Data Structures with C
(18IS32)

By:

Dr. Kiran K. Tangod

Associate Professor

Information Science & Engineering

KLS, Gogte Institute of Technology, Belagavi

UNIT 1

- Pointers, Structures: Introduction to Pointers, Pointers and Arrays, Pointers to Pointers, Pointers to functions, Dynamic memory management in C (malloc(), calloc(), free() and realloc() functions).

18IS32

- Introduction to Structures, Declaration, Initialization, Accessing Structures, Internal implementation of Structures, Union and its Definition.

Pointers:

- The **pointer in C** language is a variable which stores the address of another variable or memory location. This variable can be of type int, char, array, function, or any other **pointer**.

```
// General syntax
datatype *var_name;
```

Declaring pointer in C

```
// An example pointer "ptr" that holds
// address of an integer variable or holds
// address of a memory whose value(s) can
// be accessed as integer values through "ptr“

int *ptr;
```

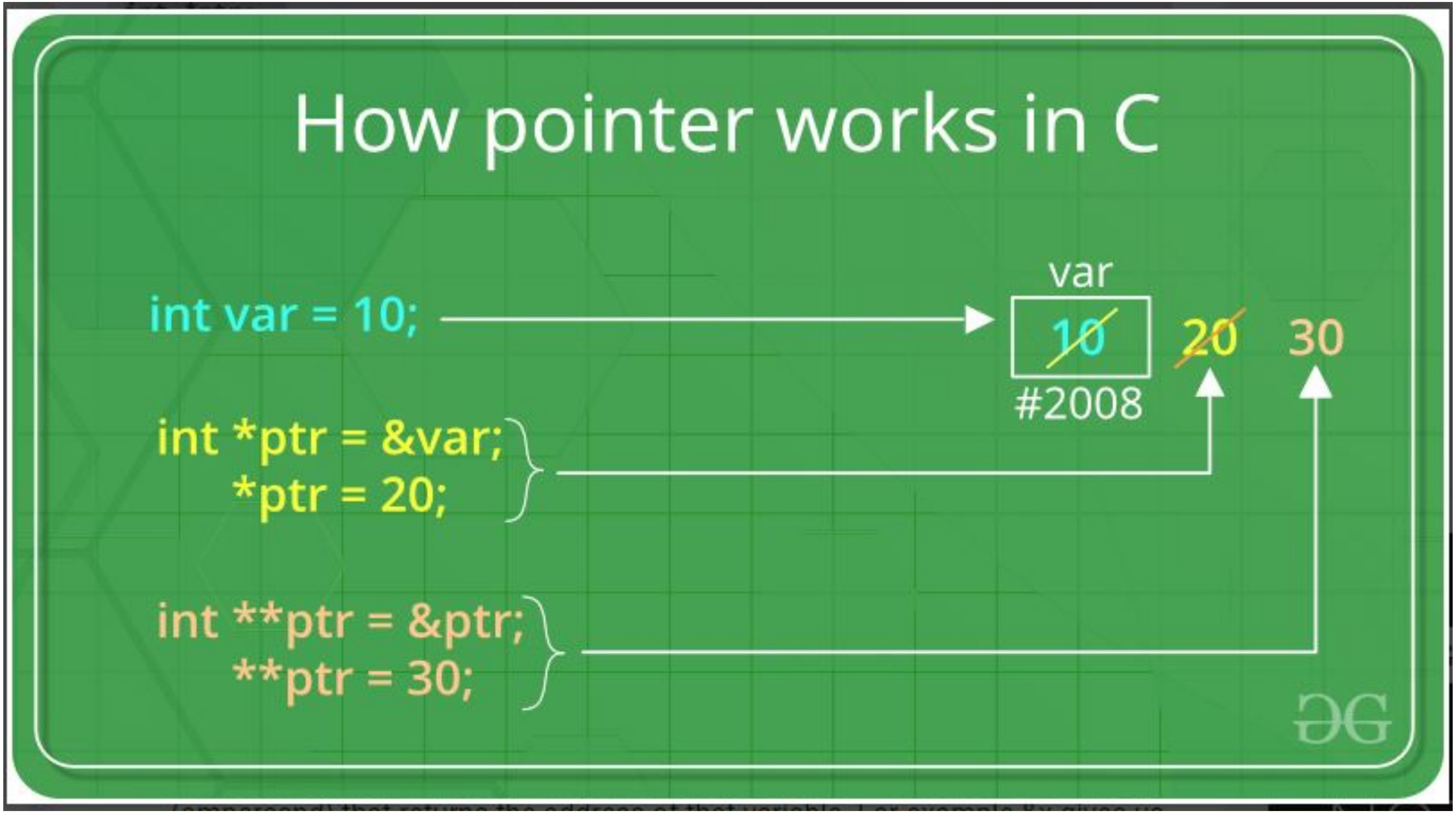
;

