Pointers

Recap

- Pointers Arithmetic
- Array Pointer

Subtraction of two pointers

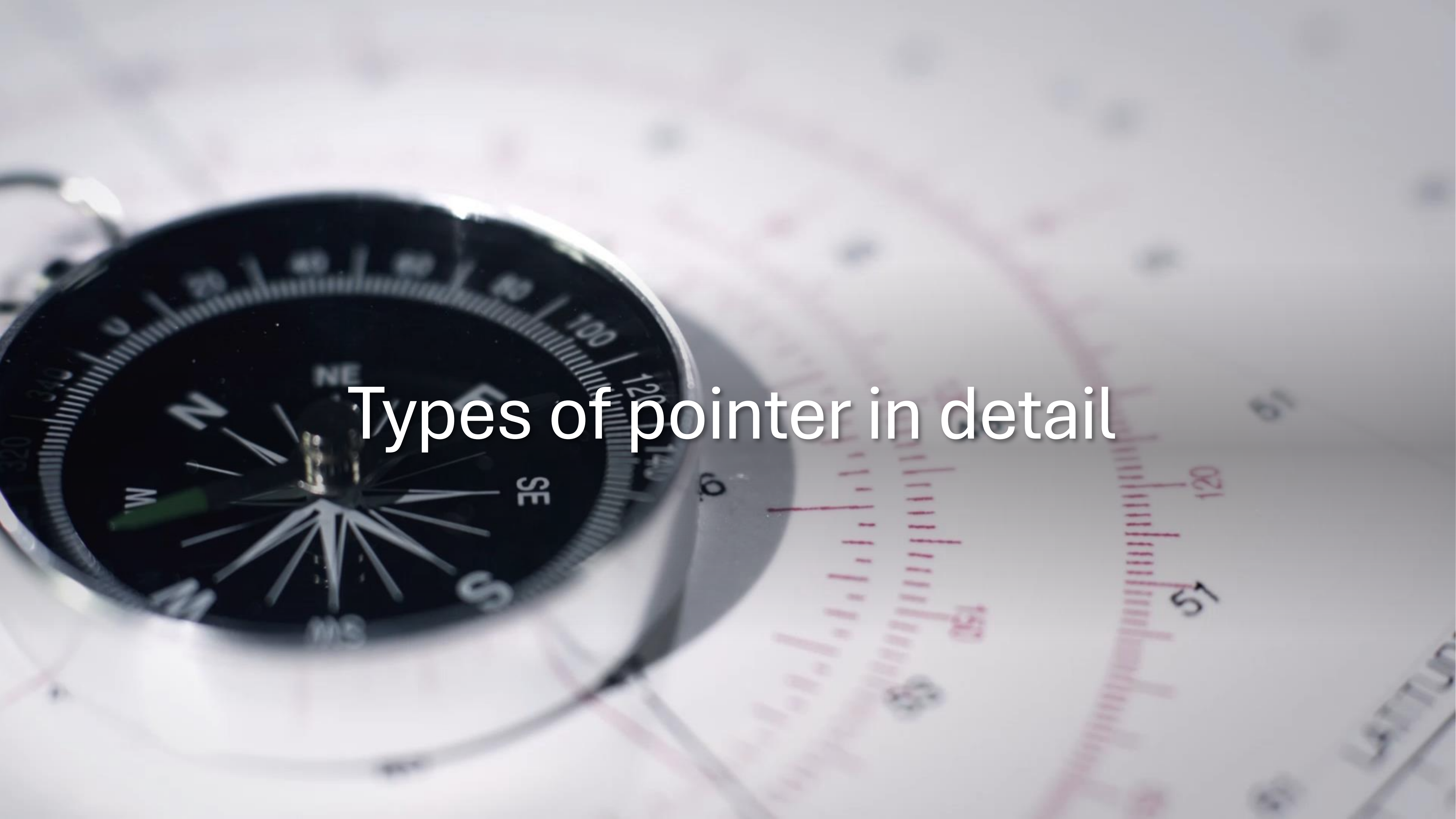
```
int main(){

   int a[]= {10, 20, 30, 40, 50, 60, 70, 80, 90, 100};
   int *x = &a[0]; // zeroth element
   int *y = &a[9]; // last element

   printf("Add of a[0]: %ld add of a[9]: %ld\n", x, y);
   printf("Subtraction of two pointers: %ld", y-x-5);    //When subtracting two pointers, the result is the number of elements between them

 printf("Addition of two pointers: %ld", y-x+5);
}
```

//When subtracting two pointers, the result is the number of elements between them

$9-0-5=4$

$y+x+5$ X $9-0+5=14$

Output:
Add of a[0]: 140729350774896 add of a[9]: 140729350774932
Subtraction of two pointers: 4
Addition of two pointers: 14



Types of pointer in detail

Pointer to an Array or Array Pointer

Difference between pointer to an integer and pointer to an array of integers.

```
#include<stdio.h>
int main()
{
int *p; // Pointer to an integer
int (*ptr)[5]; // Pointer to an array of 5 integers
int arr[5];
p = arr; // Points to 0th element of the arr.
    ptr = &arr; // Points to the whole array arr.
    printf("p = %p, ptr = %p\n", p, ptr);
    p++;
    ptr++;
    printf("p = %p, ptr = %p\n", p, ptr);
    return 0;
}
```

$5 * 4 = 20$

Output:

p = 0x7ffd199ce0b0, ptr = 0x7ffd199ce0b0
p = 0x7ffd199ce0b4, ptr = 0x7ffd199ce0c4

Sizes of pointer of array

```c
#include<stdio.h>
int main()
{
    int arr[] = { 3, 5, 6, 7, 9 };
    int *p = arr;
    int (*ptr)[5] = &arr;
    printf("p = %p, ptr = %p\n", p, ptr);
    printf("*p = %d, *ptr = %p\n", *p, *ptr);
    printf("sizeof(p) = %lu, sizeof(*p) = %lu\n", sizeof(p), sizeof(*p)); //On a 64-bit system: Pointers are generally 8 bytes
    printf("sizeof(ptr) = %lu, sizeof(*ptr) = %lu\n", sizeof(ptr), sizeof(*ptr));
    return 0;
}
```

Output:

p = 0x7fff5dd31d40, ptr = 0x7fff5dd31d40
**p = 3, *ptr = 0x7fff5dd31d40*
*sizeof(p) = 8, sizeof(*p) = 4*
*sizeof(ptr) = 8, sizeof(*ptr) = 20*

Traversing an array by incrementing the pointer

```c
#include <stdio.h>
int main(){
   int a[]= {10, 20, 30, 40, 50, 60, 70, 80, 90, 100};
   int len = sizeof(a)/sizeof(int);
   int *x = a;
   int i = 0;
   for(i = 0; i < len; i++){
      printf("Address of subscript %d = %d Value = 
%d\n", i, x, *x);
      x++;
   }
   return 0;
}
```

Output:
Address of subscript 0 = 836027440 Value = 10
Address of subscript 1 = 836027444 Value = 20
Address of subscript 2 = 836027448 Value = 30
Address of subscript 3 = 836027452 Value = 40
Address of subscript 4 = 836027456 Value = 50
Address of subscript 5 = 836027460 Value = 60
Address of subscript 6 = 836027464 Value = 70
Address of subscript 7 = 836027468 Value = 80
Address of subscript 8 = 836027472 Value = 90
Address of subscript 9 = 836027476 Value = 100

