UNIT -3

Problem Solving with C

Multi-dimensional arrays

- In C, we can define multidimensional arrays in simple words as array of arrays.
- Data in multidimensional arrays are stored in tabular form (in row major order).

General form of declaring N-dimensional arrays:

- **Data_type Array_name[size1][size2]....[Sizen];**
  - **Data_type**: Type Of Data To Be Stored In The Array. Here Data_type Is Valid C Data Type
  - **Array_name**: Name Of The Array
  - **Size1, Size2,... ,Sizen**: Sizes Of The Dimensions

- Size of multidimensional arrays
  - Total number of elements that can be stored in a multidimensional array can be calculated by multiplying the size of all the dimensions.
- For example:
  - The array int x[10][20] can store total (10*20) = 200 elements.
  - Similarly array int x[5][10][20] can store total (5*10*20) = 1000 elements.

Initializing Three-Dimensional Array:

- Initialization in Three-Dimensional array is same as that of Two- dimensional arrays. The difference is as the number of dimension increases so the number of nested braces will also increase.

Method 1:

```
int x[2][3][4] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13,
14, 15, 16, 17, 18, 19, 20, 21, 22, 23};
```

Better Method:

```
int x[2][3][4] = {      {      {0,1,2,3},
                               {4,5,6,7},
                               {8,9,10,11}
                        },
                        {      {12,13,14,15},
                               {16,17,18,19},
                               {20,21,22,23}
                        }
                };
```

- Accessing elements in Three-Dimensional Arrays:
- Accessing elements in Three-Dimensional Arrays is also similar to that of Two-Dimensional Arrays. The difference is we have to use three loops instead of two loops for one additional dimension in Three-dimensional Arrays.

- int arr [20][30][40];

- Each element can be accessed as: arr [i][j][k]

- Write a program to find the sum of elements in a 3-D array [2x3x4]

```
int numbers[2][3][4] = {
        { // First block of 3 rows
            { 10, 20, 30, 40 },
            { 15, 25, 35, 45 },
            { 47, 48, 49, 50 }
        },
        { // Second block of 3 rows
            { 10, 20, 30, 40 },
            { 15, 25, 35, 45 },
            { 47, 48, 49, 50 }
        }
};
```

- Here’s how you could sum the elements:

```
int sum = 0;
for(int i = 0 ; i < 2 ; ++i)
{
        for(int j = 0 ; j < 3 ; ++j)
        {
                for(int k = 0 ; k < 4 ; ++k)
                {
                sum += numbers[i][j][k];
                }
        }
}
printf("The sum of the values in the numbers array is %d.", sum);
```

Multi-dimensional arrays and pointers

```
#include <stdio.h>
int main(void)
{       char board[3][3] =
        {
            {'1','2','3'},
            {'4','5','6'},
            {'7','8','9'}
        };
char *pboard = *board; // A pointer to char
for(int i = 0 ; i < 9 ; ++i)
printf(" board: %c\n", *(pboard + i));
 return 0;
}
```

- What will be the output of this program?

- o/p:

board: 1

board: 2

board: 3

board: 4

board: 5

board: 6

board: 7

board: 8

board: 9

- **How it works**
- You initialize pboard with the address of the first element of the array, and then you use normal pointer arithmetic to move through the array:

```
char *pboard = *board;     // A pointer to char
// char *pboard = &board[0][0];


 for(int i = 0 ; i < 9 ; ++i)
     printf(" board: %c\n", *(pboard + i));
     //   printf(" board: %c\n", *(*board + i))
```

We could have initialized pboard by using the following:

```
 char *pboard = *board;

//  char *pboard = &board[0][0];

//   char *pboard;
//   pboard = board; ; // Wrong level of indirection!
```

This is wrong. You will at least get a compiler warning if you do this.

Strictly speaking, this isn't legal because pboard and board have *different levels of indirection.* *That's a great* jargon phrase that just means that ,

pboard refers to an address that contains a value of type char,

whereas board refers to an address that refers to an address containing a value of type char.

There's an extra level with board compared to pboard.

Consequently, pboard needs one * to get to the value and board needs two **.

Some compilers will allow you to get away with this and just give you a warning about what you've done. However, it is an error, so you shouldn't do it!

Accessing Array Elements

- Table shows **Pointer Expressions for Accessing Array Elements.**
- **It** lists different ways of accessing your **board** array. The left column contains row index values to the **board** array, and the top row contains column index values.

board	0	1	2
0	board[0][0] *board[0] **board	board[0][1] *(board[0]+1) *(*board+1)	board[0][2] *(board[0]+2) *(*board+2)
1	board[1][0] *(board[0]+3) *board[1] *(*board+3)	board[1][1] *(board[0]+4 *(board[1]+1) *(*board+4)	board[1][2] *(board[0]+5) *(board[1]+2) *(*board+5)
2	board[2][0] *(board[0]+6) *(board[1]+3) *board[2] *(*board+6)	board[2][1] *(board[0]+7) *(board[1]+4) *(board[2]+1) *(*board+7)	board[2][2] *(board[0]+8) *(board[1]+5) *(board[2]+2) *(*board+8)

```
 char board[3][3] = {
                      {'1','2','3'},
                      {'4','5','6'},
                      {'7','8','9'}
                     };

  printf("address of board        : %p\n", board);
  printf("but what is in board[0] : %p\n", board[0]);
   printf("address of board[0][0]  : %p\n",&board[0][0]);
/*
address of board        : 0x7fffab976340
but what is in board[0] : 0x7fffab976340
address of board[0][0]  : 0x7fffab976340  */
```

```
*pointer of pointer for its elements */
  printf("value of board[0][0] : %c\n", board[0][0]);
  printf("value of *board[0]   : %c\n", *board[0]);
  printf("value of **board     : %c\n", **board);

  printf("value of   elemnt : %c\n", board[2][2]);
  printf("value is           : %c\n", *(board[0]+8));
  printf("value is           : %c\n", *(*board + 8));

  printf("value of   element: %c\n", board[0][1]);
  printf("value of           : %c\n", *(board[0]+1));
  printf("value of            : %c\n", *(*board + 1));
```

typedef

typedef is a reserved keyword in the C programming language. It is used to create an alias name for another data type.

