Structures

C is a procedural programming language, which means that a program is just a collection of functions – no classes. However, C does have a construct called a struct that is similar to a data class in Java. A struct is simply a new variable type that is a collection of related data.

Syntax

Here is the format of a struct declaration:

```
struct name {
     type1 name1;
     type2 name2;
     ...
} objList;
```

Here, `name` is the **optional** name for this structure type. This name is needed if you plan to created any variables of this struct type.

The "`type name`" elements are *fields* that you want in your struct. For example, if the struct represented a person, you might want these fields:

```
char name[20];
int age;
```

Think of fields in structs as instance variables in a Java class. Finally, `objList` is an **optional** list of variable names you want constructed with this struct type.

First Example

Here is a simple struct:

```
struct person {
     char name[20];
     int age;
} p1, p2;
```

Now, `struct person` is a new datatype. `p1` and `p2` are variables of type `struct person`.

Declaring Struct Variables
You can automatically declare struct variables by listing variable names at the end of the struct definition. You can also declare them outside the definition just like you do ordinary variables. The format for declaring variables in C is:

```
type name;
```

The type of the above struct, for example, is "`struct person`". So we can declare another struct variable as follows:

```
struct person p3;
```

This declaration automatically allocates space for the struct, including space for every field in the struct.

Accessing Struct Fields
Accessing a field in a struct variable is exactly like accessing an instance variable in a Java object:

```
structVar.fieldName
```

Here, `structVar` is the name of the struct variable and `fieldName` is one of the fields in the struct. This allows us to access or change that field.

Let's declare another struct person variable, and set the person's name to "Bob" and age to 20. Here's how:

```
struct person bobPerson;       //declare struct variable
bobPerson.age = 20;            //set person's age to 20
strcpy(bobPerson.name, "Bob");//set person's name to "Bob"
```

Notice that if we are initializing a string field, we must use `strcpy`. The following will NOT compile since name is an array (a constant pointer):

```
bobPerson.name = "Bob";   //Will not compile!
```

<u>Example</u>
Structs can be declared at any point in a C program, but they are usually declared with the global variables (right after the include statements). This way, the struct type can be used throughout the file.

Here is an example that uses a struct to store a two-dimensional point (x, y location). It gets two points as input, and then prints the equation that passes through the points.

```
#include <stdio.h>

struct point {
	int x;
	int y;
};			//No variables declared here

double getSlope(int, int, int, int);
double getIntercept(int, int, double);

int main() {
	struct point p1;
	struct point p2;
	double slope;
	double intercept;

	printf("Enter point1, e.g. (1, 2): ");
	scanf("(%d, %d)", &(p1.x), &(p1.y));

	printf("Enter point2, e.g. (1, 2): ");
	scanf("(%d, %d)", &(p2.x), &(p2.y));

	slope = getSlope(p1.x, p1.y, p2.x, p2.y);
	intercept = getIntercept(p1.x, p1.y, slope);

	//prints equation in form y = mx + b
	//m: slope, b: y-intercept
	printf("y = %.2lfx + %.2lf\n", slope, intercept);

	return 0;
}

double getSlope(int x1, int x2, int y1, int y2) {
	//slope = change in y / change in x
	return (y2-y1)/(x2-x1);
}

double getIntercept(int x, int y, double slope) {
	//if y = mx + b, b = y - mx
	return y - slope*x;
}
```

Structs as Function Arguments

We could have also written to above example by passing the point structs to the `getSlope` and `getIntercept` functions (instead of passing their fields). This works just like passing other variable types, except "`struct point`" will be an argument type.

Here’s the example when we pass the points to the functions:

```
#include <stdio.h>

struct point {
     int x;
     int y;
};                  //No variables declared here

double getSlope(struct point, struct point);
double getIntercept(struct point, double);

int main() {
     struct point p1;
     struct point p2;
     double slope;
     double intercept;

     printf(“Enter point1, e.g. (1, 2): “);
     scanf(“(%d, %d)”, &(p1.x), &(p1.y));

     printf(“Enter point2, e.g. (1, 2): “);
     scanf(“(%d, %d)”, &(p2.x), &(p2.y));

     slope = getSlope(p1, p2);
     intercept = getIntercept(p1, slope);

     //prints equation in form y = mx + b
     //m: slope, b: y-intercept
     printf(“y = %.2lfx + %.2lf\n”, slope, intercept);

     return 0;
}

double getSlope(struct point p1, struct point p2) {
     //slope = change in y / change in x
     return (p2.y-p1.y)/(p2.x-p1.x);
}

double getIntercept(struct point p, double slope) {
     //if y = mx + b, b = y - mx
     return p.y - slope*p.x;
}
```