Pointer to access 2-D array elements

```
#include<stdio.h>
int main( )
{
int s[4][2] = {{ 1234, 56 },{ 1212, 33 },{ 1434, 80 },{
1312, 78 }} ;
int i, j ;
for ( i = 0 ; i <= 3 ; i++ )
{
printf ( "\n" ) ;
for ( j = 0 ; j <= 1 ; j++ )
printf ( "%d ", *( *( s + i ) + j ) ) ;
}
}
```

Output:
1234 56
1212 33
1434 80
1312 78

$*(*(s+0)+j)$

$s+0$ $s[0][0]$ $s[0][1]$

Pointer to an array

```
#include<stdio.h>
int main( )
{
int s[4][2] = {{ 1234, 56 },{ 1212, 33 },{ 1434, 80 },{ 1312, 78 }} ;
int ( *p )[2] ; //p is a pointer to an array of two integers.
int i, j, *pint ;
for ( i = 0 ; i <= 3 ; i++ )
{p = &s[i] ;
pint = *p ; // pint is a pointer to the first element of the ith row
printf ( "\n" ) ;
for ( j = 0 ; j <= 1 ; j++ )
printf ( "%d ", *( pint + j ) ) ;
}
}
```

Output:

1234 56

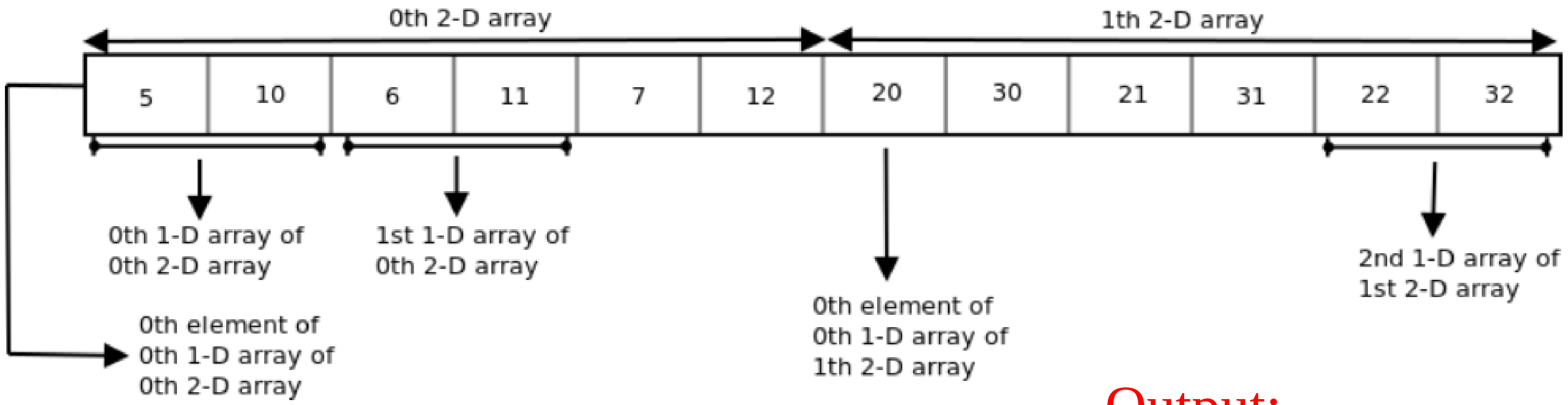
1212 33

1434 80

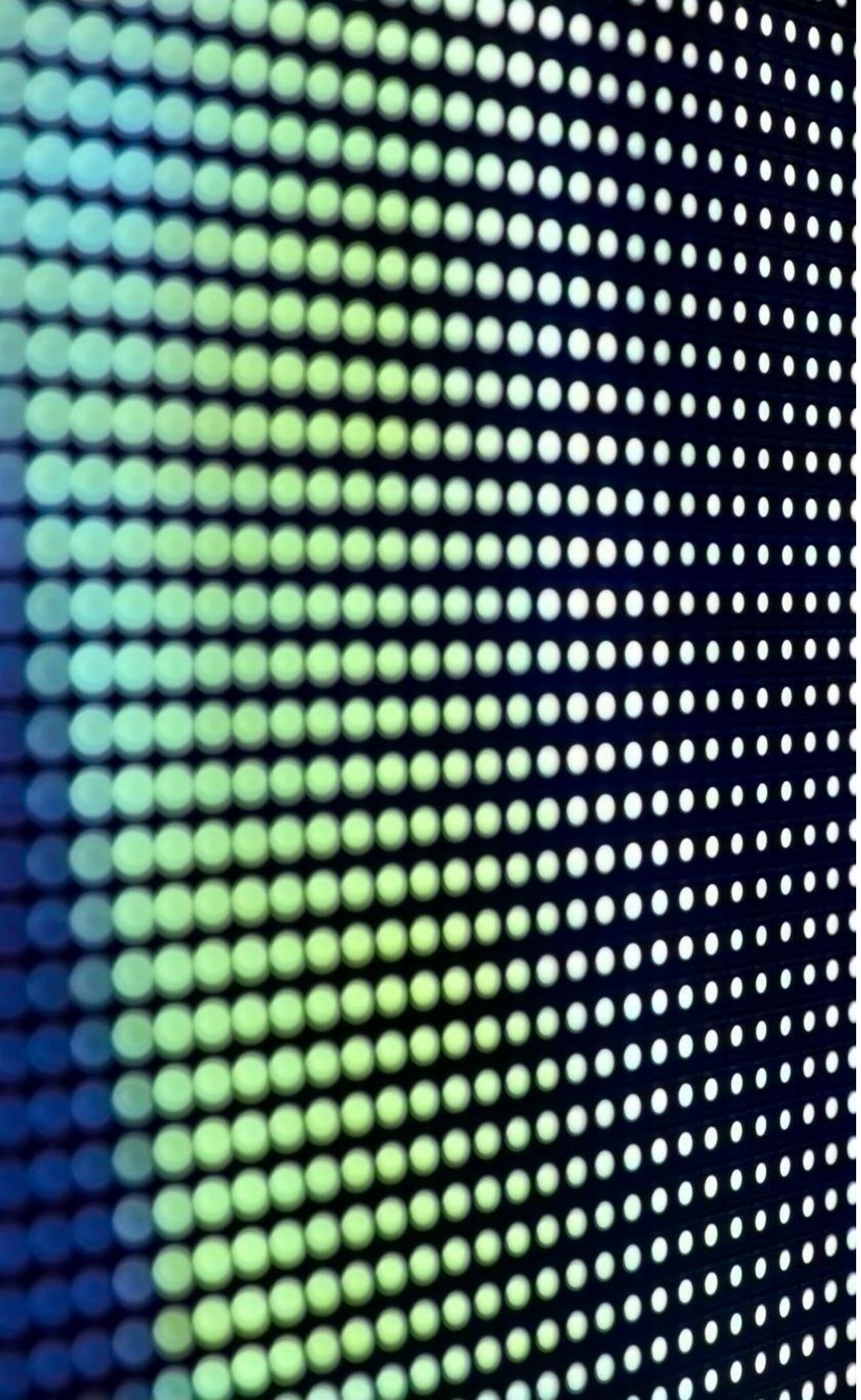
1312 78

Print the elements of 3-D array using pointer notation

```
#include<stdio.h>
int main()
{ int arr[2][3][2] = {{
                {5, 10},
                {6, 11},
                {7, 12},
              },{{20, 30},
                {21, 31},
                {22, 32},
              } };
 int i, j, k;
 for (i = 0; i < 2; i++){
  for (j = 0; j < 3; j++){
    for (k = 0; k < 2; k++)
     printf("%d\t", *(*(*(arr + i) + j) +k));
    printf("\n");
  } } }
```



