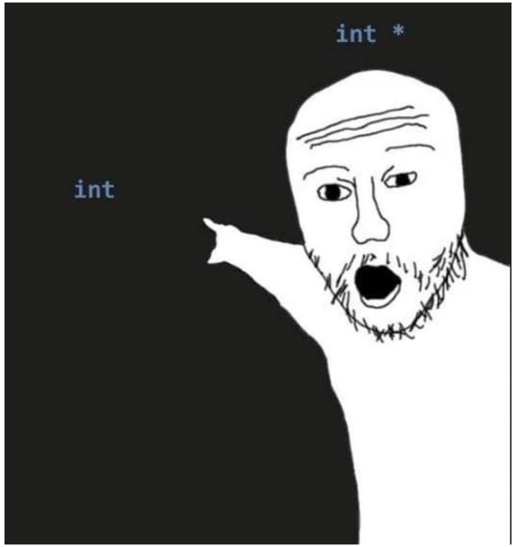


POINTERS

Week 14

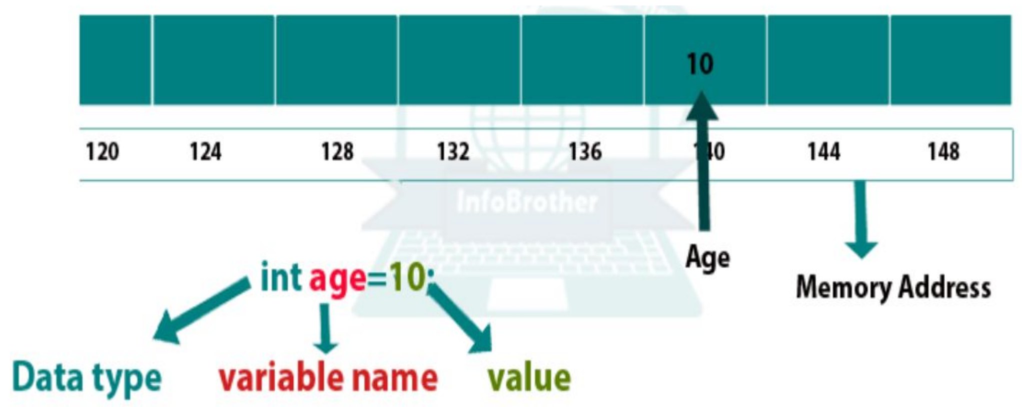
Sumaiyah Zahid

Address Space

Name	=	FAST
Value / Occupants	=	Students
Address	=	St-4, Sector 17-D, National Hwy 5, Karachi

Address Space

```
int x = 10;
```



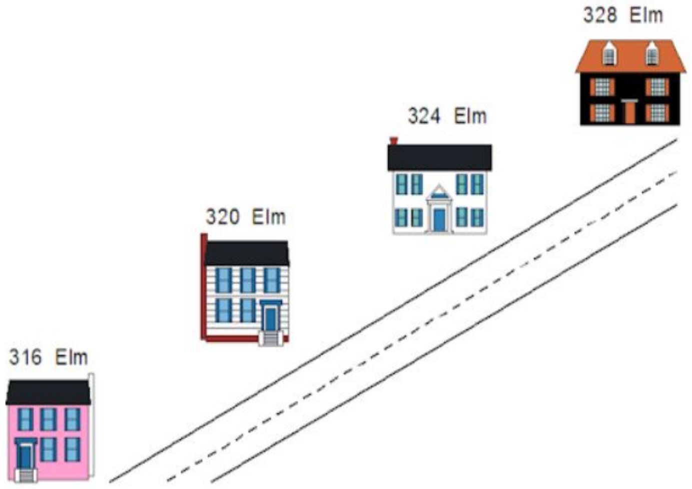
```
Name      = x

Value     = 10

Address  = Any memory location
```

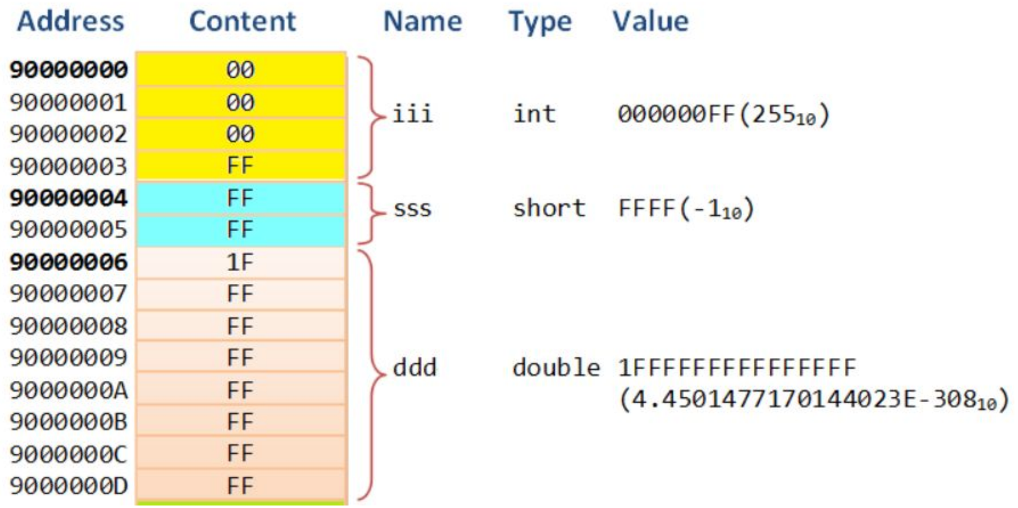
You can access that memory location using **&** operator.

Address Space

Like houses, each variable has a unique address that gets larger as you move along the street or through memory.

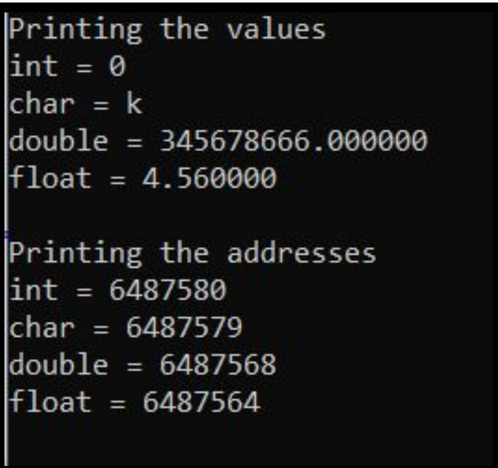
The contents of a variable, like the occupants of a house, can change over time, but the address of a variable, like the address of a house, is fixed and does not change (unless the house is picked up by a tornado and carried over a rainbow, which almost never happens).





Address	Content	Name	Type	Value
90000000	00	iii	int	000000FF(255_{10})
90000001	00			
90000002	00			
90000003	FF			
90000004	FF	sss	short	FFFF(-1_{10})
90000005	FF			
90000006	1F	ddd	double	1FFFFFFFFFFFFFFF ($4.4501477170144023E-308_{10}$)
90000007	FF			
90000008	FF			
90000009	FF			
9000000A	FF			
9000000B	FF			
9000000C	FF			
9000000D	FF			

```c
#include <stdio.h>
int main()
{
    int a=0;
    char b='k';
    double c=345678666;
    float d=4.56;
   printf("Printing the values\n");
   printf("int = %d\nchar = %c\ndouble = %lf\nfloat = %f\n",a,b,c,d);

   printf("Printing the addresses\n");
   printf("int = %d\nchar = %d\ndouble = %d\nfloat = %d \n", &a, &b,
&c, &d);

   return 0;
}
```



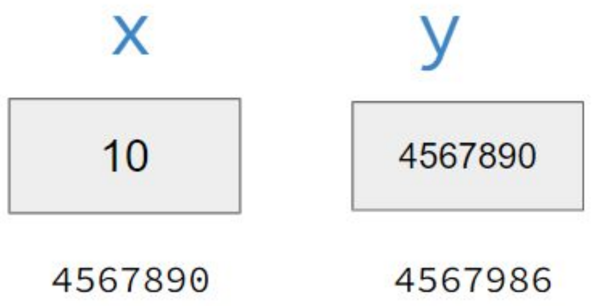
```
Printing the values
int = 0
char = k
double = 345678666.000000
float = 4.560000

Printing the addresses
int = 6487580
char = 6487579
double = 6487568
float = 6487564
```

POINTERS DEFINITION

To hold any address of variable, computer needs another storage location.

