Define multiple elements for each object

Organised & Supported by **RuggedBOARD**

Agenda

- Structures
- Accessing Structure Members.
- Nested structure
- Pointers to Structure
- Bit Fields

Structures

Def: Collection of dissimilar data elements has shared a common name is called “a Structure

Syn:

```
struct ST-NAME
{
datatype var1;
datatype var2;
… … … … … .
};
```

or

```
typedef struct
{
datatype var1;
datatype var2;
… … … … … ..
}ST-NAME;
```

Object construction :

Syn: struct ST-NAME obj; or ST-NAME obj;

Ex:

```
struct EMP
{
int eid;
char name[20];
char desgn[20];
float sal;
};
struct EMP  e;
```

Accesssing Structure members

Accessing structure members :
We can access the data members of a structure using dot(.) operator or pointer (->) operator only.

EMP e; **or**
e.eid
e.name
e.desig
e.sal

EMP *e;
e->eid
e->name
e->desig
e->sal

Structures can be defined above the main() function Or within the main() function.
Size of the structure means sum of the sizes of the variables defined in the structure definition.

To calculate the size of the structure:
int var= sizeof(structurename or object);

Ex: S= sizeof(e);

Here, e -> is the object of EMP structure
S -> is the variable of integer data type

Structure contd..

```
#include<stdio.h>

struct Point
{
int x, y;
};

int main()
{
        struct Point p1 = {0, 1};

        // Accessing members of point p1
        p1.x = 20;
        printf ("x = %d, y = %d", p1.x, p1.y);

        return 0;
}
```

Output:

x = 20, y = 1

```
#include<stdio.h>

struct Point
{
int x, y, z;
};
int main()
{
// Examples of initialization using designated  initialization
    struct Point p1 = {.y = 0, .z = 1, .x = 2};
    struct Point p2 = {.x = 20};

     printf ("x = %d, y = %d, z = %d\n", p1.x, p1.y, p1.z);
     printf ("x = %d", p2.x);
     return 0;
}
```

Output:

x = 2, y = 0, z = 1

x = 20

Nested Structures

Nested structures:
Def: A structure with in a structure is called Nested structure.

Syn:

```
Struct INNER
{
datatype var11;
datatype var12;
.....
};

struct OUTER
{
datatype var1;
datatype var2;
struct INNER in;
};
```

```
#include <stdio.h>
struct Employee
 {
      int employee_id;
      char name[20];
      int salary;
 };
struct Organisation
{
          char organisation_name[20];
          char org_number[20];
          struct Employee emp;

};
int main()
{
        struct Organisation org;
        printf("%ld", sizeof(org));
}
```

Try this

```
#include <stdio.h>
struct Organisation
{
          char organisation_name[20];
          char org_number[20];
          struct Employee
          {
                int employee_id;
                char name[20];
                int salary;
           }emp;
};
int main()
{
        struct Organisation org;
        printf("%ld", sizeof(org));
}
```

Pointers to Structures

```
struct Books
{
    char title[50];
    char author[50];
    char subject[100];
     int book_id;
}Book1;
```

Define pointers to structures in very similar way as you define pointer to any other variable.

```
struct Books   *struct_pointer;
```

Store the address of a structure variable in the above defined pointer Variable.

To find the address of a structure variable, place the **&** operator before the structure's name.

```
struct_pointer = &Book1;
```

Pointers to Structure

```
#include<stdio.h>

struct Point
{
int x, y;
};

int main()
{
        struct Point p1 = {1, 2};

        // p2 is a pointer to structure p1
        struct Point *p2 = &p1;

       // Accessing structure members using structure pointer
        printf("%d %d", p2->x, p2->y);
        return 0;
}
```

Output:

1 2

Bit-Fields

A bit-field is an integer variable that consists of a specified number of bits.
If you declare several small bit-fields in succession, the compiler packs them into a single machine word.

Syn: type [member_name] : width ;

type:

An integer type that determines how the bit-field's value is interpreted.

member_name:

The name of the bit-field, which is optional.

Width:

The number of bits in the bit-field.