Output:
5 10
6 11
7 12
20 30
21 31
22 32



Array of Pointers or Pointer array

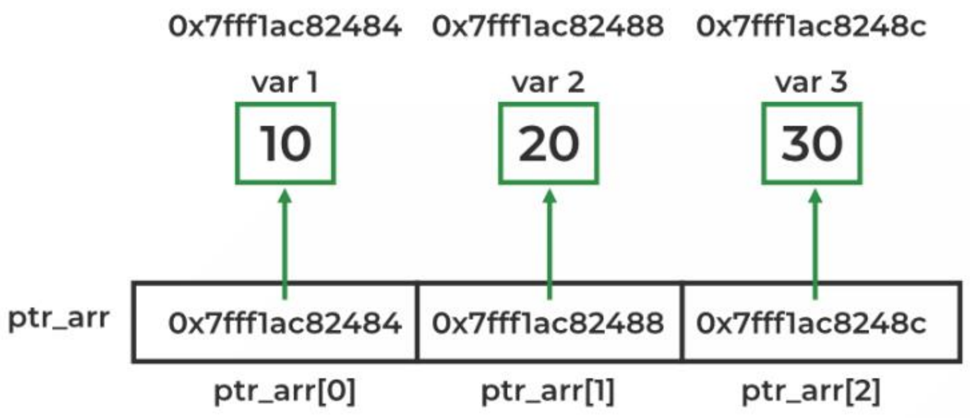
- A pointer array is a homogeneous collection of indexed pointer variables that are references to a memory location.
- Syntax:

  pointer_type *array_name [array_size];

  - **pointer_type:** Type of data the pointer is pointing to.
  - **array_name:** Name of the array of pointers.
  - **array_size:** Size of the array of pointers.

Array pointer: Example

```c
#include <stdio.h>
int main()
{
    int var1 = 10;
    int var2 = 20;
    int var3 = 30;
    int* ptr_arr[3] = { &var1, &var2, &var3 }; // array of pointers to integers
    for (int i = 0; i < 3; i++) {
        printf("Value of var%d: %d\tAddress: %p\n", i + 1, *ptr_arr[i], ptr_arr[i]);
    }
    return 0;
```



Output:

Value of var1: 10 Address: 0x7fff1ac82484
Value of var2: 20 Address: 0x7fff1ac82488
Value of var3: 30 Address: 0x7fff1ac8248c

Function Pointer

Function Pointer

- A variable that stores the address of a function is called a **function pointer** or a **pointer to a function**.
- Function pointers can be useful when you want to call a function dynamically.
- **Syntax:**

  function_return_type(*Pointer_name)(function argument list)

  Example

```
void hello(){
  printf("Hello World");
}
```

  We declare a pointer to this function as follows −

```
void (*ptr)() = &hello;
```

Function Pointer: Example

```
#include <stdio.h>
// Defining a function
void hello() {
printf("Hello World");
 }
int main() {
  void (*ptr)() = &hello; // Declaring a function pointer
  (*ptr)(); // Calling function using the function pointer

  return 0;
}
```

Output:
Hello World

Function Pointer: Example 2

```
#include <stdio.h>
int addition (int a, int b){
   return a + b;
}
int main(){
   int (*ptr)(int, int) = addition; //declaration of function pointer
   int x = 10, y = 20;
   int z = (*ptr)(x, y); // call the function through its pointer
   printf("Addition of x: %d and y: %d = %d", x, y, z);
   return 0;
}
```

Output:
Addition of x: 10 and y: 20 = 30

Function Pointer: Example 3

```
#include<stdio.h>
int areaRectangle(int, int); // function declaration
int main() {
    int length, breadth, area;
    int (*fp)(int, int); // function pointer declaration
    printf("Enter length and breadth of a rectangle\n");
    scanf("%d%d", &length, &breadth);
    fp = areaRectangle; // pointing the pointer to functions memory address
    area = (*fp)(length, breadth); // calling the function using function pointer
    printf("Area of rectangle = %d", area);
    return 0;}
// function definition
int areaRectangle(int l, int b)
{int area_of_rectangle = l * b;
    return area_of_rectangle;}
```

Output:
Enter length and breadth of a rectangle
20 4
Area of rectangle = 80

Pointer to Function with Pointer Arguments

```
#include <stdio.h>
void swap(int *a, int *b){
   int c;
   c = *a;
   *a = *b;
   *b = c;
}
int main(){
   void (*ptr)(int *, int *) = swap; // Declaration of function pointer
   int x = 10, y = 20;
   printf("Values of x: %d and y: %d before swap\n", x, y);
   (*ptr)(&x, &y); // Call the function pointer
   printf("Values of x: %d and y: %d after swap", x, y);
   return 0;
}
```

Output:
Values of x: 10 and y: 20 before swap
Values of x: 20 and y: 10 after swap

Array of Function Pointers

```c
#include <stdio.h>
float add(int a, int b){return a + b;}
float subtract(int a, int b){return a - b;}
float multiply(int a, int b){return a * b;}
float divide(int a, int b){return a / b;}

int main(){
   float (*ptr[])(int, int) = {add, subtract, multiply, divide};
   int a = 15, b = 10;
   // 1 for addition, 2 for subtraction
   // 3 for multiplication, 4 for division
   int op = 3;
   if (op > 5) return 0;
   printf("Result: %.2f", (*ptr[op-1])(a, b));
   return 0;}
```

Output:
Result: 150.00

Dynamic function calling without using if-else or switch-case statements

(*ptr[op])(a,b)

Void Pointer

- A void pointer is a pointer that has no associated data type with it.
- A void pointer can hold an address of any type and can be typecasted to any type.
- Syntax:
  - void * pointer_name;

```c
#include <stdio.h>
int main()
{
    int a = 10;
    char b = 'x';
    // void pointer holds address of int 'a'
    void* p = &a;
printf("%d\n", *p);
    // void pointer holds address of char 'b'
    p = &b;
printf("%c\n", *p);
}
```

Output:
error: invalid use of void expression

```c
#include <stdio.h>
int main()
{
    int a = 10;
    char b = 'x';
    // void pointer holds address of int 'a'
    void* p = &a;
printf("%d\n", *(int*)p);   → 10
    // void pointer holds address of char 'b'
    p = &b;   →
printf("%c\n", *(char*)p);   x

}
```

Output:
10
x

Reason: void pointers cannot be dereferenced.

Null Pointer

Pointer that does not point to any location but NULL.

Syntax of Null Pointer Declaration in C

- type pointer_name = NULL;
- type pointer_name = 0;

Uses of NULL Pointer

1. To initialize a pointer variable when that pointer variable hasn’t been assigned any valid memory address yet.
2. To check for a null pointer before accessing any pointer variable. That helps in error handling in pointer-related code, e.g., dereference a pointer variable only if it’s not NULL.
3. To pass a null pointer to a function argument when we don’t want to pass any valid memory address.
4. A NULL pointer is used in data structures like trees, linked lists, etc. to indicate the end.

