Overview of Structures

We've used arrays to provide a convenient way to store and manipulate large data sets. But arrays work only if all the data is homogeneously of the same data type. In most physical settings, the data that represents an object or a set of information has heterogeneous, multiple data types. For example, hurricanes are given names, and they are categorized by the intensity of their winds. Thus, to represent a hurricane, we might include the hurricane's name, the year in which it has occurred, and its intensity category. A character string could represent the name, and integers could represent the year and intensity. A *structure* defines a set of data, but the individual parts of the data do not have to be the same type. Thus, a structure can be defined to represent a hurricane as follows:

```c
struct Hurricane {
  char name[16];
  int year;
  int category;
};
```

We can now define variables, and even arrays, of type `struct Hurricane`. Each variable or array element would contain three data values - a character string and two integers.

Structures are often called aggregate data types, because they allow multiple data values to be collected into a single data type. Individual data values, within a structure are called *data members*, and each data member is given a name. In the previous example, the names of the data members are `name`, `year`, and `category`. A data member is referenced using the structure variable name followed by a period (called the *structure member operator*) and a data member name. Note the difference between referencing a value in a structure and in an array. A value in an array is referenced with the array name and a subscript while a value in a structure is referenced with the structure variable name, the structure member operator, and a data member name.

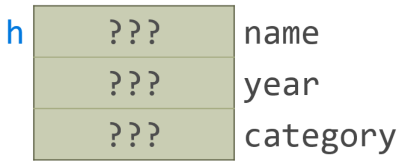
Definition and Initialization

To use a structure in a program, you must first declare the structure. The keyword `struct` is used to declare the name of the structure (also called the *structure tag*) and the data members that are included in the structure. After the structure has been defined, structure variables can be defined using definition statements. Consider the earlier definition for a structure representing a hurricane. The name of the structure type is `Hurricane`. The three data members are `name`, `year`, and `category`. Note that a semicolon is required after the structure definition. A structure declaration can appear in a source file but most often shows up in a header file. It is important to note that these statements don't reserve any memory - they only *declare* the structure.

To define a variable of this structure type, we use the following definition statement in a source file:

```c
struct Hurricane h;
```

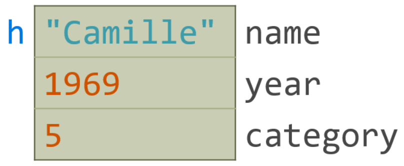
The preceding statement defines a structure variable `h` thereby allocating memory for the three data members of variable `h`. Structure variable `h` has been defined but not initialized and thus the initial values of the three data members are unknown.



The data members of a structure can be initialized in a declaration statement or with assignment statements. To initialize a structure with a declaration statement, the values are specified in a comma-separated sequence delimited by curly braces. The following declaration statement defines and initializes structure variable `h`:

```c
struct Hurricane h = {"Camille", 1969, 5};
```

The data members of `h` are now initialized:



The members of structure variable `h` can be assigned values using assignment statements. To reference an individual data member, we use the structure variable name and the data member name separated by the structure member operator `.`:

```c
strcpy(h.name, "Camille"); // copy string "Camille" to h.name
h.year = 1969;
h.category = 5;
```

Practice

Consider the following structure:

```c
struct Hurricane {
  char name[10];
  int year, category;
};
```

Show the contents of the data members of the structures defined in each set of statements:

1.
   ```c
   struct Hurricane h1 = {"Andrew", 199, 5};
   ```

2.
   ```c
   struct Hurricane h2;
   ```

3.
   ```c
   struct Hurricane h3;
   h3.name = "Hugo";
   ```

Suppose you're working as an intern at your local observatory to develop a database of the planets in our solar system. For each planet, you need to represent the following information: *Name*: Jupiter; *Diameter*: 142,800 km; *Moons*: 16; *Orbit time*: 11.6 years; *Rotation time*: 9.925 hours. Declare a structure type to encapsulate this information in a single variable.

```
// declare structure here
```

Declare a structure for a part encapsulating a character array of 30 elements for part name, an integer for the part number, a floating-point value for price, an integer for the current stock.

```
// declare structure here
```

Declare a structure that will be used to encapsulate US addresses. It will consist of four character arrays: an array of 25 elements to represent street address, an array of 20 elements to represent the city name, an array of 3 elements to represent the state, and an array of 6 elements to represent the postal code.

```
// declare structure here
```

Declare a structure to represent a person with an array of 15 elements to represent the person's first name, an array of 15 elements to represent the person's last name, and the address structure (from previous question) to represent the person's home address.

```
// declare structure here
```