Program Execution

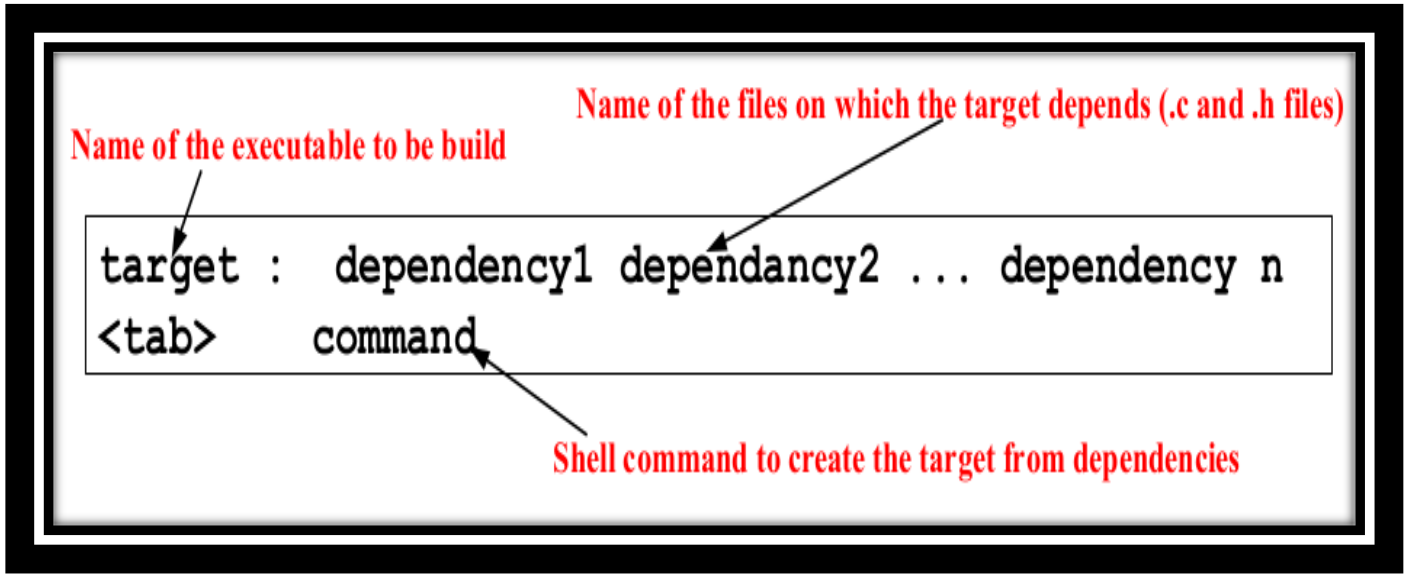
Bit-Fields

```
#include <stdio.h>
// A simple representation of the date
struct date {
unsigned int d;
unsigned int m;
unsigned int y;
};

int main()
{
printf("Size of date is %lu bytes\n",sizeof(struct date));
struct date dt = { 28,5,2022 };
printf("Date is %d/%d/%d", dt.d, dt.m, dt.y);
}
```

```
#include <stdio.h>

// Space optimized representation of the date
struct date
{
        unsigned int d : 5;
        unsigned int m : 4;
        int y;
};
int main()
{
         printf("Size of date is %lu bytes\n",sizeof(struct date));
           struct date dt = { 31, 12, 2014 };
           printf("Date is %d/%d/%d\n", dt.d, dt.m, dt.y);
           return 0;
}
```

Makefile

1.Imagine you write a program and divide it into hundred .c files and some header files
2. To make the executable you need to compile those hundred source files to create hundred relocatable object files and then you need to link those object files to final
Executable
3.make utility is a powerful tool that allows you to manage compilation of multiple modules into an executable
4. It reads a specification file called “makefile” or “Makefile”, that describes how the modules of a s/w system depend on each other. If you want to use a non- standard name you can specify that name to make using -f option
5. Make utility uses this dependency specification in the makefile and the time when various components were modified, in order to minimize the amount of recompilation



Thank You