Parul® University

Programming for Problem Solving (PPS) Chapter-6

Pointers, Dynamic memory allocation and File Management in C

Topics Covered

Topics Covered

- Pointers
  - ➢ Basics of pointers
  - ➢ Pointer to pointer
  - ➢ Pointer and array
  - ➢ Pointer to array
  - ➢ Array of pointers
  - ➢ Functions Returning Pointers
- Dynamic Memory Allocation
  - ➢ Introduction
  - ➢ malloc(), calloc(), free(), realloc().
- File Management in C
  - ➢ Introduction and standard file handling functions

Why Pointers?

- They allow you to refer to large data structures in a compact way
- They facilitate sharing between different parts of programs
- They make it possible to get new memory dynamically as your program is running
- They make it easy to represent relationships among data items.

Why Use of Pointers?

1. To access and modify variables indirectly.
2. To pass large data (like arrays, structs) efficiently to functions.
3. To dynamically allocate memory using malloc(), calloc(), etc.
4. To build complex data structures (linked lists, trees, etc.).

Pointers

A *pointer* is a reference to another variable (memory location) in a program

- Used to change variables inside a function (reference parameters)
- Used to remember a particular member of a group (such as an array)
- Used in dynamic (on-the-fly) memory allocation (especially of arrays)
- Used in building complex data structures (linked lists, stacks, queues, trees, etc.)

Pointer Operator Precedence

Operators	Associativity	Type
`()` `[]`	left to right	highest
`+` `-` `++` `--` `!` `*` `&` `(`*type*`)`	right to left	unary
`*` `/`	left to right	multiplicative
`+` `-`	left to right	additive
`<` `<=` `>` `>=`	left to right	relational
`==` `!=`	left to right	Equality
`&&`	left to right	logical and
`||`	left to right	logical OR
`?:`	right to left	conditional
`=` `+=` `-=` `*=` `/=` `%=`	right to left	assignment
`,`	left to right	comma

Pointer Expressions and Pointer Arithmetic

- Arithmetic operations can be performed on pointers
  - ➢Increment/decrement pointer (++ or −−)
  - ➢Add an integer to a pointer(+ or += , − or −=)
  - ➢Pointers may be subtracted from each other
  - ➢Operations meaningless unless performed on an array

Variable Declaration and Memory Mapping

Consider the declaration,

Int i=3;

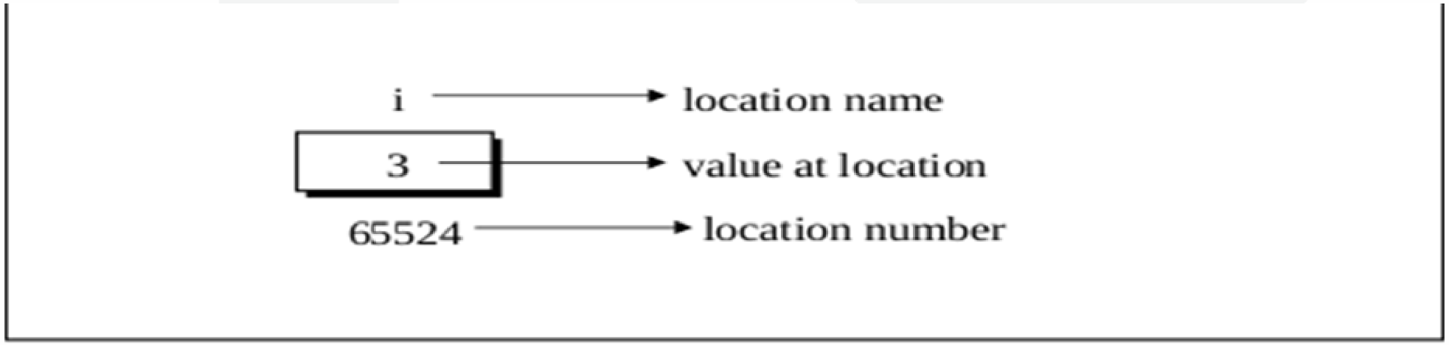
This declaration tells the c compiler to:

(a) Reserve space in memory to hold the integer value.

(b) Associate the name i with this memory location.

(c) Store the value 3 at this location.

We may represent it's location in memory by the following



Parul® University

DIGITAL LEARNING CONTENT

Pointer Variable Definitions and Initialization

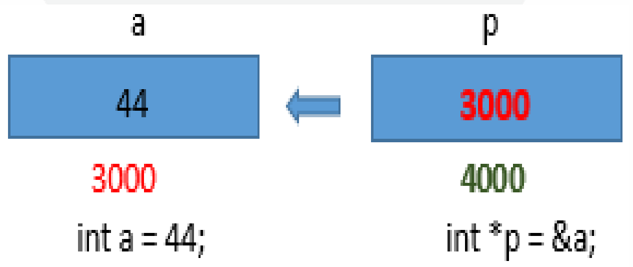
- * used with pointer variables
- e.g. int *p;
- Defines a pointer to an int (pointer of type int *)
- Multiple pointers require using a * before each variable
- int *p1, *p2;
- Can define pointers to any data type
- Initialize pointers to 0, `NULL`, or an address
- 0 or `NULL` – points to nothing
- 0 is the only integer value that can be assigned directly to a pointer variable.
- Initializing a pointer to 0 is equivalent to initializing a pointer to NULL, but NULL is preferred
- NULL is a symbolic constant defined in the <stddef.h> header and several other headers, e.g. <stdio.h>

Pointer Variable Definitions and Initialization

- A normal variable stores a value directly → Direct Reference
- A pointer variable stores the address of another variable → Indirect Reference
- In other words, instead of storing a value directly, a pointer stores memory address of the value.

```
int a = 44;     // normal variable
int *p;         // pointer variable (stores address of an int)
p = &a;         // assign address of a, to pointer p
```

- a stores the value 40.
- &a gives the address of a.
- p stores that address.
- *p gives the value stored at that address, i.e., the value of a.



Parul® University

DIGITAL LEARNING CONTENT

Example: Pointer Program

```
#include <stdio.h>
int main() {
    int a = 10;       // normal variable
    int *p;           // pointer variable

    p = &a;           // store address of a in
     pointer p

    printf("Value of a        = %d\n", a);
    printf("Address of a      = %p\n", &a);
    printf("Address stored in p = %p\n",
     p);
    printf("Value pointed by p  = %d\n",
     *p);
```

```
// change value of a using
 pointer
*p = 25;

printf("\nAfter changing
 value using pointer:\n");
printf("Value of a        =
 %d\n", a);
printf("Value pointed by p  =
 %d\n", *p);

return 0;
}
```

Output:

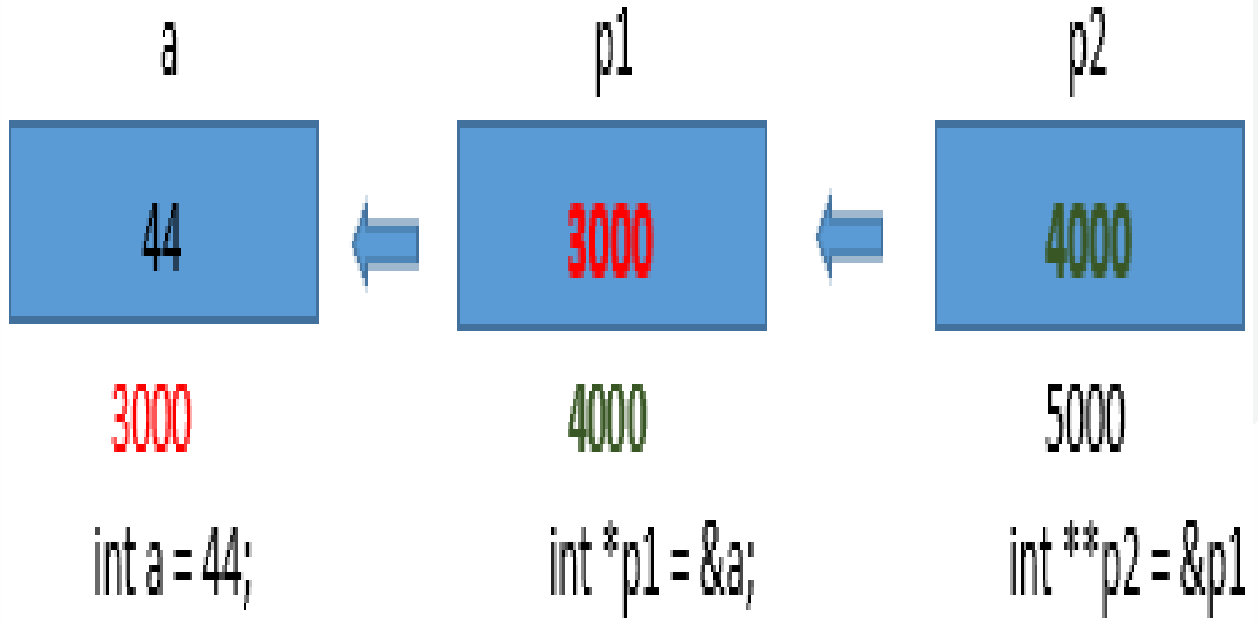
```
Value of a        = 10
Address of a      = 0x7ffe6ac97dd4
Address stored in p = 0x7ffe6ac97dd4
Value pointed by p  = 10

After changing value using pointer:
Value of a        = 25
Value pointed by p  = 25
```