Example of pointers: Null pointer

```
#include <stdio.h>
int main(){
   int *ptr = NULL;
   printf("The value of ptr is : %d\n", ptr);
   return 0;
}
```

Output:

The value of ptr is : 0

Pointer comparison with NULL value

```
int main()
{
    int* ptr = NULL;
    if (ptr == NULL) {
        printf("The pointer is NULL");
    }
    else {
        printf("The pointer is not NULL");
    }
    return 0;
}
```

Output:
The pointer is NULL

Pass NULL to a function

```
#include <stdio.h>
void passnull(int* value)
{   if (value == NULL) {
        printf("NULL Pointer Passed");
        return;
    }
    printf("Non-Null Pointer Passed");
}
int main()
{
    passnull(NULL);
    return 0;
}
```

Output:
NULL Pointer Passed

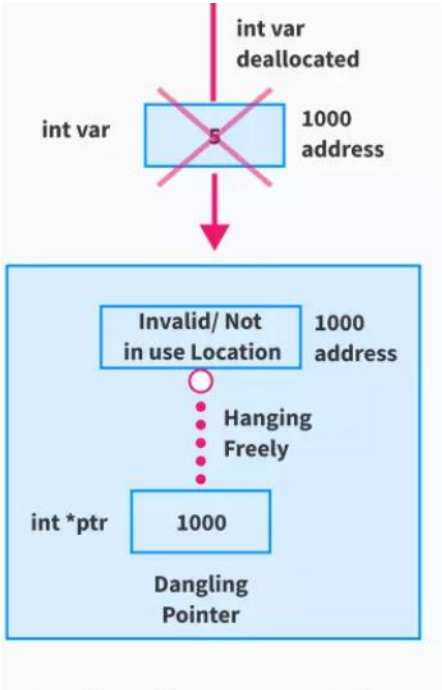
NULL Pointer	**Void Pointer**
A NULL pointer does not point to anything. It is a special reserved value for pointers.	A void pointer points to the memory location that may contain typeless data.
Any pointer type can be assigned NULL.	It can only be of type void.
All the NULL pointers are equal.	Void pointers can be different.
NULL Pointer is a value.	A void pointer is a type.
Example: int *ptr = NULL;	**Example:** void *ptr;

Dangling Pointers

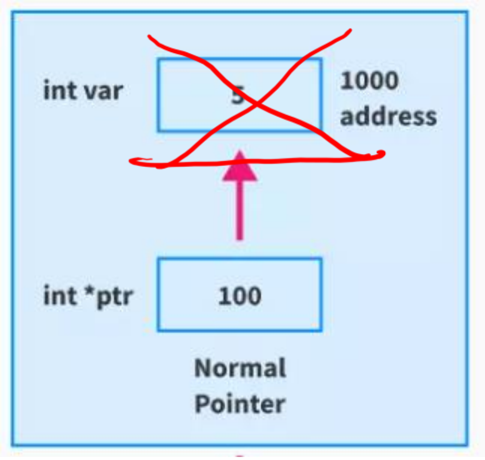
Dangling Pointers: Pointers that refer to a **memory location that has been freed.**

Three different ways are as follows**:**

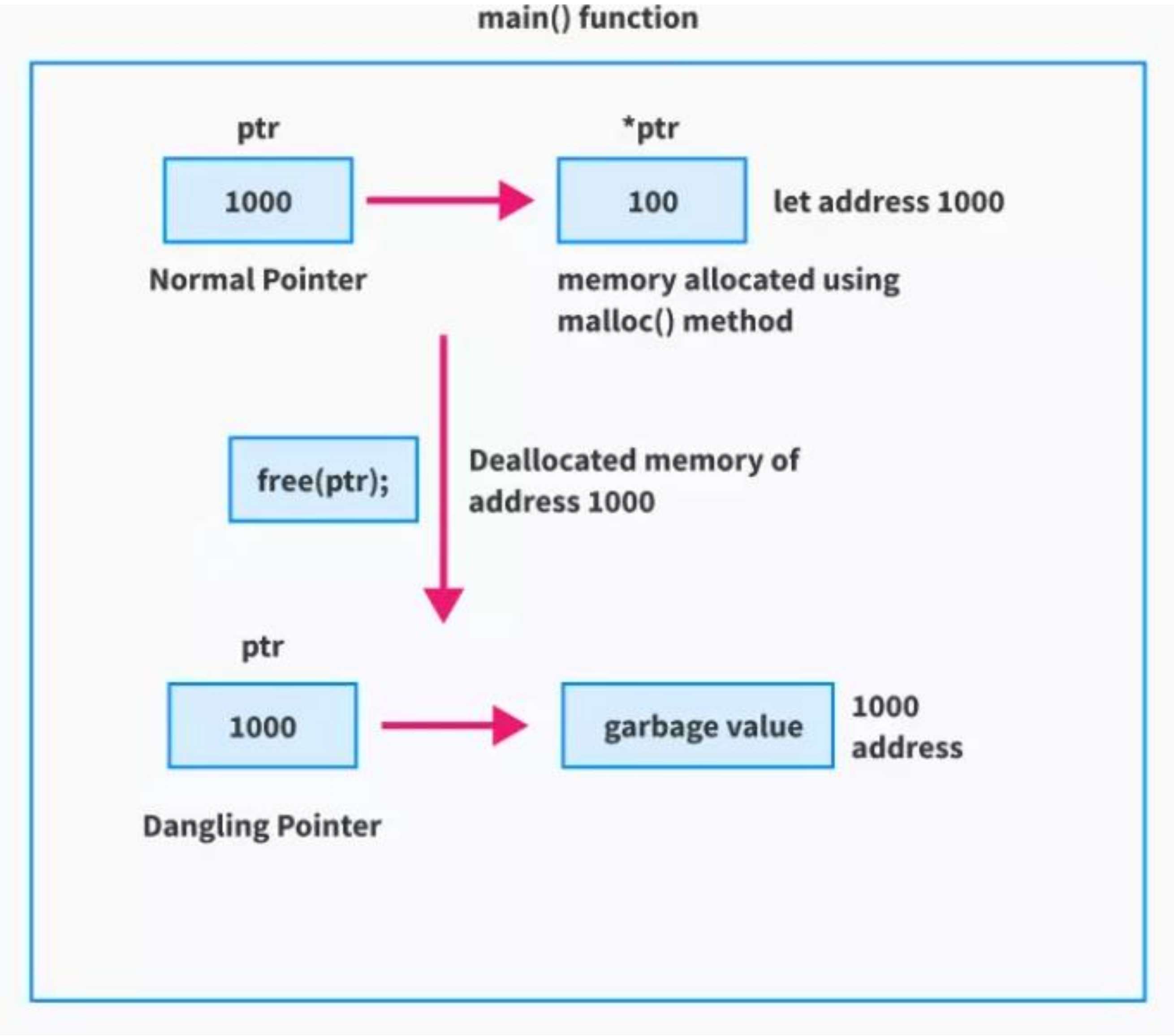
1. Deallocation of memory
2. Function Call
3. Variable goes out of scope