Initialize a pointer

- After declaring a pointer, we initialize it like standard variables with a variable address. If pointers are uninitialized and used in the program, the results are unpredictable and potentially disastrous.
- To get the address of a variable, we use the ampersand (&)operator, placed before the name of a variable whose address we need. Pointer initialization is done with the following syntax.

int *pointer = &variable

```
Example:
int n = 10;
int* p = &n; // Variable p of type pointer is
pointing to the address of
                         //the variable n of type
integer.
```


How pointer works in C
var
int var = 10;
~~10~~
~~20~~
30
#2008
int *ptr = &var;
*ptr = 20;
int **ptr = &ptr;
**ptr = 30;

Sample Program:

```
#include <stdio.h>
 int main() {
 int a=10; //variable declaration
 int *p;  //pointer variable declaration
 p=&a; //store address of variable a in pointer p
 printf("Address stored in a variable p is:%x\n",p);  //accessing the address
 printf("Value stored in a variable p is:%d\n",*p);    //accessing the value
 return 0;
}
```

Output:

```
Address stored in a variable p is:60ff08

Value stored in a variable p is:10
```

Operators used with pointers:

Operator	Meaning
*	Serves 2 purpose 1. Declaration of a pointer 2. Returns the value of the referenced variable
&	Serves only 1 purpose • Returns the address of a variable

Types of a pointer:

- Null pointer
- Void Pointer
- Wild pointer

Null pointer

- Null pointers can be created by assigning null value during the pointer declaration.
- Null pointers are useful when you do not have any address assigned to the pointer.
- A null pointer always contains value 0.

Program to illustrates the use of a null pointer:

```
#include <stdio.h>
int main()
{
        int *p = NULL;  //null pointer
        printf("The value inside variable p is:\n%x",p);
        return 0;
}
```

Output:

```
The value inside variable p is:
0
```

Void Pointer:

- Void pointer is also called as a generic pointer.
- It does not have any standard data type.
- A void pointer is created by using the keyword void.
- It can be used to store an address of any type of variable.

Program illustrates the use of a void pointer:

```
#include <stdio.h>
int main()
{
void *p = NULL;             //void pointer
printf("The size of pointer is:%d\n",sizeof(p));
return 0;
}
```

Output:

```
The size of pointer is:4
```

Wild pointer:

- A wild pointer is the one which is not initialized to anything.
- These types of pointers are not efficient because they may point to some unknown memory location.
- which may cause problems in our program and it may lead to crashing of the program.
- One should always be careful while working with wild pointers.