Input and Output

We can use `scanf` or `fscanf` statements to read values into data members of a structure and `printf` or `fprintf` to print the values of the data members. In the following code fragment, the information for a group of hurricanes is read from a file named `storms.txt` and the information is printed to standard output:

```c
#include <stdio.h>  // for fopen, fclose, printf, fscanf
#include <stdlib.h> // for exit

// declare struct Hurricane as before ...

// open text file FILENAME
#define FILENAME "storms.txt"
FILE *stream = fopen(FILENAME, "r");
if (NULL == stream) {
  printf("Error opening data file %s\n", FILENAME);
  exit(EXIT_FAILURE); // shutdown program
}

// read and print information from the file
struct Hurricane h;
while (3 == fscanf(stream, "%d %d %s", &h.year, &h.category, h.name)) {
  printf("Hurricane %s: ", h.name);
  printf("Year: %d, Category: %d\n", h.year, h.category);
}
fclose(stream); // close file
```

Note that the reference in the `fscanf` statement for hurricane name is `h.name` instead of `&h.name`. Since `name` is an array member, `h.name` evaluates to a pointer to the first element of the array.

Assume file `storms.txt` contain the following first few lines of text:

```
1954   4   Hazel
1957   4   Audrey
1960   4   Donna
```

then the first few lines for a sample run of the program would appear like this:

```
Hurricane: Hazel Year: 1954, Category: 4
Hurricane: Audrey Year: 1957, Category: 4
Hurricane: Donna Year: 1960, Category: 4
```

Short note on reading character strings with `scanf`

A problem arises if oceanographers name future hurricanes with two words, as in `Alice Bertha`. In this scenario, file `new-storms.txt` would contain lines with the following text:

```
2002 5 Alice Bertha
2004 4 Kathy Bill
```

The problem arises from the fact that `scanf` or `fscanf` functions read character strings with `%s`, where a character string is defined as a sequence of characters up to a whitespace (space, tab, newline) character. Hence the following call to `fscanf`

```c
struct Hurricane h;
while (3 == fscanf(stream, "%d %d %s", &h.year, &h.category, h.name)) {
  printf("Hurricane %s: ", h.name);
  printf("Year: %d, Category: %d\n", h.year, h.category);
}
```

would fail for the line with text `2004 4 Kathy Bill`. When reading the previous line `2002 5 Alice Bertha`, integers `2002` and `5` will be correctly formatted while the default behavior for `%s` would store partial name `"Alice"` in array `h.name` and not the entire name `"Alice Bertha"` because of the space between the two words. This means the characters in string `"Bertha"` would be read by the `%d` specifier in the next iteration causing `fscanf` to abort and return `0` (since no values were correctly formatted).

This unhappiness with the definition of a character string as far as `scanf` or `fscanf` are concerned can be changed with the special `[...]` and `[^...]` conversion characters. In the first case, the characters listed between the `[` and the `]` define *all of the valid characters in the string*. `scanf` will start reading characters and store them into the character array *until* a non-listed character is encountered on input. Ranges of characters can be abbreviated by placing a `-` between the first and last characters in the range. So, for example, the statements

```c
char letters[100];
scanf("%[a-z]", letters);
```

tell `scanf` to read characters from standard input and store them into array `letters` until a non-lowercase character is encountered. The call

```c
scanf("%[a-zA-Z]", letters);
```

is similar, except in this case `scanf` will read and store characters into `letters` until a non-alphabetic character is read.

In the second case, the `^` character that immediately follows the `[` tells `scanf` that the remaining characters listed between the brackets are to be considered the *string terminator* characters. In other words, `scanf` will continue to read characters from standard input and store them in your array until it encounters any one of the characters listed in the brackets. The `scanf` call

```c
scanf("%[^,.:]", letters);
```

says to read characters until a comma, period, or semicolon is encountered, and to store all such characters read into `letters`. The call

```c
char line[100];
scanf("%[^\n]", line);
```

tells `scanf` that the only delimiter character for this read is a newline character. Therefore, `scanf` will read and store characters inside `line` until a newline is read (which will not be stored). Therefore, the correct way to read hurricane names in `new-storms.txt` would be to replace the `%s` conversion character with `[^\n]`, as in the following code fragment:

```c
struct Hurricane h;
while (3 == fscanf(stream, "%d %d %[^\n]", &h.year, &h.category, h.name)) {
  printf("Hurricane %s: ", h.name);
  printf("Year: %d, Category: %d\n", h.year, h.category);
}
```