As such, it is often used to simplify the syntax of declaring complex data structures consisting of struct and union types, but is just as common in providing specific descriptive type names for integer data types of varying lengths.

```
typedef unsigned char USCH;
typedef short int_16;
```

```c
// C program to demonstrate typedef
#include <stdio.h>
// After this line BYTE can be used in place of
// unsigned char
typedef unsigned char BYTE;

int main(void)
{
    BYTE b1, b2;
    b1 = 'c';
    printf("%c ", b1);
    return 0;
}
```

```c
// C program to demonstrate #define
#include <stdio.h>

// After this line HYD is replaced by
// "Hyderabad"
#define HYD "Hyderabad"

int main(void)
{
    printf("%s ", HYD);
    return 0;
}
```

Difference between typedef and #define:

1. typedef is limited to giving symbolic names to types only, whereas #define can be used to define an alias for values as well, e.g., you can define 1 as ONE, 3.14 as PI, etc.
2. typedef interpretation is performed by the compiler where #define statements are performed by preprocessor.
3. #define should not be terminated with a semicolon, but typedef should be terminated with semicolon.
4. #define will just copy-paste the definition values at the point of use, while typedef is the actual definition of a new type.
5. typedef follows the scope rule which means if a new type is defined in a scope (inside a function), then the new type name will only be visible till the scope is there. In case of #define, when preprocessor encounters #define, it replaces all the occurrences, after that (No scope rule is followed).

GDB

- “GNU Debugger”
- A debugger for several languages, including C and C++.
- It allows you to inspect what the program is doing at a certain point during execution.
- Errors like segmentation faults may be easier to find with the help of gdb.

Additional steps required during compilation to help you use gdb

- Normally, you would compile a program like:

gcc [flags] <source files> -o <output file>

For example:

gcc src1.c src2.c - o outfile

To enable built-in debugging support (which gdb needs):

gcc [other flags] -g <source files> -o <output file>

Starting up “gdb”

- Just try “gdb” or “gdb **outfile**” You’ll get a prompt that looks like this:

(gdb)

If you didn’t specify a program to debug, you’ll have to load it in now:

(gdb) file **outfile**

Here, **outfile** is the program you want to load, and “file” is the command to load it.

- gdb has an interactive shell, much like the one you use as soon as you log into the linux systems. It can recall history with the arrow keys, **auto-complete words** (most of the time) with the TAB key, and has other nice features.

- **Tip**

If you’re ever confused about a command or just want more information, use the “help” command, with or without an argument: (gdb) help [command]

Running the program

To run the program, just use:
(gdb) run

This runs the program. If it has no serious problems (i.e. the normal program didn't get a segmentation fault, etc.), the program should run fine here too.

If the program did have issues, then you (should) get some useful information like the line number where it crashed, and parameters to the function that caused the error:

Program received signal SIGSEGV, Segmentation fault. 0x0000000000400524 in sum array region (arr=0x7fffc902a270, r1=2, c1=5, r2=4, c2=6) at sum-array-region2.c:12

What if bugs are present in the program?

Okay, so you've run it successfully. But you don't need gdb for that. What if the program isn't working?

Basic idea

Chances are if this is the case, you don't want to run the program without any stopping, breaking, etc. Otherwise, you'll just rush past the error and never find the root of the issue. So, you'll want to step through your code a bit at a time, until you arrive upon the error.

This brings us to the next set of commands. . .

Setting breakpoints

- Breakpoints can be used to stop the program run in the middle, at a designated point. The simplest way is the command “break.”
- This sets a breakpoint at a specified file-line pair:

```
(gdb) break file1.c:6
```

- This sets a breakpoint at line 6, of file1.c. Now, if the program ever reaches that location when running, the program will pause and prompt you for another command.

- Tip

You can set as many breakpoints as you want, and the program should stop execution if it reaches any of them.

- You can also tell gdb to break at a particular function. Suppose you have a function my_func:

```
int my_func(int a, char *b);
```

- You can break anytime this function is called:

```
(gdb) break my_func
```

What next?

Once you've set a breakpoint, you can try using the run command again. This time, it should stop where you tell it to (unless a fatal error occurs before reaching that point).

You can proceed onto the next breakpoint by typing "continue" (Typing run again would restart the program from the beginning, which isn't very useful.)

```
(gdb) continue
```

You can single-step (execute just the next line of code) by typing "step." This gives you really fine-grained control over how the program proceeds. You can do this a lot...

```
(gdb) step
```

- Similar to “step,” the “next” command single-steps as well, except this one doesn’t execute each line of a sub-routine, it just treats it as one instruction.

  (gdb) next

- Tip

  Typing “step” or “next” a lot of times can be tedious. If you just press ENTER, gdb will repeat the same command you just gave it. You can do this a bunch of times.

- So far you've learned how to interrupt program flow at fixed, specified points, and how to continue stepping line- by-line. However, sooner or later you're going to want to see things like the values of variables, etc. This might be useful in debugging. :)

- The print command prints the value of the variable specified,
  and print/x prints the value in hexadecimal:
- (gdb) print my_var
  (gdb) print/x my_var

Setting watchpoints

Whereas breakpoints interrupt the program at a particular line or function, **watchpoints act on variables**. They pause the program whenever a watched variable's value is modified. For example, the following watch command:

```
(gdb) watch my_var
```

Now, whenever my var's value is modified, the program will interrupt and print out the old and new values.

Tip

You may wonder how gdb determines which variable named my var to watch if there is more than one declared in your program. The answer (perhaps unfortunately) is that it relies upon the variable's scope, relative to where you are in the program at the time of the watch. This just means that you have to remember the tricky nuances of scope and extent

Other useful commands

backtrace - produces a stack trace of the function calls that lead to a seg fault

where - same as backtrace; you can think of this version as working even when you're still in the middle of the program

finish - runs until the current function is finished

delete - deletes a specified breakpoint

info breakpoints - shows information about all declared breakpoints

More about breakpoints

Breakpoints by themselves may seem too tedious. You have to keep stepping, and stepping, and stepping. . .