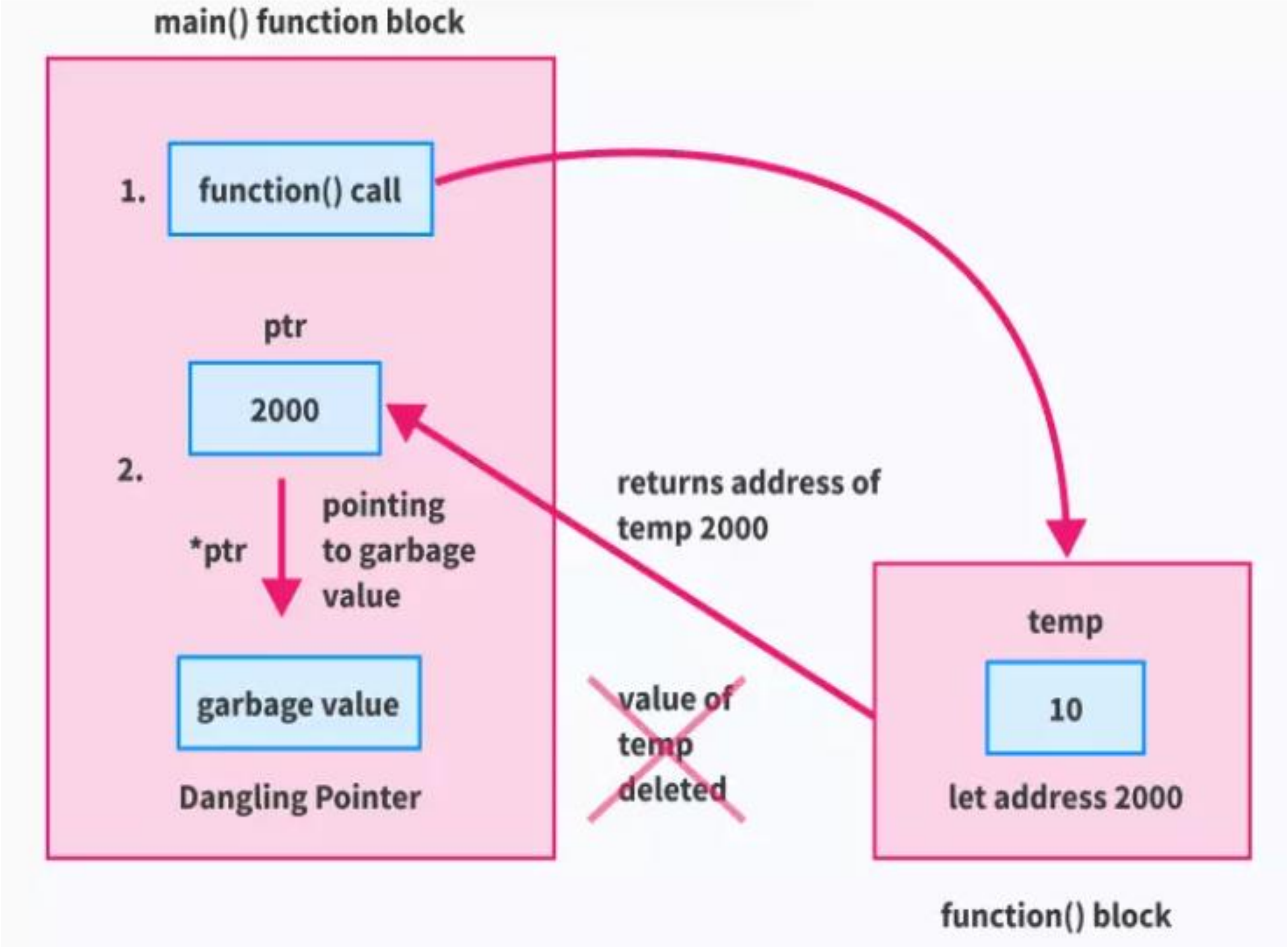
Dangling Pointers Representation



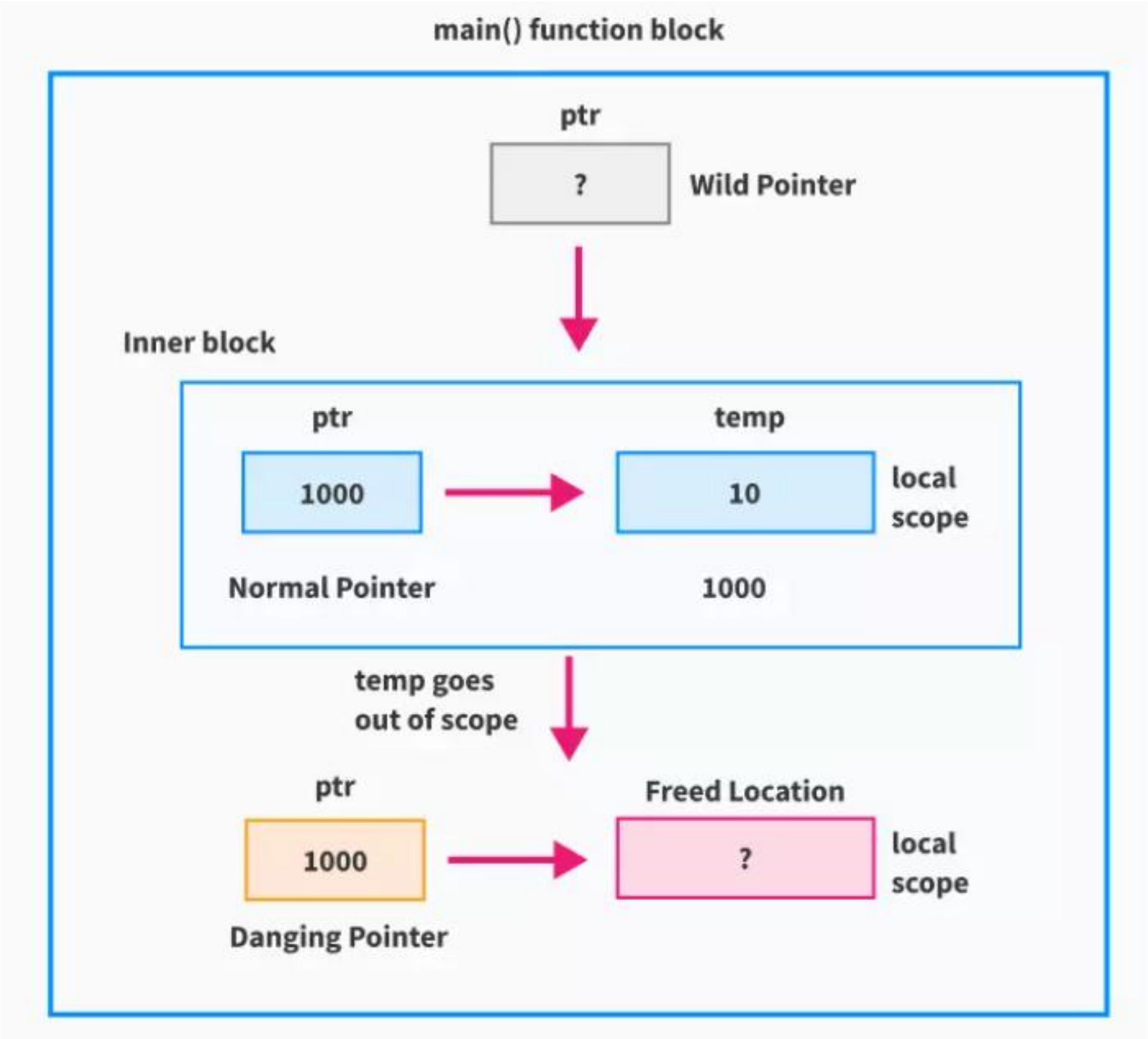
Dangling Pointers: Deallocation of memory



