Unit-9

Pointer

Pointer:

- Pointer is a derived data type.
- The Pointer in C, is a variable that stores address of another variable.
- Pointer points to the memory location.
- Pointer is used to access the information from memory.
- Simply, Pointer is used to access the information of a particular memory location.
- Whenever we are working with the pointer, memory allocation is very important because pointer concept is completely based on the memory allocation only.

Declaration of Pointer:

Syntax:

data-type* identifier; or data-type *identifier;

Example:

```
int* a;
int *a;
float* check;
```

Types of Pointer:

1. Typed Pointer
2. Untyped Pointer

3. Typed Pointer:

Typed pointer points to specific type of data.

Example:

int* ——→ int data

double* ——→ double data

struct emp* ——→ employee data

2. Untyped Pointer:

- Untyped pointer can points to any type of data.
- This type of pointer is also called **Generic Pointer**.
- Void pointer is called generic type pointer. So, it can points to any type of data.

  Example: void* a~~ny type~~ of data

The & and * operator:

- In the pointer concept, we need to take the help of two operators (& and *) to perform any operation.

1. Address Operator(&)

   Address Operator returns the **address** of the particular variable.

2. Pointer Operator(*)

   Pointer Operator returns the value which is inside the specified address.

Example:

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int x=500;
    int *ptr;
    ptr=&x;
    printf("%d\n",x);-----500
    printf("%u\n",ptr);-----3000
    printf("%u\n",&x);-----3000
    printf("%u\n",&ptr);----6700
    printf("%d\n",*ptr);----500
    printf("%d\n",*(&x));------500
    getch();
}
```

Chain of Pointers:

- In simple, Chain of pointer is known as pointer to pointer.
- It is possible to make a pointer that points to another pointer, thus creating a chain of pointers.
- A pointer is used to point to a memory location of a variable. A pointer stores the address of a variable.
- Similarly, a chain of pointers is when there are multiple levels of pointers. Simplifying, a pointer points to address of a variable, double-pointer points to a variable and so on. This is called **multiple indirections.**

Syntax:

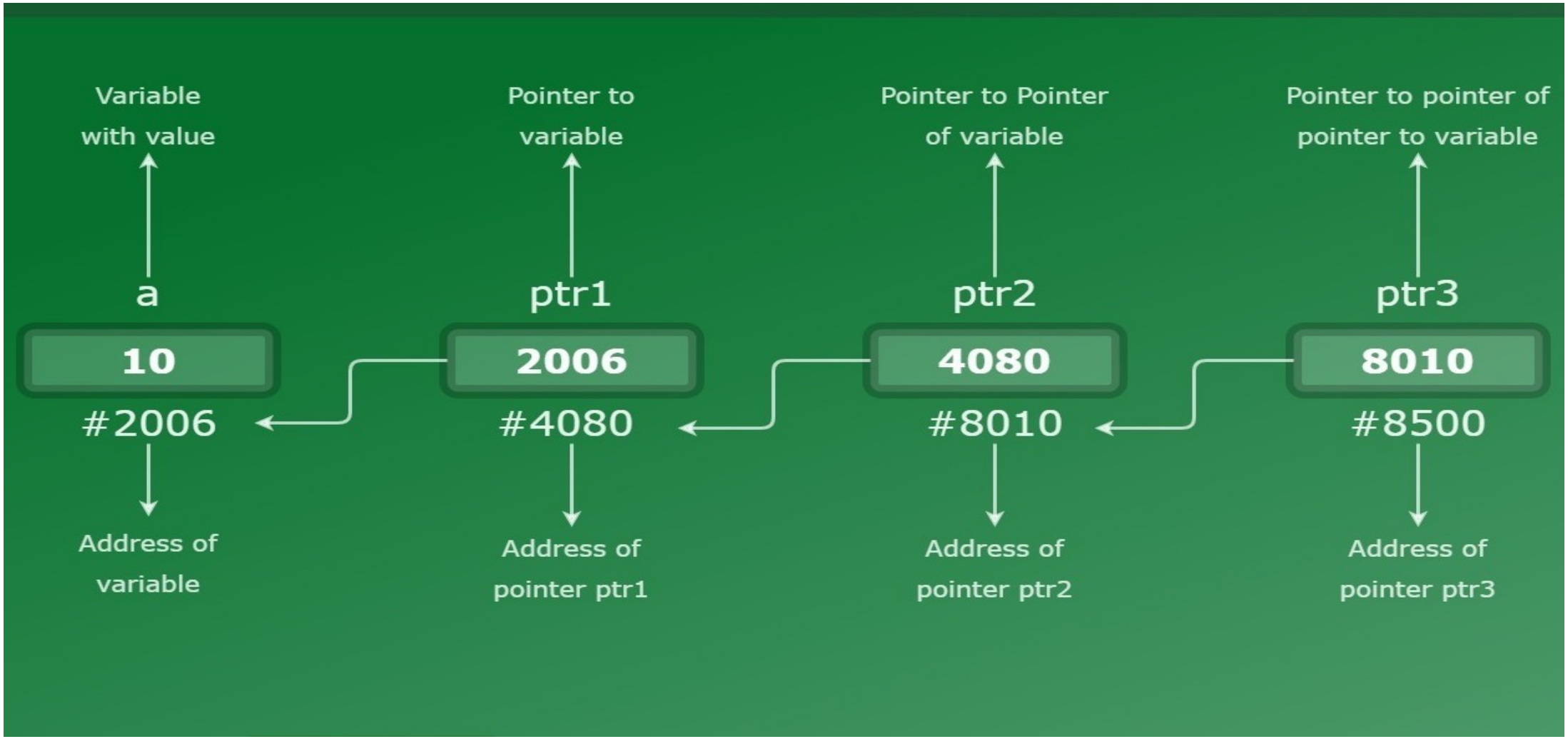
```
// level-1 pointer declaration
datatype *pointer;