Arrays of Structs
You can create arrays of structs in C just like you can create arrays of any other type. The format for creating constant-sized arrays is:

```
type name[size];
```

Consider the person struct again:

```
struct person {
     char name[20];
     int age;
};
```

Here's how to create a 3-slot array of type `struct person` called group:

```
//"struct person" is the type; "group" is the array name
struct person group[3];
```

When you create an array of type struct, C allocates space for each struct element (and its fields) in the array.

We can get out a particular struct element using an array index:

```
group[0]    //the first struct person in the array
```

For example, here's how we could set the first person's name to "Bob" and age to 20:

```
strcpy(group[0].name, "Bob");
group[0].age = 20;
```

Pointers to Structs
You can declare a pointer to a struct just like you declare a pointer to another element type. The format for declaring a pointer is:

```
type* name;
```

So, to declare a pointer to a `struct person` element, we could say:

```
struct person* personPtr;
```

Suppose we have another `struct person` variable:

```
struct person p1;
strcpy(p1.name, "Jill");
p1.age = 18;
```

Then we can make `personPtr` point to `p1`:

```
    personPtr = &p1;
```

Allocating Memory

We can also create struct variables by:

1) Declaring a pointer to a struct
2) Allocating memory for the struct

This approach (using pointers instead of standard variables) is handy when building data structures like linked lists.

To create a `struct person` in this way, we first declare a pointer:

```
    struct person* personPtr;
```

Then we allocate memory for the struct, and give `personPtr` the address of that memory. We can use `sizeof(struct person)` to get the number of bytes needed to store a variable of type `struct person`:

```
    personPtr = malloc(sizeof(struct person));
```

Now, `personPtr` points to a `struct person`, which has space for the name and age fields.

Accessing Fields

We can get at the `struct person` object itself by dereferencing `personPtr`:

```
    *personPtr
```

We can then initialize the fields:

```
    (*personPtr).age = 18;
```

This line does two things:

1) Dereferences the pointer to get at the `struct person` object
2) Changes the person's age to 18

Do NOT write something like this:

```
     *personPtr.age = 18;          //BAD!
```

The compiler will try to resolve the "`.`" operator first. Because `personPtr` is a pointer and not a struct, using a `.` doesn't make since. This line will result in a compiler error. We need to dereference the pointer before we can access any fields.

Pointers to structs are very common in C, and you'll often find yourself dereferencing a struct pointer and then accessing one of the fields. Because of this, there is a shortcut notation:

```
personPtr->age = 18;
```

- Is equivalent to -

```
(*personPtr).age = 18;
```

Linked Lists

We can use structs to help implement all the common data structures in C. For example, whereas before we would have created a `Node` class to help us write a linked list, we will now create a node struct:

```
struct node {
     int data;
     struct node *next;   //pointer to next node in list
};
```

Suppose we want to create a linked list with a single node. First, we declare a head variable:

```
struct node *head;
```

Then, we allocate memory for the node:

```
head = malloc(sizeof(struct node));
```

Finally, we initialize the fields in head:

```
head->data = 4;
head->next = NULL;
```

Here's a full C program that inserts values into a linked list, and then prints them out:

```
#include <stdio.h>

struct node {
     int data;
     struct node *next;
};
```

```c
struct node *head;

void add(int);
void print(void);

int main() {
     head = NULL;

     add(1);
     add(2);
     add(3);
     add(4);

     //prints 1, 2, 3, 4 on separate lines
     print();

     return 0;
}

void add(int num) {
     //create new node
     struct node *newnode = malloc(sizeof(struct node));
     newnode ->data = num;
     newnode ->next = NULL;

     if (head == NULL) {
          head = newnode;
     }
     else {
          //find end of list
          struct node *cur = head;
          while (cur->next != NULL) {
               cur = cur->next;
          }

          //add  newnode after  cur
          cur->next = newnode;
     }
}

void print(void) {
     struct node *cur = head;
     while (cur != NULL) {
          printf("%d\n", cur->data);
          cur = cur->next;
     }
}
```

Linked lists in C work almost exactly the same as linked lists in Java. The only differences are:

1) We use a struct instead of a class to represent a node
2) We use `->` instead of `.` to access values in the node