Sample Code:

```
#include <stdio.h>
int main()
{
int *p;          //wild pointer
printf("\n%d",*p);
return 0;
}
```

Output:

```
timeout: the monitored command dumped core
sh: line 1: 95298 Segmentation fault      timeout 10s main
```

Other types of pointers in ‘C' are as follows:

- Dangling pointer
- Complex pointer
- Near pointer
- Far pointer
- Huge pointer

Direct and Indirect Access Pointers

In C, there are two equivalent ways to access and manipulate a variable content

- **Direct access: we use directly the variable name**
- **Indirect access: we use a pointer to the variable**

Sample program:

```
#include <stdio.h> /* Declare and initialize an int variable */
int var = 1; /* Declare a pointer to int */
int *ptr; int main( void )
{
    /* Initialize ptr to point to var */
    ptr = &var;
     /* Access var directly and indirectly */
    printf("\nDirect access, var = %d", var);
    printf("\nIndirect access, var = %d", *ptr); /* Display the address of var
    two ways */
    printf("\n\nThe address of var = %d", &var);
    printf("\nThe address of var = %d\n", ptr); /*change the content of var
    through the pointer*/
    *ptr=48; printf("\nIndirect access, var = %d", *ptr);
     return 0;
}
```

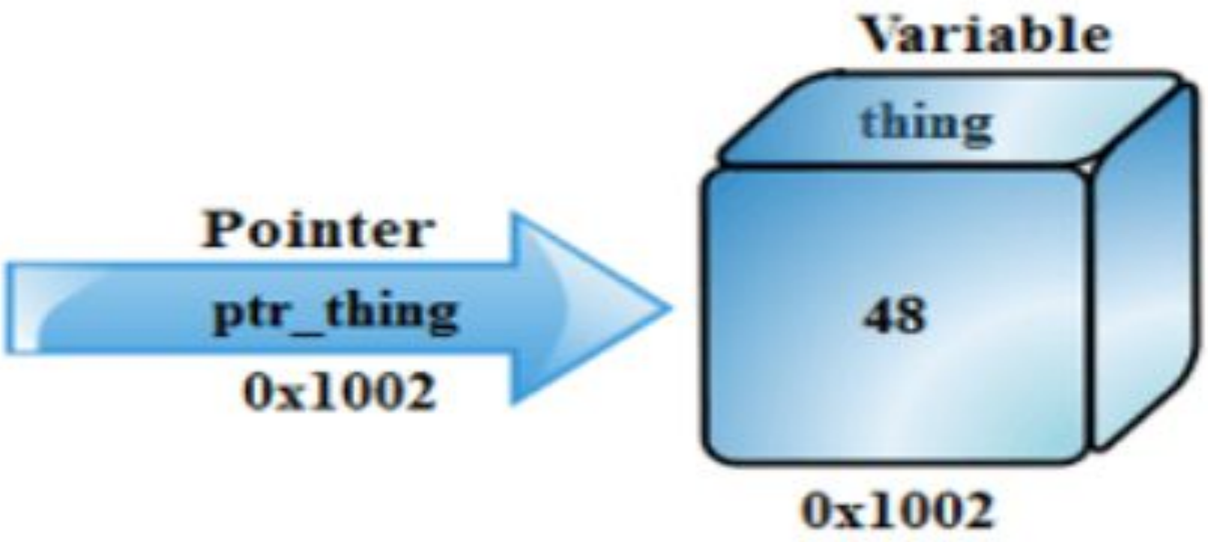
Sample Output:

```
Direct access, var = 1
Indirect access, var = 1


The address of var = 4202496
The address of var = 4202496


Indirect access, var = 48
```

