Using Structures

Section 4.4

Arrays and Structures

- Structures can contain arrays
- Can also have arrays of a particular structure
  - Think array of objects
  - **struct *structureName arrayName*[*size*];**

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

struct point points[10];
```

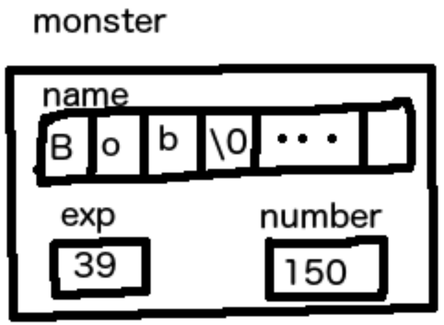
- Access the members of a structure by indexing into the array, then using the dot
  - ***arrayName*[*index*].*memberName***

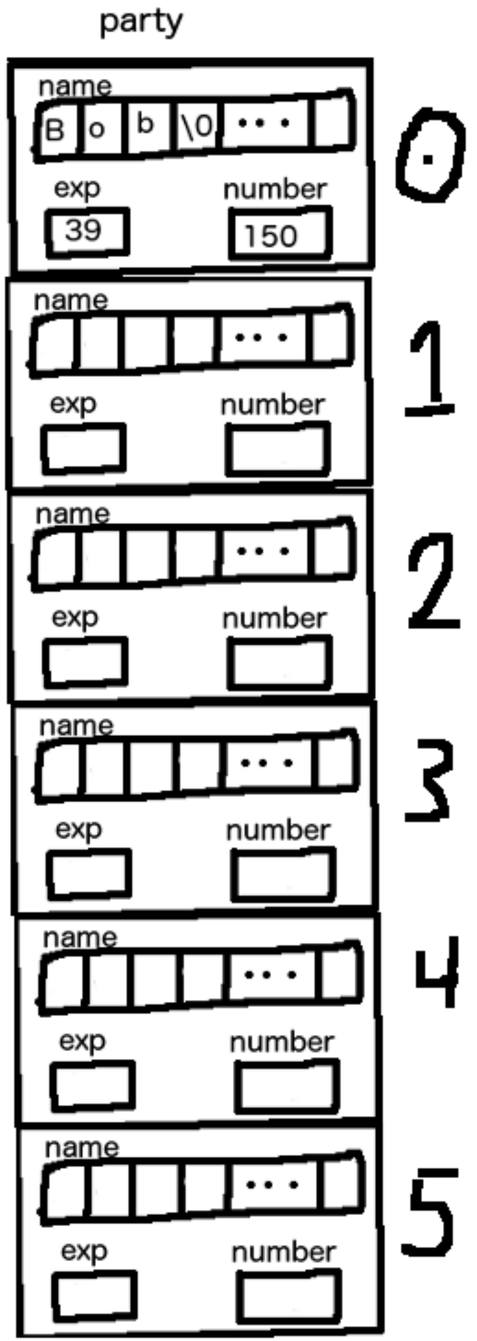
```
points[5].x = 6;
points[5].y = 5;

printf("(%d, %d)", points[5].x, points[5].y);
```

Example

- Suppose we were making a game where players collect and battle creatures
- We could make a structure to represent a creature
- And then have an array of these structures to represent the creatures the player currently had with them

```
struct Creature
{
    char name[32];
    int exp;
    int number;
};

struct Creature monster;

monster.exp = 39;
monster.number = 150;
strcpy(monster.name, "Bob");

struct Creature party[6];

party[0] = monster;
```





Making a Structure and a Structure Variable

- Most basic way of using struct is to make a template
- But can also make the template and an actual variable of that type all in one line
  - Like making a structure template, but include a name for the variable between the close } and ;

```
struct structureName
{
    /* Members */
} variableName;
```

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

Anonymous Structures

- Also possible to specify a structure variable without providing a name for the template
- Makes the structure "throw away"
- No name for the template, cannot be reused later in the program to make a new structure of that type
- Not often used by itself

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

Global and Local

- Often use the terms **global** and **local** in C programming
- Refer to the scope of a variable
  - Scope is area of the program where the variable is valid and can be accessed

- A **local** variable is a variable inside of a function
  - Can only be accessed within that function, unless passed by reference to another
  - Generally best to make variables local if possible

- A **global** variable is a variable outside of the main() function
  - Can only be accessed anywhere within the source code file
  - Not good to use excessively, can make code messy

```
int gx;
int gy;

int main(int argc, char *argv[])
{
    int lx = gx + gy;
    int ly = lx + gy;
    printf("%d %d", gx, lx);
}

void doSomething(int x)
{
    int lx;
}
```

Nested Structures

- A structure can contain another structure
- Known as a nested structure

```
struct name
{
    char first[32];
    char last[32];
};

struct person
{
    int age;
    float pay;
    struct name title;
};
```

- Have to use two dots to access the inner structure variables members
  - ***structVar*.*nestedStructVar*.*memberName***

```
struct person pirate;

strcpy(pirate.title.first, "Jack");

printf("Name: %s\n", pirate.title.first);
```

typedef Keyword

- **typedef** keyword allows us to create an alias for an existing data type
  - **typedef *oldDataType newDataType*;**
  - Can also include data type qualifier keywords in *oldDataType*

- Often used for convenience and readability
  - **typedef unsigned int uint32;**
  - **typedef signed short int sint16;**

- Also used for portability
  - Sizes of data types can vary from system to system
  - typedef allows us to define a type in a single place

- Consider a situation where the same code is intended to run on two different systems.. on one system, an int is 16 bits, on the other system, an int is 32 bits
- On first system:
  - **typedef unsigned int uint16;**
- On second system:
  - **typedef unsigned short int uint16;**
- Now the code can just use uint16 throughout

- Convenient for structures as well
  - Avoid having to continue using struct keyword

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

typedef struct _point point_t;
```

```
typedef struct _point
{
    int x;
    int y;
} point_t;
```

Structures and Pointers

- Can have a pointer to a structure variable
  - **struct *structureName* * *pointerName*;**
- Often work with pointers to structures when a structure variable passed by reference into a function

```
struct fraction
{
    int numerator;
    int denominator;
};

struct fraction fract;
fract.numerator = 1;
fract.denominator = 4;

struct fraction * ptr;
ptr = &fract;
```

Label	Address	Value
fract, fract.numerator	400 – 403	1
fract.denominator	404 – 407	4
ptr	408 – 411	400

- To access member variables from a pointer to a structure, must first dereference the pointer (put the *), then can access the members like normal
  - **(****pointerName***).*****memberName***

- Or an alternative way is to use the **-> operator**
- **-> operator** allows us to access member variables of a pointer to a structure without using the * operator
  - ***pointerName->memberName***
  - **->** is the more common way

```
struct fraction fract;

struct fraction * ptr;
ptr = &fract;

ptr->numerator = 1;
(*ptr).numerator = 1;

ptr->denominator = 4;
(*ptr).denominator = 4;
```

Dynamically Allocating Structures

- Can dynamically allocate memory for a structure or an array of structures
- Use **sizeof()** operator on a structure type
  - Conveniently counts how many bytes are in the structure

```
struct fraction * ptr;

ptr  = (struct fraction *)malloc(sizeof(struct fraction) * 5);
```