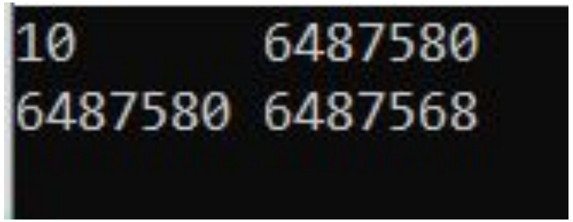
Pointer is a variable which holds address of another variable.

```
int x = 10;

int *y = &x;
```



POINTERS DEFINITION

```
& Address Operator

* Dereferencing Operator

int x = 10;

int *y = &x; or int *y; y = &x;

printf("%d %d\n",x,y);
printf("%d %d\n", &x, &y);
```

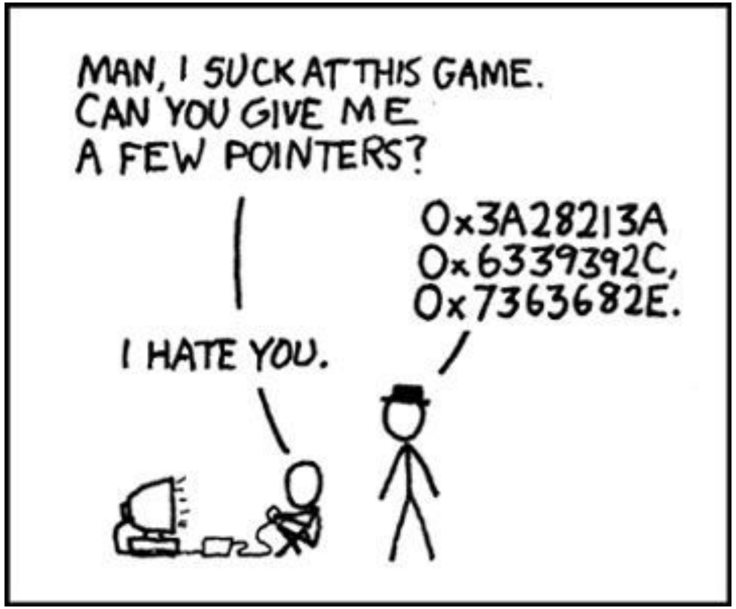


```
10          6487580
6487580 6487568
```

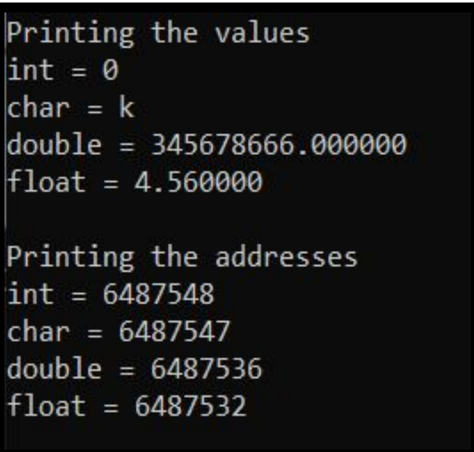
POINTERS DEFINITION

Interestingly we can access value of x through y variable too.

```
y  = holds the address of x
*y = pointer accessing value
     of x

printf("%d %d\n",y,*y);
```


MAN, I SUCK AT THIS GAME. CAN YOU GIVE ME A FEW POINTERS?
0x3A28213A 0x6339392C, 0x7363682E.
I HATE YOU.

```c
#include <stdio.h>
int main()
{
    int a=0;              int *ap;
    char b='k';           char *bp;
    double c=345678666;double *cp;
    float d=4.56;         float *dp;

   ap = &a; bp = &b; cp = &c; dp=&d;
   printf("Printing the values\n");
   printf("int = %d\nchar = %c\ndouble = %lf\nfloat = %f\n", *ap, *bp,
*cp, *dp);

   printf("Printing the addresses\n");
   printf("int = %d\nchar = %d\ndouble = %d\nfloat = %d \n", ap, bp,
cp, dp);

   return 0;
}
```



```
Printing the values
int = 0
char = k
double = 345678666.000000
float = 4.560000

Printing the addresses
int = 6487548
char = 6487547
double = 6487536
float = 6487532
```

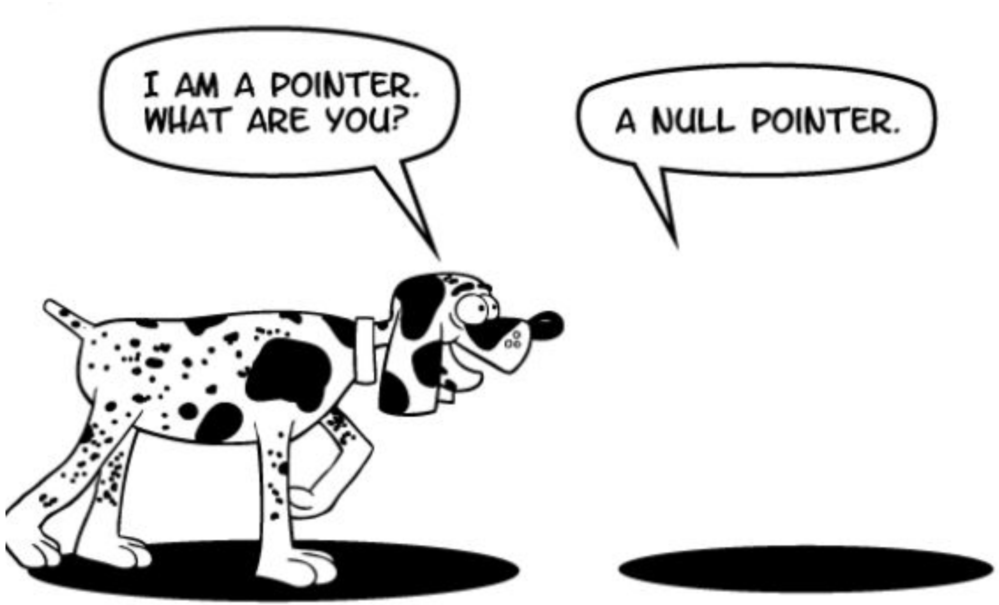
Why POinters?

- Increases the execution speed of program.
- Used for dynamic memory allocation.
- Pass by reference.
- Pointers makes possible to return more than one value in functions.
- To access variables that are declared outside the functions.
- Strings and arrays are more efficient with pointers.

NULL Pointer

"I points nothing"

```
int *p= NULL;
```



DANGLING POINTER

A pointer pointing to a memory location that has been deleted (or freed) is called dangling pointer.

Sometimes the programmer fails to initialize the pointer with a valid address, then this type of initialized pointer is known as a dangling pointer in C.

What are Dangling Pointers ?
Pointers created by Drunk Coders.

Pointers Practice

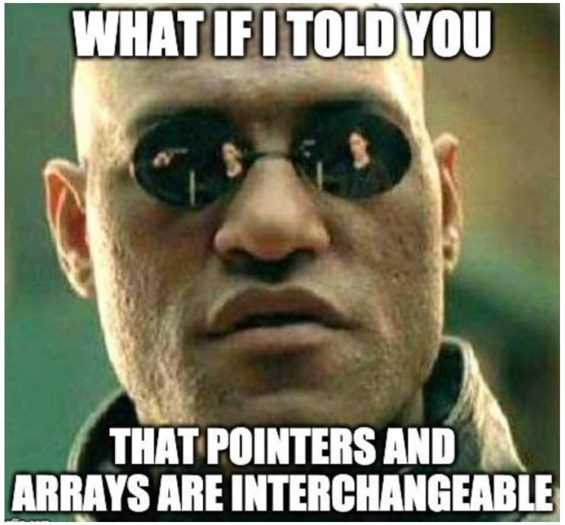
```
float num1 = 5.3;
float *ptr1 = &num1; float *ptr2 = ptr1;
printf("%0.1f, %0.1f\n", *ptr1, *ptr2);

float num2 = 7.6; ptr2 = &num2;
printf("%0.1f\n", *ptr1 + *ptr2);

float *ptr3 = ptr1;
*ptr1 = 2.2;  *ptr2 = *ptr3; *ptr1 = 1.1;
printf("%0.1f, %0.1f, %0.1f\n", *ptr1, *ptr2, *ptr3);

ptr1 = ptr2;  ptr2 = ptr3; *ptr1 = 7;
printf("%0.1f, %0.1f, %0.1f\n", *ptr1, *ptr2, *ptr3);
```

Pointers Arithmetic

```
int x = 10;
int *y = &x, *z;
*y = 5;
*y = *y + 10;
++*y;
(*y)++;
z = y;
++z;            // pointer increment
z = z + 10;
```

Arrays As a Pointers

```
int marks[]={90,86,89,76,91};

printf("%d", marks[2]);

printf("%d", marks);  ????
```



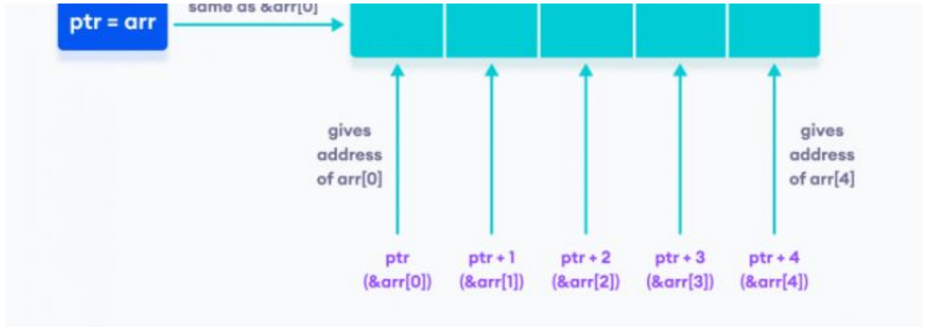
Array name holds the starting address of that array i.e.

```
    marks = & (marks[0])
```

Arrays are not variables, but pointer-variable.