Basic idea

Once we develop an idea for what the error could be (like dereferencing a NULL pointer, or going past the bounds of an array), we probably only care if such an event happens; we don't want to break at each iteration regardless.

So ideally, we'd like to condition on a particular requirement (or set of requirements). Using conditional breakpoints allow us to accomplish this goal. . .

Conditional breakpoints

Just like regular breakpoints, except that you get to specify some criterion that must be met for the breakpoint to trigger. We use the same break command as before:

```
(gdb) break file1.c:6 if i >= ARRAYSIZE
```

This command sets a breakpoint at line 6 of file file1.c, which triggers only if the variable i is greater than or equal to the size of the array (which probably is bad if line 6 does something like arr[i]). Conditional breakpoints can most likely avoid all the unnecessary stepping, etc.

STRUCTURES.

Structure

What is a structure?

- A structure is a user defined data type in C.
- A structure is a named collection of items not necessarily homogeneous.
- A structure creates a data type that can be used to group items of possibly different types into a single type.

How to create a structure?

```
struct student
{
        int srn;
        char name[20];
        int marks;
};
```

- contains 3 elements – called Fields/ Data members.

How to declare structure variables?

- A structure variable can either be declared with structure declaration or as a separate declaration like basic types.
    - struct student

```
{
    int srn;
    char name[20];
    int marks;
} s1;
```

or

- struct student s1;

How to initialize structure members?

- Structure members **cannot be** initialized with declaration.

For example the following C program fails in compilation.

```
struct Point
{
    int x = 0;   // COMPILER ERROR: cannot initialize members here
    int y = 0;   // COMPILER ERROR: cannot initialize members here
};
```

Reason?

- The reason for above error is simple, when a data type is declared, no memory is allocated for it.
- Memory is allocated only when variables are created.

```
struct individual
{
    int age;
    int height;
    char name[20];
    char father[20];
    char mother[20];
};
```

```
struct individual S1 = {
                        24, 170 ,“Robbin", “ABC", “XYZ"
                        };
```

```
struct student
        {
            int srn;
            char name[20];
            int marks;
        } ;
```

- Structure s1 is not initialized. So, the fields are uninitialized.

```
struct student s1;
```

- The structure s2 is created and also initialized. Observe the order of values initialized matches the order of fields in the structure.

```
struct student s2={23,"pes",99};
```

- Partially initialized explicitly and the rest are filled with 0.

```
struct student s3={23,"pes"};
```

By using **a typedef definition**

typedef struct individual STUDENT;

- This defines STUDENT to be the equivalent of struct individual.
- If you put this definition at the beginning of a source file, you can define a variable of type INDIVIDUAL like this:

```
STUDENT rama= {
                30, 168, “Rama", “Dasharath", “kousalya”
                };
```

- The struct keyword is no longer necessary. This makes the code less cluttered and makes your structure type look like a first-class type.

Accessing Structure Members

- You refer to a member of a structure by writing the structure variable name followed by a period, (.) ,followed by the member variable name.

- The period between the structure variable name and the member name is called the *member selection operator.*

  Ex: rama.age

- Structure members are the same as variables of the same type. You can set their values and use them in expressions in the same way as ordinary variables.
  - *Ex: rama.age = 40 ;*

Designated initialization

- The fields are initialized by specifying there names in the initializer. This is called designated initialization.

```
STUDENT sita = { .height = 15, .age = 30,
.name = “sita", .mother = “Sunayna ", .father = " janaka "
};
```

- Now there is no doubt about which member is being initialized by what value. **The order of the initializers is now unimportant**.

Unnamed Structures

- You don’t have to give a structure a tag name.
- When you declare a structure and any instances of that structure in a single statement, you can omit the tag name.
- In the previous example, instead of the structure declaration for type INDIVIDUAL, followed by the instance declaration for RAMA, you could have written this statement.

```
struct
    { // Structure declaration and...
        int age;
        int height;
        char      name[20];
        char      father[20];
        char mother[20];
    } RAMA  ;  // ...structure variable declaration combined
```

- A serious disadvantage with this is that you can no longer define further instances of the structure in another statement. All the variables of this structure type that you want in your program must be defined in the one statement.

Pointer to a structure

STUDENT *pstudent;

How do you access the individual members of the structure?

pstudent ->name
pstudent ->age
pstudent ->father
pstudent -mother

1) Write a program to accept the details of a student and print out the same.

```
#include <stdio.h>
        struct student {
                char name[50];
                int roll;
                float marks;
              } s;
int main() {
   printf("Enter information:\n");
   printf("Enter name: ");
   scanf("%[^\n]", s.name);
                printf("Enter roll number: ");
                 scanf("%d", &s.roll);
                 printf("Enter marks: ");
                 scanf("%f", &s.marks);
   printf("Displaying Information:\n");
   printf("Name: ");
   printf("%s\n", s.name);
   printf("Roll number: %d\n", s.roll);
   printf("Marks: %.1f\n", s.marks);
   return 0;
}
```

2) Write a program to accept the details of five students and print out the same.

Array of Structures

struct Student student[n];

struct student s2={23,“ram",99};

- This is not a direct assignment involving assignment operator. Here we're using a brace-enclosed initializer list to provide the initial values of the object. That follows the law of initialization.

s2.name=“sita"; // throws error during compilation.

- Because, in the LHS, you're using an array type, which is not assignable.

Assignment operator should have a modifiable lvalue as its left operand.
A modifiable lvalue is an lvalue that does not have array type.

char *b = "Hello" ,
char c[] = "hello" , both works ,
if for first case, if we do , b = "helloworld" , it works too ,
but for second case if we do, c = "helloworld" , it does not why so ? –
You cannot assign to an char array.

- 2 ways to assign values to char array:

1) use strcpy() to copy into the array.

```
strcpy(s1.name, “ sita");
```

2) you can define a pointer which points to char arrays of any length.

```
char *name;
```

3) Write a program to accept the details of five students with each student having 2 subject marks. print out total marks and average marks secured by each student.