Pointers Arithmetic

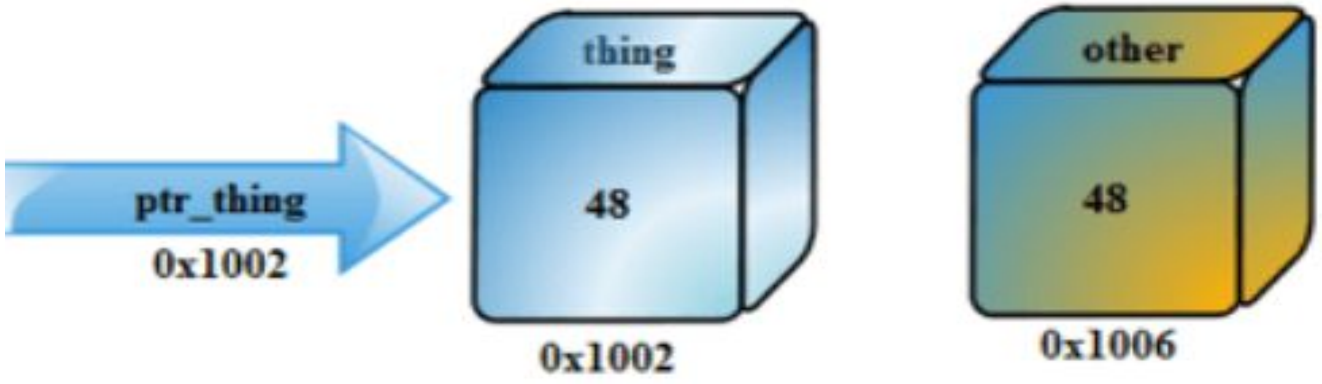
- The pointer operations are summarized in the following figure

A) ptr_thing = &thing;



Assigns thing's address to ptr_thing

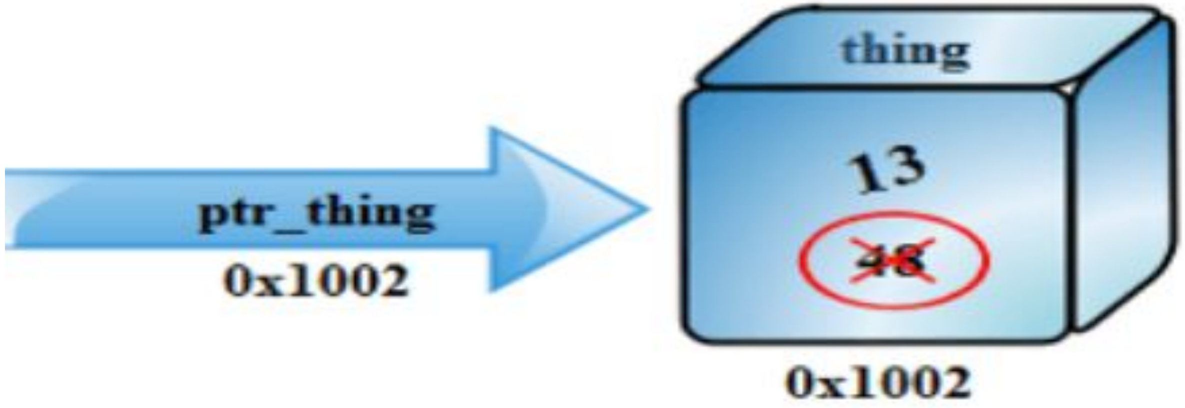
B) other = *ptr_thing;



Assigns to other the value

pointed by ptr_thing

C) *ptr_thing=13;



Assigns new value to

the pointed variable

Priority operation (precedence):

- When working with pointers, we must observe the following priority rules

✔ The operators * and & have the same priority as the unary operators.

✔ In the same expression, the unary operators *, & are evaluated from right to left.

✔ If a P pointer points to an X variable, then * P can be used wherever X can be written.

The following expressions are equivalent:

```
int X =10
int *P = &X;
```

For the above code, below expressions are true

Expression	Equivalent Expression
Y=*P+1	Y=X+1
*P=*P+10	X=X+10
*P+=2	X+=2
++*P	++X
(*P)++	X++

(*P)++ □ X++

parentheses are needed in the above statement: as the unary operators * and ++ are evaluated from right to left, without the parentheses the pointer P would be incremented, not the object on which P points.