Practice

Assume that the structure variables `h1` and `h2` have been defined with the following statements:

```c
struct Hurricane {
  char name[10];
  int year, category;
};

int main(void) {
  struct Hurricane h1 = {"Audrey", 1957, 4}, h2 = {"Frederic", 1979, 3};
  // show the output for each of the following sets of statements:

  printf("%s \n%s \n", h2.name, h1.name); // answer:

  printf("Category %d hurricane: %s\n", h1.category, h1.name); // answer:
}
```

Computations

We've seen that the structure member operator `.` is used with the name of the structure variable to access individual data members of the structure. When the name of the structure variable is used without the structure member operator, it refers to the entire structure. The assignment operator can be used with structure variables of the same type to assign an entire structure to another structure, as shown in this code fragment:

```c
struct Hurricane h = {"Audrey", 1957, 4}, h2;
// to assign is to copy entire memory block of storage for h to h2
h2 = h;
```

However, relational operators cannot be applied to an entire structure. To compare one structure with another, you must compare individual data members. This is shown in the following code fragment:

```c
// read all hurricanes from file but print about category 5 hurricanes
while (3 == fscanf(stream, "%d %d %s", &h.year, &h.category, h.name)) {
  if (5 == h.category) {
    printf("%s\n", h.name);
  }
}
```

Practice

Use the following structure definition to answer the problems listed below.

```c
struct computer {
  char manufacturer[10];
  double price;
  int speed;
};
struct computer pc1, pc2;
```

1. Provide the correct declaration of a function that is called to read values from standard input and store them in data members of structure variable `pc1`:

   ```

   ```

2. Provide the correct declaration of a function that is called to print values of data members of structure variable `pc1`:

   ```

   ```

3. Write the correct statement that assigns a value of $800 to data member `price` of structure variable `pc1` is:

   ```

   ```

4. Define and initialize a pointer with the address of structure variable `pc1`:

   ```

   ```

Using Functions with Structures

Structures can be used as arguments to functions, and functions can return structures. Each of these cases is considered separately.

Structures as Function Arguments

Entire structures can be passed as arguments to functions. When a structure is used as a function argument, the value of each data member of the argument will be the initializer of the corresponding data member of the function parameter. Thus, changing the value of a parameter doesn't change the corresponding argument. Remember, arguments are expressions while parameters are variables that are initialized with the values obtained by evaluating the expressions. The code fragment in the previous section is modified to print information for a hurricane from a function:

```c
// declare and define function print_hurricane
void print_hurricane(struct Hurricane h) {
  printf("Hurricane %s: ", h.name);
  printf("Year: %d, Category: %d\n", h.year, h.category);
}

// read and print information from the file
struct Hurricane h;
while (3 == fscanf(stream, "%d %d %s", &h.year, &h.category, h.name)) {
  print_hurricane(h);
}
```

When functions are written to modify the value of a data member within a structure, we must use a pointer to the structure as the function argument. This allows the function direct access to the data members of the structure. When data members are accessed via a pointer to the structure, the right arrow selection operator `->` is used instead of the structure member operator. The following function takes a pointer to a variable of type `struct Hurricane`, reads information entered from the keyboard, and saves it in the structure variable using the structure pointer operator `->`:

```c
// declare and define function get_info_ptr to read hurrican parameters
void get_info_ptr(struct Hurricane *ptr) {
  printf("Enter hurricane information in following order:\n");
  printf("Enter year and hurricane category: ");
  scanf("%d %d", &ptr->year, &ptr->category);
  printf("Enter location (<30 characters, no whitespace): ");
  scanf("%s", ptr->name);
}

struct Hurricane h;
get_info_ptr(&h);
print_hurricane(h);
```

The following code fragment rewrites the `print_hurricane` function to take a pointer to a structure as its parameter:

```c
// notice that ph is pointer to read-only structure variable ...
void print_hurricane_ptr(struct Hurricane const *ph) {
  printf("Hurricane %s: \n", ph->name);
  printf("Year: %d, Category: %d\n", ph->year, ph->category);
}

// read and print information from the file
struct Hurricane h;
get_info_ptr(&h);
print_hurricane_ptr(&h);
```

Functions that Return Structures

A function can be defined to return a value of type `struct`. After the function is called, an entire structure is returned to the calling function. To illustrate, we rewrite function `get_info_ptr` as function `get_info` that reads information entered from the keyboard, saves it in the structure, and then returns the information to the caller:

```c
struct Hurricane get_info(void) {
  struct Hurricane h;

  printf("Enter hurricane information in following order:\n");
  printf("Enter year and hurricane category: ");
  scanf("%d %d", &h.year, &h.category);
  printf("Enter location (<30 characters, no whitespace): ");
  scanf("%s", h.name);

  return h;;
}
```