// level-2 pointer declaration
datatype **pointer;

// level-3 pointer declaration
datatype ***pointer;
.
.
and so on
```

Note: The level of the pointer depends on how many asterisks the pointer variable is preceded with at the time of declaration.

Chain of Pointers:



Example:

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int x=90;
    int *p1,**p2;
    p1=&x;
    p2=&p1;
    printf("%d",**p2);
    getch();
}
```

Pointer Arithmetic:

- C supports four arithmetic operators that can be used with pointers.

S.N.	Operator	Symbol
1.	Addition	+
2.	Subtraction	-
3.	Increment	++
4.	Decrement	--

The expression “ptr++” points to the memory location of the next element of its base type.

For example: ptr be an integer pointer with the current holding address 6220, after the expression “ptr++” ptr contains 6224. In the same way “ptr--” contains the memory address.

Continue…

We may add or subtract integers to or from pointers.

The expression “ptr=ptr+2” points to the address of element to forward by two position in the forward direction.

For example: Initially “ptr” holds the address 1020 then after the expression “ptr=ptr+2” holds the address 1028. In the same way we can also subtract the integer from pointers.

Example:

```
#include<stdio.h>
#include<conio.h>
void main()
{
      int x=90;
      int *ptr;
      ptr=&x;
      printf("%u\n",ptr);
      ptr++;
      printf("%u\n",ptr);
      getch();
}
```

Example:

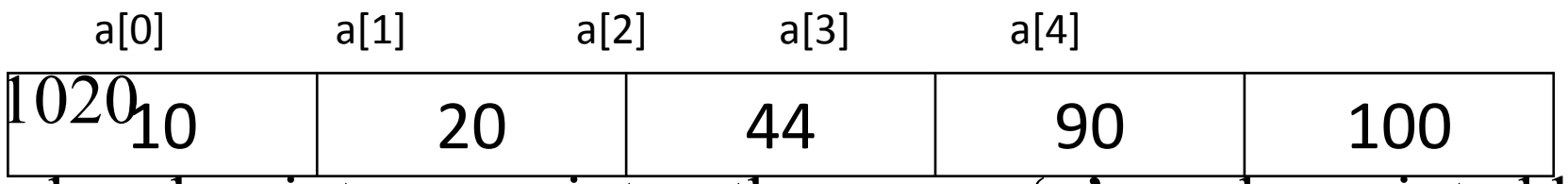
```
#include<stdio.h>
#include<conio.h>
void main()
{
	int x=90;
	int *ptr;
	ptr=&x;
	printf("%u\n",ptr);
	ptr=ptr+2;
	printf("%u\n",ptr);
	getch();
}
```

Arrays and Pointers:

- An array name by itself is an address, or pointer.
- When we declare a primitive type array it can able to store integers, characters, float value etc. An array can store references(address) as an elements and this is called array of pointers and those pointer can be of any type such as integer pointer, float pointer, character pointer and so on.
- Contiguous memory locations are allocated for all the elements of the array by the compiler.
- The base address is the location of the first element (index 0) of an array.