Pointer to Pointer (Double Pointer)

* Means Single Pointer

** Means Double Pointer

- a, p1 and p2 are Variable Names
- 3000, 4000 and 5000 are Memory Addresses



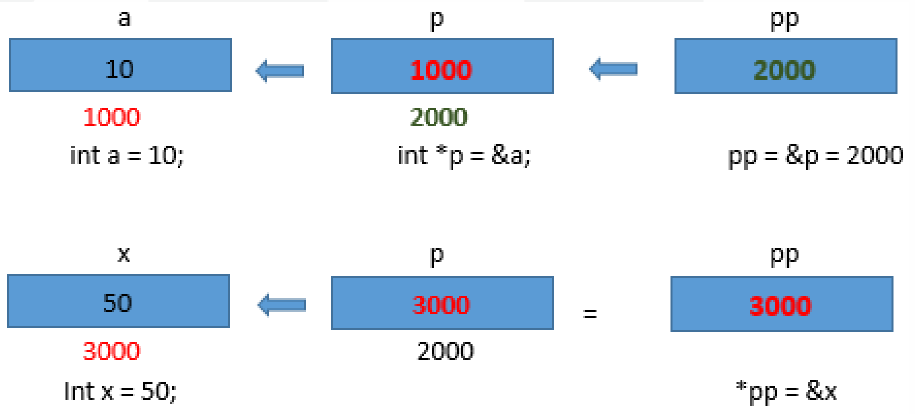
Pointer to Pointer (Double Pointer)

Example:

```
#include <stdio.h>
void change(int **pp) {
    static int x = 50;
    *pp = &x;   // changes original pointer
}

int main() {
    int a = 10;
    int *p = &a;

    change(&p);
    printf("%d", *p);  // prints 50
}
```

Explanation:

1. p is a pointer and pp is a pointer to pointer
2. pp is storing address of p (or points to p). *pp means value at 2000 which is p = 1000. Meaning *pp is actually p. By doing *pp = &x, makes p = 3000.
3. Address: 2000 Value changed from 1000 to 3000
4. Previously p → a, now p → x



Output:

50

CALL BY VALUE & CALL BY REFERENCE

- Call by value
  - If data is passed by value, the data is copied from the variable used in for example main() to a variable used by the function. So if the data passed (that is stored in the function variable) is modified inside the function, the value is only changed in the variable used inside the function.
- Call by reference
  - If data is passed by reference, a pointer to the data is copied instead of the actual variable as is done in a call by value. Because a pointer is copied, if the value at that pointers address is changed in the function, the value is also changed in main().

CALL BY VALUE & CALL BY REFERENCE

- **Call by value**

1. Copy of variable is sent to function
2. Change in variable inside function will not affect original value
3. Scope is limited to function only

- **Call by reference**

1. Send actual address of variable storage
2. Change in variable anywhere affect original value
3. Scope is global

CALL BY VALUE & CALL BY REFERENCE

- **Call by value**

```
#include<stdio.h>
void swapv(int x, int y) {
    int t;
    t = x;
    x = y;
    y = t;

printf("\nx = %d, y = %d\n", x, y);
}

int main()
{
    int a = 10, b = 20;
    swapv(a, b);
    printf("\na = %d, b = %d\n", a, b);
    return 0;
}
```

Output

```
x = 20, y = 10
a = 10, b = 20
```

- **Call by reference**

```
#include<stdio.h>
void swapr(int *x, int *y) {
    int t;
    t = *x;
    *x = *y;
    *y = t;

printf("\nx = %d, y = %d\n", *x, *y);
}

int main()
{
    int a = 10, b = 20;
    swapr(&a, &b);
    printf("\na = %d, b = %d\n", a, b);
    return 0;
}
```

Output

```
x = 20, y = 10
a = 20, b = 10
```

Pointer and Array

The **name of an array** acts like a **pointer** to its **first element**.

```
int arr[5] = {10, 20, 30, 40, 50};
int *p = arr;   // same as p = &arr[0];
```

- p is a pointer to int — it points to the first element of the array.
- p + 1 → moves to the next element in an array (4 bytes ahead for int)

Here

arr → address of first element (&arr[0])
*arr → value of first element (arr[0])
*(arr + 1) → value of second element (arr[1])

Dereferencing
- *p → 10
- *(p + 1) → 20

Pointer and Array

```
#include <stdio.h>

int main() {
    int arr[5] = {10, 20, 30, 40, 50};
    int *p;      // pointer to int
    int i;

    p = arr;     // same as p = &arr[0];

    printf("Array elements using pointer:\n");
    for(i = 0; i < 5; i++) {
        printf("arr[%d] = %d\t", i, *(p + i));
        printf("Address = %p\n", (p + i));
    }

    return 0;
}
```

p = arr; assigns the base address of the array to pointer p.
*(p + i) gives the value of the i-th element.
(p + i) gives the address of the i-th element.

So both these statements are equivalent:
arr[i] == *(p + i)
&p[i] == p + i

```
Array elements using pointer:
arr[0] = 10    Address = 0x7ffd12345670
arr[1] = 20    Address = 0x7ffd12345674
arr[2] = 30    Address = 0x7ffd12345678
arr[3] = 40    Address = 0x7ffd1234567c
arr[4] = 50    Address = 0x7ffd12345680
```

Pointer to an Array

- A **pointer to an array** is a **pointer variable that stores the address of an entire array**, not just the first element.

In simple terms:

- A normal pointer (like int *p) points to a single element.
- A pointer to an array (like int (*ptr)[5]) points to a whole array of fixed size.

e.g.

int (*ptr)[5];

Explanation:

ptr → is a pointer

(*ptr) → points to an array

[5] → means it points to an array of 5 integers

So ptr is a pointer to an array of 5 integers.

