Agenda

C Structures
Introduction

- Structures, sometimes referred to as aggregates, are collections of related variables under one name. Structures may contain variables of many different data types, in contrast to arrays, which contain only elements of the same data type.

- Structures are commonly used to define records to be stored in files. Pointers and structures facilitate the formation of more complex data structures such as linked lists, queues, stacks and trees.

Structure Definitions

- Structures are derived data types, they're constructed using objects of other types. Consider the following structure definition:

```
struct card{
  char *face;
  char *suit;
}; // end struct card
```

- Keyword struct introduces a structure definition. The identifier card is the struct tag, which names the structure definition and is used with struct to declare variable of the structure type, struct card.

- Variables declared within the braces of the structure definition are the structure's members. Members of the same structure type must have unique names, but two different structure types may contain members of the same name without conflict.

- Each structure definition must end with a semicolon.

Structure Definitions

Structure members can be of different types. For example, the following struct contains character array members for an employee's first and last names, an unsigned int member for the employee's age, a char membber that would contain 'M' or 'F' for the employee's gender and a double member for the employee's hourly salary:

```
struct employee{
   char firstName[ 20 ];
   char lastName[ 20 ];
   unsigned int age;
   char gender;
   double hourlySalary;
}; // end struct employee
```

Self-Referential Structures

- A structure cannot contain an instance of itself. For example, a variable of type struct employee cannot be declared in the definition for struct employee. A pointer to struct employee may be included. For example:

```
struct employee{
  char firstName[ 20 ];
  char lastName[ 20 ];
  unsigned int age;
  char gender;
  double hourlySalary;
  struct employee person; // Error
  struct employee *pointerEmployee; // pointer
}; // end struct employee
```

struct employee contains an instance of itself, which is an error. Because pointerEmployee is a pointer, it's permitted in the definition. A structure containing a member that's a pointer to the same structure type is referred to as a self-referential structure.

Defining Variables of Structure Types

- Structure definitions do not reserve any space in memory, rather, each definition creates a new data type that's used to define variables. Structure variables are defined like variables or other types.The definition

```
struct card aCard, deck[ 52 ], *cardPtr;
```

declares aCard to be a variable of type struct card, declares deck to be an array with 52 elements of type struct card and declares cardPtr to be a pointer to struct card.

- Variables of a given structure type may also be declared by placing a comma-separated list of the variable names between closing brace of the structure definition and the semicolon that ends the structure definition. For example, the preceding definition could have been incorporated into the struct card definition as follows:

```
struct card{
  char *face;
  char *suit;
} aCard, deck[ 52 ], *cardPtr; // end struct card
```

Initializing Structures

- Structures can be initialized using initializer lists as with arrays. To initialize a structure, follow the variable name in the definition with an equals sign and a brace-enclosed, comma-separated list of initializers. For example;

```
struct card aCard = { “Three”, “Hearts” };
```

creates variable aCard to be of type struct card and initializes member face to “Three” and member suit to “Hearts”. If there are fewer initializers in the list than members in the structure, the remaining members are automatically initialized to 0 (or NULL if the member is a pointer).

- Structure variables defined outside a function definition are initialized to 0 or NULL if they’re not explicitly initialized in the external definition. Structure variables may also be initialized in assignment statements by assigning a structure variable of the same type, or by assigning values to the individual members of the structure.

Accessing Structure Members

- Two operators are used to access members of structures: the structure member operator (.), also called the dot operator, and the structure pointer opeartor, (->), also called the arrow operator.

- The structure member operator accesses a structure member via the structure variable name. For example, to print member suit of structure variable aCard defined in the previous section, use the statement

```
printf( "%s", aCard.suit );
```

- The structure point operator, consisting of a minus (-) sign and a greater than (>) sign with no intervening spaces, accesses a structure member via a pointer to the structure. Assume that the pointer cardPtr has been declared to point to struct card and that the address of structure aCard has been assigned to cardPtr. To print member suit of structure aCard with pointer cardPtr, use the statement

```
printf( "%s", cardPtr->suit );
```

- The expression cardPtr->suit is equivalent to (*cardPtr).suit, which dereferences the point and accesses the member suit using the structure member operator. The parentheses are needed here because the structure member operator (.) has a higher precedence than the pointer dereferencing operator (*).

C Structures

Operations That Can Be Performed on Structures

- The only valid operations that may be performed on structures are:

  Assigning structure variables to structure variables of the same type

  Taking the address (&) of a structure

  Accessing the members of a structure variable

  Using the sizeof operator to determine the size of a structure variable

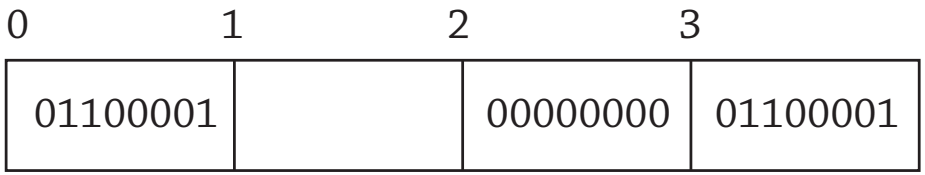
- Structures may not be compared using operators == and !=, becase structure members are not necessarily stored in consequtive bytes of memory. Because, computers may store specific data types only on certain memory boundaries such as half-word, word or double-word boundaries. Consider the following structure definition, in which sample1 and sample2 of type struct example are declared:

```
struct example{
  char c;
  int i;
} sample1, sample2;
```

Operations That Can Be Performed on Structures

- A computer with 2-byte words may require that each member of struct example be aligned on a word boundary. Figure below shows a sample storage alignment for a variable of type struct example that has been assigned the character ‘a’ and the integer 97.

- If the members are stored beginning at word boundaries, there’s a 1-byte hole in the storage for variables of type struct example. The value in the 1-byte hole is undefined. Even if the member values of sample1 and sample2 are in fact equal, the structures are not necesarrily equal, because the undefined 1-byte holes are not likely to contain identical values.



Using Structures with Functions

- Structures may be passed to functions by passing individual structure members, by passing an entire structure or by passing a point to a structure. When structures or individual structure members are passed to a function, they're passed by value.

- To pass a structure by reference, pass the address of the structure variable. Arrays of structures, like all other arrays, are automatically passed by reference.

- To pass an array by value, create a structure with the array as a member. Structure are passed by value, so the array is passed by value.

Common Programming Error

Assuming that structures, like arrays, are automatically passed by reference and trying to modify the caller's structure values in the called function is a logic error.

Performance Tip

Passing structures by reference is more efficient than passing structures by value.

- The keyword typedef provides a mechanism for creating synonyms (or aliases) for previously defined data types. Names for structure types are often defined with typedef to create shorter type names. For example, the statement

typedef struct card Card;

defines the new type name Card as a synonym for type struct card. C programmers often use typedef to define a structure type, so a structure tag is not required. For example, the following definition creates the structure type Card without the need for a separate typedef statement.

```
typedef struct{
  char *face;
  char *suit;
} Card;
```