Arrays and Pointers:

Example: int a[5]={10,20,44,90,100};



a=&a[0]=1020

If ‘p’ is declared as integer pointer, then array ‘a’ can be pointed by pointer ‘p’ as

p=&a[0]; Here, p is holding the address of another variable

Arrays and Pointers:

- Every element of an array ‘a’ can be accessed by using pointer ‘p’ as:

p=&a[0];

p+1=&a[1];

p+2=&a[2];

p+3=&a[3];

p+4=&a[4];

- Address of an particular element can be calculated using its index and the scale factor of the data type.

  For Example: address of a[4]= base address+(index number*scale factor of integer data type)

  =1020+(4*2)

  =1028

Array and Pointers:

- Instead of using array index number, we can access the every element of an array using pointer variable as: **a[i]=*(p+i)**

*p=10;

*(p+1)=20;

*(p+2)=44;

*(p+3)=90;

*(p+4)=100;

Example:

```
#include<stdio.h>
#include<conio.h>
void main()
{
	int a[5]={10,20,44,90,100};
	int i, *p;
	p=&a[0];
	printf("Your array elements are:\n");
	for(i=0;i<5;i++)
	{
		printf("%d\t",*(p+i));
	}
	getch();
}
```

Arrays of Pointers:

- Collection of address or collection of pointers

Declaration:

Syntax: data-type *pointer-name[size];

int *p[5]; ----it represents an array of pointers that can store five integer element's address.

Example:

```
#include<stdio.h>
#include<conio.h>
void main()
{
	int a[5]={10,20,44,90,100};
	int *p[5],i;
	for(i=0;i<5;i++)
	{
		p[i]=&a[i];
	}

	printf("Your array elements are:\n");
	for(i=0;i<5;i++)
	{
		printf("%d\t",*p[i]);
	}
	getch();
}
```

```
//Two-dimensional array using pointer
#include<stdio.h>
#include<conio.h>
void main()
{
	int m[2][2]={1,2,4,5};
	int i,j;
	printf("Your matrix:\n");
	for(i=0;i<2;i++)
	{
		for(j=0;j<2;j++)

		{
			printf("%d\t",*(*(m+i)+j));
		}
		printf("\n");
	}
	getch();
}
```

Pointers and Character Strings:

- String is a one dimensional character array so we can also point to the string by using a character pointer variable.

Example:

```
#include<stdio.h>
#include<conio.h>
void main()
{
        char s[]="C Programming";
        char *p;
        p=s;
        while(*p!='\0')
        {
                printf("%c",*p);
                p++;
        }

        getch();
}
```

```
#include<stdio.h>
#include<conio.h>
int main()
{
    char *s="Programming";
    printf("%s\n",s);--programming
    printf("%c\n",s);--Null
    printf("%c\n",*s);--P
    printf("%c\n",*(s+2));---0
    printf("%c\n",*s+3);----s
    getch();
    return 0;
}
```

```
#include<stdio.h>
#include<conio.h>
int main()
{
    char *s1="Ram";
    char *s2;
    s2=s1;
    printf("%s",s2);
    getch();
    return 0;
}
```

Function Pointer:

- A pointer that can points to a function is called function pointer.
- Normal pointer variable stores the address of another variable but function pointer stores the address of another function.
- Function pointer is completely based on the prototype of the function.
- Syntax: **return-type (*pointer-name)(argument list);**
- Example: int (*p)(int);

  int (*ptr)(int, int);

where ptr can points to any function which is taking two integer arguments and returning integer data.

Example:

```
#include<stdio.h>
#include<conio.h>
int fun(int); //function prototype
void main()
{
	int x;
	int (*ptr)(int); //declaration of function pointer
	ptr=&fun;
	x=ptr(50); //function call
	printf("%d",x);
	getch();
}
int fun(int a)
{
	return a;
}
```

Example:

```
#include<stdio.h>
#include<conio.h>
int fun(int,int);
void main()
{
      int x;
      int (*ptr)(int,int);
      ptr=&fun;
      x=ptr(100,30);
      printf("%d",x);
      getch();
}
int fun(int a,int b)
{
      int m;
      m=a+b;
      return m;
}
```

Pointer as Function Arguments (vimp):

1. Call by value
2. Call by reference or Call by address

3. Call by value:

In a call by value, values of variables is passed to the function from the calling section.