Pointer to an Array: Example

```
#include <stdio.h>

int main() {
    int arr[5] = {10, 20, 30, 40, 50};

    int (*ptr)[5];   // pointer to an array of 5 integers
    ptr = &arr;      // store address of the entire array

    printf("Accessing array elements using pointer to
array:\n");
    for (int i = 0; i < 5; i++) {
        printf("(*ptr)[%d] = %d\n", i, (*ptr)[i]);
    }

    return 0;
}
```

Output

```
Accessing array elements using pointer to array:
(*ptr)[0] = 10
(*ptr)[1] = 20
(*ptr)[2] = 30
(*ptr)[3] = 40
(*ptr)[4] = 50
```

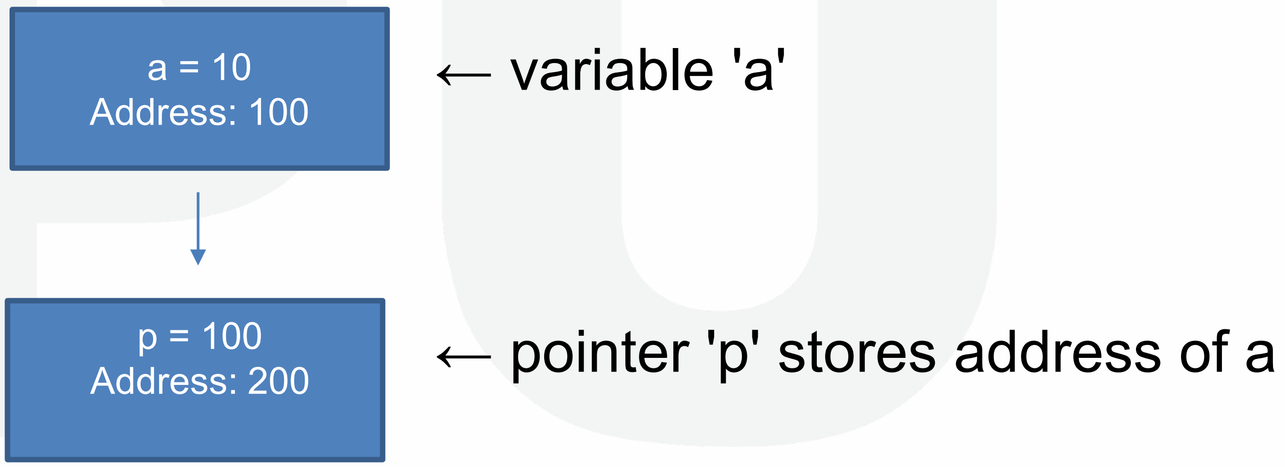
Explanation:

- arr is a normal array.
- &arr gives the address of the whole array.
- ptr is a pointer that can store that address.
- (*ptr) means "the array pointed to by ptr".
- (*ptr)[i] accesses each element inside that array.

Difference between a single variable pointer (int a) and an array pointer (int arr[5]).

Case 1: Single Variable – Memory Layout

```
int a = 10;
int *p;
p = &a;
```



☞ p = &a; means “store the address of a into pointer p.”
❌ You cannot write p = a; because a is not an address — it’s just an integer value (10).

Parul® University

DIGITAL LEARNING CONTENT

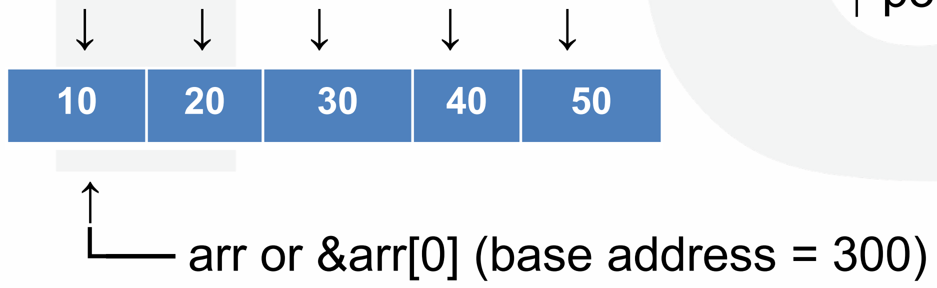
Difference between a single variable pointer (int a) and an array pointer (int arr[5]).

Case 2: Array – Memory

```
int arr[5] = {10, 20, 30, 40, 50};
int *p;
p = arr;      // same as p = &arr[0];
```

Array elements in consecutive memory:

Address: 300   304   308   312   316



p = 300
Address: 400

↑ pointer 'p' stores base address

☞ arr automatically acts like the address of the first element (&arr[0]).
So p = arr; and p = &arr[0]; are the same thing.

Summary: Single Element Pointer and Array Pointer

Concept	Expression	Meaning
Single variable	p = &a;	Pointer gets address of a
Array	p = arr;	Pointer gets base address of array (same as &arr[0])
Invalid	p = a;	✗ a is a value, not an address

Array of Pointers

An **array of pointers** is an array where **each element stores an address** (not an actual value).

Each pointer can point to different variables or data (like integers or strings).

```
int a = 10, b = 20, c = 30;
int *arr[3];     // array of 3 pointers to
int

arr[0] = &a;
arr[1] = &b;
arr[2] = &c;
```

```
Variables in memory:
 Address   Value
 1000  →   a = 10
 1004  →   b = 20
 1008  →   c = 30

Array of pointers:
 Index   arr[i] (stores address)   *arr[i] (value)
 arr[0]  → 1000  ----------------→  10
 arr[1]  → 1004  ----------------→  20
 arr[2]  → 1008  ----------------→  30
```

Example 1: Array of Pointers to Integers

```
#include <stdio.h>
int main() {
    int a = 10, b = 20, c = 30;
    int *arr[3];   // array of 3 integer pointers

    arr[0] = &a;
    arr[1] = &b;
    arr[2] = &c;

    printf("Values using array of pointers:\n");
    for (int i = 0; i < 3; i++) {
        printf("*arr[%d] = %d\n", i, *arr[i]);
    }
    return 0;
}
```

Explanation

arr is an array with 3 elements — each element is a pointer to int.

So:

arr[0] → points to a

arr[1] → points to b

arr[2] → points to c

*arr[i] dereferences the pointer to get the value.

Output

```
Values using array of pointers:
*arr[0] = 10
*arr[1] = 20
*arr[2] = 30
```

Parul® University

Example 2: Array of Pointers to Strings

```
#include <stdio.h>

int main() {
    const char *names[] = {"Alice",
"Bob", "Charlie", "David"};

    printf("Names stored in array of
pointers:\n");
    for (int i = 0; i < 4; i++) {
        printf("names[%d] = %s\n", i,
names[i]);
    }

    return 0;
}
```

Explanation

Each element of names points to the first character of a string.
names[i] gives the address of the string.
%s prints the string at that address.

Output

```
Names stored in array of
pointers:
names[0] = Alice
names[1] = Bob
names[2] = Charlie
names[3] = David
```

Functions Returning Pointer

A **function can return a pointer** — that is, it can return the **address** of a variable or memory location instead of a direct value.

Syntax:

```
datatype* function_name(parameters);
```

Example 1: Returning Pointer to a Local Variable (❌ Wrong Way)

```
#include <stdio.h>

int* wrongFunction() {
    int x = 10;
    return &x;   // ❌ x is local, destroyed after function ends
}

int main() {
    int *p = wrongFunction();  // dangling pointer
    printf("%d", *p);          // undefined behavior
    return 0;
}
```

Output

```
5 |     return &x;   // ❌ x is local, destroyed after function ends
```

Because x is a local variable, it is destroyed when the function ends.

The returned address no longer points to valid memory — this causes undefined behavior.

Example 2: Returning Pointer to a Static Variable

```
#include <stdio.h>

int* getNumber() {
    static int x = 42;   // static variable → retains value after function ends
    return &x;           // safe to return its address
}

int main() {
    int *p = getNumber();
    printf("Value = %d\n", *p);   // works correctly
    return 0;
}
```

Explanation:
static int x remains in memory even after the function returns.
So returning &x is safe.
The pointer p in main() correctly points to that memory.

Output
```
Value = 42
```

Example 3: Returning Pointer to a an array element

```
#include <stdio.h>

int* findMax(int arr[], int n) {
    int *max = &arr[0];
    for(int i = 1; i < n; i++) {
        if(arr[i] > *max)
            max = &arr[i];
    }
    return max;   // returns address of max element
}

int main() {
    int nums[5] = {10, 25, 5, 40, 30};
    int *p = findMax(nums, 5);
    printf("Maximum value = %d\n", *p);
    return 0;
}
```

Explanation:

findMax() returns the address of the largest number in the array.

p in main() receives that address and dereferences it.