Arrays As a Pointers

```
int arr[]={90,86,89,76,91};

int *ptr=arr;

ptr = arr

arr = &arr[0]
```

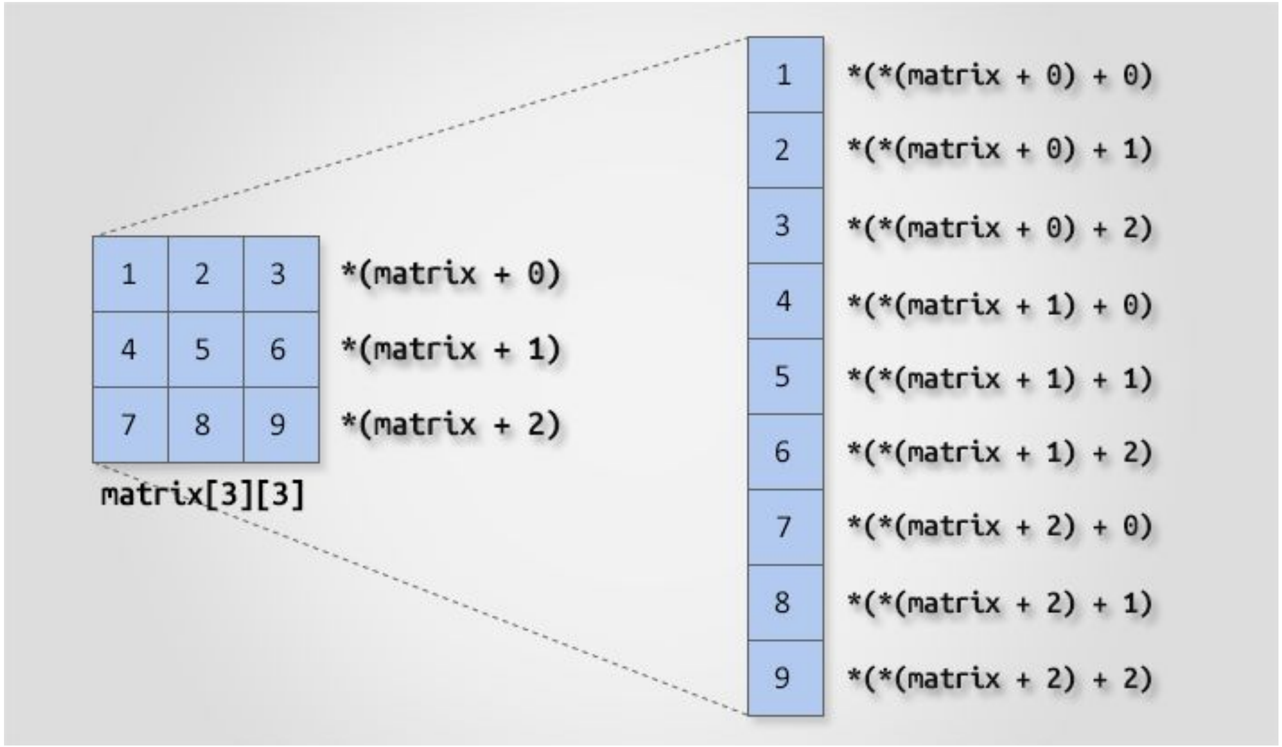


```
arr[0] = *(arr +0) or *(ptr +0)  // Array indexing is actually
arr[1] = *(arr +1) or *(ptr +1)  //dereferencing memory location
arr[i] = *(arr +i) or *(ptr +i)  //with pointer addition
```

Arrays As a Pointers

```
#include <stdio.h>
int main()
{
 int num[ ] = { 24, 34, 12, 44, 56, 17 } ;
 int i, *j ;
 j = num;
 for ( i = 0 ; i <= 5 ; i++ )
 {
 printf ( "\naddress = %u ", j ) ;
 printf ( "element = %d", *j ) ;
 j++ ; /* increment pointer to point to next location */
 }
}
```

2D Arrays As a Pointers



Strings As a Pointers

```
char name[ ] = "FAST" ;
char *ptr ;
ptr = name ; /* store base address of string */
while ( *ptr != `\0' )
{
    printf ( "%c", *ptr ) ;
    ptr++ ;
}
```

Strings As a Pointers

```
char str[ ] = "Hello" ;
char *p = "Hello" ;
```

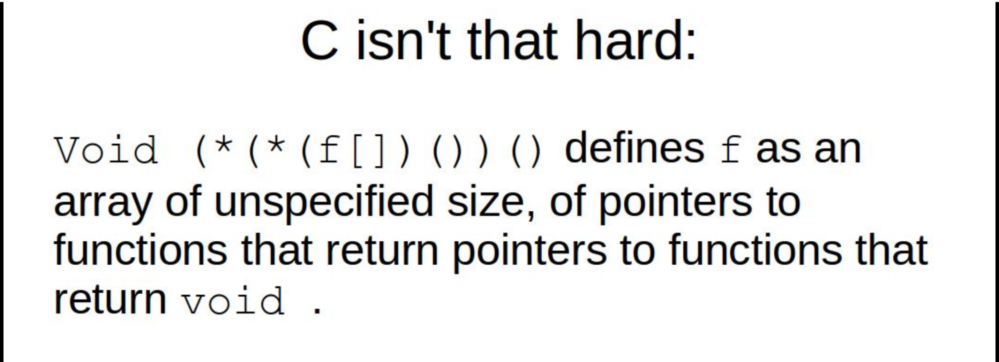
- We cannot assign a string to another, whereas, we can assign a char pointer to another char pointer.

- Once a string has been defined it cannot be initialized to another set of characters. Unlike strings, such an operation is perfectly valid with char pointers.

Strings As a Pointers

```
char str1[ ] = "Hello" ;
char str2[10] ;
char *s = "Good Morning" ;
char *q ;
str2 = str1 ; /* error */
q = s ; /* works */
------------------------------

char str1[ ] = "Hello" ;
char *p = "Hello" ;
str1 = "Bye" ; /* error */
p = "Bye" ; /* works */
```

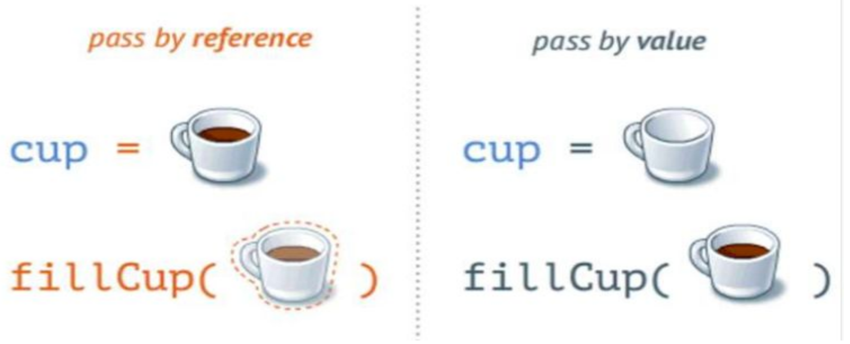
Functions & Pointers



C isn't that hard:

`Void (*(*(f[])())()` defines `f` as an array of unspecified size, of pointers to functions that return pointers to functions that return `void` .

Calling Functions



Call by value

- Copy of argument passed to function
- Changes in function do not affect original
- Use when function does not need to modify argument
- Avoids accidental changes

All the examples mentioned before are using call by value.

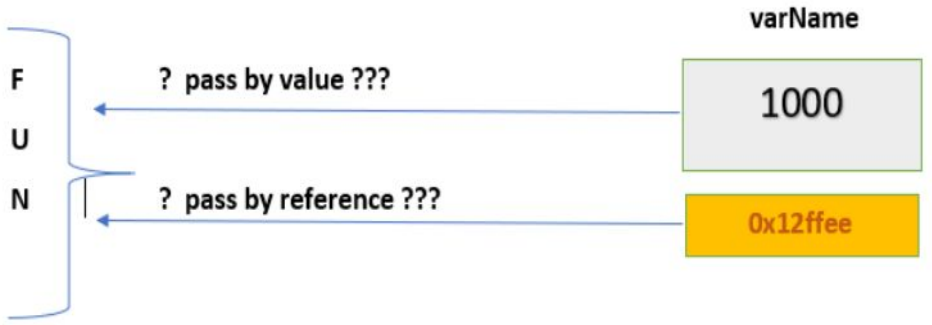
Call by reference (We will study later with Pointers)

- Passes original argument
- Changes in function effect original
- Only used with trusted functions

Pass By Reference

```
void function( int *p);
```

```
int *p,num=6;
p = &num;
function( p );
```

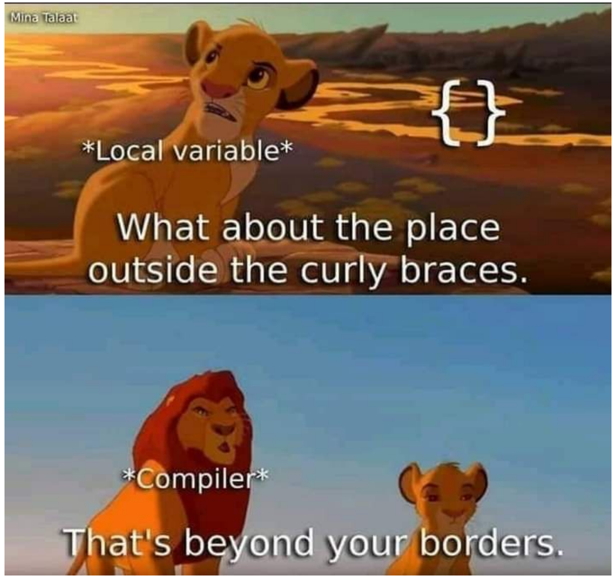


```c
#include <stdio.h>
int cube(int *a);
int main()
{
   int num=4, *p;
   p=&num;
   cube(p); // cube(&num);
   printf("%d", num);
   return 0;
}
int cube(int *a)   // a is an alias or nickname of p
{
     *a= (*a) * (*a) *(*a);
}
```

Returning A Pointer

```
int *function();
```

```
int *ptr;
ptr=function();
```


Mina Talaat
{}
Local variable
What about the place outside the curly braces.
Compiler
That's beyond your borders.