This method copies the value of actual parameters into formal parameters.

When the function is called, a separate copy of the variable is created in the memory and the value of original variables is given to these variables. So, if any changes are made in the value of the called function is not reflected the original variable of the calling function.

Example of call by value:

```
#include<stdio.h>
#include<conio.h>
int swap(int,int);
void main()
{
        int a=100,b=200;
        printf("Before Swapping:\n");
        printf("a=%d b=%d\n",a,b);
        swap(a,b);
        printf("a=%d b=%d\n",a,b);
}
int swap(int x,int y)
{
        int temp=x;
        x=y;
        y=temp;
        printf("x=%d y=%d\n",x,y);
}
```

2. Call by Reference:

- In a call by reference, we pass the address or location of a variable to the function during the function call. Pointer are used to call a function by reference. When a function is call by reference, then the formal argument becomes reference to the actual argument. This means that it refers to the original values only by reference name. This function works with original data and the changes are made in the original data itself.

Example of call by Reference:

```
#include<stdio.h>
#include<conio.h>
int swap(int*,int*);
void main()
{
        int a=100,b=200;
        printf("Before Swapping:\n");
        printf("a=%d b=%d\n",a,b);
        swap(&a,&b);
        printf("a=%d b=%d\n",a,b);
}
int swap(int *x,int *y)
{
        int temp=*x;
        *x=*y;
        *y=temp;
        printf("x=%d y=%d\n",*x,*y);
}
```

Function Returning Pointers:

```
#include<stdio.h>
#include<conio.h>
int* fun();
void main()
{
      int *p=fun();
      printf("%d",*p);
      getch();
}
int* fun()
{
      int x=100;
      return &x;
}
```

Function Returning Pointers:

```
#include<stdio.h>
#include<conio.h>
int* sum(int*,int*);
void main()
{
    int a=100,b=200;
    int *ptr=sum(&a,&b);
    printf("Sum=%d",*ptr);
}
int* sum(int *x,int *y)
{
    static int c;
    c=(*x)+(*y);
    return &c;
}
```

Structure and Pointer:

- We can access the members of structure using structure pointer variable.
- To access the member of structure using pointer variable we use structure pointer operator or arrow operator.
- Example: struct emp *ptr=&e;

  Where ptr is a pointer to some variable of type struct emp.

Example:

```
#include<stdio.h>
#include<conio.h>
struct emp
{
        int id;
        char name[20];
        float salary;
};
void main()
{
        struct emp e={101,"Kiran",45000};
        struct emp *ptr=&e;
        printf("%d\n",(*ptr).id);
        printf("%s\n",(*ptr).name);
        printf("%.2f",(*ptr).salary);
        getch();

}
```

Example:

```
struct emp
{
	int id;
	char name[20];
	float salary;
};
void main()
{
	struct emp e={101,"Kiran",45000};
	struct emp *ptr=&e;
	printf("%d\n",ptr->id);
	printf("%s\n",ptr->name);
	printf("%.2f",ptr->salary);
	getch();

}
```

Example:

```
struct emp
{
	int id;
	char name[20];
	float salary;
};
void main()
{
	struct emp e={101,"Kiran",45000};
	struct emp *ptr=&e;
	printf("%d\n",(*&e).id);
	printf("%s\n",(*&e).name);
	printf("%.2f",(*&e).salary);
	getch();

}
```

Example:

```
struct emp
{
	int id;
	char name[20];
	float salary;
};
void main()
{
	struct emp e;
	struct emp *ptr=&e;
	printf("Enter id:");
	scanf("%d",&(*ptr).id);
	printf("Enter name:");
	scanf("%s",(*ptr).name);
	printf("Enter salary:");
	scanf("%f",&(*ptr).salary);
	printf("%d\t%s\t%.2f",(*ptr).id,(*ptr).name,(*ptr).salary);
	getch();

}
```

Dynamic Memory Allocation:

Static Memory Allocation:

- Memory allocated during compile time is called static memory.
- The memory allocated is fixed and can not be increased or decreased during run time.
- Example: int a[3]={10,20,30};

Here the size of an array is fixed so that we can not stored more than 3 elements.

