1 Working with Pointers and Structures
Pointers can be defined to point to structures, just as with other data types. Using the `date` structure example,

```c
struct date
{
    int month;
    int day;
    int year;
};
```

you declare the following structure:

```c
struct date todaysDate;
```

then you can declare a pointer of that type:

```c
struct date *datePtr;
```

and set that pointer to point to the `todaysDate` variable declared above:

```c
datePtr = &todaysDate;
```

to access a member of that structure, like the day member, the syntax looks like the following:

```c
(*datePtr).day = 21;
```

the parentheses are required because the structure member operator `.`, has a higher precedence than the indirection operator `*`.

Pointers to structures are used so often in C that a special operator exists in the language: `->`

so the previous statement above:

```c
(*datePtr).day = 21;
```

can be rewritten as:

```c
datePtr->day = 21;
```

2 Structures Containing Pointers
A structure can also contain a pointer as one of its members. For example:

```c
struct intPtrs
{
    int *p1;
    int *p2;
};
```

is a structure called `intPtrs` that contains two integer pointers. You can declare a variable of type `struct intPtrs` in the usual way:

```c
struct intPtrs pointers;
```

This variable `pointers` (a structure that has two pointers) can now be used in the normal fashion, such as:

```c
int i1=100, i2;
pointers.p1 = &i1;
pointers.p2 = &i2;
*pointers.p2 = -97;
```
3 Linked Lists
The concepts of pointers to structures and structures containing pointers are very powerful ones in C because they enable you to create sophisticated data structures, such as linked lists, doubly linked lists, and trees. Suppose you define a structure as the following:

```c
struct entry
{
    int value;
    struct entry *next;
};
```

The first member is a simple integer; the second one is a pointer to another entry structure.

Suppose you define two variables to be of type struct entry as follows:
```
struct entry n1, n2;
```

Then you can set the next pointer of n1 to point to n2 like this:
```
n1.next = &n2;
```

which makes a link between the two structures.

You can also declare a variable n3 and link to it from n2, like this:
```
struct entry n3;
n2.next = &n3;
```

which would result in this chain of linked structures, a "linked list":

To assign values to these structs, you could do the following:
```
n1.value = 100;
n2.value = 200;
n3.value = 300;
```

additional code:
```
int i = n1.next->value;  // assigns 200 to i
printf("%i
", n2.next->value);  // would print 300
```

4 Why Linked Lists?
The power of linked lists becomes more apparent when dealing with large lists by greatly simplifying operations such as the insertion and removal of elements from large sets of sorted items. For example, with the above example, if you wanted to remove n2 from the linked list, just simply do the following:
```
n1.next = n2.next;  // safer, more general way
```

or
```
n1.next = &n3;  // but, this assumes that you know that n2 points to n3
```

Either way, n2 has now been effectively removed from the list.
Inserting a new element is just as straightforward. If you want to insert a struct entry called n2_3 after n2 in the original list, you can simply set n2_3.next to point to what n2.next was pointing to, and then set n2.next to point to n2_3, like the following:

```
n2_3.next = n2.next;
n2.next = &n2_3;
```

Note that the sequence of the preceding two statements is important. The second statement makes n2 point to n2_3, and if that statement was executed first, then you wouldn’t be able to then make n2_3 point to n3 because you would have just lost that link.

Notice that these entries are not necessarily next to each other in order (especially when inserting items). This is one of the main motivations of using linked lists. If you were to use arrays instead, the elements of an array need to be stored sequentially in memory, so insertion and removal of items to/from large lists can be more difficult and time consuming.

5 Two More Issues With Linked Lists

Usually associated with a linked list is at least one pointer to the list. Often, a pointer to the start of the list is kept. So, for the example above of the list with n1, n2, and n3, you could define a variable called list_pointer and set it to point to the beginning of the list with the statement:

```
struct entry *list_pointer = &n1;
```

The second issue involves the idea of having some way to identify the end of the list. This is needed so that a function that searches through the list, for example, can tell when it has reached the final element in the list. By convention, a constant value of 0 is used and is known as the null pointer. You can use the null pointer to mark the end of the list by storing this value in the pointer field of the last entry of the list.

```
n3.next = (struct entry *) 0;
```

The type cast operator is used to cast the constant 0 to the appropriate type (“pointer to struct entry”). It’s not required, but it makes the statement more readable.

A while loop to print each value in the linked list to the right might look like the following:

```
while (list_pointer != (struct entry *) 0) {
    printf ("%i \n", list_pointer->value);
    list_pointer = list_pointer->next;
}
```