## Enumeration Constants

- C provides a user-defined type called an *enumeration*. An enumeration, introduced by the keyword `enum`, is a set of integer constants represented by identifiers. The values in an `enum` start with 0, unless specified otherwise, and are incremented by 1.

**Example:**

```
enum months {JAN, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC};
             0    1    2    3    4    5    6    7    8    9    10   11


enum months {JAN = 1, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC};
```

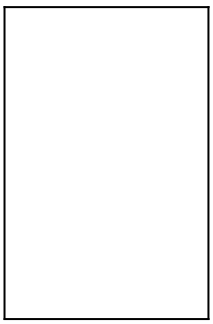
- The identifiers in an enumeration must be unique.
- The value of each constant in an enumeration can be set explicitly in the definition by assigning a value to the identifier.
- Multiple members of an enumeration can have the same constant value.

**Example:**

```
enum colors {WHITE, RED = 4, GREEN = 3, BLUE, BLACK};
              0       4          3        4     5
```

**Example:** What does the following program display?

```
#include <stdio.h>
enum days {MON = 1, TUE, WED, THU, FRI, SAT,
SUN};
int main(void)
{  int  i, j = SUN;
   for (i = MON; i <= SUN; i++)
      printf("%d %d\n", i, j--);
   return(0);
}
```



**Example:**

```
enum day            /* Defines an enumeration type    */
{   SATURDAY,       /* named day and declares a       */
    SUNDAY = 0,     /* variable named workday with    */
    MONDAY,         /* that type                      */
    TUESDAY,
    WEDNESDAY,      /* WEDNESDAY is associated with 3 */
    THURSDAY,
    FRIDAY
} workday;
```

- As in the above example, we can use the enumerations to declare variables. The variable `workday` is declared with the enumeration type `day`. We can use the name of an enumeration constant to assign a value to `workday`.

```
workday = MONDAY;   // workday becomes 1
```

- A variable of the enumeration type may also be declared later using the enumeration tag `day`.

```
enum day today = WEDNESDAY;
```

- To explicitly assign an integer value to a variable of an enumerated data type, use a type cast:

```
int day_value = 5;
workday = (enum day)(day_value - 1); //workday becomes 4
printf("workday = %d\n", workday);
```

**Example:** Define an enumeration data type called `BOOLEAN`, where `FALSE` means **0**, `TRUE` means **1**.

```
enum boolean { FALSE, TRUE }; /* FALSE = 0, TRUE = 1 */
```

*READ Sec 10.11 (pg 403 – 404) from Deitel & Deitel.*

## Structures

- A *structure* is a set of declarations which may consist of arrays or variables of mixed data types, grouped under a common name. You can define a structure to store all the necessary information about one student, one course, one book, or one inventory item.

- For instance, the structure for one student may consist of his id (an integer), his surname (a string), his name (another string), and his cgpa (a double):

  Student Information:

  - id (int)
  - surname (string)
  - name (string)
  - cgpa (double)

- The structure for one inventory item may consist of

  - item no (int)
  - item name (string)
  - quantity (int)
  - unit price (double)

- The syntax for defining a structure is as follows:

```
struct     struct_name     {
     field_type  field_name;
     field_type field_name;
     ...
} variable_name;
```

- Keyword `struct` introduces the structure definition. Then, you give a name to the structure. It is called the *structure tag*.

- Variables declared within the braces of the structure definition are the structure's *members*.

- Members of the same structure must have unique names. However, using the same names for members of different structures is allowed. But, since it may cause confusion, it is not recommended.

- Each structure definition must end with a semicolon.

- A variable of a structure type can be declared just after the structure definition or later separately, using the keyword `struct` with the structure tag.

```
struct struct_name variable_name;
```

**Example:**

```
struct student {
     int     id;
     char    surname[25];
     char    name[25];
     double  cgpa;
} ceng_student;

struct student eee_student;
```

- `student` is the structure tag. Two variables called `ceng_student` and `eee_student` are allocated in the memory.
- We could finish the structure definition just after the curly brace, without the variable name, and declare the variable `ceng_student` near `eee_student`.

