2-D array

- Logically 2-D array represents m x n matrix i.e. m rows and n columns.

```
int arr[3][4] = { {1, 2, 3, 4}, {10, 20, 30, 40}, {11, 22, 33, 44} };
```

- Array declaration:

```
int arr[3][4] = { {1, 2, 3, 4}, {10, 20, 30, 40}, {11, 22, 33, 44} };
int arr[3][4] = { {1, 2 }, {10}, {11, 22, 33 } };
int arr[3][4] = { 1, 2, 10, 11, 22, 33 };
int arr[ ][4] = { 1, 2, 10, 11, 22, 33 };
```

- 2-D array is collection of 1-D arrays in contiguous memory locations.
  - Each element is 1-D array.
- Pointer to array is pointer to 0th element of the array.
  - Scale factor of the pointer = number of columns * sizeof(data-type).

```
int arr[3][4] = { {1, 2, 3, 4}, {10, 20, 30, 40}, {11, 22, 33, 44} };
int (*ptr)[4] = arr;
```

- 2-D array is passed to function by address.
- It can be collected in formal argument using array notation or pointer notation.
- While using array notation, giving number of rows is optional. Even though mentioned, will be ignored by compiler.

Dynamic memory allocation

- Dynamic memory allocation allow allocation of memory at runtime as per requirement.
- This memory is allocated at runtime on Heap section of process.
- Library functions used for Dynamic memory allocation are
  - malloc() - allocated memory contains garbage values.
  - calloc() - allocated memory contains zero values.
  - realloc() - allocated memory block can be resized (grow or shrink).
- All these function returns base address of allocated block as void*.
- If function fails, it returns NULL pointer.

Memory leakage

- If memory is allocated dynamically, but not released is said to be "memory leakage".
- Such memory is not used by OS or any other application as well, so it is wasted.
- In modern OS, leaked memory gets auto released when program is terminated.
- However for long running programs (like web-servers) this memory is not freed.

- More memory leakage reduce available memory size in the system, and thus slow down whole system.

```
int main() {
    int *p = (int*) malloc(20);
    int a = 10;
    // …
    p = &a; // here addr of allocated block is lost, so this memory can never be
freed.
    // this is memory leakage
    // …
    return 0;
}
```

- In Linux, valgrind tool can be used to detect memory leakage.

Dangling pointer

- Pointer keeping address of memory that is not valid for the application, is said to be "dangling pointer".
- Any read/write operation on this may abort the application. In Linux it is referred as "Segmentation Fault".
- Examples of dangling pointers
  - After releasing dynamically allocated memory, pointer still keeping the old address.
  - Uninitialized (local) pointer
  - Pointer holding address of local variable returned from the function.
- It is advised to assign NULL to the pointer instead of keeping it dangling.

```
int main() {
    int *p = (int*) malloc(20);
    // …
    free(p); // now p become dangling
    // …
    return 0;
}
```

Structure

- Structure is a user-defined data type.
- Structure stores logically related (similar or non-similar) elements in contiguous memory location.
- Structure members can be accessed using "." operator via struct variable.
- Structure members can be accessed using "->" operator via struct pointer.
- Size of struct = Sum of sizes of struct members.
- If struct variable initialized partially at its point of declaration, remaining elements are initialized to zero.

```
// struct data-type declaration (global or local)
struct emp {
    int empno;
```

```
    char ename[20];
    double sal;
};
// struct variable declaration
struct emp e1 = {11, "John", 20000.0};
// print struct members
printf("%d%s%lf", e1.empno, e1.ename, e1.sal);
```

- A variable of a struct can be member of another struct.
- This can be done with nested struct declaration.

```
struct emp {
    int empno;
    char ename[20];
    double sal;
    struct {
        int day, month, year;
    }join;
};
```

```
struct date {
    int day, month, year;
};
struct emp {
    int empno;
    char ename[20];
    double sal;
    struct date join;
};
struct emp e = { 11, "John", 2000.0, {1, 1, 2000} };
printf("%d %s %d-%d-%d\n", e.empno, e.ename, e.join.day, e.join.month,
e.join.year);
```

Structure padding

- For efficient access compiler may add hidden bytes into the struct called as "struct padding" or "slack bytes".
- On x86 architecture compiler add slack bytes to make struct size multiple of 4 bytes (word size).
- These slack bytes not meant to be accessed by the program.
- Programmer may choose to turn off this feature by using #pragma.
    - #pragma pack(1)

```
struct test {
    int a;
    char b;
};
```

```
printf("%u\n", sizeof(struct test));

#pragma pack(1)
struct test {
    int a;
    char b;
};
printf("%u\n", sizeof(struct test));
```

Bit Fields

- A bit-field is a data structure that allows the programmer to allocate memory to structures and unions in bits in order to utilize computer memory in an efficient manner.
- Bit-fields can be signed or unsigned.
- Signed bit-field, MSB represent size + or -.
- Unsigned bit-field, all bits store data.
- Limitations of bit-fields
    - Cannot take address of bit-field (&)
    - Cannot create array of bit-fields.
    - Cannot store floating point values.

```
struct student {
     char name[20];
     unsigned int age: 7;
     unsigned int roll: 6;
};
struct student s1 = { "Ram", 10, 21 };
printf("%s, %d, %d", s1.name, s1.age, s1.roll);
```

Union

- Union is user defined data-type.
- Like struct it is collection of similar or non-similar data elements.
- All members of union share same memory space i.e. modification of an member can affect others too.
- Size of union = Size of largest element
- When union is initialized at declaration, the first member is initialized.
- Application:
    - System programming: to simulate register sharing in the hardware.
    - Application programming: to use single member of union as per requirement.

```
union test {
    int num;
    char arr[2];
}u = { 65 };
printf("%d, %c, %s\n", u.num, u.arr[0], u.arr);
```