

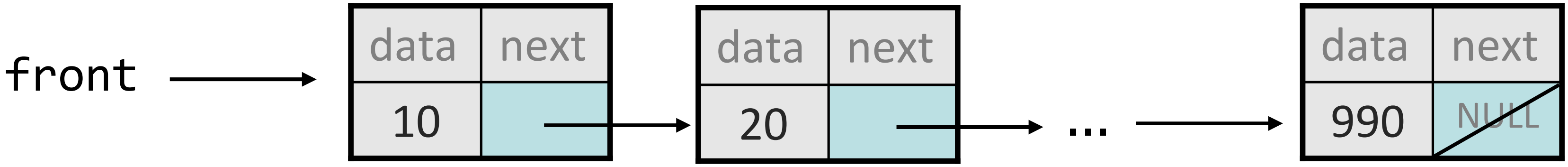
  - a Student cannot fit another entire Student inside of it!

```
typedef struct Student {        // correct
    int id, year;
    double gpa;
    struct Student* sigother;
} Student;
```

Linked data structures

- C does not include collections like Java's `ArrayList`, `HashMap`
  - must build any needed data structures manually
  - to build a linked list structure, create a chain of structs/pointers

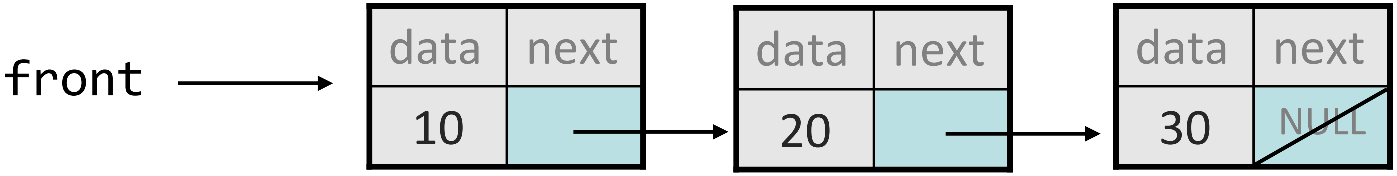
```
typedef struct Node {
    int data;
    struct Node* next;
} Node;

Node* front = ...;
```



Manipulating a linked list

- there is only a node type (`struct`), no overall list class
- list methods become functions that accept a front node pointer:

```
int list_length(Node* front) {
    Node* current = front;
    int count = 0;
    while (current != NULL) {
        count++;
        current = current->next;
    }
    return count;
}
```



Exercise

- Write a complete C program that allows the user to create a basic stack of `int`s. The user should be able to:
  - *push* : put a new `int` onto the top of the stack.
  - *pop* : remove the top `int` from the stack and print it.
  - clear : remove all `int`s from the stack.

- Do not make any assumptions about the size of the stack.
  - Do not allow any memory leaks in your program.