Never return a pointer to local variable from a function.

Local variable exists only inside the function and as soon as function ends the variable cease to exists. Even though the address returned by the function is assigned to ptr inside main(), the variable to which ptr points is no longer available. On dereference the ptr you will get some garbage value.

Returning A Pointer

Static variables have a property of preserving their value even after they are out of their scope!

Static variables help local variables to exist even when the function ends.

There is an always ocean of knowledge which depends how deep you want to dive.
https://www.geeksforgeeks.org/static-variables-in-c/

```c
#include <stdio.h>
int * cube(int a);
int main()
{
   int num=4, *p;
   p= cube(num);
   printf("%d", *p);
   return 0;
}
int * cube(int a)
{
     static int x=0;
     x= a*a*a;
     return &x;
}
```

Passing Arrays By Reference

Array name is already a pointer, so passing an array name is itself passing by reference.

That means, if you update array element in function, it also changes in main.

offthepack • 12m

wait can u explain pointers to me

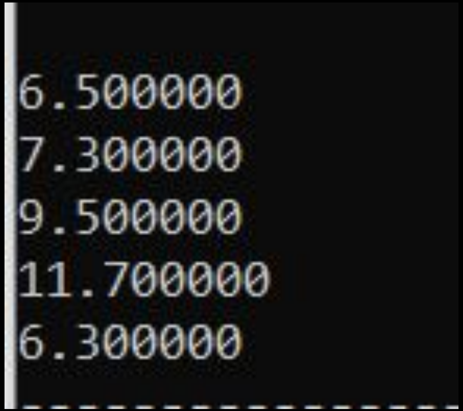
4

Noah5900 • 8m

just keep putting & and * until it works

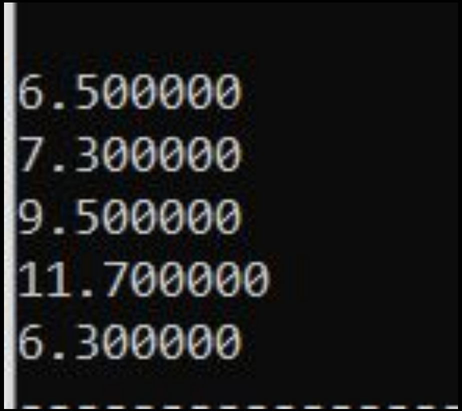
3

```c
#include <stdio.h>
float array_add(float arr[], int size) ;
int main()
{
   float array[5]={1.5, 2.3,4.5,6.7,1.3};
   array_add(array, 5);

   int i;
   for(i=0; i<5; i++)
      printf("%f", array[i]);
   return 0;
}
float array_add(float arr[], int size)
{
     int i;
     for(i=0; i<size; i++)
        arr[i]=arr[i]+5;
}
```



```
6.500000
7.300000
9.500000
11.700000
6.300000
```

```c
#include <stdio.h>
float array_add(float *p, int size)    ;
int main()
{
   float array[5]={1.5, 2.3,4.5,6.7,1.3};
   array_add(array, 5);

   int i;
   for(i=0; i<5; i++)
      printf("%f", array[i]);
   return 0;
}
float array_add(float *p, int size)
{
     int i;
     for(i=0; i<size; i++)
        p[i]=p[i]+5; // *p=*p+5; p++;
}
```



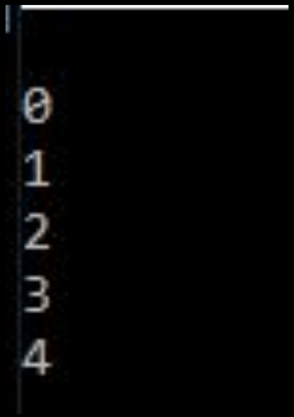
```
6.500000
7.300000
9.500000
11.700000
6.300000
```

Returning Array Using Pointers

We can return array by returning the address of array first element i.e. array name.

Your array should be static.

```c
#include <stdio.h>
int * initialize_array(int size);
int main()
{
   int *p,i;
   p=initialize_array(5);
   for(i=0; i<5; i++)
      printf("\n%d", *p++);
   return 0;
}
int * initialize_array(int size)
{
     static int arr[10];
     int i;
     for(i=0; i<size; i++)
        arr[i]=i;
     return arr;
}
```

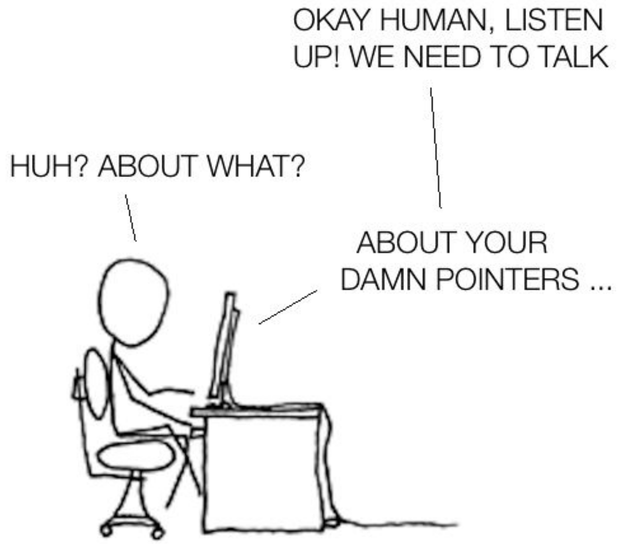


```

0
1
2
3
4
```

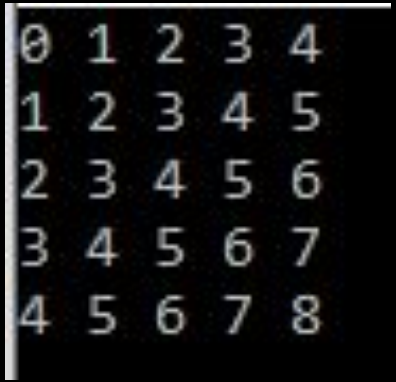
Passing A 2d array To Function

A two-dimensional array is treated as an array of arrays.

That is, when we access the array using only one subscript, we get a pointer to the corresponding row.


OKAY HUMAN, LISTEN UP! WE NEED TO TALK
HUH? ABOUT WHAT?
ABOUT YOUR DAMN POINTERS ...

```c
#include <stdio.h>
void initialize_array(int m, int n,int(*arr)[n]);
int main()
{
   int m=5, n=5,i,j;
   int arr[m][n];
   initialize_array(m, n, arr);
     for(i=0; i<m; i++)
     {
       for(j=0; j<n; j++)
         printf("%d ", *(*(arr + i) + j));
       printf("\n");
     }
   return 0;}
void initialize_array(int m, int n,int(*arr)[n])
{    int i,j;
     for(i=0; i<m; i++)
       for(j=0; j<n; j++)
           *(*(arr + i) + j) = i + j;
}
```

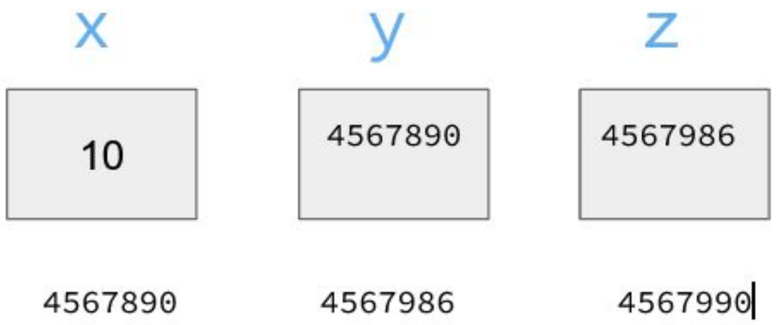


```
0 1 2 3 4
1 2 3 4 5
2 3 4 5 6
3 4 5 6 7
4 5 6 7 8
```