Dynamic Memory Allocation:

- The process of allocating memory at the time of execution is called dynamic memory allocation.
- There are certain built-in functions that can help in allocating or deallocating some memory space at run time and these built-in functions are:
    1. malloc( )
    2. calloc( )
    3. realloc( )
    4. free( )
- Pointer play an important role in dynamic memory allocation.
- Allocated memory can only be accessed through pointers.

malloc( ):

- malloc is a built-in function declared in the header file <stdlib.h>
- malloc is the short name for “memory allocation” and is used to dynamically allocate a single large block of contiguous memory according to the size specified.
- Syntax: (void*)malloc(size_t size);

  Where size_t is defined in <stdlib.h> as unsigned int.

  -On success, it returns base address of memory blocks.

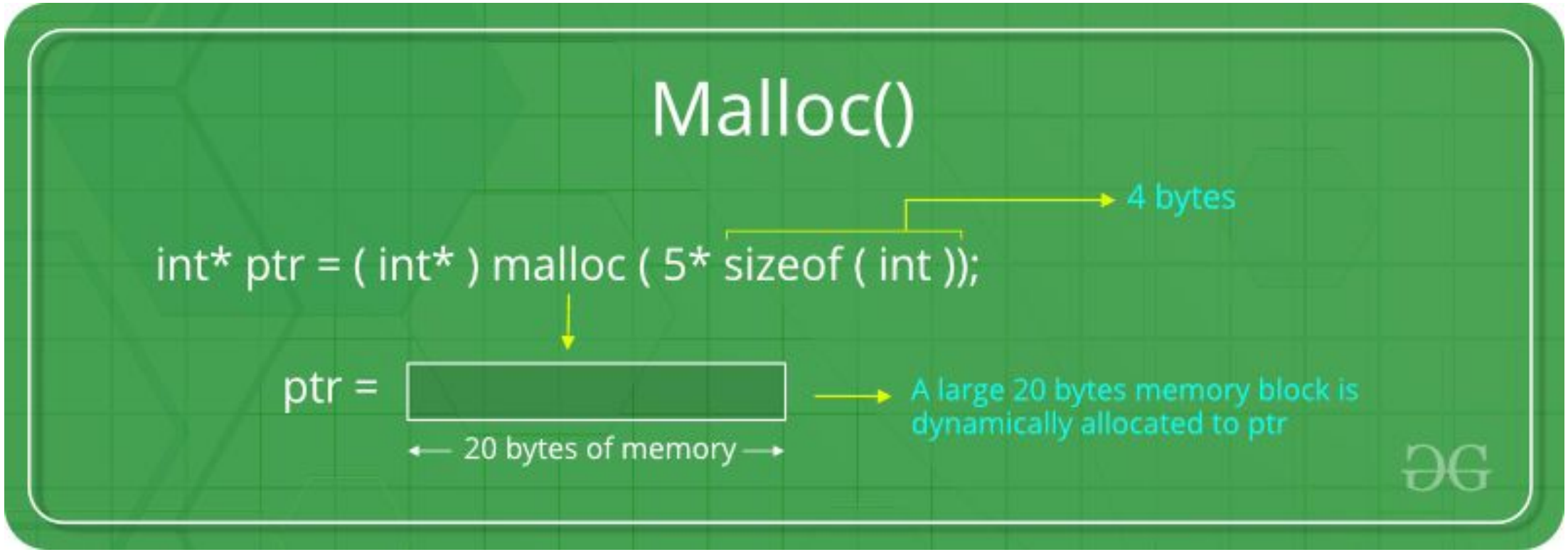
  -On failure, it returns NULL pointer.

malloc( ):

- It doesn’t initialize memory at execution time so that it has initializes each block with the default garbage value initially.
- The void pointer can be type casted to an appropriate type.

```
int *ptr=(int* )malloc(4);
```

Where malloc allocates the 4 bytes of memory in the memory and the address of the first byte is stored in the pointer ptr.