Pointers and Arrays :

- Traditionally, the array elements are accessed using its index.
- An alternate method of accessing is possible using pointers.
- Pointers make it easy to access each array element.

```
#include <stdio.h>
int main()
{
    int a[5]={1,2,3,4,5};   //array initialization
    int *p;     //pointer declaration
               /*the ptr points to the first element of the
array*/

    p=a; /*We can also type simply ptr==&a[0] */

    printf("Printing the array elements using pointer\n");
    for(int i=0;i<5;i++)    //loop for traversing array elem
ents
    {
            printf("\n%x",*p);  //printing array element
s
            p++;    //incrementing to the next element,
you can also write p=p+1
    }
    return 0;
}
```

```
#include <stdio.h>
int main()
{
    int a[5]={1,2,3,4,5}; //array initialization
    int *p; //pointer declaration
    //the “p” points to the first element of the array
    p=a; /*We can also type simply p==&a[0] */
    printf("Printing the array elements using pointer\n");
    for(int i=0;i<5;i++) //loop for traversing array elements
    {
      printf("\n%x",*p); //printing array elements
      p++;
      //incrementing to the next element, you can also write p=p+1
    }
    return 0;
}
```

Output:

1

2

3

4

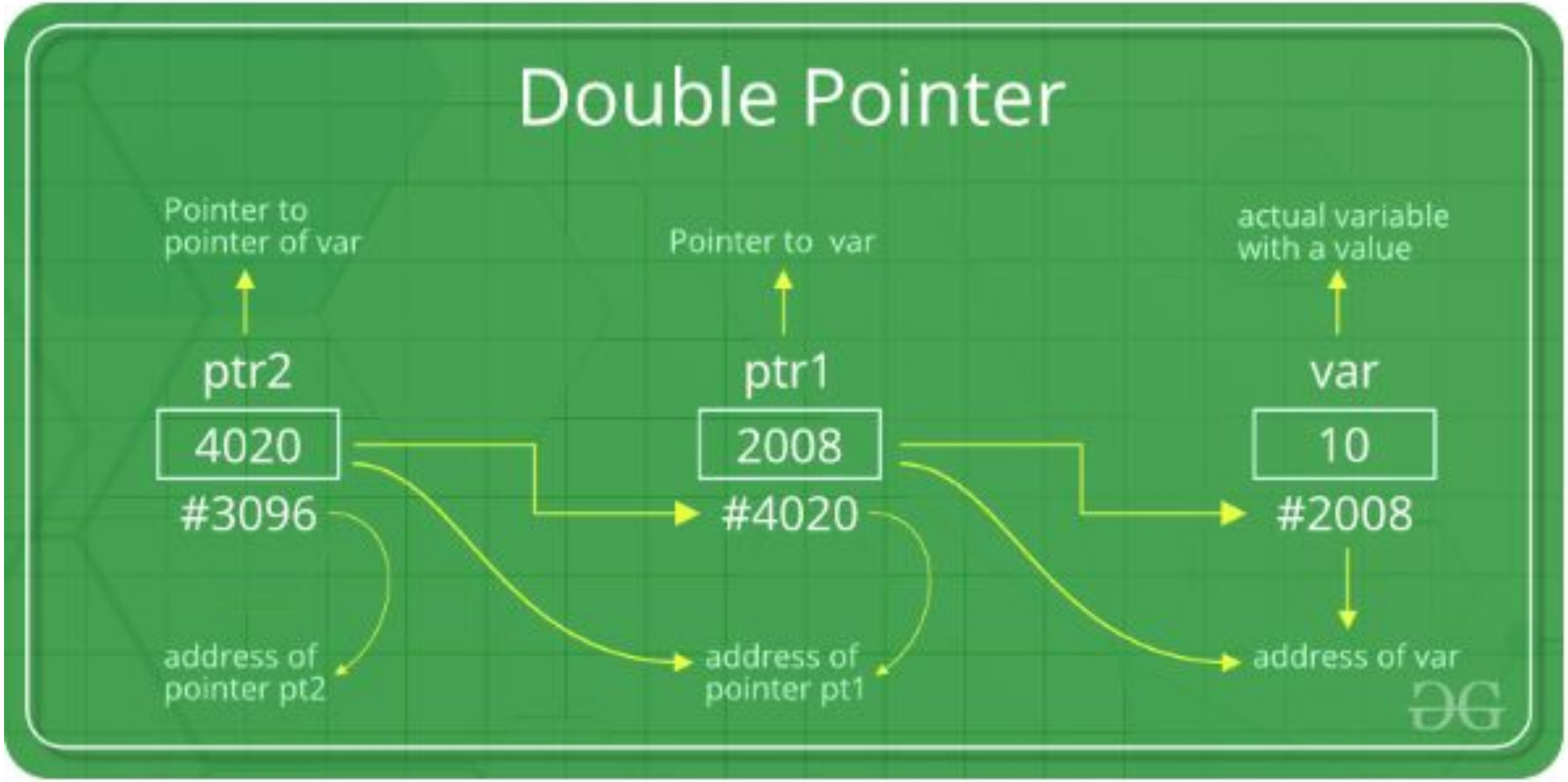
5

Pointer to Pointer:

A pointer to a pointer is a form of multiple indirection, or a chain of pointers. Normally, a pointer contains the address of a variable.
When we define a pointer to a pointer, the first pointer contains the address of the second pointer, which points to the location that contains the actual value as shown below.

A variable that is a pointer to a pointer must be declared as such.
This is done by placing an additional asterisk in front of its name.

Example:


Double Pointer
Pointer to
pointer of var
Pointer to var
actual variable
with a value
ptr2
ptr1
var
4020
2008
10
#3096
#4020
#2008
address of
pointer pt2
address of
pointer pt1