Double Pointer

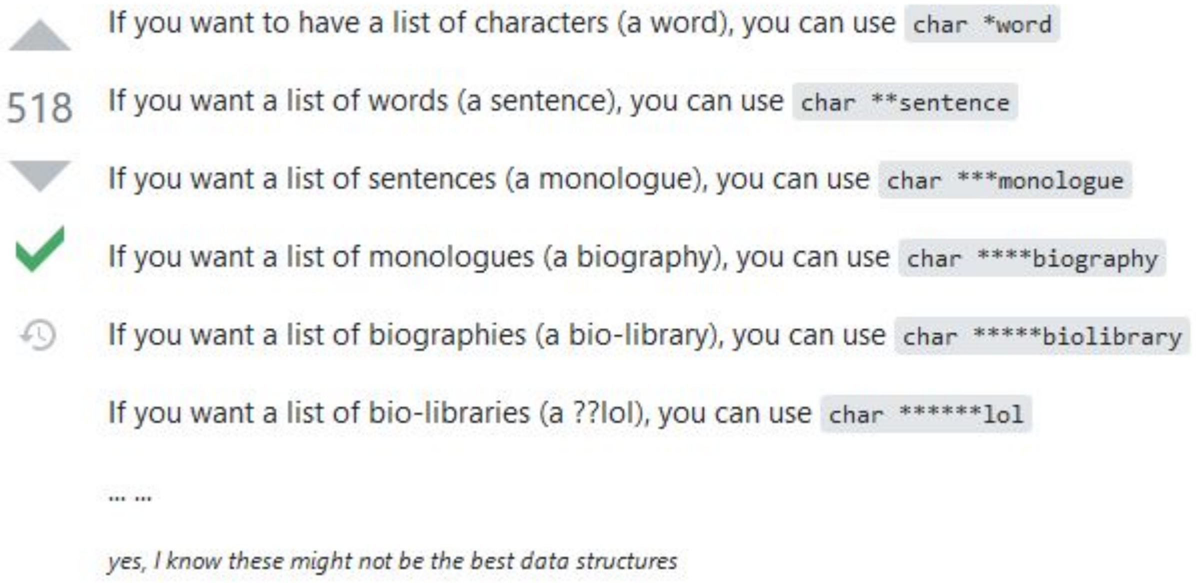
Double Pointer is a pointer to pointer.

```c
int x = 10;
int *y = &x;
int **z = &y;
```



```c
printf("%d %d %d\n", x, *y, **z);
```

Why Use Double Pointers


518

If you want to have a list of characters (a word), you can use `char *word`

If you want a list of words (a sentence), you can use `char **sentence`

If you want a list of sentences (a monologue), you can use `char ***monologue`

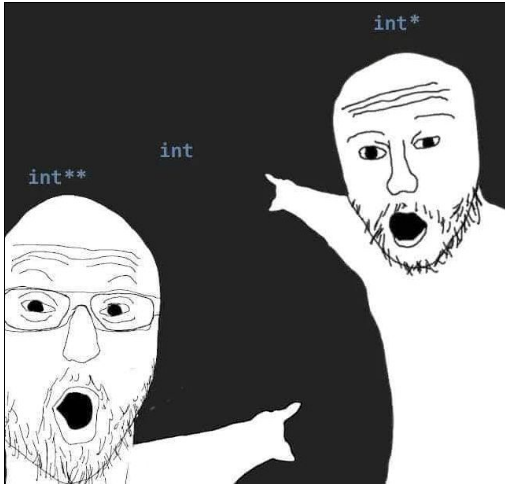
If you want a list of monologues (a biography), you can use `char ****biography`

If you want a list of biographies (a bio-library), you can use `char *****biolibrary`

If you want a list of bio-libraries (a ??lol), you can use `char ******lol`

... ...

yes, I know these might not be the best data structures

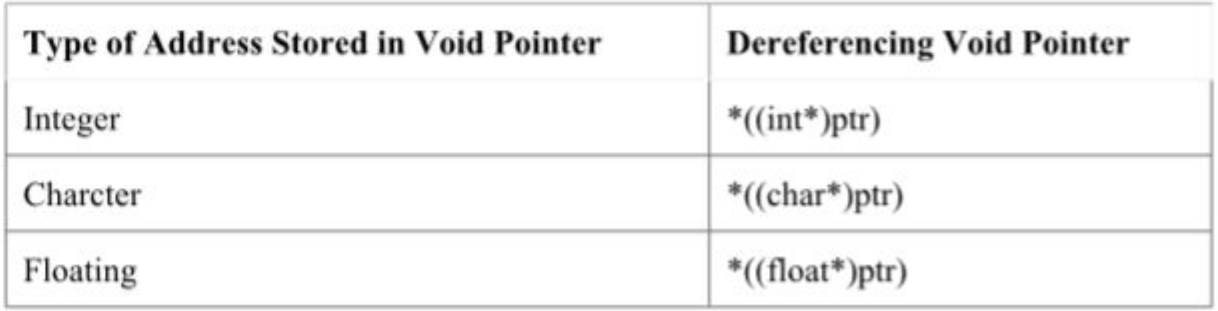


VOID POINTER

It is a general purpose pointer.

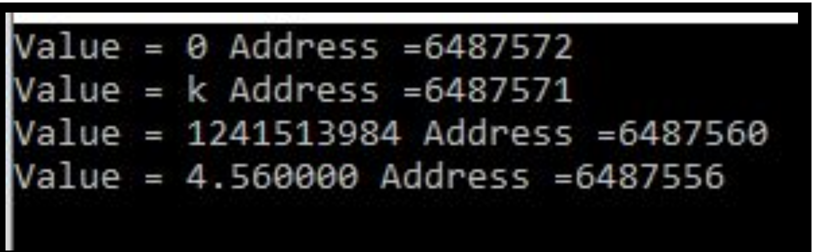
A pointer that has no associated data type with it.

It holds address of any type and can be typcasted to any type.



Type of Address Stored in Void Pointer	Dereferencing Void Pointer
Integer	*((int*)ptr)
Charcter	*((char*)ptr)
Floating	*((float*)ptr)

```c
#include <stdio.h>
int main()
{
    int a=0;
    char b='k';
    double c=345678666;
    float d=4.56;
    void *ptr=NULL;
   ptr = &a;
   printf("Value = %d Address =%d\n", *(int *)ptr, ptr);
   ptr = &b;
   printf("Value = %c Address =%d\n", *(char *)ptr, ptr);
   ptr = &c;
   printf("Value = %ld Address =%d\n", *(double *)ptr, ptr);
   ptr = &d;
   printf("Value = %f Address =%d\n", *(float *)ptr, ptr);

   return 0;
}
```



```
Value = 0 Address =6487572
Value = k Address =6487571
Value = 1241513984 Address =6487560
Value = 4.560000 Address =6487556
```

Casting Pointer

When assigning a memory address of a variable of one type to a pointer that points to another type it is best to use the cast operator to indicate the cast is intentional (this will remove the warning).

```
int V = 101;
float *P = (float *) &V; /* Casts int address to float * */
```

Removes warning, but is still a somewhat unsafe thing to do.

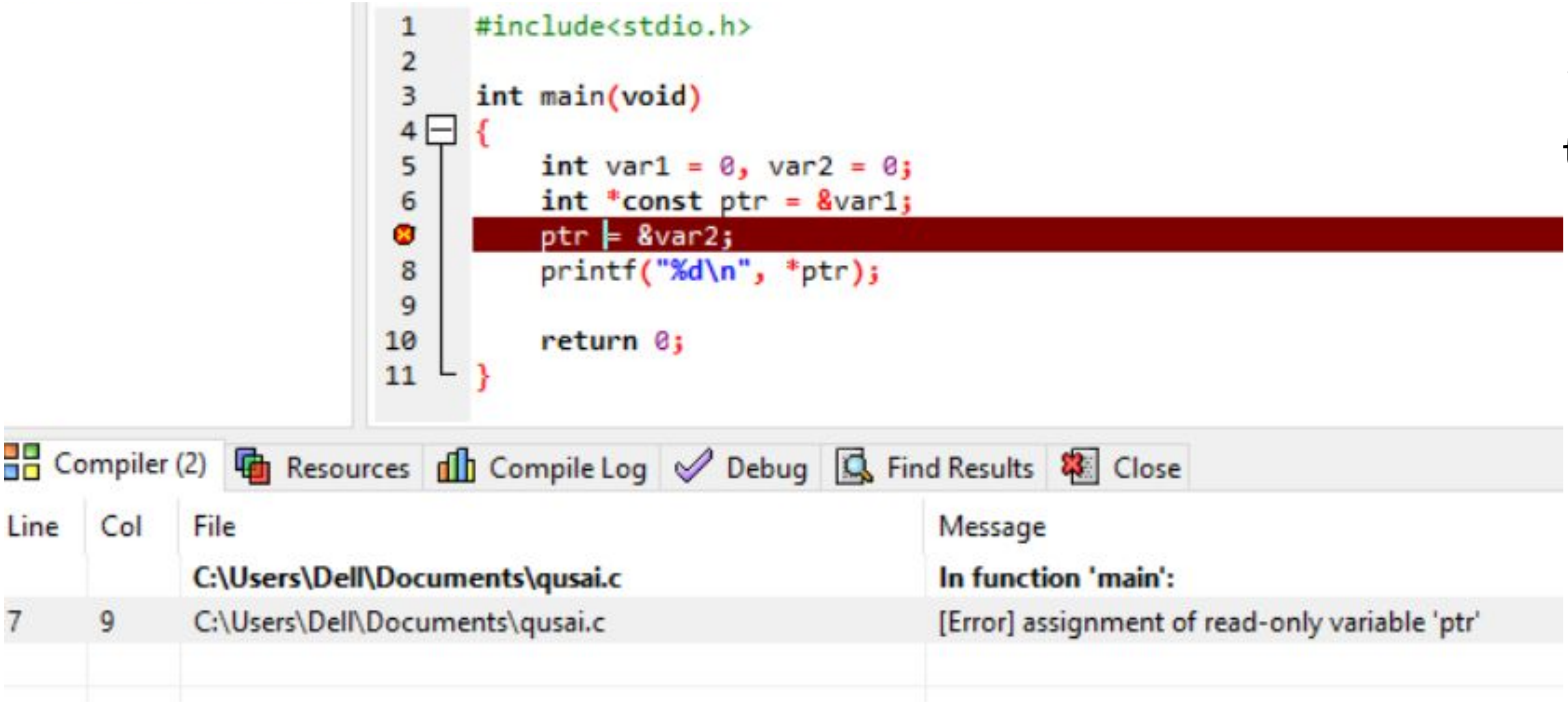
Constant Pointer

It is a pointer that cannot change the address its holding.