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

Example of malloc( ):

```
void main()
{
	int i,n;
	printf("Enter the number of elements:");
	scanf("%d",&n);
	int *ptr=(int *)malloc(n*sizeof(int));
	if(ptr==NULL)
	{
		printf("Memory not available.");
	}
	else
	{
		for(i=0;i<n;i++)
		{
			printf("Enter the element:");
			scanf("%d",ptr+i);
		}
		for(i=0;i<n;i++)
		{
			printf("%d\t",*(ptr+i));
		}
	}
	getch();
}
```

calloc( ):

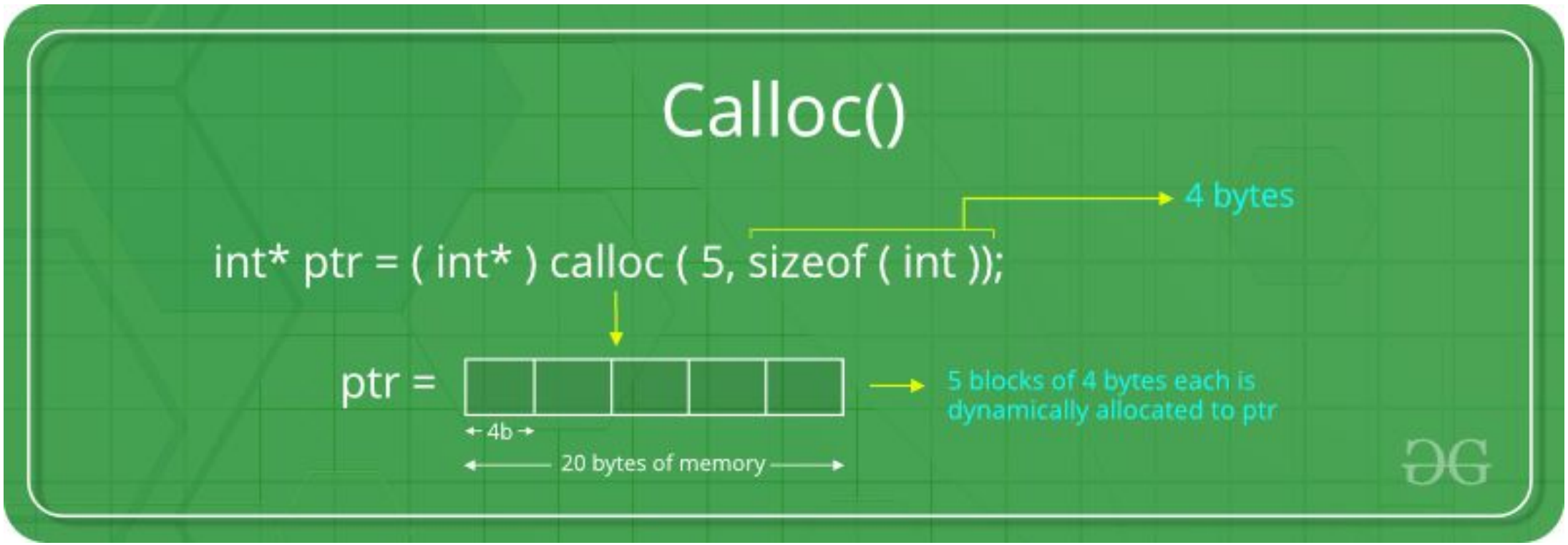
- “calloc” or “contiguous allocation” method in C is used to dynamically allocate the specified number of blocks of memory of the specified type. it is very much similar to malloc() but has two different points and these are:
- It initializes each block with a default value ‘0’.
- It has two parameters or arguments as compare to malloc().
- Syntax: ptr = (cast-type*)calloc(n, element-size);

Where, n is the no. of elements and element-size is the size of each element.

- Example: ptr = (float*) calloc(5, sizeof(float));

This statement allocates contiguous space in memory for 5 elements each with the size of the float.

Calloc()


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

Example of calloc( ):

```
void main()
{
	int i,n,sum=0;
	printf("Enter the number of elements:");
	scanf("%d",&n);
	int *ptr=(int *)calloc(n,sizeof(int));
	if(ptr==NULL)
	{
		printf("Memory not available.");
	}
	else
	{
		for(i=0;i<n;i++)
		{
			printf("Enter the element:");
			scanf("%d",ptr+i);
		}
		for(i=0;i<n;i++)
		{
			sum+=(*(ptr+i));
		}
		printf("Sum=%d",sum);
	}
	getch();
}
```

realloc( ):

- “realloc” or “re-allocation” method in C is used to dynamically change the size of memory of a previously allocated memory.
- In other words, 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.