Output

```
Maximum value = 40
```

Pointer v/s array

Pointer	Array
1. A pointer is a place in memory that keeps address of another place inside	1. An array is a single, pre allocated chunk of contiguous elements (all of the same type), fixed in size and location.
2. Pointer can't be initialized at definition.	2. Array can be initialized at definition. Example int num[] = { 2, 4, 5}
3. Pointer is dynamic in nature. The memory allocation can be resized or freed later.	3. They are static in nature. Once memory is allocated , it cannot be resized or freed dynamically.

Dynamic Memory Allocation

- Allows you to manage memory during a program's **runtime** (execution time), rather than at compile time.
- This is essential when the size of data needed is unknown in advance, such as when handling user input of variable size or creating complex data structures like linked lists that can grow or shrink dynamically.

Dynamic Memory Allocation

- Allows you to manage memory during a program's **runtime** (execution time), rather than at compile time.
- This is essential when the size of data needed is unknown in advance, such as when handling user input of variable size or creating complex data structures like linked lists that can grow or shrink dynamically.

Parul® University

DIGITAL LEARNING CONTENT

stdlib.h

Key Functions for Dynamic Memory Management

The C standard library, via the <stdlib.h> header file, provides four primary functions for dynamic memory allocation:

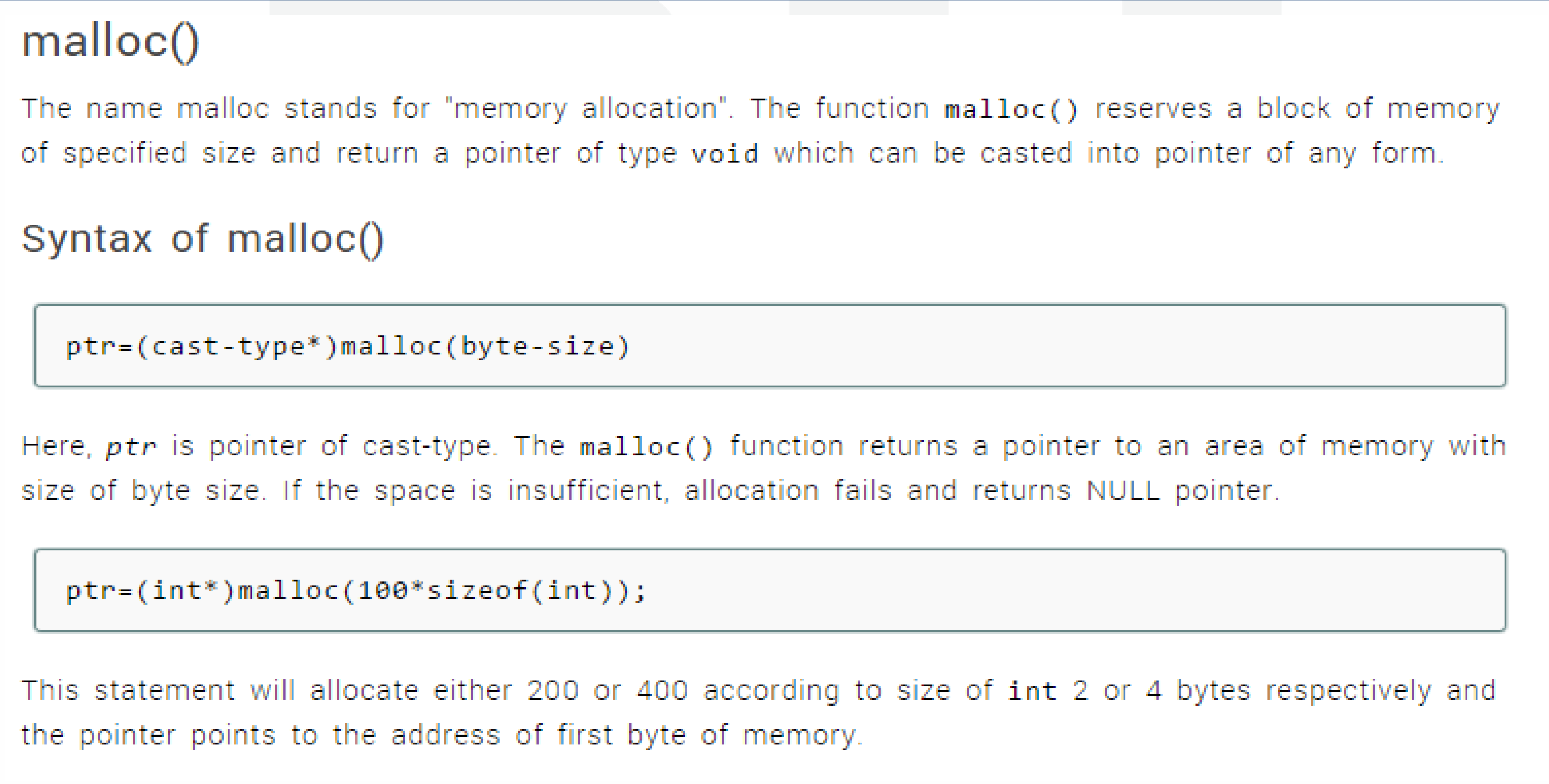
Function	Purpose	Initialization
malloc()	Allocates a single block of a specified size (in bytes)	Memory is uninitialized (contains garbage values).
calloc()	Allocates memory for a specified number of elements of a given size.	All allocated memory is initialized to zero.
free()	Deallocates memory that was previously allocated by malloc(), calloc(), or realloc().	Releases memory back to the system.
realloc()	Resizes a previously allocated memory block to a new size.	Retains the original data; new memory is uninitialized.

stdlib.h

Although, C language inherently does not has any technique to allocated memory dynamically, there are 4 library functions under "`stdlib.h`" for dynamic memory allocation.

Function	Use of Function
malloc()	Allocates requested size of bytes and returns a pointer first byte of allocated space
calloc()	Allocates space for an array elements, initializes to zero and then returns a pointer to memory
free()	dellocate the previously allocated space
realloc()	Change the size of previously allocated space

malloc()



malloc()

The name malloc stands for "memory allocation". The function **`malloc()`** reserves a block of memory of specified size and return a pointer of type **`void`** which can be casted into pointer of any form.

Syntax of malloc()

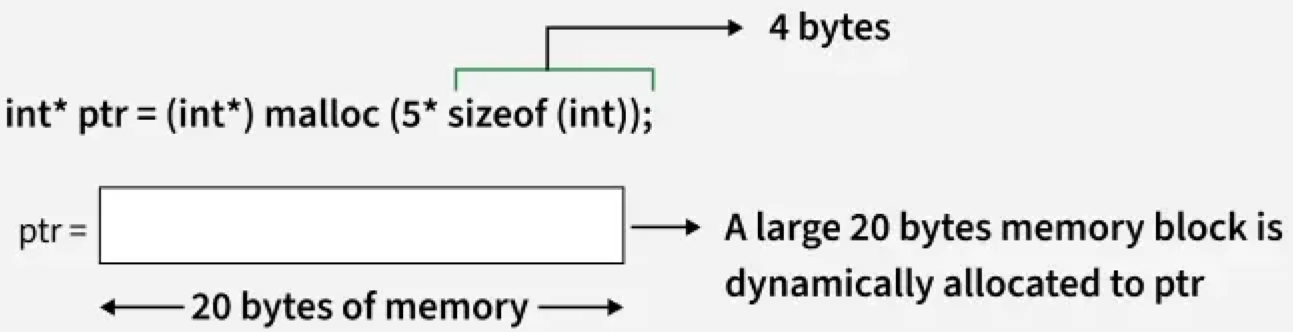
```
ptr=(cast-type*)malloc(byte-size)
```

Here, *`ptr`* is pointer of cast-type. The **`malloc()`** function returns a pointer to an area of memory with size of byte size. If the space is insufficient, allocation fails and returns NULL pointer.

```
ptr=(int*)malloc(100*sizeof(int));
```

This statement will allocate either 200 or 400 according to size of **`int`** 2 or 4 bytes respectively and the pointer points to the address of first byte of memory.

malloc()



4 bytes
int* ptr = (int*) malloc (5* sizeof (int));
ptr =
A large 20 bytes memory block is dynamically allocated to ptr
20 bytes of memory

