Structures in C

Memory Allocation

C Structures and Memory Allocation

- There is no class in C, but we may still want non-homogenous structures
  - So, we use the struct construct
    - struct for structure
  - A struct is a data structure that comprises multiple types, each known as a member
    - each member has its own unique name and a defined type
  - Example:
    - A student may have members: name (char[ ]), age (int), GPA (float or double), sex (char), major (char[ ]), etc
  - If we want to create a structure that can vary in size, we will allocate the struct on demand and attach it to a previous struct through pointers
    - Here, we examine structs, allocation techniques, and linked structures

The struct Definition

- struct is a keyword for defining a structured declaration
- Format:

```
struct name {
    type1 name1;
    type2 name2;
    …
};
```

name1 and name2 are *members* of name

- name represents this structure's tag and is optional
  - we can either provide name
  - or after the } we can list variables that will be defined to be this structure
- We can also use typedef to declare name to be this structure and use name as if it were a built-in type
  - typedef will be covered later in these notes

Examples

```
struct point {
    int x;
    int y;
};

struct point p1, p2;
```

p1 and p2 are both points, containing an x and a y value

```
struct {
    int x;
    int y;
}  p1, p2;
```

p1 and p2 both have the defined structure, containing an x and a y, but do not have a tag

```
struct  point {
    int x;
    int y;
}  p1, p2;
```

same as the other two versions, but united into one set of code, p1 and p2 have the tag point

For the first and last sets of code, point is a defined tag and can be used later (to define more points, or to declare a type of parameter, etc) but in the middle code, there is no tag, so there is no way to reference more examples of this structure

Accessing structs

- A struct is much like an array
  - The structure stores multiple data
    - You can access the individual data, or you can reference the entire structure
  - To access a particular member, you use the . operator
    - as in student.firstName or p1.x and p1.y
      - we will see later that we will also use - > to reference a field if the struct is pointed to by a pointer
  - To access the struct itself, reference it by the variable name
    - Legal operations on the struct are assignment, taking its address with &, copying it, and passing it as a parameter
      - p1 = {5, 10}; // same as p1.x = 5; p1.y = 10;
      - p1 = p2; // same as p1.x = p2.x; p1.y = p2.y;

structs as Parameters

- We may pass structs as parameters and functions can return structs
  - Passing as a parameter:
    - void foo(struct point x, struct point y) {…}
      - notice that the parameter type is not just the tag, but preceded by the reserved word struct
  - Returning a struct:

```
struct point createPoint(int a, int b)
{
    struct point temp;
    temp.x = a;
    temp.y = b;
    return temp;
}
```

Inputting a struct in a Function

- We will need to do multiple inputs for our struct
  - Rather than placing all of the inputs in main, let's write a separate function to input all the values into our struct
    - The code to the right does this
  - But how do we pass back the struct?
    - Remember C uses pass by copy
      - the struct is *copied* into the function so that p in the function is different from y in main
      - after inputting the values into p, nothing is returned and so y remains {0, 0}

```
#include <stdio.h>

struct point  {
            int x;
            int y;  };

void getStruct(struct point);
void output(struct point);
void main( )    {
            struct point y = {0, 0};
            getStruct(y);
            output(y);  }

void getStruct(struct point p)   {
            scanf("%d", &p.x);
            scanf("%d", &p.y);
            printf("%d, %d", p.x, p.y);  }

void output(struct point p)   {
            printf("%d, %d", p.x, p.y);  }
```

One Solution For Input

- In our previous solution, we passed the struct into the function and manipulated it in the function, but it wasn't returned
  - Why not? Because what was passed into the function was a copy, not a pointer
    - So structs differ from arrays as structs are not pointed to
- In our input function, we can instead create a temporary struct and return the struct rather than having a void function

```
void main( )
{
        struct point y = {0, 0};
        y = getStruct( );
        output(y);
}
```

```
struct point inputPoint( )
{
    struct point temp;
    scanf("%d", &temp.x);
    scanf("%d", &temp.y);
    return temp;
}
```

We could also pass the address of y and treat the struct like an array, we will see this next, but it requires a change in how we handle the members of the struct

Pointers to Structs

- The previous solution had two flaws:
  - It required twice as much memory
    - we needed 2 points, one in the input function, one in the function that called input
  - It required copying each member of temp back into the members of the original struct
    - with our point type, that's not a big deal because there were only two members, but this may be undesirable when we have a larger struct
  - So instead, we might choose to use a pointer to the struct, we pass the pointer, and then we don't have to return anything – scanf will follow the pointer and place the datum in our original struct
    - We see an example next, but first…
- If a is a pointer to a struct, then to access the struct's members, we use the - > operator as in a->x

Pointer-based Example

```
 #include <stdio.h>
struct foo  {              // a global definition, the struct foo is known in all of
    int a, b, c;           // these functions
};

//  function prototypes
void inp(struct foo *);            // both functions receive a pointer to a struct foo
void outp(struct foo);

void main( )   {
            struct foo x;          // declare x to be a foo
            inp(&x);               // get its input, passing a pointer to foo
            outp(x);               // send x to outp, this requires 2 copying actions
}

void inp(struct foo *x)
{                          // notice the notation here:  &ptr->member
            scanf("%d%d%d", &x->a, &x->b, &x->c);
}

void outp(struct foo x)            // same notation, but without the &
{
            printf("%d %d %d\n", x.a, x.b, x.c);
}
```