Structure layout and size:

```
struct s {
        int i;
        char ch;
        double d;
};
```

```
#include <stdio.h>
struct s {
                int i;
                char ch;
                double d;
      };

int main()
{
        struct s A;
       printf("Size of A is: %lu", sizeof(A));
}
```

- The address of any particular data type may start on a byte boundary, word boundary, paragraph boundary and so on.
- The size of an int normally is a size of a word 4 bytes. If the int field is word aligned, accessing the int field will be faster.
- So structures could align for each of the types and there could be some space in between the fields which is not used – this is called padding.

#pragma Directive

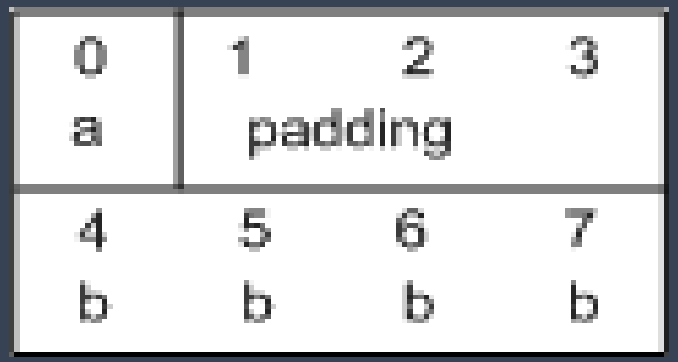
- This directive is a special purpose directive and is used to turn on or off some features.

Syntax

```
#pragma pack(n)
```

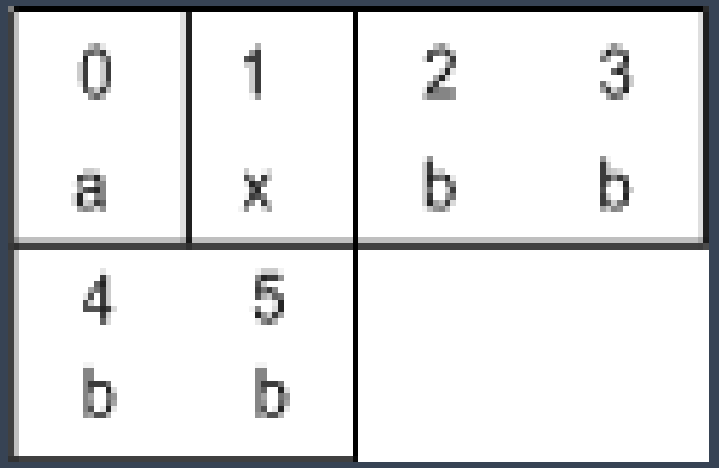
- Where: n is the alignment in bytes, valid alignment values being 1, 2, 4 and 8.
- This pragma aligns members of a structure.
- The default #pragma pack() -> value of n=8.
- Run the program with various values of pack ie. ‘n’ and convince yourself about the sizes.

```
struct s1
{
    char a;
    int b;
} S1;
```



This layout has 3 bytes of padding.
S1 is a 8-byte structure.

```
#pragma pack(2)
struct s2
{
    char a;
    int b;
} S2;
```



In this layout, x denotes one byte of padding.
S2 is a 6-byte structure.
There is no padding after b.

Structures and Functions in C

- The C Programming allows us to pass the structures as the function parameters.
- If you define the structure inside main(), **THE SCOPE IS LIMITED TO MAIN() ONLY**. Any other function cannot see that definition and hence, cannot make use of that structure definition.
- If you **DEFINE THE STRUCTURE IN A GLOBAL SCOPE**, (i.e., outside main() or any other function, for that matter), that definition is available globally and **ALL THE FUNCTIONS CAN SEE AND MAKE USE OF THE STRUCTURE DEFINITION**

We can pass the C structures to functions in 3 ways:

1) Passing each item of the structure as a function argument. It is similar to passing normal values as arguments. Although it is easy to implement, we don't use this approach because if the size of a structure is a bit larger, then the program becomes clumsy.
2) Pass the whole structure as a value.
3) We can also Pass the address of the structure (pass by reference).

```
struct test
{
        int i;
        char a[20];
        float j;
};
      void f1(struct test t)
      {
                strcpy(t.a,"pqrstuv");
      }
            void f2(struct test* p)
            {
                  strcpy(p->a,"pqr");
            }
```

```
int main()
{
            struct test t1={20,"xyz",78.5};
            printf("t1 is %s\n", t1.a);
            f1(t1);
            printf("t1 is %s\n", t1.a);
            f2(&t1);
            printf("t1 is %s\n", t1.a);
            return 0;
}
```

t GETS A COPY OF t1. CHANGING t.a DOES NOT AFFECT t1.a

o/p:

xyz

xyz

pqr

- Passing a structure by value is considered a bad programming practice. This would occupy more space, would require more time to copy and also will cause problems if the members of the structure are pointers.
- When we do not want to change the structure, pass a pointer to a structure and make the parameter a pointer to a constant structure.
- This function f3 shows how to pass a structure if we do not intent to change the structure.

```
void f3(const struct test* p)
{
//strcpy(p->a,"pqr"); // Error
printf("%s\n", p->a);
}
f3(&t1);
```

C – Nested Structure

- Nested structure in C is nothing but structure within structure. One structure can be declared inside other structure as we declare structure members inside a structure.
- The structure variables can be a normal structure variable or a pointer variable to access the data.

```
struct point
{
        int x;  int y;
};
struct rect
{
    struct point pt1;
    struct point pt2;
};
```

- One representation of a rectangle is a pair of points that denote the diagonally opposite corners

```
struct point
{
        int x;  int y;
};
struct rect
{
    struct point pt1;
    struct point pt2;
};
```

```
struct rect
{
    struct point{
        int x;  int y;
        } pt1;
    struct point pt2;
};
```