Img src: https://www.geeksforgeeks.org/c/dynamic-memory-allocation-in-c-using-malloc-calloc-free-and-realloc/

Example: malloc()

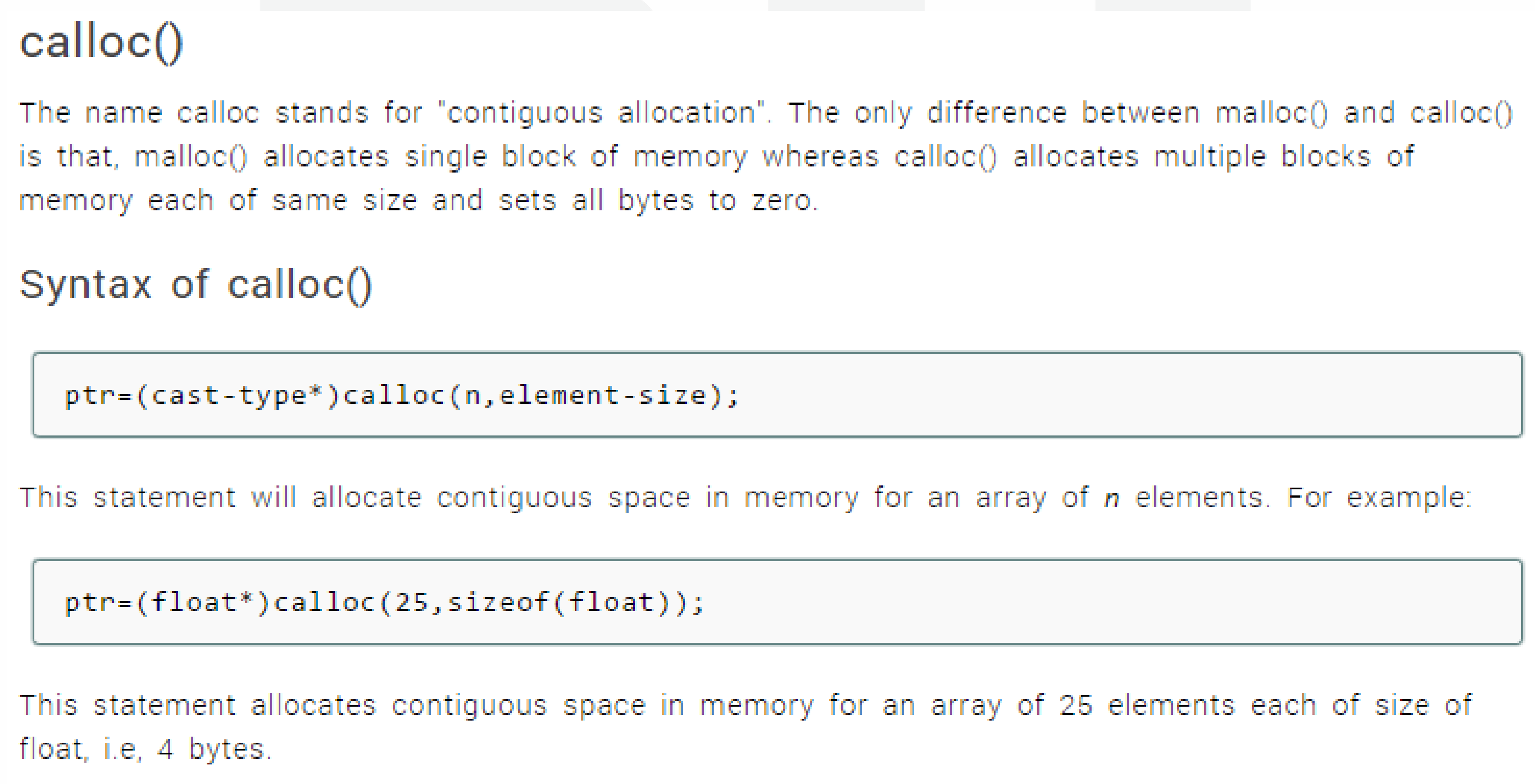
“Write a C program to find sum of n elements entered by user. Allocate memory dynamically using malloc() function.”

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

int main() {
    int n, i, *ptr, sum = 0;

    printf("Enter number of elements: ");
    scanf("%d", &n);

    // memory allocated using malloc
    ptr = (int*) malloc(n * sizeof(int));

    if (ptr == NULL) {
        printf("Error! memory not allocated.");
        exit(0);
    }

    printf("Enter elements of array: ");
    for (i = 0; i < n; i++) {
        scanf("%d", ptr + i);
        sum += *(ptr + i);
    }

    printf("Sum = %d", sum);

    free(ptr);
    return 0;
}
```

Output

```
Enter number of elements: 3
Enter elements of array: 10
20
30
Sum = 60
```

calloc()



calloc()

The name calloc stands for "contiguous allocation". The only difference between malloc() and calloc() is that, malloc() allocates single block of memory whereas calloc() allocates multiple blocks of memory each of same size and sets all bytes to zero.

Syntax of calloc()

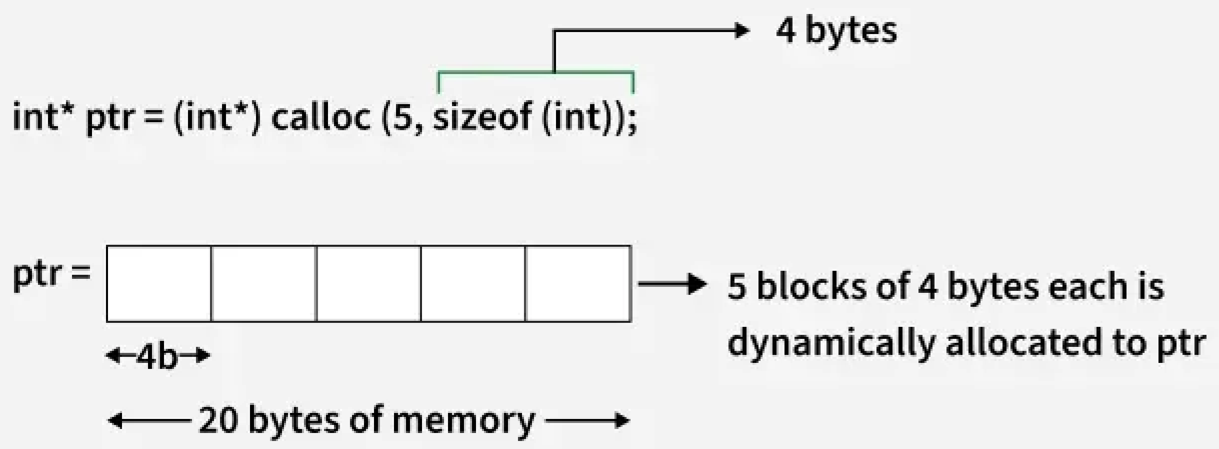
```
ptr=(cast-type*)calloc(n,element-size);
```

This statement will allocate contiguous space in memory for an array of *n* elements. For example:

```
ptr=(float*)calloc(25,sizeof(float));
```

This statement allocates contiguous space in memory for an array of 25 elements each of size of float, i.e, 4 bytes.

calloc()


4 bytes
int* ptr = (int*) calloc (5, sizeof (int));
ptr =
4b
20 bytes of memory
5 blocks of 4 bytes each is dynamically allocated to ptr

Example: calloc()

“Write a C program to find sum of n elements entered by user. Allocate memory dynamically using calloc() function.”

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

int main() {
    int n, i, *ptr, sum = 0;

    printf("Enter number of elements: ");
    scanf("%d", &n);

    // memory allocated using malloc
    ptr = (int*) calloc(n, sizeof(int));

    if (ptr == NULL) {
        printf("Error! memory not allocated.");
        exit(0);
    }

    printf("Enter elements of array: ");
    for (i = 0; i < n; i++) {
        scanf("%d", ptr + i);
        sum += *(ptr + i);
    }

    printf("Sum = %d", sum);

    free(ptr);
    return 0;
}
```

