STRUCTURE

EPROG

COMPUTER FUNDAMENTALS & PROGRAMMING FOR ENGINEERING STUDENTS

<<professor>>

STRUCTURES (STRUCT)

- A *struct* is a derived data type composed of members that are each fundamental or derived data types.

- A single *struct* would store the data for one object. An array of *structs* would store the data for several objects.

- A *struct* can be defined in several ways as illustrated in the following examples:

THE STRUCTURE TYPE

- The **structure type** allows the programmer to aggregate components into a single, named variable.
  - A structure has components that are individually named.
    - These components are called **members**.
  - The members of a structure can be of various types.
    - This allows the programmer to create aggregates of data that are suitable for each specific problem.
  - Like arrays and pointers, structures are considered a derived type.

MEMBER AND STRUCTURE POINTER OPERATORS "." AND "->"

- Members of structures are accessed using either:
  - the member operator **.** or
  - the structure pointer operator **->**
- These operators along with **()** and **[]** have the highest precedence.

DECLARING STRUCTURES (STRUCT)

DOES NOT RESERVE SPACE

```
struct my_example
{
    int label;
    char letter;
    char name[20];
};
/* The name "my_example" is called a
    structure tag    */
```

RESERVES SPACE

```
struct my_example
{
    int label;
    char letter;
    char name[20];
} mystruct ;
```

ACCESSING STRUCT MEMBERS

Individual members of a *struct* variable may be accessed using the structure member operator (the dot, "."):

```
mystruct.letter ;
```

Or , if a pointer to the *struct* has been declared **and** initialized

```
Some_name   *myptr = &mystruct ;
```

by using the structure pointer operator (the "->"):

```
myptr -> letter ;
```

which could also be written as:

```
(*myptr).letter ;
```

UNIQUENESS OF MEMBER NAMES

- A **member name** must be unique within a specified structure.
- Since the member must always be prefaced or accessed through a unique structure variable identifier, there is no confusion between members of different structures having the same name.

EXAMPLE OF SAME MEMBER NAMES IN DIFFERENT STRUCTURES

```
struct fruit {
   char  name[15];
   int   calories;
};
struct vegetable {
   char name[15];
   int calories;
}
struct fruit      a;
struct vegetable  b;
```

It can access a.calories and b.calories without ambiguity.

SAMPLE PROGRAM WITH STRUCTS

```
/* This program illustrates creating structs and then declaring
   and using struct variables.  Note that struct personal is an
   included data type in struct "identity". */
#include <stdio.h>
struct personal  //Create a struct but don't reserve space.
 {  long id;
    float gpa;
 } ;
struct identity    //Create a second struct that includes the first
   one.
 {  char name[30];
    struct personal person;
 } ;
```

SAMPLE PROGRAM WITH STRUCTS

```
int main ( )
{
   struct identity js = {"Joe Smith"}, *ptr = &js ;

   js.person.id = 123456789 ;
   js.person.gpa = 3.4 ;
   printf ("%s %ld %f\n", js.name, js.person.id,
           js.person.gpa) ;
   printf ("%s %ld %f\n", ptr->name, ptr->person.id,
           ptr->person.gpa) ;
}
```

EXAMPLES OF TWO ACCESSING MODES

DECLARATIONS AND ASSIGNMENTS

```
struct student temp, *p = &temp;
temp.grade = 'A';
temp.last_name = "Bushker";
temp.student_id = 590017;
```

EXPRESSION	EQUIVALENT EXPRESSION	CONCEPTUAL VALUE
temp.grade	p -> grade	A
temp.last_name	p -> last_name	Bushker
temp.student_id	p -> student_id	590017
(*p).student_id	p -> student_id	590017