Once a constant pointer points to a variable then it cannot point to any other variable.

```
int * const ptr;
```

Constant Pointer



```
#include<stdio.h>

int main(void)
{
    int var1 = 0, var2 = 0;
    int *const ptr = &var1;
    ptr = &var2;
    printf("%d\n", *ptr);

    return 0;
}
```

Compiler (2) | Resources | Compile Log | Debug | Find Results | Close

```
Line  Col  File                              Message
           C:\Users\Dell\Documents\qusai.c   In function 'main':
7     9    C:\Users\Dell\Documents\qusai.c   [Error] assignment of read-only variable 'ptr'
```

- t

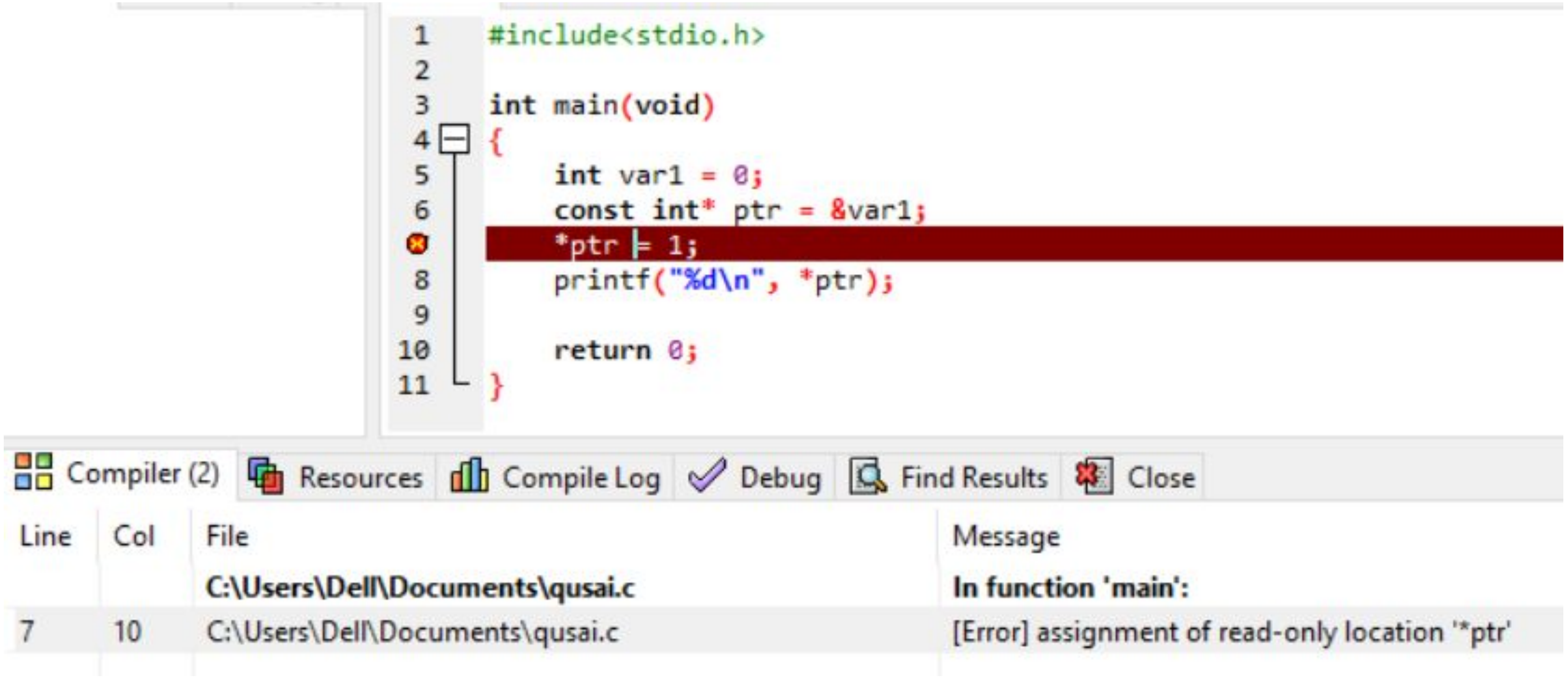
Pointer To Constant

A pointer through which one cannot change the value of variable it points is known as a pointer to constant.

These type of pointers can change the address they point to but cannot change the value kept at those address.

```
const int * ptr;
```

Pointer To Constant



```c
#include<stdio.h>

int main(void)
{
	int var1 = 0;
	const int* ptr = &var1;
	*ptr = 1;
	printf("%d\n", *ptr);

	return 0;
}
```

Line	Col	File	Message
		C:\Users\Dell\Documents\qusai.c	**In function 'main':**
7	10	C:\Users\Dell\Documents\qusai.c	[Error] assignment of read-only location '*ptr'

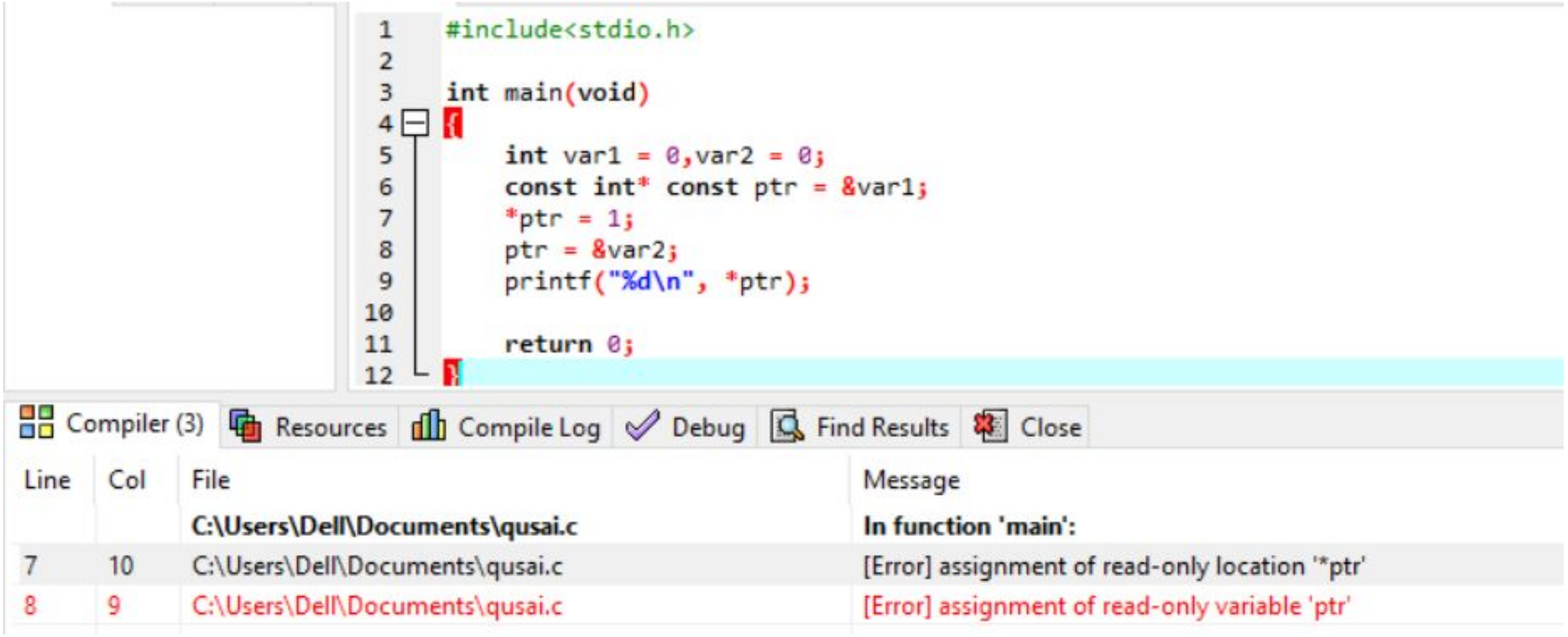
```c
#include <stdio.h>
int cube(const int *x,int *a);
int main()
{
   int num=4, result=0;
   cube(&num, &result);
   printf("%d", num);
   return 0;
}
int cube(const int *x,int *a)
{
     *a= (*x) * (*x) *(*x); // This is correct
    //*x= (*x) * (*x) *(*x); This is wrong
}
```

Constant Pointer To Constant

A constant pointer to constant is a pointer that can neither change the address its pointing to and nor it can change the value kept at that address.

```
const int * const ptr;
```

Constant Pointer To Constant



```c
#include<stdio.h>

int main(void)
{
	int var1 = 0,var2 = 0;
	const int* const ptr = &var1;
	*ptr = 1;
	ptr = &var2;
	printf("%d\n", *ptr);

	return 0;
}
```

```
Line  Col  File                                Message
           C:\Users\Dell\Documents\qusai.c     In function 'main':
7     10   C:\Users\Dell\Documents\qusai.c     [Error] assignment of read-only location '*ptr'
8     9    C:\Users\Dell\Documents\qusai.c     [Error] assignment of read-only variable 'ptr'
```

Function Pointer

A pointer which keeps address of a function is known as function pointer.

```
void *(*ptr)();
```

- We are simply declaring a double pointer.
- Write () symbol after “Double Pointer“.
- void represents that, function is not returning any value.
- () represents that, function is not taking any parameter.