Practice

Let `color` be the following structure:

```c
struct color { int red; int green; int blue; };
```

Define a `const` variable named `MAGENTA` of type `struct color` whose members have the values 255, 0, and 255, respectively.

```c


```

Define the following function that returns a `color` structure containing the specified red, green, and blue values. If any argument is less than zero, the corresponding member of the structure will contain zero instead. If any argument is greater than 255, the corresponding member of the structure will contain 255.

```c
struct color make_color(int red, int green, int blue);




```

Define the following function that returns the value of `c`'s `red` member.

```c
int get_red(struct color c);



```

Define the following function that returns `true` if the corresponding members of `color1` and `color2` are equal.

```c
bool equal_color(struct color color1, struct color color2);




```

Define the following function that returns a `color` structure that represents a brighter version of the color `c`. The structure is identical to `c`, except that each member has been divided by 0.7 (with the result truncated to an integer). However, there are three special cases: (1) If all members of `c` are zero, the function returns a color whose members all have the value 3. (2) If any member of `c` is greater than 0 but less than 3, it is replaced by 3 before the division by 0.7. (3) If dividing by 0.7 causes a member to exceed 255, it is reduced to 255.

```c
struct color brighter(struct color c);










```

Define the following function that returns a `color` structure that represents a darker version of the color `c`. The structure is identical to `c`, except that each member has been multiplied by 0.7 (with the result truncated to an integer).

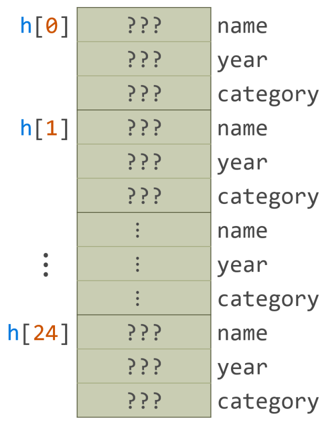
```c
struct color darker(struct color c);









```

Arrays of Structures

In many applications, it is often convenient to store information in arrays for analysis. However, an array can contain only a single type of information, such as an array of integers or an array of character strings. If we need to use an array to store information that contains different types of values, then we can use an array of structures. For example, if we want to store an array of information for hurricanes, we can use an array of the structure type `struct Hurricane`; if we need to store an array of information on tsunamis, we can use an array of the structure type `struct Tsunami`. We can write a statement to define an array of 25 elements, each of type `struct Hurricane`:

```c
struct Hurricane h[25];
```

Each element in the array is a structure containing the three variables, as illustrated in the following picture:



To access an individual data member of a structure in the array, we must specify the array name, a subscript, and the data member name. As an example, we'll assign values to the first element of type `struct Hurricane` in array `h`:

```c
strcpy(h[0].name, "Camille");
h[0].year = 1969;
h[0].category = 5;
```

To access an entire structure within the array, we must specify the name of the array and a subscript. As an example, we can call the previously defined output function `print_hurricane`. We'll print the first hurricane in the array with this statement:

```c
print_hurricane(h[0]);
```

The output would be

```
Hurricane: Camille
Year: 1969, Category: 5
```

Instead of passing the structure by value, we can print the first hurricane in the array with this statement:

```c
print_hurricane_ptr(&h[0]);
```

Since `&h[0]` and `h` both evaluate to address of first element of array `h`, the previous statement can also be written as:

```c
print_hurricane_ptr(h); // pass address of first element of array h
```

We'll now present code that reads information for a group of hurricanes from a data file into an array. This code fragment determines the maximum category in the array and then prints the names of all hurricanes with the maximum category. It should be clear that we cannot print this information as we read the data file. We will not know the maximum category until we've reviewed all the information in the file.

```c
// define required variables here
struct Hurricane h[100];
FILE *stream;
#define FILENAME "storms.txt"

```

```c
// insert code from before to open text file storms.txt

// read information from the file and keep track of
// maximum hurricane category encountered so far ...
printf("Hurricanes with Maximum Category\n");
int k = 0, max_cat = 0;
while (3 ==
       fscanf(stream, "%d %d %s", &h[k].year, &h[k].category h[k].name)) {
  max_cat = (h[k].category > max_cat) ? h[k].category : max_cat;
  ++k;
}

// after conclusion of while loop,
// k contains number of hurricanes in file ...

// only print hurricanes with maximum category ...
for (int i = 0; i < k; ++i) {
  if (h[i].category == max_category) {
    print_hurricane_ptr(&h[i]);
  }
}
```