Solution: ptr =NULL



Dangling Pointers: Function Call



Dangling Pointers: Variable goes out of scope

Uninitialized Pointers: wild pointer

- **Uninitialized Pointers:** Using pointers without assigning a valid address.

```
#include <stdio.h>
int main() {
    int *p;  // Uninitialized pointer
    *p = 10;  // Attempt to dereference and assign value
    printf("Value of *p: %d\n", *p);  // Undefined behavior
    return 0;
}
```

Output:
Segmentation fault

Uninitialized Pointers: Correct way to Initialized

```
#include <stdio.h>

int main() {
    int x = 10;
    int *p = &x;  // Initialized pointer, pointing to the address of x

    *p = 20;  // Assign a value through the pointer

    printf("Value of *p: %d\n", *p);  // Outputs 20
    printf("Value of x: %d\n", x);    // Outputs 20 (because *p modifies x)
    return 0;
}
```

Output:
Value of *p: 20
Value of x: 20

Common pointer Mistakes

Uninitialized Pointers: Using pointers without assigning a valid address.

Dangling Pointers: Pointers that refer to a memory location that has been freed.

Pointer Arithmetic Errors: Incorrect pointer increment/decrement.

Memory Allocation: Static vs Dynamic

Static vs Dynamic Memory Allocation

Static memory allocation	Dynamic memory allocation
Memory is allocated at compile time.	Memory is allocated at run time.
Memory can't be increased while executing program.	Memory can be increased while executing program.
Used in array.	Used in linked list.

Dynamic Memory Allocation

- The concept of dynamic memory allocation enables us to allocate memory at runtime.
- Dynamic memory allocation in c language is possible by 4 functions of stdlib.h header file.
  - malloc()
  - calloc()
  - realloc()
  - free()

malloc()

- The name "malloc" stands for memory allocation.
- The malloc() function reserves a block of memory of the specified number of bytes.
- It returns a pointer of void which can be casted into pointers of any form.

- Syntax:
  - ptr = (castType*) malloc(size);
- Example:
  - ptr = (float*) malloc(100 * sizeof(float)); // allocates 400 bytes of memory
- The expression results in a NULL pointer if the memory cannot be allocated.

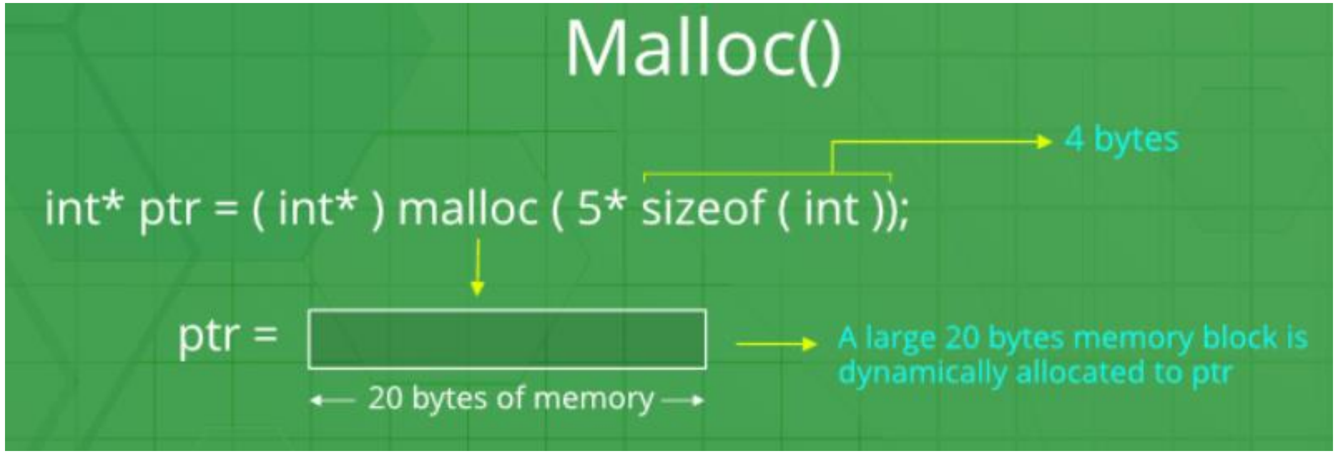
calloc()

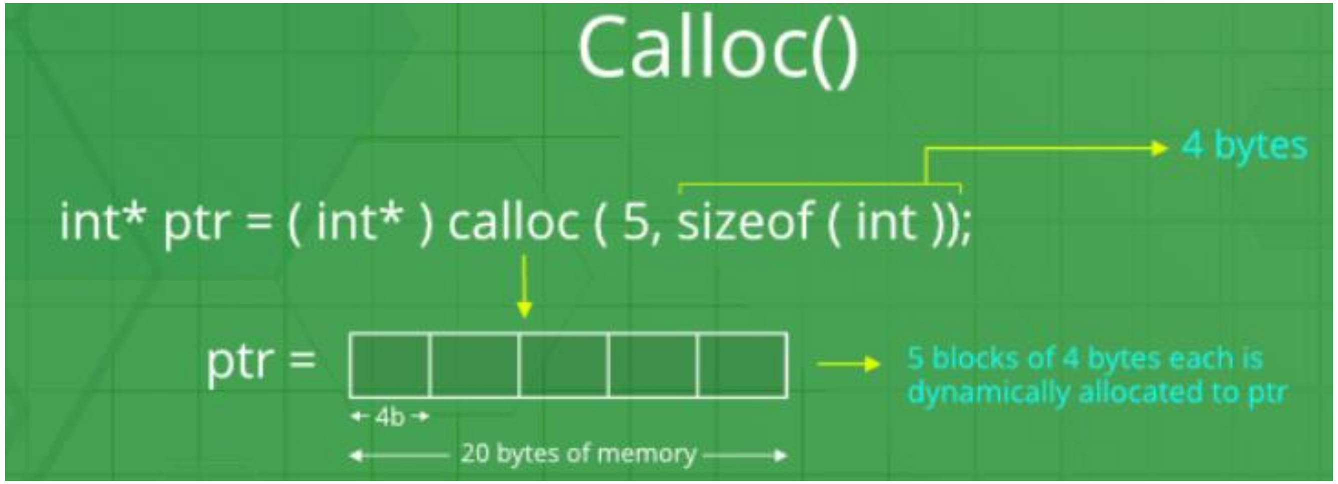
- “calloc” or “contiguous allocation” method is used to dynamically allocate the specified number of blocks of memory of the specified type. It is similar to malloc except:
  - It initializes each block with a default value ‘0’.
  - It has two parameters or arguments as compared to malloc()
- Syntax:
  - ptr = (cast-type*)calloc(n, element-size);

  Here, n is the no. of elements and element-size is the size of each element.
- Example:
  - ptr = (float*) calloc(25, sizeof(float));

  It allocates contiguous space in memory for 25 elements each with the size of the float.