**Example:**

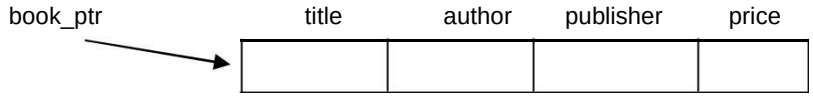
```
struct student {
     int     id;
     char    surname[25];
     char    name[25];
     double  cgpa;
}; /* no variable name, only type is defined
*/ struct student ceng_student, eee_student;
```

**Example:**

```
struct book {  /* description of a book */
     char    title[40];
     char    author[40];
     char    publisher[25];
     double  price;
};
struct book text_book, ref_book;
struct book *book_ptr;   /* points to a book structure */
```

- After the above declarations, we can assign the address of `text_book` or `ref_book` to `book_ptr`, or we can make a separate memory allocation for it, so that it will point to a book structure:

```
book_ptr = &text_book;
```




or

```
book_ptr = &ref_book;
```

or

```
book_ptr = (book *)malloc(sizeof (book));
```

- The keyword `typedef` provides a mechanism for creating synonyms for previously defined data types.

**Example:**

```
typedef int length;
...
int main (void)
{    length a, b;    // means int a, b;
```

- We can also use it to define a new data type.

**Example:**

```
typedef struct book {
     char    title[40];
     char    author[40];
     char    publisher[25];
     double  price;
} book_t;

book_t text_book, ref_book;
```

- Now, `book_t` is no longer a variable name but a synonym for the type `struct book`.

- The structure tag name is optional. If a structure definition does not contain a structure tag name, variables of the structure type may be declared only in the structure definition, not in a separate declaration, or with a synonym created with `typedef`.

- Structures may be initialized during declaration by putting the list of elements in curly braces as with arrays.

**Example:**

```
struct student {
     int     id;
     char    surname[25];
     char    name[25];
     double  cgpa;
} ceng_student = {20900555, "TEKIN", "ALI", 2.86};
```

- If there are fewer initializers in the list than members in the structure, the remaining members are automatically initialized to 0 (if it is a number), NULL (if it is a pointer), or empty string (if it is a string).

- 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.

**Example:** `text_book = ref_book;`

## Accessing Structure Members

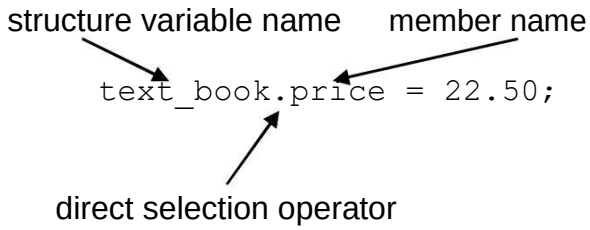
- Individual members of a structure are accessed using a structure member selection operator. There are two selection operators:

  `.` direct member selector (dot operator)

  `->` indirect member selector (arrow operator)

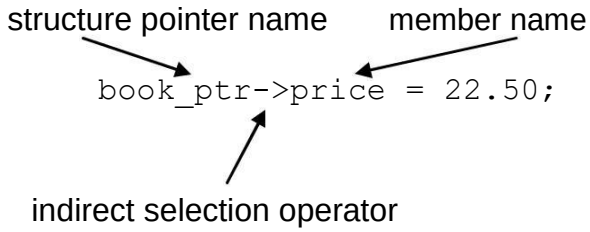
- Dot operator accesses a structure member via the structure variable name.

**Example:**




- Arrow operator accesses a structure member via a pointer to the structure.

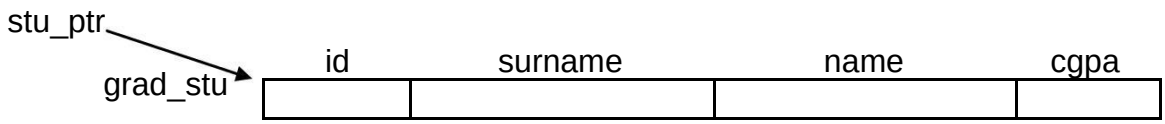
**Example:**




**Example:**

```
struct student grad_stu, *stu_ptr;

stu_ptr = &grad_stu;
stu_ptr->cgpa = 3.24; // (*stu_ptr).cgpa = 3.24;
                      // grad_stu.cgpa = 3.24;
```




- The expression `stu_ptr->cgpa` is equivalent to `(*stu_ptr).cgpa`, which dereferences the pointer and accesses the member `cgpa` using dot operator.

- Parantheses are needed because the dot operator has higher precedence than the indirection operator *****. Not using the parantheses (e.g. `*stu_ptr.cgpa`) causes a compilation error.