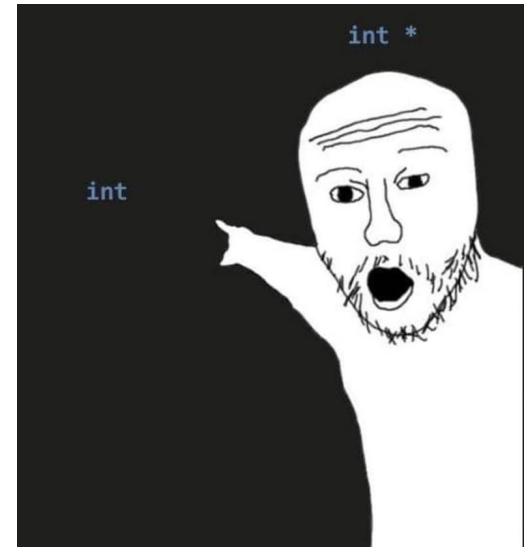




POINTERS



Sumaiyah Zahid

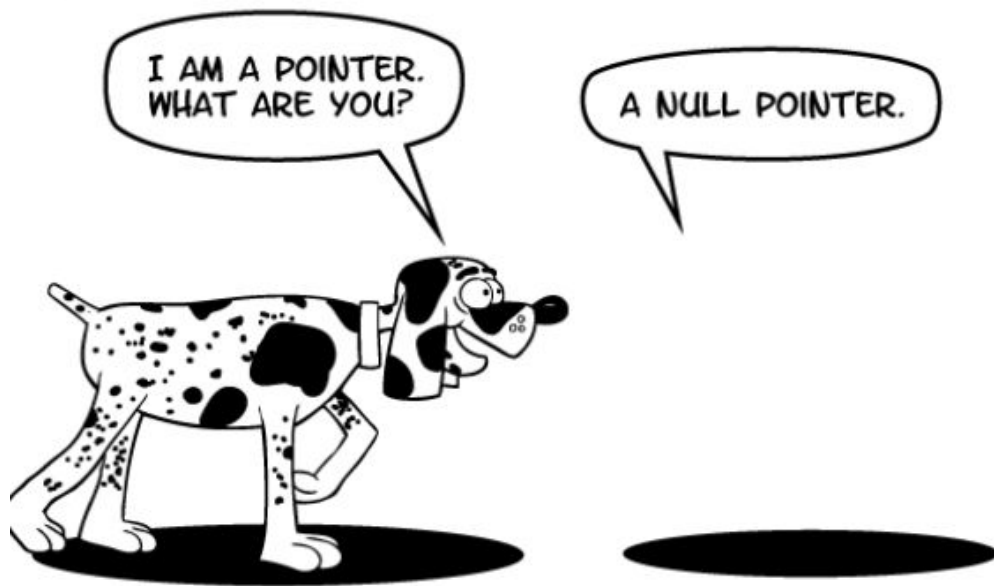
WHY POINTERS?

- 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"

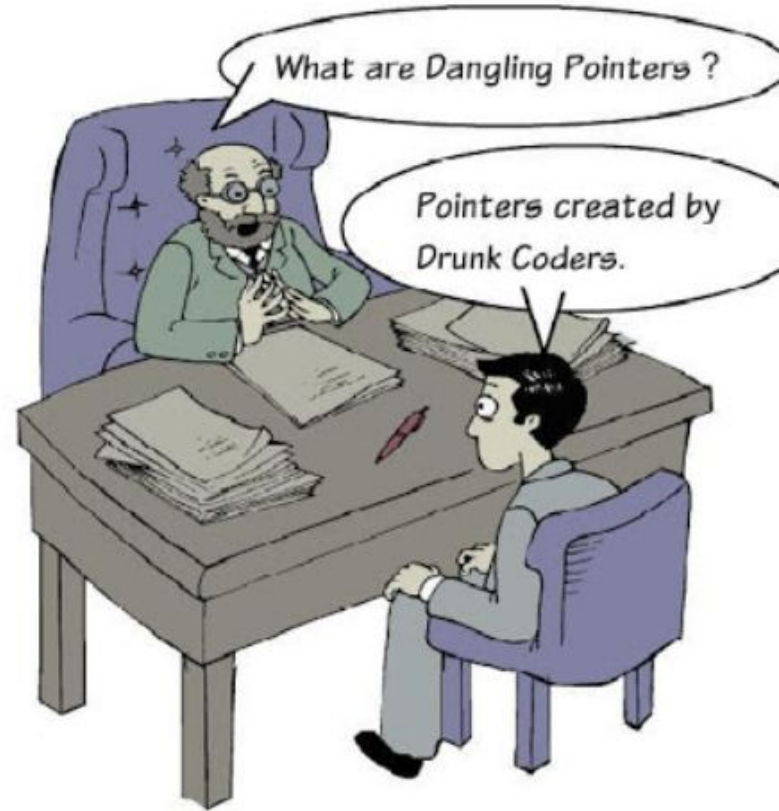
```
int *p= NULL;
```



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.