Malloc vs. Calloc

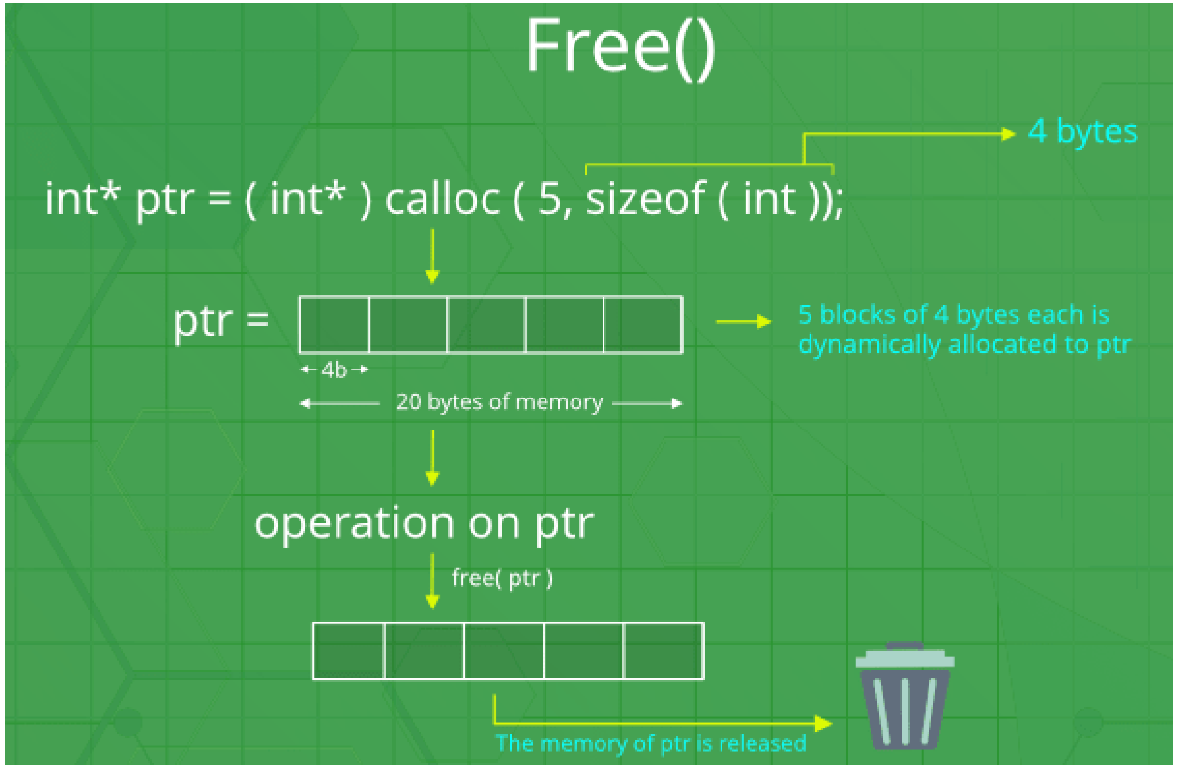




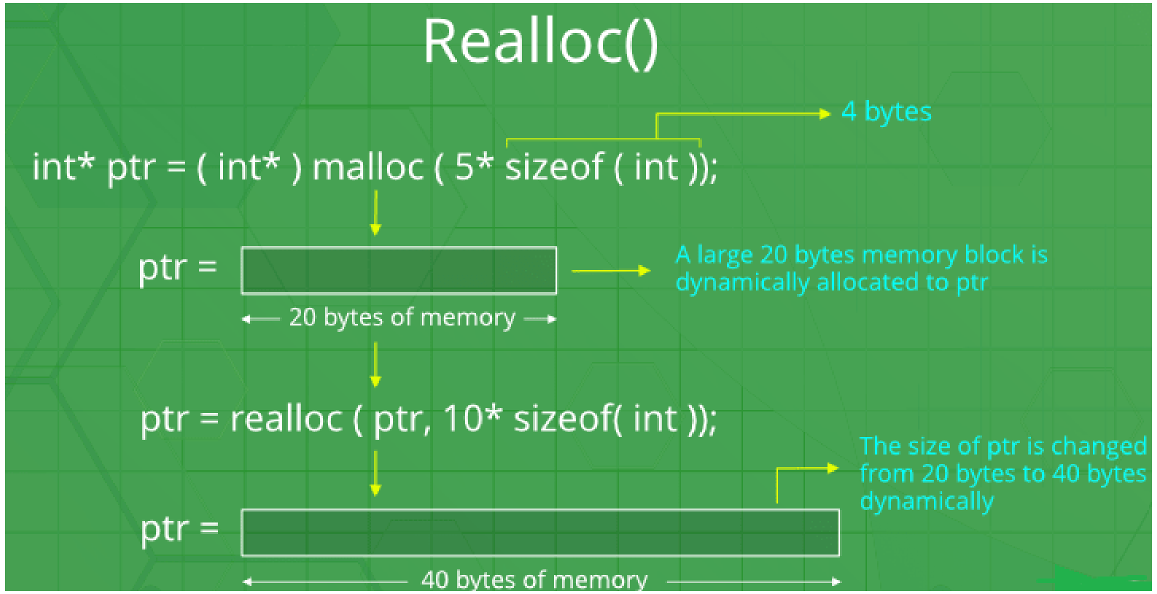
S.No.	malloc()	calloc()
1.	A function that creates one block of memory of a fixed size.	A function that assigns a specified number of blocks of memory to a single variable.
2.	It only takes one argument	Takes two arguments.
3.	It is faster than calloc.	slower than malloc()
4.	It is used to indicate memory allocation	Used to indicate contiguous memory allocation
5.	Syntax : void* malloc(size_t size);	Syntax : void* calloc(size_t num, size_t size);
6.	It does not initialize the memory to zero	Initializes the memory to zero
7.	Does not add any extra memory overhead	Adds some extra memory overhead

free()

- **“free”** method is used to dynamically **de-allocate** the memory.
- The memory allocated using functions **malloc()** and **calloc()** is not de-allocated on their own.
- **free()** method is used to deallocate the memory and reduces wastage of memory by freeing it.
- Syntax: **free(ptr);**



realloc()



- **"realloc"** or **"re-allocation"** method is used to dynamically change the memory allocation of a previously allocated memory.
- If the memory previously allocated with the help of malloc or calloc is insufficient, realloc can be used to **dynamically re-allocate memory**.
- Re-allocation of memory maintains the already present value and new blocks will be initialized with the default garbage value.
- Syntax:
  - ptr = realloc(ptr, newSize);

Example: malloc

```
#include <stdio.h>
#include <stdlib.h>
int main()
{int* ptr,n, i;
    printf("Enter number of elements:");
    scanf("%d",&n);
    printf("Entered number of elements: %d\n", n);
    ptr = (int*)malloc(n * sizeof(int));
if (ptr == NULL) {
        printf("Memory not allocated.\n");
        exit(0);
    }
    else {
        printf("Memory successfully allocated using
malloc.\n");
        printf("Now enter the element of the array");
        for (i = 0; i < n; ++i) {
            int x;
            scanf("%d",&x);
            ptr[i] = x;
        }       // Print the elements of the array
        printf("The elements of the array are: ");
        for (i = 0; i < n; ++i) {
            printf("%d, ", ptr[i]);
        }
    return 0;}
```