1. STRUCTURE WITHIN STRUCTURE IN C USING NORMAL VARIABLE

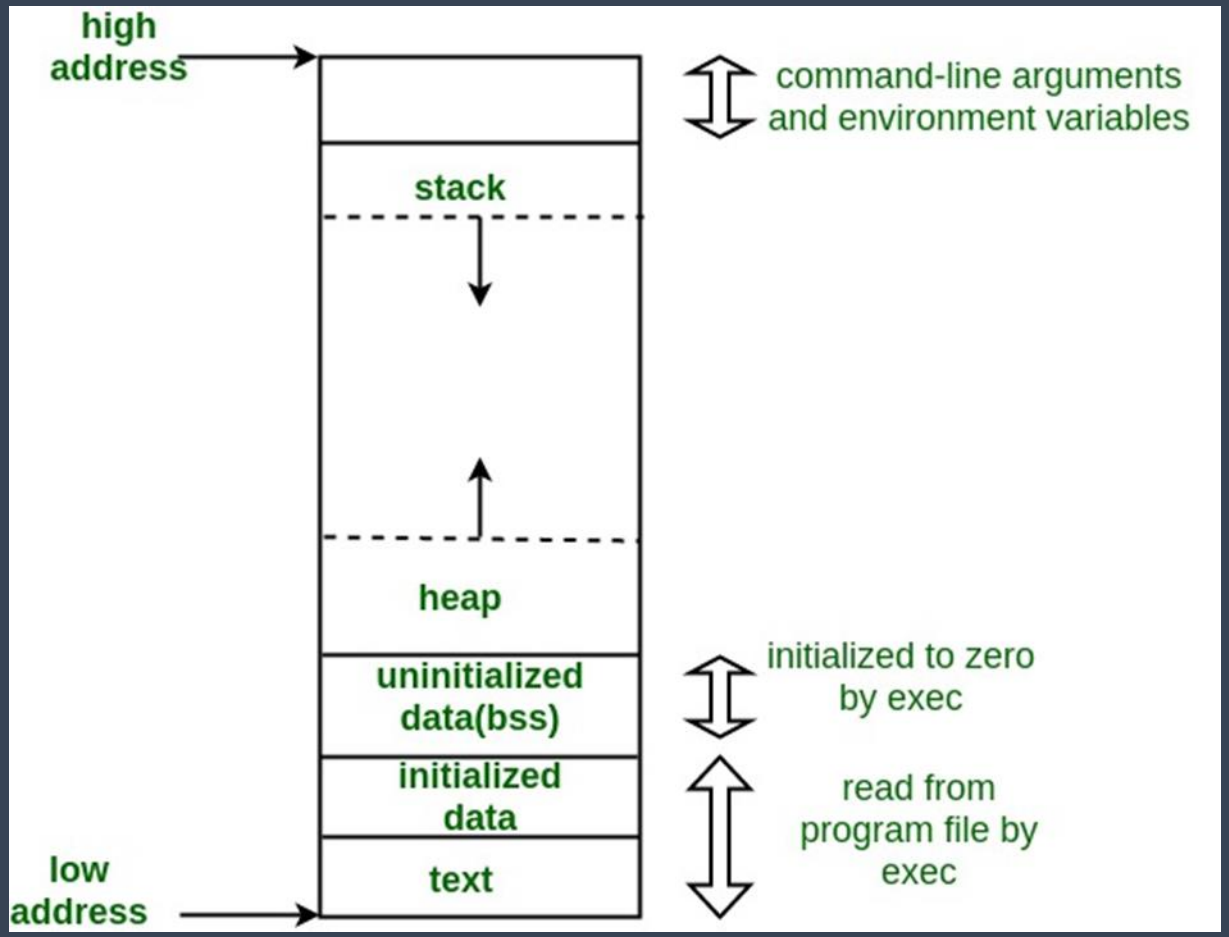
```
struct Employee
{
        int age;
        char Name[50];
        char Department[20];
        float salary;
        struct address // Declaring the address structure
         {
                    int Door_Number;
                    char Street_Name[20];
                    char City[20];
                    int Postcode;
         } add;     // creating the address variable = add
};
```

```
struct address
{
  int Door_Number;
  char Street_Name[20];
  char City[20];
  int Postcode;
};
struct Employee
{
  int age;
  char Name[50];
  char Department[20];
  struct address add; // creating the structure variable = add
  float salary;
};
```

2)STRUCTURE WITHIN STRUCTURE IN C USING POINTER VARIABLE:

```
 struct address
{
        int Door_Number;
        char City[20];
};
struct Employee
{
        int Age;
        char Name[50];
        struct address add;
        float Salary;
}*emp3;
```

Dynamic memory allocation



Memory layout of C Program

Memory can be allocated for variables using different techniques:

1. STATIC ALLOCATION

- decided by the compiler
- allocation at load time [before the execution or run time]

2. AUTOMATIC ALLOCATION

- decided by the compiler
- allocation at run time allocation on entry to the block(variables defined inside functions), deallocation on exit

3. DYNAMIC ALLOCATION

- code generated by the compiler.
- allocation and deallocation on call to memory allocation functions: malloc, calloc, realloc

Dynamic Memory Allocation

- The process of allocating memory at runtime is known as dynamic memory allocation.
- Memory is allocated or deallocated during run-time (during the execution of the program) in the heap region.
- Library routines known as "memory management functions" are used for allocating and freeing/releasing memory during execution of a program.
- These functions are defined in stdlib.h

1. malloc(): Allocates requested size of bytes and returns a void pointer pointing to the first byte of the allocated space

2 calloc(): Allocates space for an array of elements, initialize them to zero and then return a void pointer to the memory

3. realloc(): Modifies the size of previously allocated space using above functions

4. free(): Releases the allocated memory

SMA(Static Memory Allocation)

- 1. The memory is allocated before execution/run time
- 2. The size of the memory to be allocated is fixed during compile time and cannot be altered during run time.
- 3. Memory is allocated in stack and other segments
- 4. Used only when the data size is known in advance.