POINTERS PRACTICE

```
float num1 = 5.3;  
float *ptr1 = &num1; float *ptr2 = ptr1;  
printf("%.1f, %.1f\n", *ptr1, *ptr2);
```

```
float num2 = 7.6; ptr2 = &num2;  
printf("%.1f\n", *ptr1 + *ptr2);
```

```
float *ptr3 = ptr1;  
*ptr1 = 2.2; *ptr2 = *ptr3; *ptr1 = 1.1;  
printf("%.1f, %.1f, %.1f\n", *ptr1, *ptr2, *ptr3);
```

```
ptr1 = ptr2; ptr2 = ptr3; *ptr1 = 7;  
printf("%.1f, %.1f, %.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;
```



MORE POINTERS :(

DOUBLE POINTER

Double Pointer is a pointer to pointer.

```
int x = 10;
```

```
int *y = &x;
```

```
int **z = &y;
```

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



Double Pointers in C be like :

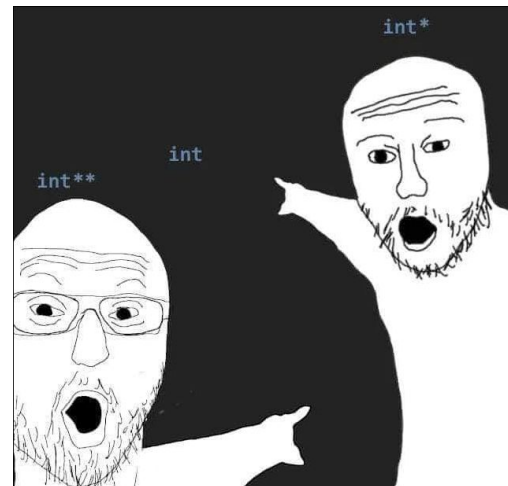


WHY USE DOUBLE POINTERS

- ▲ If you want to have a list of characters (a word), you can use `char *word`
- 518 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 typecasted to any type.

| Type of Address Stored in Void Pointer | Dereferencing Void Pointer |
|--|-----------------------------|
| Integer | <code>*((int*)ptr)</code> |
| Character | <code>*((char*)ptr)</code> |
| Floating | <code>*((float*)ptr)</code> |

```
#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

```
1  #include<stdio.h>
2
3  int main(void)
4  {
5      int var1 = 0, var2 = 0;
6      int *const ptr = &var1;
7      ptr = &var2;
8      printf("%d\n", *ptr);
9
10     return 0;
11 }
```

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' |
| | | | |

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

```
1  #include<stdio.h>
2
3  int main(void)
4  {
5      int var1 = 0;
6      const int* ptr = &var1;
7      *ptr = 1;
8      printf("%d\n", *ptr);
9
10     return 0;
11 }
```

Compiler (2) Resources Compile Log Debug Find Results Close

| 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' |


```
#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

```
1  #include<stdio.h>
2
3  int main(void)
4  {
5      int var1 = 0, var2 = 0;
6      const int* const ptr = &var1;
7      *ptr = 1;
8      ptr = &var2;
9      printf("%d\n", *ptr);
10
11     return 0;
12 }
```

Compiler (3) Resources Compile Log Debug Find Results Close

| 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' |

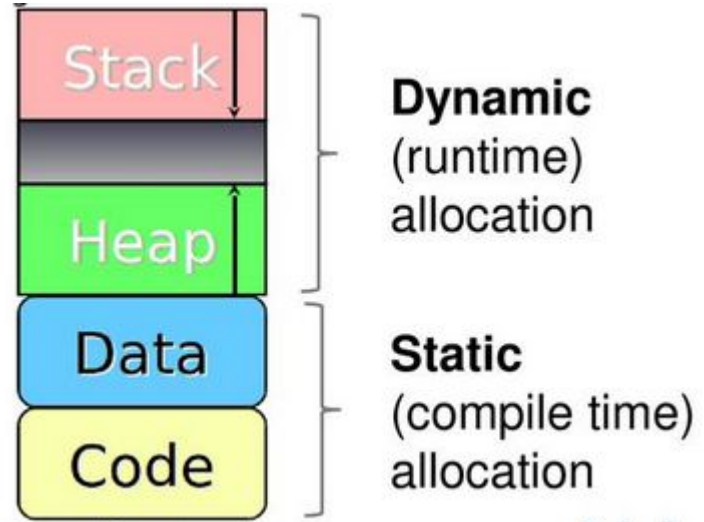
DYNAMIC MEMORY ALLOCATION

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)