Output:
Enter number of elements:5
Entered number of elements: 5
Memory successfully allocated using malloc.
Now enter the element of the array3 4 6 7 1
The elements of the array are: 3 4 6 7 1

Example: calloc

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
 // This pointer will hold the base address of the block created
    int* ptr,n, i;
    printf("Enter number of elements:");
    scanf("%d",&n);
    printf("Entered number of elements: %d\n", n);
    ptr = (int*)calloc(n, sizeof(int));
printf("Now enter the element of the array");
        for (i = 0; i < n; ++i) {
            int x;
            scanf("%d",&x);
            ptr[i] = x;
        }       // Print the elements of the array
        printf("The elements of the array are: ");
        for (i = 0; i < n; ++i) {
            printf("%d, ", ptr[i]);
        }
    return 0;
}
```

Output:

Enter number of elements:4
Entered number of elements: 4
Now enter the element of the array3 2 4 9
The elements of the array are: 3 2 4 9

Example: free

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
int *ptr, *ptr1,n, i;
n = 5;
printf("Enter number of elements: %d\n", n);
// Dynamically allocate memory using malloc()
ptr = (int*)malloc(n * sizeof(int));
// Dynamically allocate memory using calloc()
ptr1 = (int*)calloc(n, sizeof(int));
 if (ptr == NULL || ptr1 == NULL) {
        printf("Memory not allocated.\n");
        exit(0);}
else {
printf("Memory successfully allocated using malloc.\n");
free(ptr);
printf("Malloc Memory successfully freed.\n");
printf("\nMemory successfully allocated using calloc.\n");
free(ptr1);
printf("Calloc Memory successfully freed.\n");}
    return 0;
}
```

Output:

Enter number of elements: 5
Memory successfully allocated using malloc.
Malloc Memory successfully freed.
Memory successfully allocated using calloc.
Calloc Memory successfully freed.

Example: realloc

```
#include <stdio.h>
#include <stdlib.h>
int main()
{int* ptr,n, i;
 n = 5;
 printf("Enter number of elements: %d\n", n);
 ptr = (int*)calloc(n, sizeof(int));
for (i = 0; i < n; ++i) {ptr[i] = i + 1;}
for (i = 0; i < n; ++i) {printf("%d ", ptr[i]);}
n = 10;
printf("\nEnter the new size of the array: %d\n", n);
ptr = (int*)realloc(ptr, n * sizeof(int));
printf("Memory re-allocated using realloc.\n");
for (i = 5; i < n; ++i) {
        ptr[i] = i + 1;
    }
printf("The elements of the array are: ");
    for (i = 0; i < n; ++i) {
        printf("%d ", ptr[i]);
    }
    free(ptr);
  return 0;
}
```

Output:

Enter number of elements: 5
1 2 3 4 5
Enter the new size of array: 10
Memory re-allocated using realloc.
The elements are: 1 2 3 4 5 6 7 8 9 10

Program to calculate the sum of n numbers entered by the user

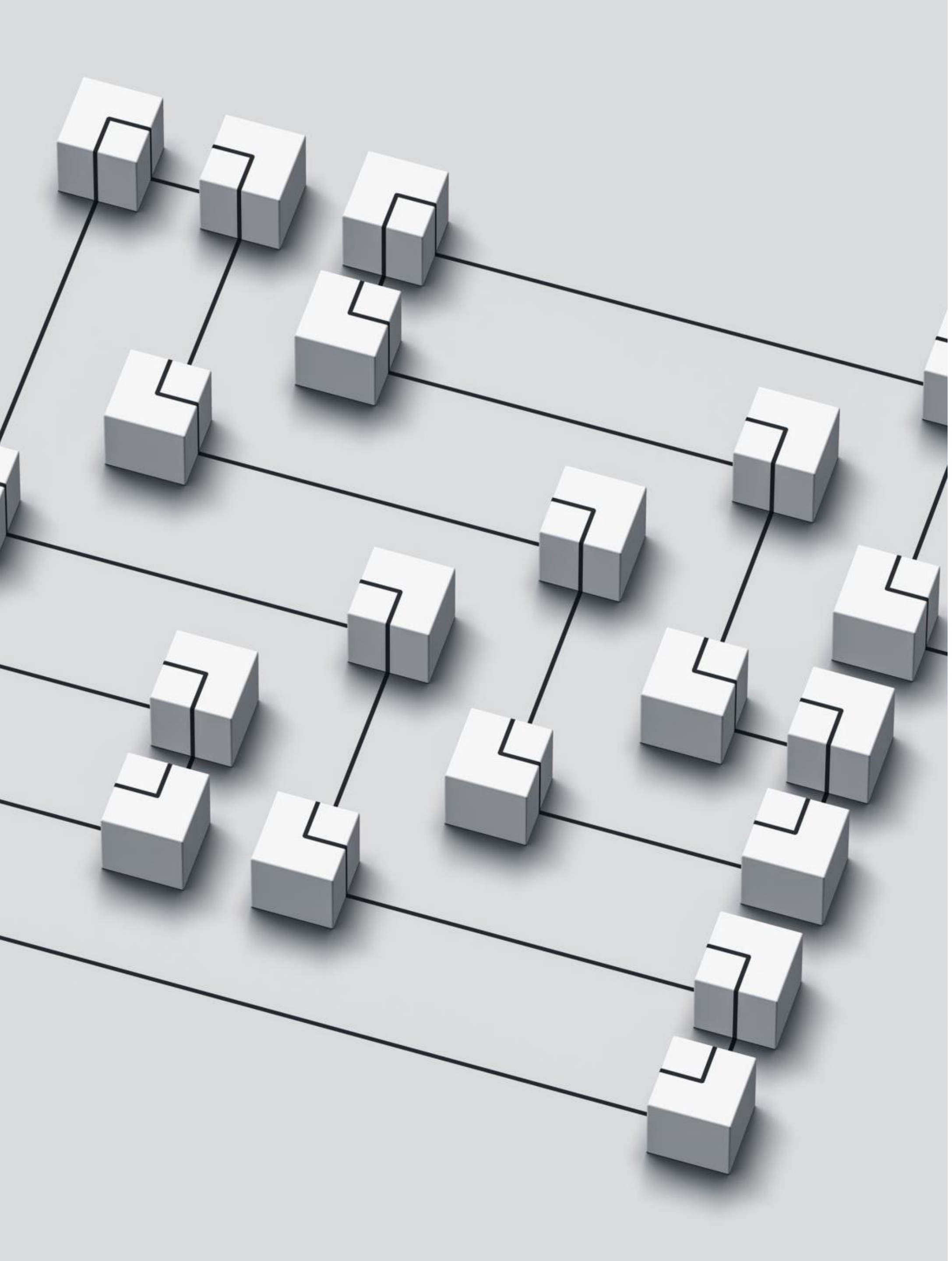
```
#include <stdio.h>
#include <stdlib.h>
int main() {
  int n, i, *ptr, sum = 0;
  printf("Enter number of elements: ");
  scanf("%d", &n);
  ptr = (int*) malloc(n * sizeof(int));
  if(ptr == NULL) {
    printf("Error! memory not allocated.");
    exit(0);
  }
  printf("Enter elements: ");
  for(i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

Output:

Enter number of elements: 5

Enter elements: 9 8 7 3 4

Sum = 31



Upcoming Slides

- **Searching & Sorting**