Structures in C

An array allows you to store related variables of the same type in the same essential structure. But what if you wanted to store related information in the same structure that was NOT of the same type. For example, consider the situation where you want to store information about a block that has a number on it, a letter on it, and is painted a particular color.

C allows you to declare a struct. Once you do so, you can create variables of this type. Here is the general syntax:

```
struct <struct_name> {
   <type1>      <var1>;
   <type2>      <var2>;
   ....
   <typen>      <varn>;
};
```

Here is how you would declare a struct for a block:

```
struct block {
    int number;
    char letter;
    char color[15];
};
```

Typically, you would include this definition at the top of your file, after your #defines.

Once you declare a type, you may declare variables of that type. Let's looks at a small example using the block type:

In order to access a component of a record, you have to use the . operator. Thus, if the name of your block is first, as it is below, you can access the number component of THAT specific block by the expression first.number.
Let's look at an example utilizing this syntax:

```
#include <stdio.h>

struct block {
    int number;
    char letter;
    char color[15];
};

int main() {

    struct block first;

    printf("Enter the number on the first block.\n");
    scanf("%d", &first.number);
    printf("Enter the letter on the first block.\n");
    scanf("%c", &first.letter);
    printf("Enter the color of the first block.\n");
    scanf("%s", &first.color);

    printf("Block info: %d %c %s\n", first.number, first.letter,
                                                first.color);
}
```

Although we only have simple components for this particular record, there is no reason why one of the components of a record couldn't be another record, or an array, for example. All the syntax would work accordingly. Keep in mind that every expression such as first.number acts EXACTLY like an integer variable.

Alternate Syntax for Defining a struct

Since it's somewhat cumbersome to type "struct block" every time one wants a variable of type struct block, a typedef allows us to shorten our struct definition so that we just have to use the name that we give the struct and not the keyword struct as well. Here how we'd do it for a struct block:

```
typedef struct block {
    int number;
    char letter;
    char color[15];
} block;
```

In main we could declare a block variable as follows:

```
block myBlock;
```

Passing a struct into a function

This works exactly like passing a normal variable into a function, in most respects. You may pass a struct into a function as a pass by value OR pass by reference parameter. The syntax works accordingly.

One piece of information that is useful for functions is the syntax used when dealing with a pointer to a struct instead of the stuct itself. For example, if p is a pointer to a block, instead of a block itself, to access one of the fields of the block that p was pointing to, we would write:

p->number, instead of (*p).number. This, along with other mechanics of passing structs into functions is illustrated in the example below:

```
// Arup Guha
// Used 10/22/01, 11/17/03, 4/1/04 for C.
// Program Description: A short example to
// illustrate the use of structs. The struct
// created is a set of blocks. Each block is
// initialized and then compared with each
// other to see if there are any duplicates.
#include <stdio.h>

struct block {
  int number;
  char letter;
  char color[15];
};

void setup(struct block *b);
void print_block(struct block b);
int equal(struct block a, struct block b);

#define SIZE   3

int main() {
  int i, j;
  struct block my_set[SIZE];

  // Initialize all blocks
  for (i=0; i<SIZE; i++) {
    setup(&my_set[i]);
    print_block(my_set[i]);
  }

// See if any pair are equal
  for (i=0; i<SIZE; i++) {
    for (j=i+1; j<SIZE; j++) {
      if (equal(my_set[i],my_set[j]))
```

```
        printf("Blocks %d and %d are
                identical.\n",i,j);
    }
  }
}

// Post-condition: b will be initialized
// with information entered by the user.
void setup(struct block *b) {
  printf("Enter number, letter & color\n");
  scanf("%d %c %s", &(b->number),
       &(b->letter), &(b->color));
}

// Post-condition: b's components will be
// printed out.
void print_block(struct block b) {
  printf("%d %c %s\n", b.number, b.letter,
                       b.color);
}

// Post-condition: Will return 1 if each
// corresponding component of blocks a and b
// are equal, 0 otherwise.
int equal(struct block a, struct block b) {

  if (a.number == b.number &&
      a.letter  == b.letter &&
      !strcmp(a.color, b.color))
    return 1;
 else
    return 0;
}
```

In this program there are a few important things to notice:

1) Only the setup function takes in a block by reference. It needs to do so so that changes made in the function to the block are reflected in the actual block passed to the function.

2) Since the other functions take a block (or blocks) in by value, right when the function is called, the entire block is COPIED into the formal parameter. Thus, changes made in the function are NOT reflected in the actual object passed into that function.

3) If a struct is large, it is customary to pass that struct into a function by reference. The reason for this is that the "copying" step for pass by value parameters that occurs implicitly becomes very, very expensive, whereas copying a pointer is very quick!

4) There are times when attempting to pass a large struct with varied components (which may also be structs) by value does not work perfectly. In essence, it's possible that a perfect copy of the struct in question is not made. This is another reason why people often prefer passing structs by reference.

5) Passing in structs by value works fine for relatively smaller structs. It ensures that changes will not be made to the actual struct passed to the function. *Another way to ensure this is to make the parameters const parameters.* (This is illustrated in the example block2.c)

Scrabble Example

In this example, we will have a struct that stores a single tile from the game of Scrabble. We will declare an array of these structs to represents the tiles on one person's tray. We will simply read these in and then print them out, along with their aggregate score. In this example, pay attention to the use of arrays of structs and accessing components of the individual structs.

```
#include <stdio.h>

struct tile {
  char letter;
  int score;
};

void printTiles(struct tile alltiles[],
                int length);
void printScore(struct tile alltiles[],
                int length);

int main() {

  int index;
  struct tile mine[7];
  // Read in all tiles from user.
  for (index=0; index<7; index++) {
    scanf("%c", &mine[index].letter);
    scanf("%d", &mine[index].score);
  }
  printTiles(mine, 7);
  printScore(mine, 7);
  return 0;
}
```

```
// Precondition: The length of the array
// alltiles is length and each tile is
// already initialized.
// Postcondition: All the letters on the
// tiles will be printed with no spaces
// followed by a newline character.
void printTiles(struct tile alltiles[],
                int length) {

  int index;
  for (index=0; index<length; index++)
    printf("%c", alltimes[index].letter);
  printf("\n");
}

// Precondition: The length of the array
// alltiles is length and each tile is
// already initialized.
// Postcondition: The sum of the scores of
// all the tiles will be printed.
void printScore(struct tile alltiles[],
                int length) {

  int index, sum=0;
  for (index=0; index<length; index++)
    sum+=alltimes[index].score;
  printf("Score = %d\n", sum);
}
```