DMA(Dynamic Memory Allocation):

- 1. The memory is allocated during run time
- 2. As and when memory is required, memory can be allocated. If memory is not required it can be deallocated.
- 3. Memory is allocated in heap area when functions are used.
- 4. Used for unpredictable memory requirement

malloc()

Prototype: void *malloc(size_t size);

- The malloc() function allocates size bytes and returns a pointer to the allocated memory.
- The memory is not initialized.
- If size is 0, then malloc() returns either NULL, or a unique pointer value.

Return Value

- The malloc() function returns a void pointer to the allocated memory that is suitably aligned for any kind of variable.
- On error, these functions return NULL.
- NULL may also be returned by a successful call to malloc() with a size of zero.

calloc()

- prototype: void *calloc(size_t nmemb, size_t size);
- The calloc() function allocates memory for an array of nmemb elements of size bytes each and returns a pointer to the allocated memory.
- The memory is set to zero.

Return Value

- If n membor size is 0, then calloc() returns either NULL, or a unique pointer value.
- The calloc() function return a pointer to the allocated memory that is suitably aligned for any kind of variable.
- On error, this function return NULL.
- NULL may also be returned by a successful call to calloc() with nmemb or size equal to zero.

realloc()

Prototype: void *realloc(void *ptr, size_t size);

- This function changes the size of the memory block pointed to by ptr to size bytes.
- The contents will be unchanged in the range from the start of the region up to the minimum of the old and new sizes.
- If the new size is larger than the old size, the added memory will not be initialized.
- If ptr is NULL, then the call is equivalent to malloc(size), for all values of size;
- If size is equal to zero, and ptr is not NULL, then the call is equivalent to free(ptr).
- Unless ptr is NULL, it must have been returned by an earlier call to malloc(), calloc() or realloc().
- If the area pointed to was moved, a free(ptr) is done.
- Return Value : The realloc() function returns a pointer to the newly allocated memory, which is suitably aligned for any kind of variable and may be different from ptr, or NULL if the request fails.
- If size was equal to 0, either NULL or a pointer suitable to be passed to free() is returned.

free():

void free(void *ptr);

- The free() function frees the memory space pointed to by ptr, which must have been returned by a previous call to malloc(), calloc() or realloc().
- Otherwise, or if free(ptr) has already been called before, undefined behavior occurs.
- If ptr is NULL, no operation is performed.
- **Return Value**
- The free() function returns no value.

malloc

1. stands for memory allocation
2. This function takes single argument.
3. Syntax:

void *malloc(size_t size);

the required number of bytes to be allocated is specified as an argument.

4. Allocates a block of memory of size bytes.
5. Allocated space will be initialized to undefined values.
6. malloc is faster than calloc.

calloc

1. Stands for contiguous allocation
2. This function takes two arguments
3. Syntax:

void *calloc(size_t n, size_t size);

n is number of blocks to be allocated. size is number of bytes to be allocated for each block

4. Allocates multiple blocks of memory, each block with the same size.
5. Each byte of allocated space is initialized to zero
6. calloc takes little longer than malloc because of the extra step of initializing the allocated memory by zero. However, in practice the difference in speed is very tiny and not recognizable.

Releasing Dynamically Allocated Memory

When you allocate memory dynamically, you should always release the memory when it is no longer required.

Memory that you allocate on the heap will be automatically released when your program ends, but it is better to explicitly release the memory when you are done with it, even if it's just before you exit from the program.

In more complicated situations, you can easily have a memory leak.

What is a memory leak?

A memory leak occurs when you allocate some memory dynamically and you do not retain the reference to it, so you are unable to release the memory.

This often occurs within a loop, and because you do not release the memory when it is no longer required, your program consumes more and more of the available memory on each loop iteration and eventually may occupy it all.

```
#include<stdio.h>
int main()
{
   int *p = (int *)malloc(sizeof(int));

   p = NULL;

   free(p);
}
```

- free() can be called for NULL pointer, so no problem with free function call.
- The problem is memory leak, p is allocated some memory which is not freed, but the pointer is assigned as NULL. The correct sequence should be following:

  ```
  free(p);
  p = NULL;
  ```

What is the return type of malloc() or calloc()

A) void *

B) Pointer of allocated memory type

C) void **

D) int *

o/p:

a) malloc() and calloc() return void *. We may get warning in C if we don't type cast the return type to appropriate pointer.

Which pgm necessarily need heap allocation in the run time environment?

a) Those that support recursion

b)Those that use static variables.

c)Those that use global variables

d)Those that allow dynamic data structure

o/p:

d)

Consider the following program, where are i, j and k are stored in memory?

```
int i;
int main()
{
   int j;
   int *k = (int *) malloc (sizeof(int));
}
```

O/P: i is stored in BSS part of data segment, j is stored in stack segment. *k is stored on heap

```
# include<stdio.h>
# include<stdlib.h>

void fun(int *a)
{
   a = (int*)malloc(sizeof(int));
}

int main()
{
   int *p;
   fun(p);
   *p = 6;
   printf("%dn",*p);
   return(0);
}
```

O/P:The program is not valid.

Try replacing “int *p;” with “int *p = NULL;” and it will try to dereference a null pointer. This is because fun() makes a copy of the pointer, so when malloc() is called, it is setting the copied pointer to the memory location, not p. p is pointing to random memory before and after the call to fun(), and when you dereference it, it will crash. If you want to add memory to a pointer from a function, you need to pass the address of the pointer (ie. double pointer).

```
#include‹stdio.h›
int main()
{
   struct site
   {
      char name[] = “PES";
      int no_of_pages = 200;
   };
   struct site *ptr;
   printf("%d ", ptr->no_of_pages);
   printf("%s", ptr->name);
   getchar();
   return 0;
}
```

o/p:

When we declare a structure or union, we actually declare a new data type suitable for our purpose. So we cannot initialize values as it is not a variable declaration but a data type declaration

Assume that size of an integer is 4 byte. What is the output of following program?

```
#include<stdio.h>
struct st
{
   int x;
   static int y;
};

int main()
{
   printf("%d", sizeof(struct st));
   return 0;
}
```

O/P: COMPILE ERROR.

In C, struct and union types cannot have static members.

```
#include<stdio.h>
struct st
{
   int x;
   struct st next;
};

int main()
{
   struct st temp;
   temp.x = 10;
   temp.next = temp;
   printf("%d", temp.next.x);
   return 0;
}
```

o/p: compile error.