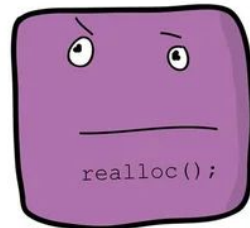
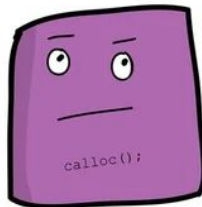


DYNAMIC MEMORY ALLOCATION

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

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

Joop Pokes



memory allocation

16

your program has
memory

10MB program
binary

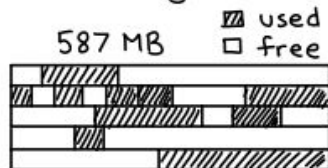
3MB stack

587MB heap

↑
the heap is what
your allocator
manages

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!



malloc: oh no! I'm being asked
for 40 MB and I don't
have it.

malloc: can I have
60 MB more?
OS: here you go!

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).

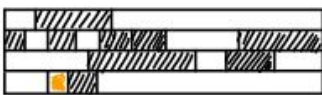
`realloc (void* pointer, size_t size)`
ask for more/less memory for `pointer`.

`calloc (size_t members, size_t size)`
allocate array + initialize to 0.

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

your code: can I have 512
bytes of memory?

YES! malloc



malloc isn't magic!
it's just a function!

you can always:

→ use a different malloc
library like `jemalloc`
or `tcalloc` (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);
```



```
#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; }
```



Int a[10];

`Int* a = (int*)malloc(10*sizeof(int));`

```
#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);
```

```
#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)
```

```
#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 |


```
#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(arr); // deallocating the memory
    return 0;}
```

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



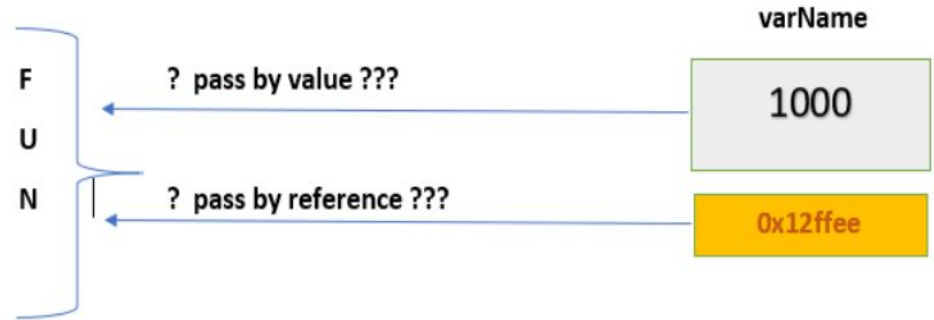
pass by value



PASS BY REFERENCE

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

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



```
#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();
```

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 ceases to exist. 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.

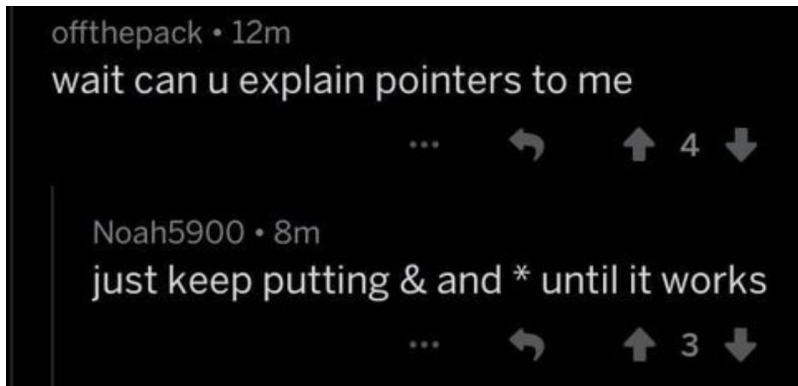
There is an always ocean of knowledge which depends how deep you want to dive.
<https://www.geeksforgeeks.org/static-variables-in-c/>

```
#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

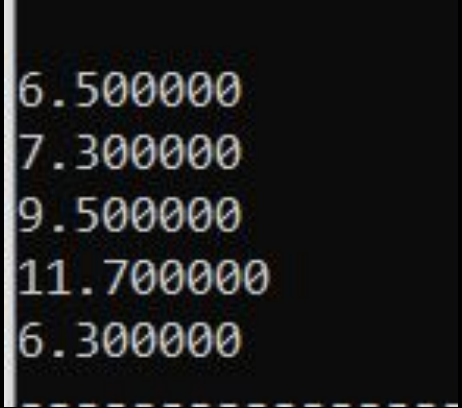
Array name is already a pointer, so passing an array name is itself passing by reference.

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