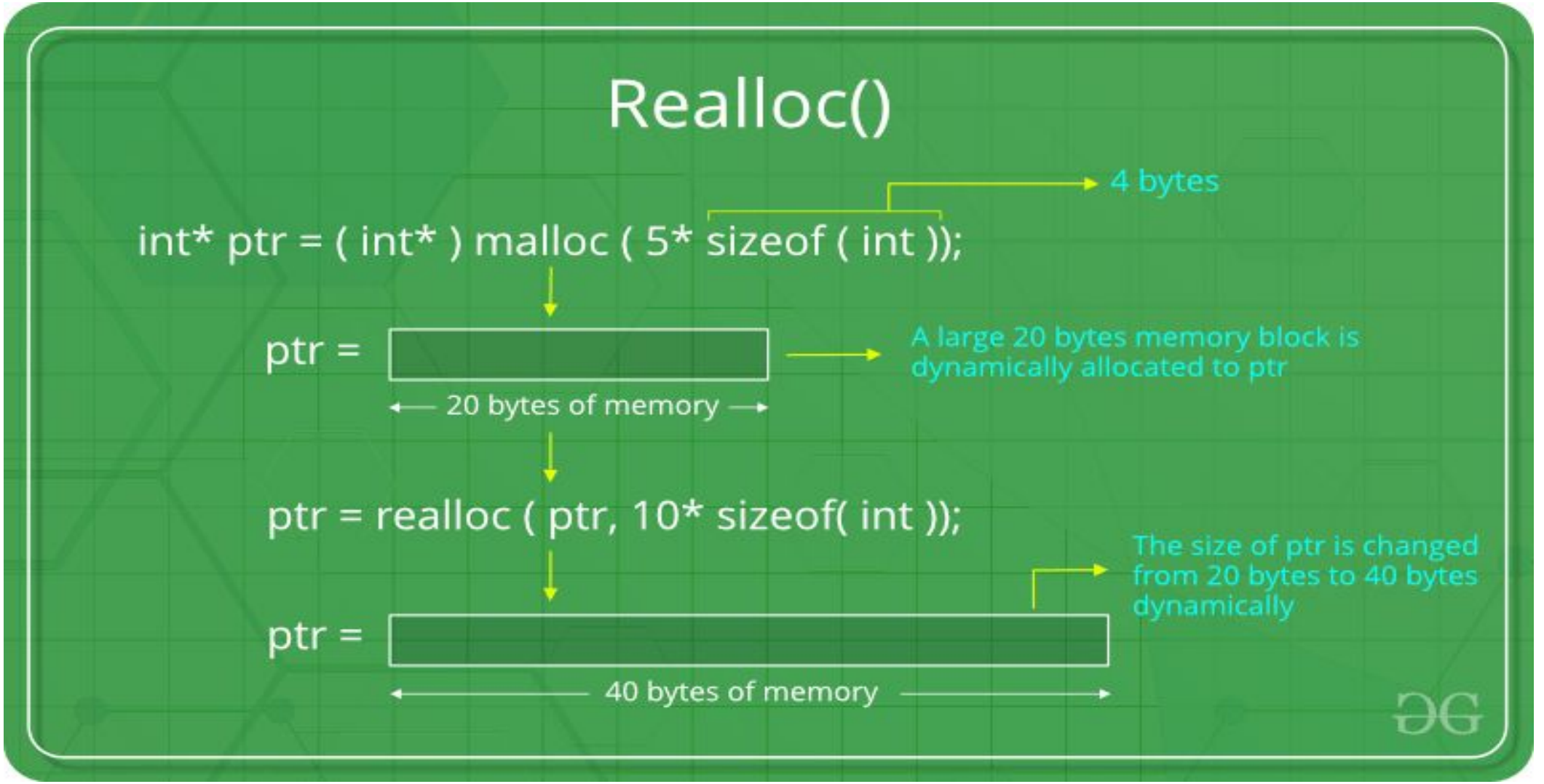
realloc( ):

Syntax: void *realloc(void *ptr, size_t newsize)

Where, ptr is the pointer to the previously allocated memory and newsize represents the new size of the memory.

-On failure, it returns the NULL pointer.