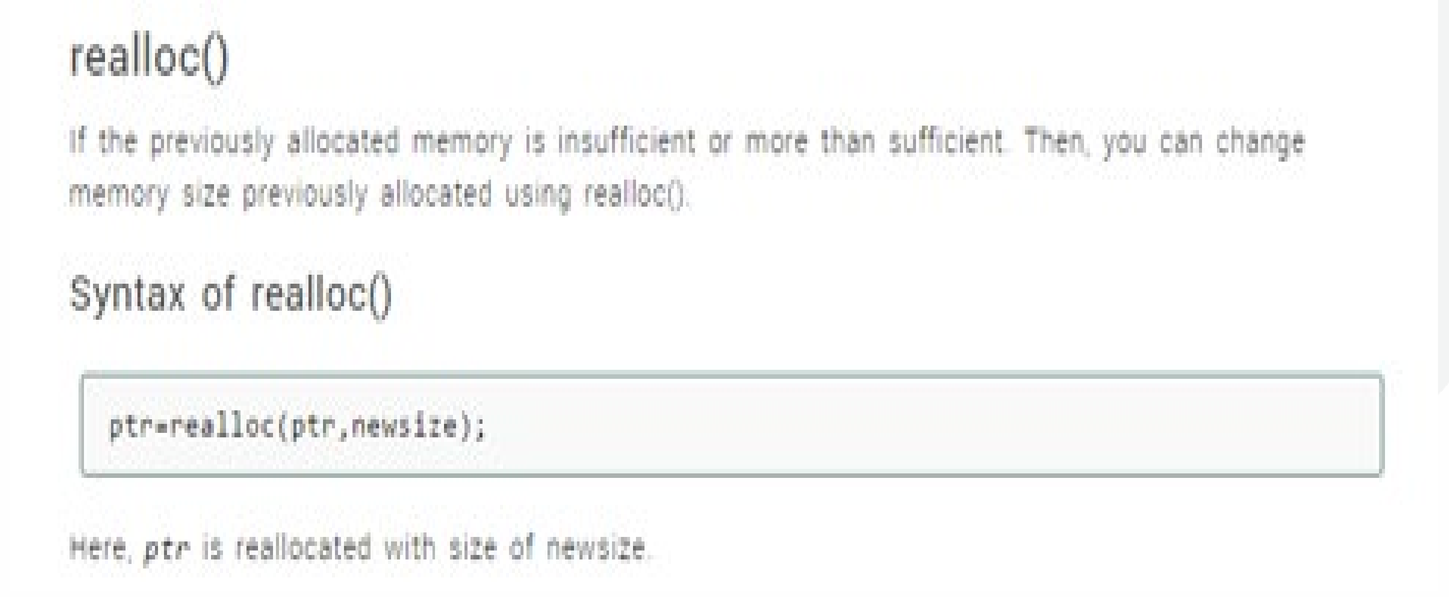
Output

```
Enter number of elements: 3
Enter elements of array: 10
20
30
Sum = 60
```

Parul® University

DIGITAL LEARNING CONTENT

realloc()



realloc()

If the previously allocated memory is insufficient or more than sufficient. Then, you can change memory size previously allocated using realloc().

Syntax of realloc()

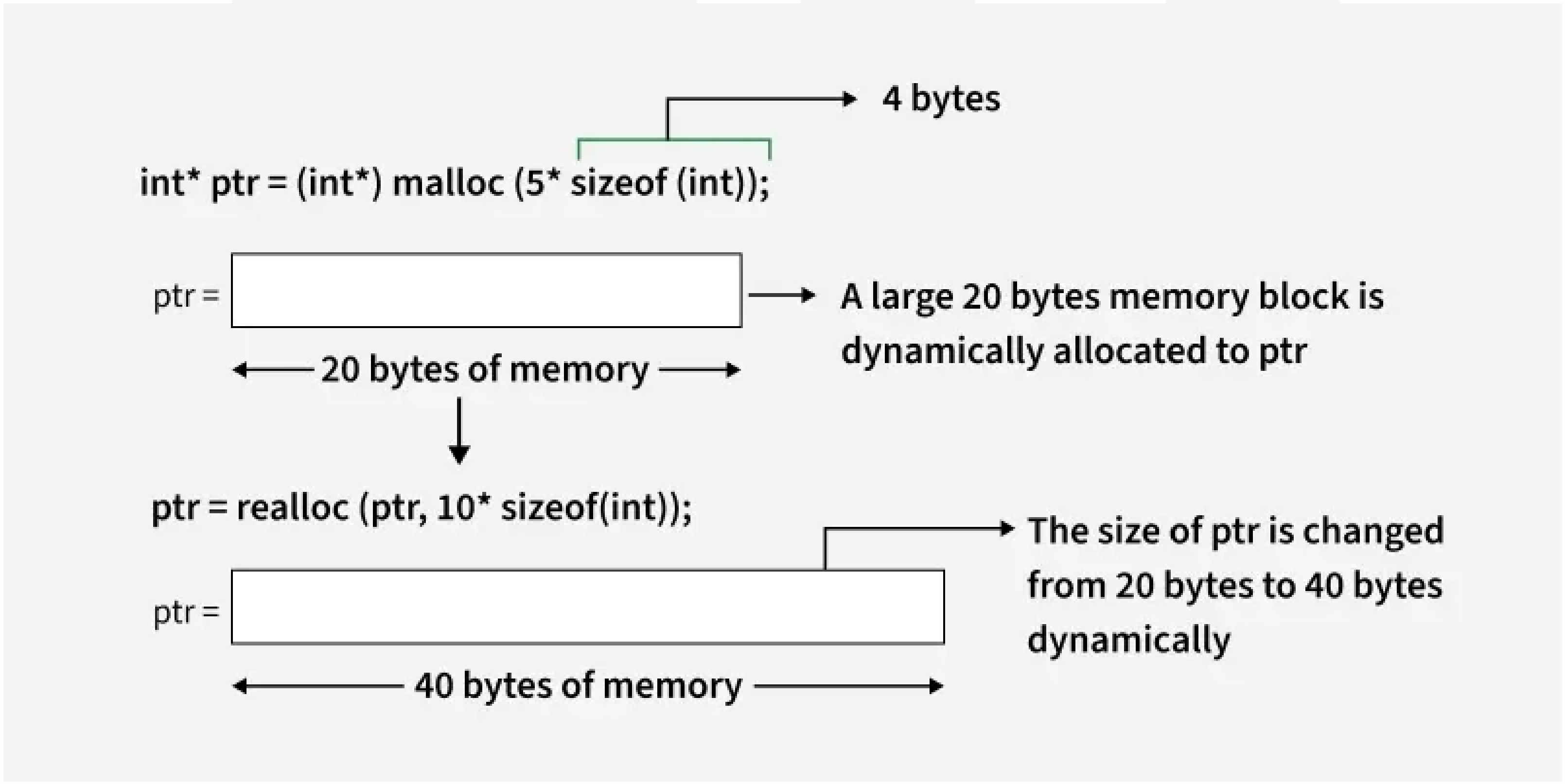
```
ptr=realloc(ptr,newsize);
```

Here, *ptr* is reallocated with size of newsize.