A structure cannot contain a member of its own type because if this is allowed then it becomes impossible for compiler to know size of such struct.

Pointers to Structures as structure members

- Any pointer can be a member of a structure.
- This includes a pointer that points to a structure.
- A pointer structure member that points to the same type of structure is also permitted. This is called a Self-referential Structure.

Linked List

Linked List

- Like arrays, Linked List is a linear data structure. Unlike arrays, linked list elements are not stored at contiguous location; the elements are linked using pointers.

Why Linked List?

- Arrays can be used to store linear data of similar types, but arrays have following limitations.

1) The size of the arrays is fixed: So we must know the upper limit on the number of elements in advance. Also, generally, the allocated memory is equal to the upper limit irrespective of the usage.

2)Inserting a new element in an array of elements is expensive, because room has to be created for the new elements and to create room existing elements have to shifted.

3)Deletion is also expensive with arrays unless some special techniques are used.

- Advantages of linked lists over arrays
- Dynamic size
- Ease of insertion/deletion

- Representation:
- A linked list is represented by a pointer to the first node of the linked list. The first node is called head. If the linked list is empty, then value of head is NULL.

- Each node in a list consists of at least two parts:
  1)data
  2)Pointer (Or Reference) to the next node
- In C, we can represent a node using structures. Below is an example of a linked list node with an integer data.

- // A linked list node

```
struct Node
{
int data;
struct Node *next;
};
```

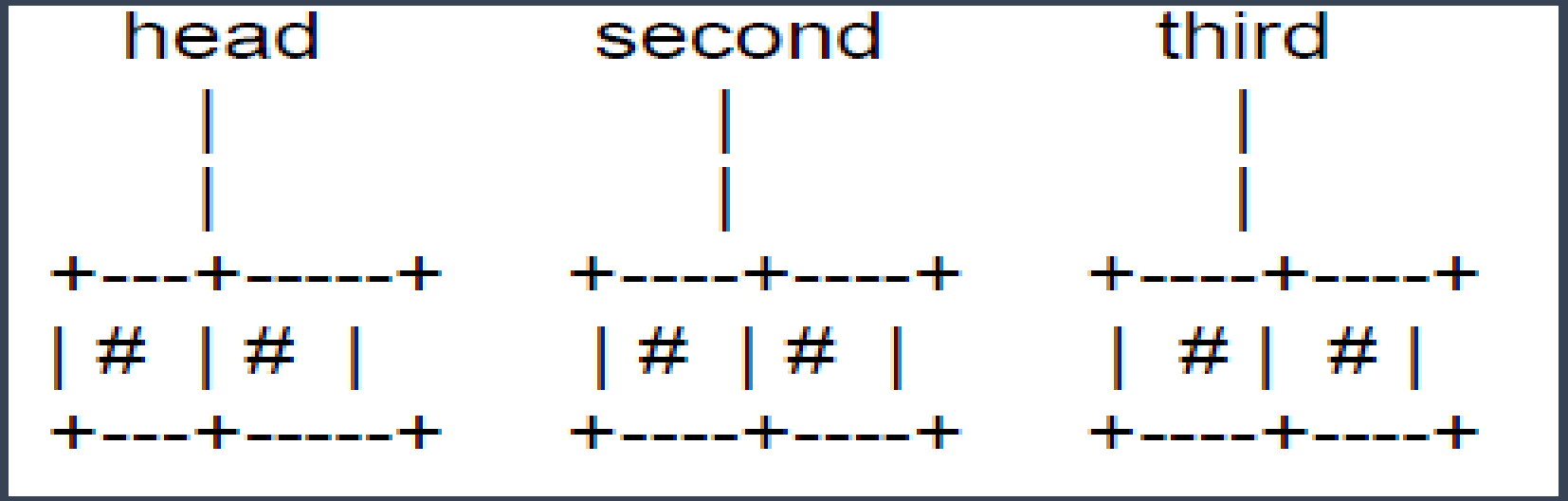
Create a linked list with 3 nodes

```
#include<stdio.h>
#include<stdlib.h>

struct Node
{
int data;
struct Node *next;
};

// Program to create a simple linked list with 3 nodes
int main()
{
struct Node* head = NULL;
struct Node* second = NULL;
 struct Node* third = NULL;
```

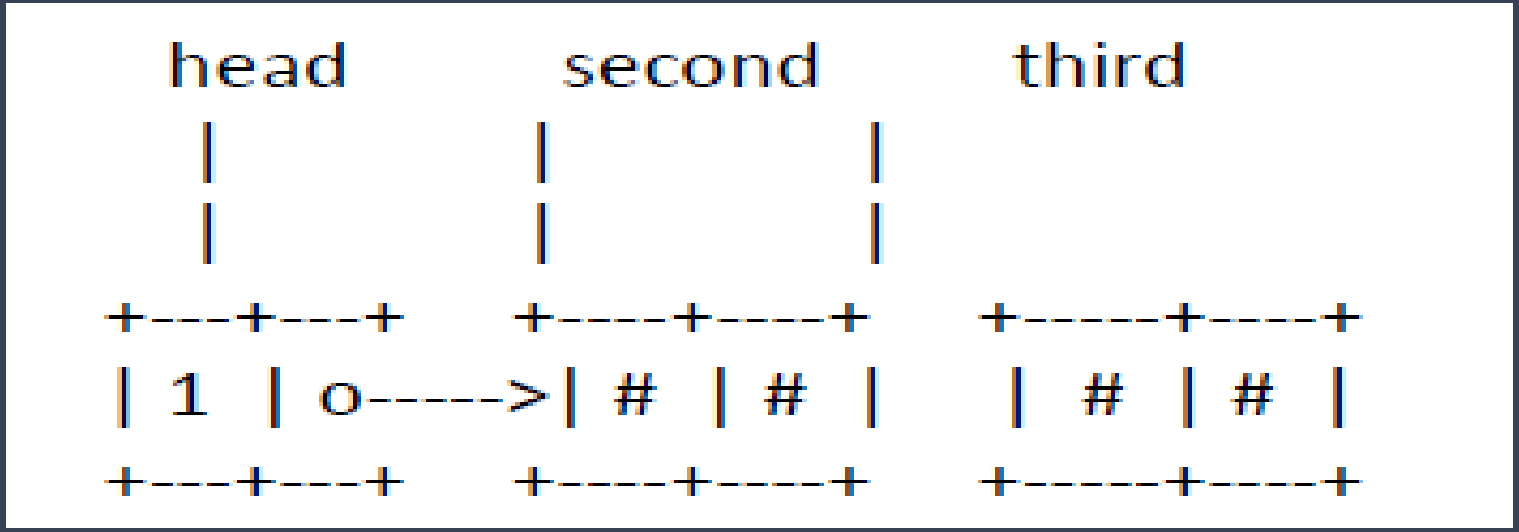
```
// allocate 3 nodes in the heap
        head = malloc(sizeof(struct Node));
       second = malloc(sizeof(struct Node));
       third = malloc(sizeof(struct Node));
/* Three blocks have been allocated dynamically.
We have pointers to these three blocks as head, second and  third
   " # "represents any random value.  Data is random because we haven't assigned
anything yet    */
```



```
head->data = 1;              //assign data in first node
head->next = second;   // Link first node with the second node
```