```
#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);

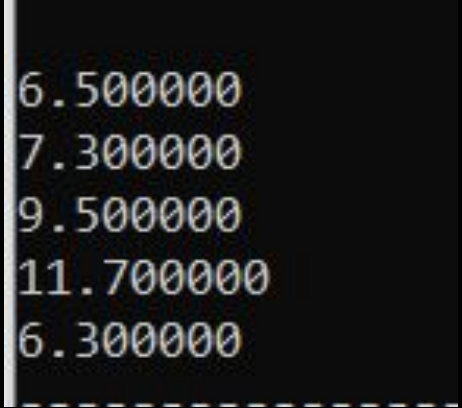
    int i;
    for(i=0; i<5; i++)
        printf("%f", array[i]);
    return 0;
}
float array_add(float arr[], int size)
{
    int i;
    for(i=0; i<size; i++)
        arr[i]=arr[i]+5;
}
```



6.500000
7.300000
9.500000
11.700000
6.300000

```
#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.

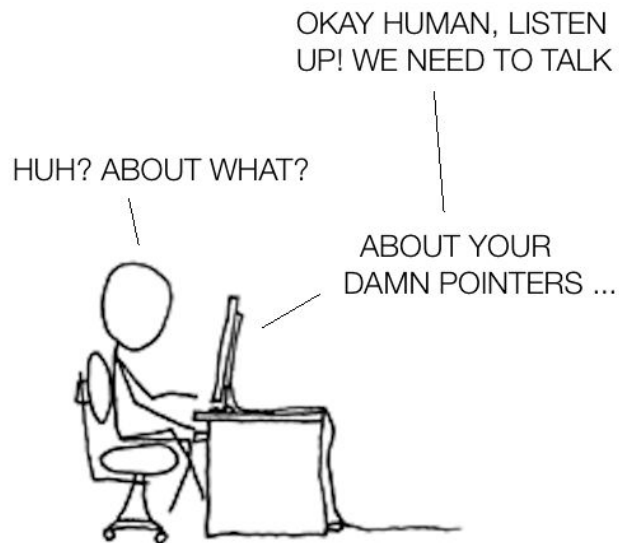
```
#include <stdio.h>
int * initialize_array(int size);
int main()
{
    int *p,i;
    p=initialize_array(5);
    for(i=0; i<5; i++)
        printf("\n%d", *p++);
    return 0;
}
int * initialize_array(int size)
{
    static int arr[10];
    int i;
    for(i=0; i<size; i++)
        arr[i]=i;
    return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



```
#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 |

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 | <code>void *(*ptr)();</code> | <code>ptr = &display;</code> | <code>(*ptr)();</code> |
| Return Type : Integer Parameter : None | <code>int *(*ptr)();</code> | <code>ptr = &display;</code> | <code>int result;</code> <code>result = (*ptr)();</code> |
| Return Type : Float Parameter : None | <code>float *(*ptr)();</code> | <code>ptr = &display;</code> | <code>float result;</code> <code>result = (*ptr)();</code> |
| Return Type : Char Parameter : None | <code>char *(*ptr)();</code> | <code>ptr = &display;</code> | <code>char result;</code> <code>result = (*ptr)();</code> |

```
#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

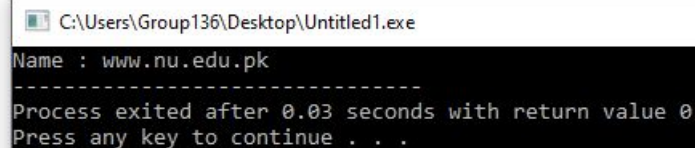
```
char *getName(int *ivar, float *fvar)
{
char *str="www.nu.edu.pk";
return(str);
}
```

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;
}
```

```
#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 . . .
```

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

```
#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.

```
#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