As we already discussed, an array can only hold stuff of *the same type*. On the other hand, a C *structure* is a collection of one or more variables, possibly of different types. It is similar to the *Java class*, except it contains no method(s).

A structure thus helps us to organize complicated data, particularly for larger programs, since we now can place a group of related data into an integrated piece, instead of a bunch of separate pieces.

The following declares a *point* in a 2-d plane.

```c
struct point {
    int x;
    int y;
}
```
Define, and use, a structure

We can now define a few point type variables:

```c
struct point pt;
struct point maxpt={230, 340};
```

and later on print out a pointer as follows:

```c
printf("%d,%d", pt.x, pt.y);
```

We can also calculate the distance from (0,0) to pt, using the formula, with `math.h`:

\[
\sqrt{x^2 + y^2}
\]

```c
dist=sqrt((double)pt.x*pt.x+(double)pt.y*pt.y);
```

We can define more structures, e.g., a rectangle with two points:

```c
struct rect {
    struct point p1; //bottom left
    struct point p2; //top right
}
```
Structures and functions

We can copy a structure, assign it as a unit, take its address using ‘&’, or get its members with the ‘.’ operator, similar to Java.

We cannot compare structures, although a structure can be initialized with a list of constant member values. There is no constructor for a C structure, though.

Let’s write a few functions to work with structures, although they are not affiliated with structures in the Java sense, i.e., these functions are not part of a structure.

Keep in mind that C is not an object oriented language. 😊 Thus, it does not support such a notion of class.

Again, we started with C, then C++, Java...
What could we do?

The following looks like a constructor of the point structure. 😊

```c
struct point makepoint(int x, int y){
    struct point temp;

    temp.x=x;
    temp.y=y;
    return temp;
}
```

We can then do the following:

```c
struct rect screen;
struct point middle;
struct point makepoint(int, int);

screen.pt1=makepoint(0, 0);
screen.pt2=makepoint(XMA, YMAX);
middle=makepoint((screen.pt1.x+screen.pt2.x)/2,
                 (screen.pt1.y+screen.pt2.y)/2);
```

**Question:** Is this where the cursor points initially? In the middle of a screen...
We can also write a function `pointInRect` to tell if a point is inside a rectangle:

```c
int pointInRect(struct point p, struct rect r){
    return (p.x>=r.pt1.x) && (p.x<r.pt2.x)
           && (p.y>=r.pt1.y) && (p.y < r.pt2.y);}
```

*Is the mouse cursor “pointing at” that icon so that we can click?*

When working with large structures, it is much more efficient to use a pointer to a structure, e.g.,

```c
struct point origin, *pp;

pp=&origin;
printf("origin is (%d, %d)\n", (*pp).x, (*pp).y);
```

My personal preference is the following: 😊

```c
printf("origin is (%d, %d)\n", pp->x, pp->y);
```

Something else
An application

**Question:** What does the following do?

```c
int global_positives=0;
struct List {
    struct List* next;
    double val;
};

void count_positives(List * l){
    List * p;

    /*When will it wrap up?*/
    for(p=l; p; p=p->next)
        if(p->val>0.0)
            ++global_positives;
}
```

**Answer:** It counts the number of positive `val` stuff as contained in `l`, a list of structures.

It returns 35 for this example.
Remember register?

We talked about it on Page 28 of the Function notes.

If you use some “optimizing” compiler, including gcc, to compile the above segment, it will turn the above function into the following:

```c
void count_positives(List * l){
    List * p;
    register int r;

    r=global_positives;
    for(p=l; p; p=p->next)
        if(p->val>0.0)
            ++r;
    global_positives=r;
}
```

Instead of repeatedly accessing `global_positives` from the memory, we do it just twice, thus saving time when `l` is really long and contains many positive numbers. 😊
Let’s check out an application that counts the occurrence of each of the C keywords, i.e., their frequencies.

We need a list of records, each of which contains the keyword itself and its frequency, in a file.

```c
struct key {
    char * word;
    int count;
} keytab[NKEYS];
```

This is the same as

```c
struct key {
    char * word;
    int count;
};
struct key keytab[NKEYS];
```

It can be used to construct a Huffman tree (Course page) in CS 3221 Algorithm Analysis.
Initialization

We can then initialize the keytab array as follows:

keytab[]= {
"auto", 0,
"break", 0,
"case", 0,
"char", 0,
"const", 0,
"continue", 0,
"default", 0,
/* all those in between */
"unsigned", 0,
"void", 0,
"volatile", 0,
"while", 0
}

It looks like the following at this point:

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>…</th>
<th>?</th>
</tr>
</thead>
<tbody>
<tr>
<td>keytab</td>
<td>“auto”</td>
<td>“break”</td>
<td>“case”</td>
<td>…</td>
<td>“while”</td>
</tr>
<tr>
<td>p</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>…</td>
<td>0</td>
</tr>
</tbody>
</table>

For the pointer p, wait until Page 11.
The program

#include <stdio.h> 
#include <ctype.h> \For isalpha(1) 
#include <string.h> 
#define MAXWORD 100

int getword(char *, int);
int binsearch(char *, struct key *, int);

main(){
  int n;
  char word[MAXWORD];

  while(getword(word, MAXWORD)!=EOF){
    //Is word a "word"?
    if(isalpha(word[0]))
      //Where is it?
      if((n=binsearch(word, keytab, NKEYS))>=0)
        //Its freq goes up by one....
        keytab[n].count++;
      //Show off
      for(n=0; n<NKEYS, n++)
        if(keytab[n].count>0)
          printf("%4d %s\n", keytab[n].count, keytab[n].word);
    return 0;
  }

Check out, in §6.3 of the textbook, and the course page, for details of getword (char *, int) and binsearch(char *, struct key[], int).
Show me your other face...

We can simplify the above program using a pointer to the keytab structure array as follows:

```c
main()
{
    int n;
    char word[MAXWORD];

    while(getword(word, MAXWORD)! = EOF)
        if(isalpha(word[0]))
            if((p = binsearch(word, keytab, NKEYS))!=NULL)
                p->count++;
        for(p = keytab; p<keytab+NKEYS, p++)
            if(p->count>0)
                printf("%4d %s\n", p->count, p->word);
    return 0;
}

Notice that, in this case, the return type of binsearch is no longer int, but a pointer to a key structure.

For the initial position of the pointer p in for loop, check out the image on Page 9.
Labwork

1. Come up with a complete list of all the reserved key words of C. How many are there? 😊 (1)

2. Complete the keytab table, as partly shown in page 9. (0.5)

3. Come up with a pointer version of the program, where the main() is drafted in Page 11; and make the program work that sends in a correct frequency table of all the collected reserved words that occur in your program. (3)

4. Which reserved word is the most frequently used, and which is the least frequently used? (0.5)
A little piece

C provides a facility to create a new data type name, which might be handy. For example, the following makes Length a synonym for int.

typedef int Length;

We can then use it to declare some variables.

Length len, maxlen;
Length *length[];

We can certainly make something else, e.g.,

typedef char * String;

then, we can say

String p, linePtr[MAXLINES], alloc(int);
int strcmp(String, String);
p=(String) malloc(100);

It is just a string substitution as we saw earlier with #define (Cf. Page 38 of Function notes).
Another example

A list has only one successor, while a binary tree could have at most two, a left child and a right child. We can thus define the following tree structure.

typedef struct tnode {
    char *word;
    int count;
    treeptr left;
    treeptr right;
} Treenode;

tnode * treePtr;

This binary search tree is a much better choice to count the occurrences of reserved words, as we discuss in CS 3221.