Function Pointer

```
ptr = &display;  // initializing the pointer
(*ptr)();    // calling a function


(*ptr)() = (*ptr)();
         = (*&display)();
         = (display)();
         = display();
```

Requirement	Declaration of Function Pointer	Initialization of Function Pointer	Calling Function using Pointer
Return Type : None Parameter : None	`void *(*ptr)();`	`ptr = &display;`	`(*ptr)();`

Return Type : Integer
Parameter : None
	`int *(*ptr)();`	`ptr = &display;`	`int result;`
			`result = (*ptr)();`

Return Type : Float
Parameter : None
	`float *(*ptr)();`	`ptr = &display;`	`float result;`
			`result = (*ptr)();`

Return Type : Char
Parameter : None
	`char *(*ptr)();`	`ptr = &display;`	`char result;`
			`result = (*ptr)();`

```c
#include <stdio.h>
void smile();    // Function Declaration or Prototype
int main()
{
   void *(*ptr)();
    ptr = &smile;  // Initializing the pointer

   (*ptr)();   // Function Call

   return 0;
}
void smile()     // Function Definition
{
   printf("\nSmile, and the world smiles with you...");
}
```

Function Prototype

```
#include<stdio.h>

char * getName(int *,float *);
```

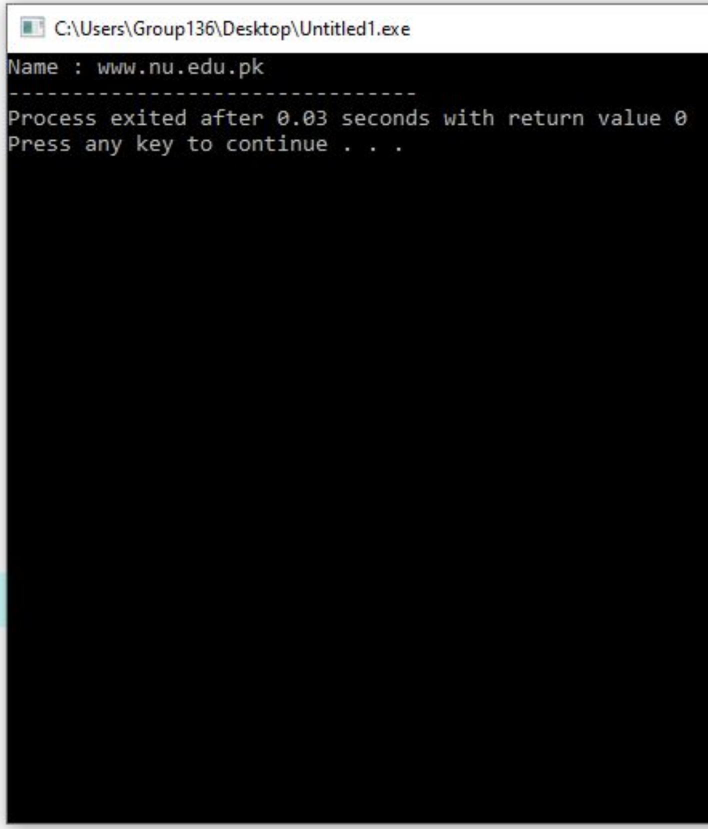
Function Definition

```
char *getName(int *ivar,float *fvar)
{
char *str="www.nu.edu.pk";
return(str);
}
```

Main

```
int main()

{

char *name; int num = 100; float marks = 99.12;

char *(*ptr)(int*,float *);

ptr=&getName;

name = (*ptr)(&num,&marks);

printf("Name : %s",name);

return 0;

}
```

```c
#include<stdio.h>
char *getName(int *ivar,float *fvar)
{
char *str="www.nu.edu.pk";
return(str);
}
int main()
{
char *name;
int num = 100;
float marks = 99.12;

char *(*ptr)(int*,float *);
ptr=&getName;
name = (*ptr)(&num,&marks);
printf("Name : %s",name);
return 0;
}
```


C:\Users\Group136\Desktop\Untitled1.exe

```
Name : www.nu.edu.pk
--------------------------------
Process exited after 0.03 seconds with return value 0
Press any key to continue . . .
```

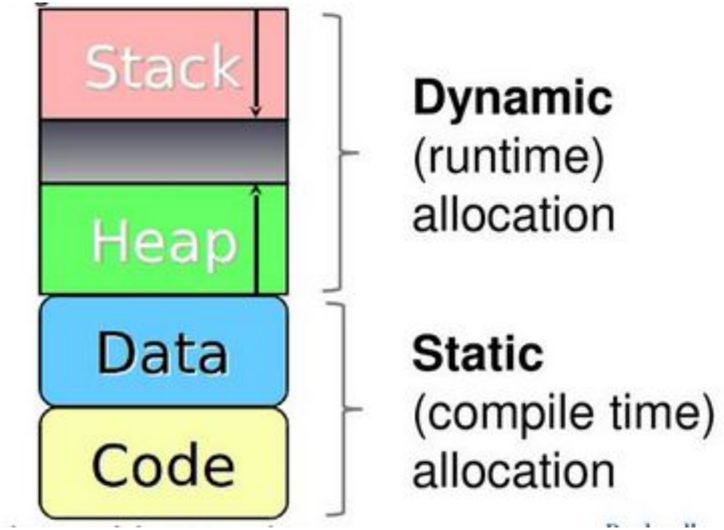
Dynamic Memory Allocation

Static Vs Dynamic Memory Allocation

Programmers can dynamically allocate storage space while the program is running, but cannot create new variable names "on the fly".

For this reason, dynamic allocation requires two criteria:

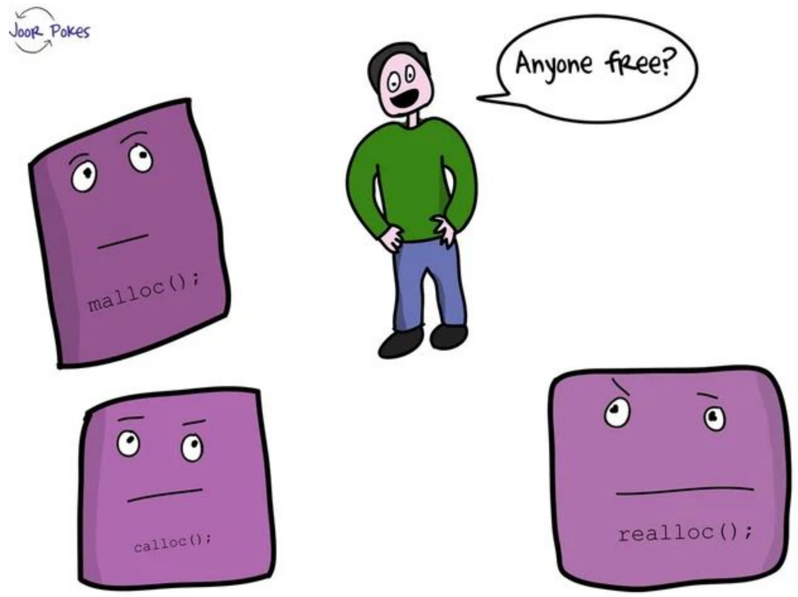
- Creating the dynamic space in memory
- Storing its address in a pointer (so that space can be accessed)


Stack
Heap
Data
Code
Dynamic (runtime) allocation
Static (compile time) allocation

Dynamic Memory Allocation

To allocate memory dynamically, <stdlib.h> has following functions:

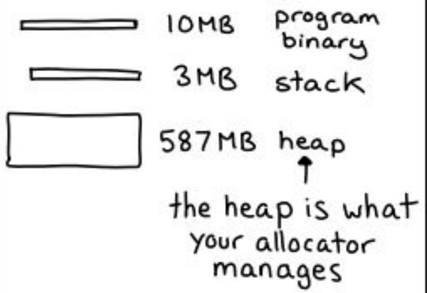
- malloc()
- calloc()
- realloc()
- free()



JULIA EVANS
@børk

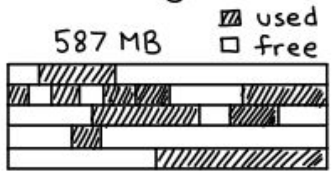
memory allocation

your program has memory

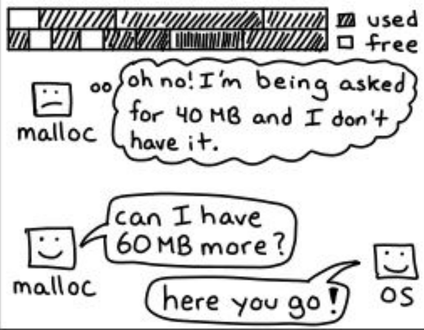


your memory allocator (malloc) is responsible for 2 things.