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

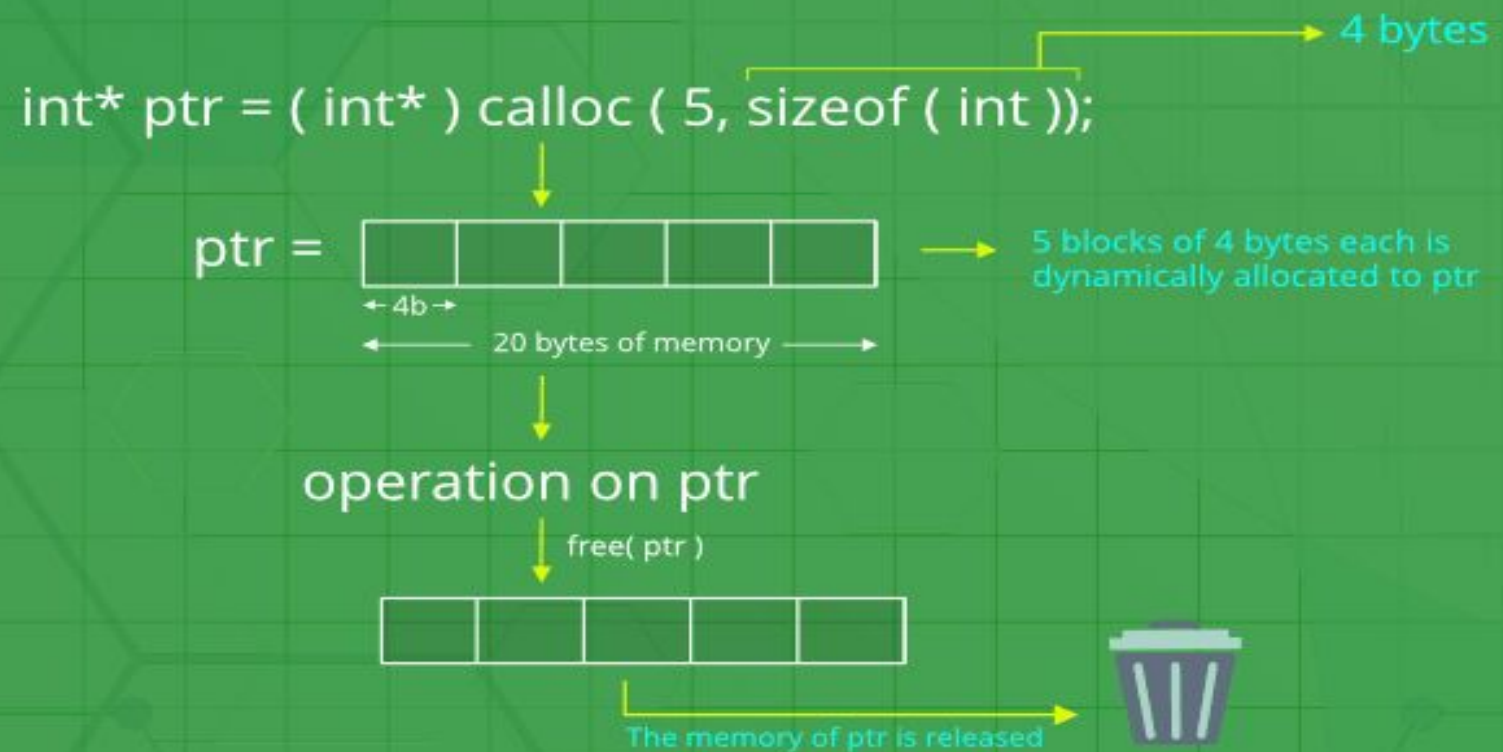
Example of realloc( ):

```
void main()
{
	int i;
	int *ptr=(int *)malloc(2*sizeof(int));
	if(ptr==NULL)
	{
		printf("Memory not available.");
	}
	else
	{
		printf("Enter the two numbers:");
		for(i=0;i<2;i++)
		{
			scanf("%d",ptr+i);
		}
	}
	ptr=(int *)realloc(ptr,4*sizeof(int));
	if(ptr==NULL)
	{
		printf("Memory not available.");
	}
	else
	{
		printf("Enter two more numbers:");
		for(i=2;i<4;i++)
		{
			scanf("%d",ptr+i);
		}
		for(i=0;i<4;i++)
		{
			printf("%d\t",*(ptr+i));
		}
	}
	getch();
}
```

free( ):

- “free” method in C is used to dynamically de-allocate the memory.
- The memory allocated using functions malloc() and calloc() is not de-allocated on their own.
- Hence the free() method is used, whenever the dynamic memory allocation takes place.
- It helps to reduce wastage of memory by freeing it.
- Syntax: free(ptr);

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 of free( ):

```
void main()
{
	int *ptr, *ptr1;
	int n, i;
	n = 5;
	printf("Enter number of elements: %d\n", n);
	ptr = (int*)malloc(n * sizeof(int));
	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");
	}
}
```