Parul® University

DIGITAL LEARNING CONTENT

realloc()


4 bytes
int* ptr = (int*) malloc (5* sizeof (int));
ptr =
A large 20 bytes memory block is dynamically allocated to ptr
20 bytes of memory
ptr = realloc (ptr, 10* sizeof(int));
The size of ptr is changed from 20 bytes to 40 bytes dynamically
ptr =
40 bytes of memory

Img src: https://www.geeksforgeeks.org/c/dynamic-memory-allocation-in-c-using-malloc-calloc-free-and-realloc/

Example: realloc()

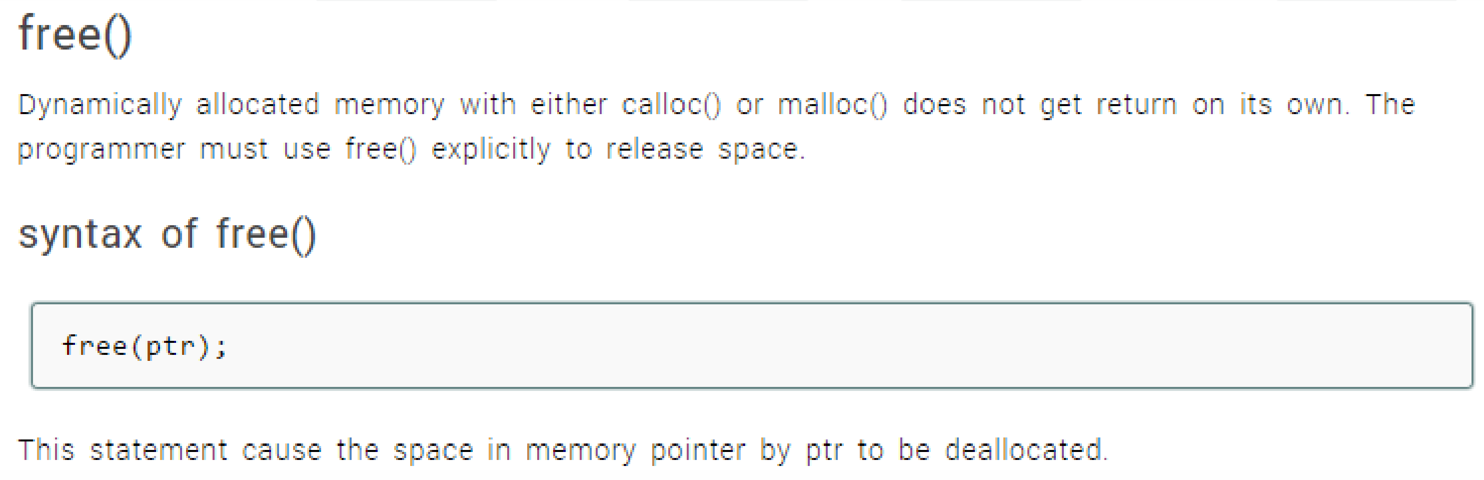
Write a C program to demonstrate the use of realloc() by first allocating memory for an integer array using malloc(), displaying the allocated addresses, then reallocating the memory to a new size using realloc() and displaying the new addresses.

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

int main() {
    int *ptr, i, n1, n2;

    printf("Enter size of array:
");
    scanf("%d", &n1);

    ptr = (int*) malloc(n1 *
sizeof(int));

    printf("Address of
previously allocated
memory:\n");
for (i = 0; i < n1; i++)
        printf("%u\t", ptr + i);

printf("\n\nEnter new size of
array: ");
    scanf("%d", &n2);

    ptr = realloc(ptr, n2);

    for (i = 0; i < n2; i++)
        printf("%u\t", ptr + i);

    return 0;
}
```

Output

```
Enter size of array: 2
Address of previously allocated memory:
3962188080      3962188084

Enter new size of array: 3
3962188080      3962188084      3962188088
```

free()



free()

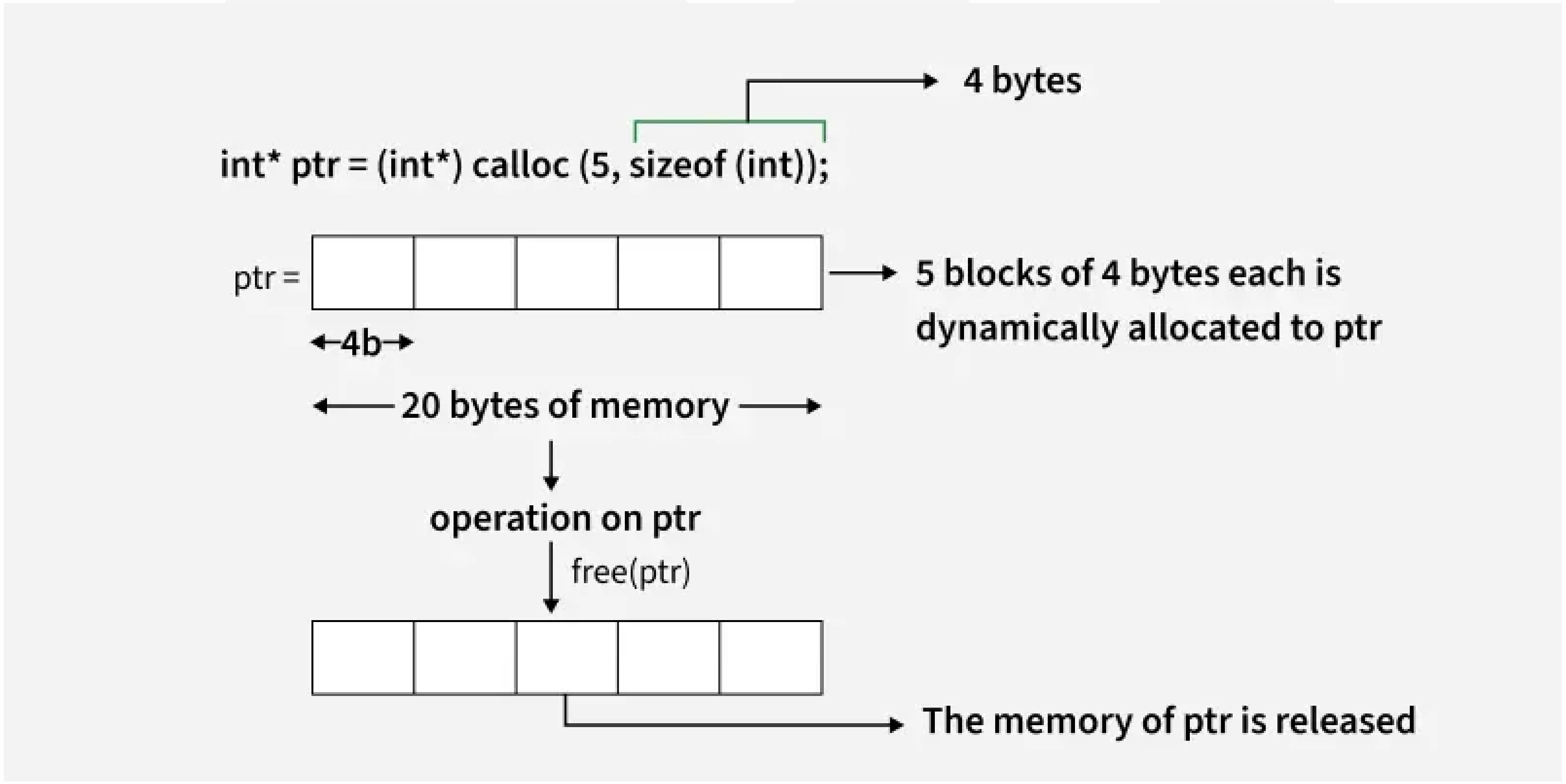
Dynamically allocated memory with either calloc() or malloc() does not get return on its own. The programmer must use free() explicitly to release space.

syntax of free()

```
free(ptr);
```

This statement cause the space in memory pointer by ptr to be deallocated.

free()


4 bytes
int* ptr = (int*) calloc (5, sizeof (int));
ptr =
5 blocks of 4 bytes each is dynamically allocated to ptr
4b
20 bytes of memory
operation on ptr
free(ptr)
The memory of ptr is released

Example: free()

Write a C program to demonstrate the use of realloc() by first allocating memory for an integer array using malloc(), displaying the allocated addresses, then reallocating the memory to a new size using realloc() and displaying the new addresses.

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

int main() {
    int *ptr, n, i;

    printf("Enter number of elements: ");
    scanf("%d", &n);

    // Allocate memory using malloc
    ptr = (int*) malloc(n * sizeof(int));

    if (ptr == NULL) {
        printf("Memory not allocated!\n");
        exit(0);
    }
printf("Enter elements:\n");
    for (i = 0; i < n; i++) {
        scanf("%d", ptr + i);
    }

    printf("You entered:\n");
    for (i = 0; i < n; i++) {
        printf("%d ", *(ptr + i));
    }
    // Free the allocated memory
    free(ptr);

    printf("\nMemory successfully
freed.\n");

    return 0;
}
```

Output

```
Enter elements:
10
20
30
You entered:
10 20 30
Memory successfully freed.
```

File Management in C

- File management in C refers to the process of **creating, opening, reading, writing, and closing files** using standard library functions.
- Files allow programs to **store data permanently** on disk rather than temporarily in memory.