THING 1: keep track of what memory is used/free.



THING 2: Ask the OS for more memory!



your memory allocator's interface

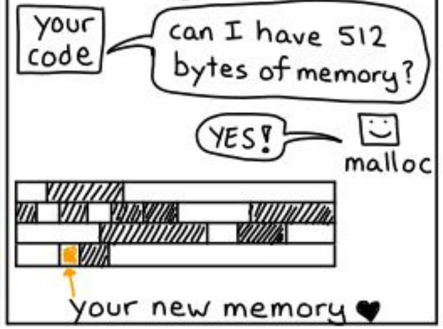
malloc (size_t size)
allocate size bytes of memory & return a pointer to it.

free (void* pointer)
mark the memory as unused (and maybe give back to the OS).

realloc(void* pointer, size_t size)
ask for more/less memory for pointer.

calloc (size_t members, size_t size)
allocate array + initialize to Ø.

malloc tries to fill in unused space when you ask for memory



malloc isn't magic!

it's just a function!

you can always:

→ use a different malloc library like jemalloc or tcmalloc (easy!)

→ implement your own malloc (harder)

C malloc()

"malloc" stands for memory allocation.

It reserves a block of memory of the specified number of bytes. And, returns a pointer of void which can be type casted in any form.

```
ptr = (castType*) malloc(size);

ptr = (float*) malloc(100 * sizeof(float)); // 400 bytes reserved
```

C calloc()

"calloc" stands for contiguous allocation.

It allocates memory and initializes all bits to zero.

```
ptr = (castType*) calloc(n, size);

ptr = (float*) calloc(100 , sizeof(float));

// 400 consecutive bytes reserved
```

C free()

Dynamically allocated memory created with either calloc() or malloc() doesn't get freed on their own.

You must explicitly use free() to release the space.

```
free(ptr);
```

```c
#include <stdio.h>
#include <stdlib.h>
int main()
{
  int n,i, *ptr, sum=0;
  printf("Enter the number of elements: ");
  scanf("%d", &n);
  ptr = (int*) malloc(n * sizeof(int));
   if(ptr == NULL)
  {
     printf("Error! Memory not allocated");
     return 0; }
  printf("Enter elements: ");
  for(i = 0; i < n; ++i) {
    scanf("%d", ptr+i);
    sum += *(ptr + i);
  }
  printf("Sum = %d", sum);
  free(ptr); // deallocating the memory
  return 0;  }
```

```c
#include <stdio.h>
#include <stdlib.h>
int main()
{
  int n,i, *ptr, sum=0;
  printf("Enter the number of elements: ");
  scanf("%d", &n);
  ptr = (int*) calloc(n , sizeof(int));
   if(ptr == NULL)
  {
     printf("Error! Memory not allocated");
     return 0; }
  printf("Enter elements: ");
  for(i = 0; i < n; ++i) {
    scanf("%d", ptr+i);
    sum += *(ptr + i);
  }
  printf("Sum = %d", sum);
  free(ptr); // deallocating the memory
  return 0;  }
```

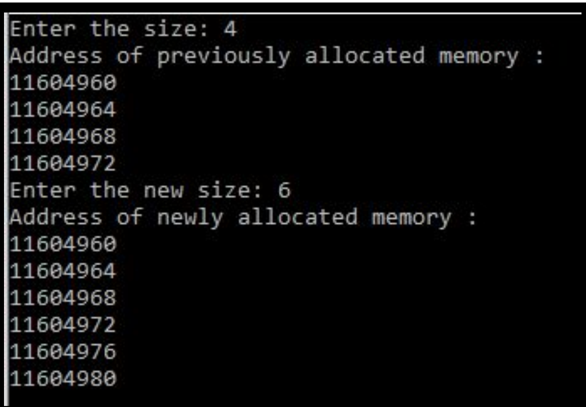
C realloc()

If the dynamically allocated memory is insufficient or more than required, you can change the size of previously allocated memory using the realloc() function.

If enough space doesn’t exist in memory of current block to extend, new block is allocated for the full size of reallocation, then copies the existing data to new block and then frees the old block.

```
ptr = realloc(ptr, size);
```

```c
#include <stdio.h>
#include <stdlib.h>
int main()
{
  int n1,n2,i, *ptr, sum=0;
  printf("Enter the size: ");
  scanf("%d", &n1);
  ptr = (int*) malloc(n1 * sizeof(int));
  printf("Address of previously allocated memory : ");
  for(i = 0; i < n1; i++)
    printf("%d\n", ptr+i);
  printf("Enter the new size: ");
  scanf("%d", &n2);
  ptr = realloc(ptr, n2 * sizeof(int));// reallocating the memory
  printf("Address of newly allocated memory : ");
  for(i = 0; i < n2; i++)
    printf("%d\n", ptr+i);
  free(ptr); // deallocating the memory
  return 0;
}
```



```
Enter the size: 4
Address of previously allocated memory :
11604960
11604964
11604968
11604972
Enter the new size: 6
Address of newly allocated memory :
11604960
11604964
11604968
11604972
11604976
11604980
```

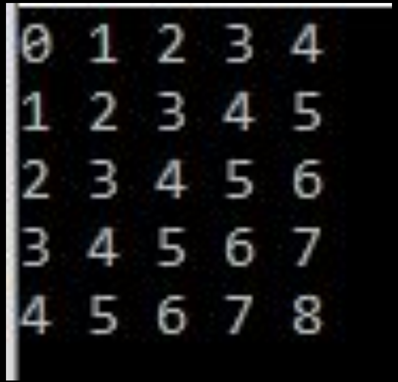
2D Array Using DMA

```
int *arr = (int*) malloc(m*n * sizeof(int));
```

To access any element:

```
*(arr + i*n + j)
```

```c
#include <stdio.h>
void initialize_array(int m, int n,int(*arr)[n]);
int main()
{
   int m=5, n=5,i,j;
   int arr[m][n];
   initialize_array(m, n, arr);
     for(i=0; i<m; i++)
     {
       for(j=0; j<n; j++)
         printf("%d ", *(*(arr + i) + j));
       printf("\n");
     }
   return 0;}
void initialize_array(int m, int n,int(*arr)[n])
{    int i,j;
     for(i=0; i<m; i++)
       for(j=0; j<n; j++)
           *(*(arr + i) + j) = i + j;
}
```



```
0 1 2 3 4
1 2 3 4 5
2 3 4 5 6
3 4 5 6 7
4 5 6 7 8
```

```c
#include <stdio.h>
#include <stdlib.h>
void initialize_array(int m, int n,int *arr)
{    int i,j;
     for(i=0; i<m; i++)
       for(j=0; j<n; j++)
           *(arr + i*n + j) = i + j;  }
int main()
{
   int m=5, n=5,i,j;
   int *arr = (int*) malloc(m*n * sizeof(int));
   initialize_array(m, n, arr);
     for(i=0; i<m; i++)
     {
       for(j=0; j<n; j++)
         printf("%d ", *(arr + i*n + j));
       printf("\n");
     }
   free(arrr); // deallocating the memory
   return 0;}
```

Structures & Pointers

Struct pointer

```
struct student {

char name[20]; float marks;int batch; char city; int roll_num; };


struct student *ptr, student1;

ptr=&student1;

(*ptr).batch= ptr->batch

-> member access operator for pointers
```

```c
#include <stdio.h>
struct student
{
char name[20]; float marks; int batch; char city; int roll_num;
};
int main()
{
  struct student student1, *ptr;
  ptr=&student1;
  printf("Enter your name :");      scanf("%s", ptr->name);
  printf("Enter your marks:");      scanf("%f", &ptr->marks);
  printf("Enter your batch:");      scanf("%d", &ptr->batch);
  printf("Enter your city:");       scanf(" %c",&ptr->city);
  printf("Enter your roll_num:");   scanf("%d", &ptr->roll_num);

  printf("Your data is\nName : %s \nMarks : %f\nBatch :
%d\nCity:%c\nRoll Number :%d", ptr->name, ptr->marks,   ptr->batch,
ptr->city, ptr->roll_num);
  return 0;
}
```

DMA of Struct

Sometimes, the number of struct variables you declared may be insufficient.

You may need to allocate memory during run-time.

```c
#include <stdio.h>
#include <stdlib.h>
struct student
{
char name[20]; float marks; int batch; char city; int roll_num;
};
int main()
{
   struct student *ptr;   int i,n;
   printf("Enter the number of students: ");
   scanf("%d", &n);
   ptr = (struct student *) malloc(n * sizeof(struct student));
   for(i=0; i<n; i++)
     {
  printf("Enter your name :");     scanf("%s", (ptr+i)->name);
  printf("Enter your marks:");     scanf("%f", &(ptr+i)->marks);
  printf("Enter your batch:");     scanf("%d", &(ptr+i)->batch);
     }
   free(ptr); // deallocating the memory
   return 0;}
```

Thank You

PICTURE TIME