Structures

- C structures are similar to records in Pascal, they allow us to collect
together several pieces of related data into one block. The individual
pieces of data are called structure elements or structure members.

- The declaration of a structure does not allocate memory, it just provides a
template for the structure. It specifies the quantities that are being
grouped together.

- For example, we could have the following for the declaration of a name
structure

```c
struct Person {
    char* first;
    char* last;
};
```

- This structure has two elements, the `first` and `last`, which are
used to access an array of characters. Other variables can have the same
name as structure elements, and the same element name can be used in
different structure declarations. You will not find this to be a problem,
because the compiler recognizes all potential ambiguities.

- The. (dot) operator is used to extract the individual elements from a
structure variable. In the case of our Person structure we have:

```c
char* FirstName;
char* LastName;
FirstName = andrew.first
LastName = andrew.last
```

- A slightly different syntax is used for pointers to structures, for example

```c
struct Person* ptr;
```

- The variable `ptr` is now a pointer to a structure of type `Person`. From our
previous knowledge about pointers we can use the following to get the
value of the first element of the structure that is accessed via `ptr`
either

```c
(*ptr).first
```
or

```c
ptr->first
```

- Thus `(*ptr).first` is equivalent to `ptr->first`.

- There are several ways to create a variable that has a structure type

```c
struct StructureName VariableName;
```

- So with our example Person structure (struct Person) we could create two
instances, fred and andrew:

```c
struct Person fred, andrew;
```

This declaration creates two variables (called `fred` and `andrew`) and allocates
space for them. Each variable (structure) has two elements, each element
points to an array of characters that may be used to hold the first and last
names, respectively.

We use the . (dot) operator to form individual instances of a structure
element. For example:

```c
fred.first = "Fred";
fred.last = "Flintstone";
```

equally one might write

```c
andrew = {"Andrew", "Choi"};
```

- The -> operator takes a pointer to a struct, follows the pointer to the
structure value and then extracts the field--this is a shorthand, but it
makes sense.

- We can have arrays of structures, just as we can have arrays of any other
data type. This is often quite convenient, and is done in the following way:

```c
struct StructureName VariableName [ size ];
```

- In the case of our Person structure, we could have an initialized array of
names formed in the following way:

```c
struct Person People[ ] = {
    { "Andrew", "Choi" },
    { "Fred", "Flintstone" },
    
    { NULL, NULL }
};
```

- We can use an explicit NULL value to indicate the end of the array, if we
wish.
• We can include pointers to a structure within the declaration of the structure. We might use this technique to build linked lists and binary trees.

• We can use the following structure declaration for a node in a binary tree:

```c
typedef struct Tnode* NodePtr;

struct Tnode {
    NodePtr left;
    char data;
    NodePtr right;
};
```

```c
NodePtr Tree;
```

• Note that `left` and `right` must be pointers to structures, they cannot be structure variables, otherwise we will have a structure that includes two copies of itself.

```
typedef struct Tnode* NodePtr;

struct Tnode {
    struct Person* Nameptr;
    NodePtr left;
    NodePtr right;
};
```

```
NodePtr Tree;
```

// Creates only the entry node Tree, not the sample structure below.

```
Tree

Person_1

Person_2

Person_3
```

Commonly the data item itself is a pointer to another structure or string. Consider:

```
Branch

NULL data NULL

NULL data NULL

NULL data NULL
```