Nested structs

- In order to provide modularity, it is common to use already-defined structs as members of additional structs
- Recall our point struct, now we want to create a rectangle struct
  - the rectangle is defined by its upper left and lower right points

```
struct point {
    int x;
    int y;
 }
struct rectangle {
    struct point pt1;
    struct point pt2;
}
```

If we have

```
struct rectangle r;
```

Then we can reference

r.pt1.x, r.pt1.y,
r.pt2.x and r.pt2.y

Now consider the following

```
struct rectangle r, *rp;
rp = &r;
```

Then the following are all equivalent

```
r.pt1.x
rp->pt1.x
(r.pt1).x
(rp->pt1).x
```

But not rp->pt1->x (since pt1 is not a pointer to a point)

Arrays of structs

- To declare an array of structs (once you have defined the struct):
  - struct rectangle rects[10];
  - rects now is a group of 10 structures (that consist each of two points)
  - You can initialize the array as normal where each struct is initialized as a { } list as in {5, 3} for a point or {{5, 3}, {8, 2}} for a rectangle
- The array of structs will be like the array of classes that we covered in 260/262, we will use this data structure if we want to create a database of some kind and apply such operations as sorting and searching to the structure

Example

```
struct point{
    int x
    int y;
};

struct rectangle {
     struct point p1;
     struct point p2;
};

void printRect(struct rectangle r)
{
            printf("<%d, %d> to <%d, %d>\n", r.p1.x, r.p1.y, r.p2.x, r.p2.y);
}

void main( )
{
    int i;
    struct rectangle rects[ ] = {{{1, 2}, {3, 4}}, {{5, 6}, {7, 8}}}; // 2 rectangles
    for(i=0;i<2;i++) printRect(rects[i]);
}
```

Memory Allocation

- To this point, we have been declaring pointers and having them point at already created variables/structures
  - However, the primary use of pointers is to create dynamic structures
    - structures that can have data added to them or deleted from them such that the amount of memory being used is equal to the number of elements in the structure
    - this is unlike an array which is static in size
  - An ordinary variable has its memory created at compile time so is fixed in size
  - The pointer can point to a piece of memory that has just been created (allocated)
  - We will use this approach (memory allocation + pointers) to create data structures like linked lists and trees

Note: in Java, allocation was done whenever you used the *new* reserved word as in ClassName x = new ClassName(…);

malloc and calloc

- The two primary memory allocation operations in c are malloc and calloc
  - The main difference between them is that calloc provide a chunk of contiguous memory – we will use this for the creation of an array
  - For most situations, we will use malloc, which has the form:
    - pointer = (type *) malloc(sizeof(type));
    - This sets pointer to point at a newly allocated chunk of memory that is the type specified and the size needed for that type
      - NOTE: pointer will be NULL if there is no more memory to allocate
  - The cast may not be needed, but is good practice
  - calloc has the form:
    - pointer = (type *) calloc(n, sizeof(type)); // n is the size of the array
  - Another C instruction is free, to free up the allocated memory when you no longer need it as in free(pointer);

calloc example

- The most common use of calloc is for flexible sized arrays (to change the size)
- free() is used to free the allocated space.

An example of malloc is given on my web site rather than here as it is too large to fit

```
#include <stdio.h>
#include <stdlib.h>                 // needed for calloc

void main()
{
        int i;
        int *x, *y;                                 // two pointers to int arrays
        x = (int *) calloc(10, sizeof(int));        // x now points to an array of 10 ints
        for(i=0;i<10;i++) x[i] = i;                 // fill the array with values
        …                                           // oops, need more room than 10
        y = (int *) calloc(20, sizeof(int));        // create an array of 20, temporarily
                                                    //    pointed to by y
        for(i=0;i<10;i++) y[i] = x[i];              // copy old elements of x into y
        free(x);                                    // release memory of old array
        for(i=10;i<20;i++) y[i] = i;                // add the new elements
        x = y;                                      // reset x to point at new, bigger array
}
```

Unions

- Like structures, but every member occupies the same region of memory!
  - Structures: members are “and”ed together: “name and species and owner”
  - Unions: members are “xor”ed together

```
union VALUE {
  float f;
  int i;
  char *s;
};
/* either a float xor an int xor a string */
```

Unions

- Up to programmer to determine how to interpret a union (i.e. which member to access)
- Often used in conjunction with a “type” variable that indicates how to interpret the union value

```
enum TYPE { INT, FLOAT, STRING };
struct VARIABLE {
  enum TYPE type;
  union VALUE value;
};
```

Access `type` to determine how to interpret `value`

Unions

- Storage
  - size of union is the size of its largest member
  - avoid unions with widely varying member sizes; for the larger data types, consider using pointers instead
- Initialization
  - Union may only be initialized to a value appropriate for the type of its first member

Assignment

- Differentiate between arrays and structure with example.
- Explain how we can nest a structure into another structure.
- Explain the use of free() in dynamic memory allocation.
- Differentiate between structure and union.