address of var

```
int **var;
```

```
#include <stdio.h>
int main ()
{
   int var; int *ptr; int **pptr;
   var = 3000;
   /* take the address of var */ ptr = &var;
   /* take the address of ptr using address of operator &
   */ pptr = &ptr; /* take the value using pptr */
   printf("Value of var = %d\n", var );
   printf("Value available at *ptr = %d\n", *ptr );
   printf("Value available at **pptr = %d\n", **pptr);
   return 0;
}
```

Output:

```
Value of var = 3000

Value available at *ptr = 3000

Value available at **pptr = 3000
```

Pointer to function:

- A **function pointer** can point to a specific **function** when it is assigned the name of that **function.**

  int sum(int, int);

  int (*s)(int, int);

  s = ∑

  Here “s” is a **pointer** to a **function** ”sum” .

- Now sum can be called using **function pointer** s along with providing the required **argument** values.

Example code:

```
#include <stdio.h>
// A normal function with an int parameter
// and void return type
void fun(int a)
{
	printf("Value of a is %d\n", a);
}

int main()
{
// fun_ptr is a pointer to function fun()
	void (*fun_ptr)(int) = &fun;


	/* The above line is equivalent of
		following two lines
	void (*fun_ptr)(int);
```

Why do we need an extra bracket around function pointers like fun_ptr in above example?

If we remove bracket, then the expression

“void (*fun_ptr)(int)” becomes “void *fun_ptr(int)”

which is declaration of a function that returns void pointer.

Following are some interesting facts about function pointers.

1) Unlike normal pointers, a function pointer points to code, not data. Typically a function pointer stores the start of executable code.

2) Unlike normal pointers, we do not allocate de-allocate memory using function pointers.

3) A function's name can also be used to get functions' address.

```
int sum(int, int);
int (*s)(int, int);
s = sum;      //  same as s=&sum
```

4) Like normal pointers, we can have an array of function pointers.

