structs

What is a struct?

- A "record"/group of related variables, e.g. name, address, phone #
- Precursor of C++/Java object
- Similar to a Java/C++/Python object:
  - Related data (variables) are grouped together
  - Members accessed with '.' operator. (person.name, person.addr)
  - Members may have different types (unlike arrays)
  - Members can be primitives, arrays, structs, etc. Recursive definitions and nesting allowed.
  - First declared. Then "instances" are defined
- Different from a a Java/C++/Python object:
  - No functions! Just variables
  - All members are public. (No 'private'. No encapsulation.)
  - Copying a struct copies all members
  - Passing a struct to a function puts copy of struct on stack; likewise when returning a struct

jo
10 Walnut St
215.898.2468

mo
25 South St
215.990.5532

Example: A simple person

```
// Here, we define a struct person. No memory is allocated.
struct person {
   char name;
   int id;
};

int main() {
    // Here we declare two variables of type struct person. Memory is allocated.
    struct person p1;
    struct person p2;
    p1.name = 'a';  // initialize the name field/member
    p1.id = 25;     // initialize the id field/member
    p2 = p1;        // UNLIKE JAVA, all the fields/members are copied
                    // Now, p2.name is 'a', and p2.id is 25.

    // Comparing structs with <, <=, >, >=, ==, etc. doesn't compute/compile.
    // Instead, compare on a field-by-field basis:
    if (pi.id == p2.id)
     // do something
```

arrays of structs

```
// Here, we define a struct person. No memory is allocated.
struct person {
   char name;
   int id;
};

int main() {
    // Here we declare (and allocate memory for) 1000 struct person's
    struct person data[1000];
    data[0].name = 'a';
    data[0].id = 25;
    data[1].name = 'f';
    data[1].id = 500;

    int i = 0;
    for (i = 0; i < 1000; i++)
       printf("name: %c, id: %d\n", data[i].name, data[i].id);
}
```

functions and structs

```
// Here, we define a struct person. No memory is allocated.
struct person {
   char name;
   int id;
};

int main() {
    // Here we declare and intialize one struct person
    struct person p1 = {'a', 25};
    struct person p2;
    p2 = f(p1); // A COPY of person1 is placed on the stack
    printf("name is %c, id is %d\n", p2.name, p2.id);  // What is printed?
}

struct person f(struct person p){
    p.id = 500;
    return p;  // A COPY of p is placed on the stack (which is assigned to person2)
}
```

typedef's for readability

```
// Wouldn't it be nice to use Person instead of struct person?
// We can if we use a typedef.
// A typedef is a "type definition". We use it to define a new type.
// No memory is allocated here; this is a definition:
typedef struct {
   char name;
   int id;
} Person;   // By convention, the new type name begins with a capital letter

int main() {
    Person person1, person2;  // memory allocated for two Persons
    person1.name = 'a';
    person1.id = 25;

    person2 = foo(person1);
}

Person foo(Person p){
    ...
}
```