Why File Management?

•Data remains **stored permanently** even after the program ends.

•Useful for:

- Storing records (students, employees, etc.)
- Logging program output
- Reading configuration settings
- Handling large data that doesn’t fit in memory.

Types of File in C

C supports two types of files:

1. Text Files
   - ❖ Stored as readable characters. Each line ends with a newline character.
   - ❖ **Contains ASCII codes only**
   - ❖ **The last byte of a file contains the end-of-file character (`EOF`), with ASCII code 1A (hex).**
   - ❖ **While reading a text file, the EOF character can be checked to know the end.**
   - ❖ Example: .txt, .csv, .ini

Types of File in C

2. Binary Files
   - Data stored in binary format (0s and 1s).Faster
   - and efficient.
   - **Can contain non-ASCII characters**
   - **Image, audio, video, executable, etc.**
   - **To check the end of file here, the *file size* value (also stored on disk) needs to be checked.**
   - Example: .bin, program data files

File Pointer

To work with files, C uses a **FILE pointer**:

```
FILE *fp;
```

This pointer stores the reference to the opened file.

Basic File Operations

Operation	Function	Purpose
Open	fopen()	Opens a file
Read	fscanf(), fgetc(), fgets(), fread()	Reads from a file
Write	fprintf(), fputc(), fputs(), fwrite()	Writes to a file
Close	fclose()	Closes a file

- fscanf() reads formatted text data, while, fread() reads raw, binary data in fixed-size blocks.
- fprintf writes data as formatted text, while fwrite writes data as raw binary blocks

Command File modes in C – Text Files

Mode	Meaning
"r"	Open file for reading (file must exist)
"w"	Open for writing (creates new or overwrites existing)
"a"	Append mode (writes at end of file)
"r+"	Read + write
"w+"	Write + read (overwrites file)
"a+"	Append + read

Parul® University

Command File modes in C – Binary Files

Mode	Description	Behavior
"rb"	Read binary	Opens a binary file for reading. The file must exist, or fopen() returns NULL.
"wb"	Write binary	Opens a binary file for writing. If the file exists, its contents are overwritten (truncated to zero length); if it doesn't exist, a new file is created.
"ab"	Append binary	Opens a binary file for appending. Data is added to the end of the file. A new file is created if it does not exist.
"rb+"	Read and Write binary	Opens a binary file for both reading and writing. The file must exist, or fopen() returns NULL.
"wb+"	Write and Read binary	Opens a binary file for both writing and reading. If the file exists, its contents are overwritten; if it doesn't exist, a new file is created.
"ab+"	Append and Read binary	Opens a binary file for both reading and appending. Data is appended to the end of the file. A new file is created if it does not exist.

b – used for handing binary files

1. fopen() – Open a File

Syntax: FILE *fopen(const char *filename, const char *mode);

Example: Opening a file for writing

```
#include <stdio.h>
int main() {
    FILE *fp;
    fp = fopen("data.txt", "w");

    if (fp == NULL) {
        printf("File cannot be opened!");
        return 1;
    }
    printf("File opened successfully!");
    fclose(fp);
    return 0;
}
```

2. fprintf() – Writing Text to a File

Syntax: int fprintf(FILE *stream, const char *format, ...);

Example: Example: Writing formatted text

```
#include <stdio.h>
int main() {
    FILE *fp = fopen("data.txt", "w");
    if (fp == NULL) {
        printf("Error opening file!");
        return 1;
    }
    fprintf(fp, "Name: Ridha\nAge: 25\n");
    fclose(fp);
    return 0;
}
```

3. fscanf() – Reading Text from a File

Syntax: int fscanf(FILE *stream, const char *format, ...);

Example: Reading formatted text

```
#include <stdio.h>
int main() {
    FILE *fp;
    fp = fopen("data.txt", "w");

    if (fp == NULL) {
        printf("File cannot be opened!");
        return 1;
    }
    printf("File opened successfully!");
    fclose(fp);
    return 0;
}
```

4. fgetc() – Read a Single Character

Syntax: int fgetc(FILE *stream);

Example: Reading a Single Character

```
#include <stdio.h>
int main() {
    FILE *fp = fopen("data.txt", "r");
    char ch;
    if (fp == NULL) {
        printf("Error opening file!");
        return 1;
    }
    while ((ch = fgetc(fp)) != EOF) {
        printf("%c", ch);
    }
    fclose(fp);
    return 0;
}
```

5. fputc() – Write a Single Character

Syntax: int fputc(int character, FILE *stream);

Example: Writing a Single Character

```
#include <stdio.h>
int main() {
    FILE *fp = fopen("data.txt", "w");
    if (fp == NULL) {
        printf("Error opening file!");
        return 1;
    }
    fputc('A', fp);
    fputc('\n', fp);
    fclose(fp);
    return 0;
}
```

6. fgets() – Read a String

Syntax: char *fgets(char *str, int n, FILE *stream);

Example: Reading a String

```
#include <stdio.h>
int main() {
    FILE *fp = fopen("data.txt", "r");
    char line[100];
    if (fp == NULL) {
        printf("File not found!");
        return 1;
    }
    fgets(line, sizeof(line), fp);
    printf("Read: %s", line);
    fclose(fp);
    return 0;
}
```

7. fputs() – Write a String

Syntax: int fputs(const char *str, FILE *stream);

Example: Writing a String

```
#include <stdio.h>
int main() {
    FILE *fp = fopen("data.txt", "w");
    if (fp == NULL) {
        printf("Error opening file!");
        return 1;
    }
    fputs("Hello C Programming!", fp);
    fclose(fp);
    return 0;
}
```

8. fclose() – Close a File

Syntax: `fclose(fp);`

Summary Table

Function	Purpose
fopen()	Open file
fclose()	Close file
fprintf()	Write formatted text
fscanf()	Read formatted text
fputc()	Write single character
fgetc()	Read single character
fputs()	Write string
fgets()	Read string

The exit() function

- Sometimes error checking means we want an "*emergency exit*" from a program.
- In main()we can use returnto stop.
- In functions we can use exit()to do this.
- Exit is part of the stdlib.h library.

```
exit(-1);
```

in a function is exactly the same as

```
return -1;
```

in the main routine

The exit() function - Example

```
#include <stdio.h>
#include <stdlib.h>
int main() {
    FILE *fptr;
    char filename[] = "file2.dat";
    fptr = fopen(filename, "w");
    if (fptr == NULL) {
        printf("ERROR IN FILE CREATION\n");
        exit(-1);
    }
    /* Do something with the file */
    fclose(fptr);
    return 0;
}
```

Three special streams

- Three special file streams are defined in the <stdio.h> header
  - stdin reads input from the keyboard
  - stdout send output to the screen
  - stderr prints errors to an error device (usually also the screen)

- What might this do?

```
fprintf (stdout,"Hello World!\n");
```

An example program

```
#include <stdio.h>

int main()
{
    int i;
    fprintf(stdout, "Give value of i \n");
    fscanf(stdin, "%d", &i);
    fprintf(stdout, "Value of i = %d \n", i);
    fprintf(stderr, "No error: But an example to show error
message.\n");

    return 0;
}
```

- printf → simpler, screen only
- scanf → simpler, keyboard only
- fprintf/fscanf → more powerful, can work with:
  - ❖ keyboard (stdin)
  - ❖ screen (stdout)
  - ❖ error messages (stderr)
  - ❖ files (via file pointers)

Output

```
Give value of i
3
Value of i = 3
No error: But an example to show error message.
```

Note: In place of file pointer, stdout, stdin or stderr can also be used

DIGITAL LEARNING CONTENT

Parul® University

www.paruluniversity.ac.in