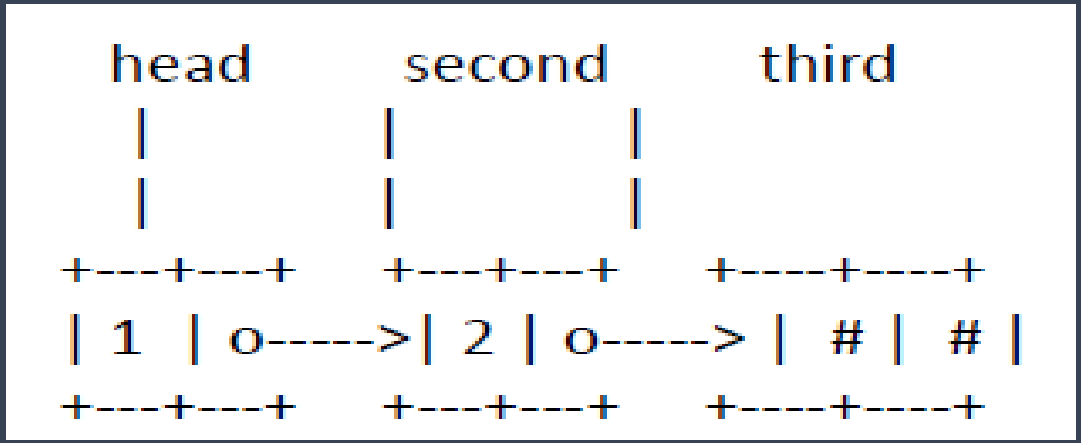
/* data has been assigned to data part of first block (block pointed by head). And next pointer of first block points to second.

So they both are linked. */



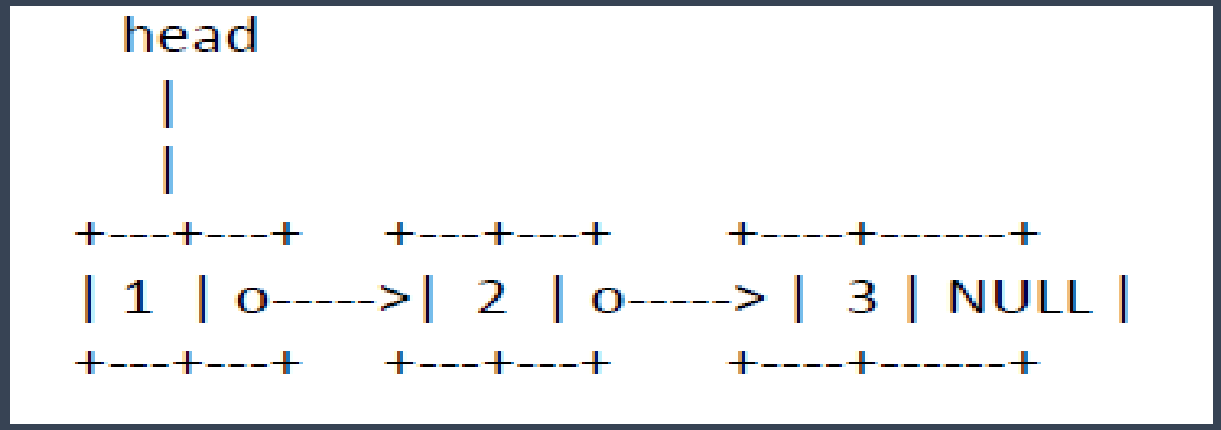
```
second->data = 2;          //assign data to second node
second->next = third  ;    // Link second node with the third node
```

- /* data has been assigned to data part of second block (block pointed by second). And next pointer of the second block points to third block. So all three blocks are linked.



```
third->data = 3;
third->next = NULL;
```

- /* data has been assigned to data part of third block (block pointed by third). And next pointer of the third block is made NULL to indicate that the linked list is terminated here.



- We have the linked list ready.
- Note that only head is sufficient to represent the whole list. We can traverse the complete list by following next pointers.

```
    return 0;
}
```

Traversing through a list

```
// This function prints contents of linked list starting from
// the given node
void printList(struct Node *n)
{
        while (n != NULL)
        {
        printf(" %d ", n->data);
        n = n->next;
        }
}
```

Drawbacks:

- Random access is not allowed. We have to access elements sequentially starting from the first node. So we cannot do binary search with linked lists efficiently with its default implementation.
- Extra memory space for a pointer is required with each element of the list.
- Not cache friendly. Since array elements are contiguous locations, there is locality of reference which is not there in case of linked lists.

Inserting / Deleting a node in a list

1. At the front of the linked list
2. At given ‘n’th position.
3. At the end of the linked list

- Find the length of a linked list.
- Search for an element /data in a linked list.