5) Function pointer can be used in place of switch case

```
#include <stdio.h>
void add(int a, int b)
{
	printf("Addition is %d\n", a+b);
}
void subtract(int a, int b)
{
	printf("Subtraction is %d\n", a-b);
}
void multiply(int a, int b)
{
	printf("Multiplication is %d\n", a*b);
}

int main()
{
	// fun_ptr_arr is an array of function pointers
	void (*fun_ptr_arr[])(int, int) = {add, subtract, multiply};
```

Dynamic memory management in C:

- **Dynamic Memory Allocation** can be defined as a procedure in which the size of a data structure (like Array) is changed during the runtime.
- There are 4 library functions provided by C defined under **<stdlib.h>** header file to facilitate dynamic memory allocation in C programming. They are:
  1. malloc()
  2. calloc()
  3. free()
  4. realloc()

malloc() function:

- **“malloc”** or **“memory allocation”** method in C is used to dynamically allocate a single large block of memory with the specified size. It returns a pointer of type void which can be cast into a pointer of any form. It initializes each block with default garbage value.
- Syntax:

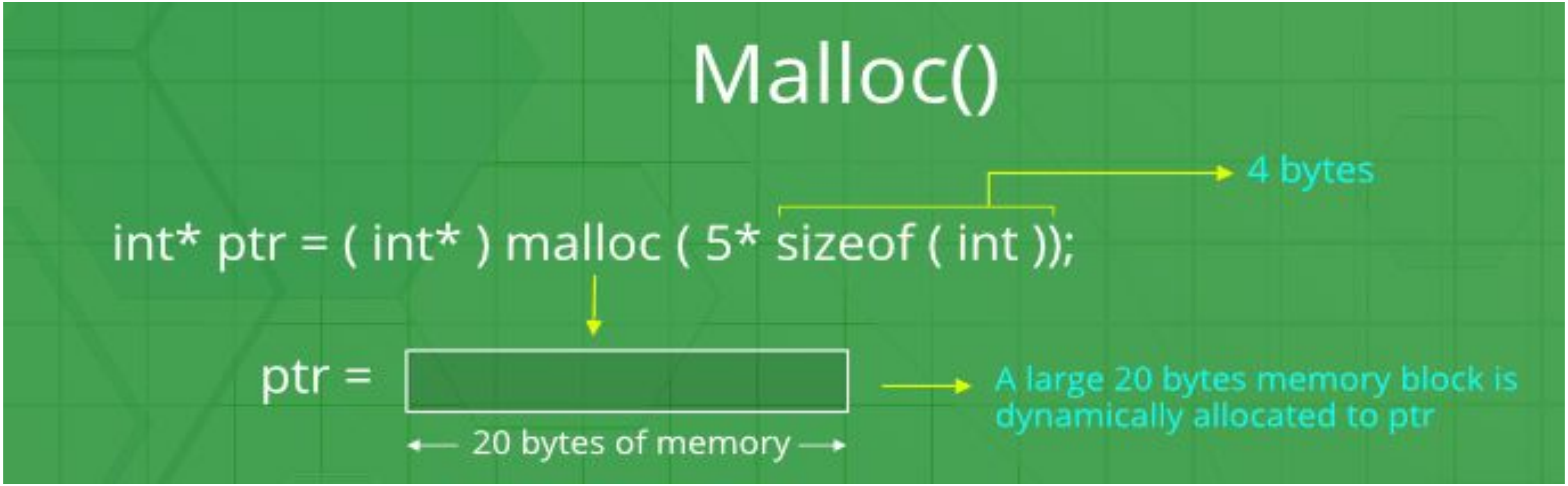
```
ptr = (cast-type*) malloc(byte-size);
```

Example:

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

- *Since the size of int is 4 bytes, this statement will allocate 400 bytes of memory. And, the pointer ptr holds the address of the first byte in the allocated memory.*

Example:



Important Note: If space is insufficient, allocation fails and returns a NULL pointer.

calloc() function:

- **“calloc”** or **“contiguous allocation”** method in C is used to dynamically allocate the specified number of blocks of memory of the specified type. It initializes each block with a default value ‘0’.
- Syntax:

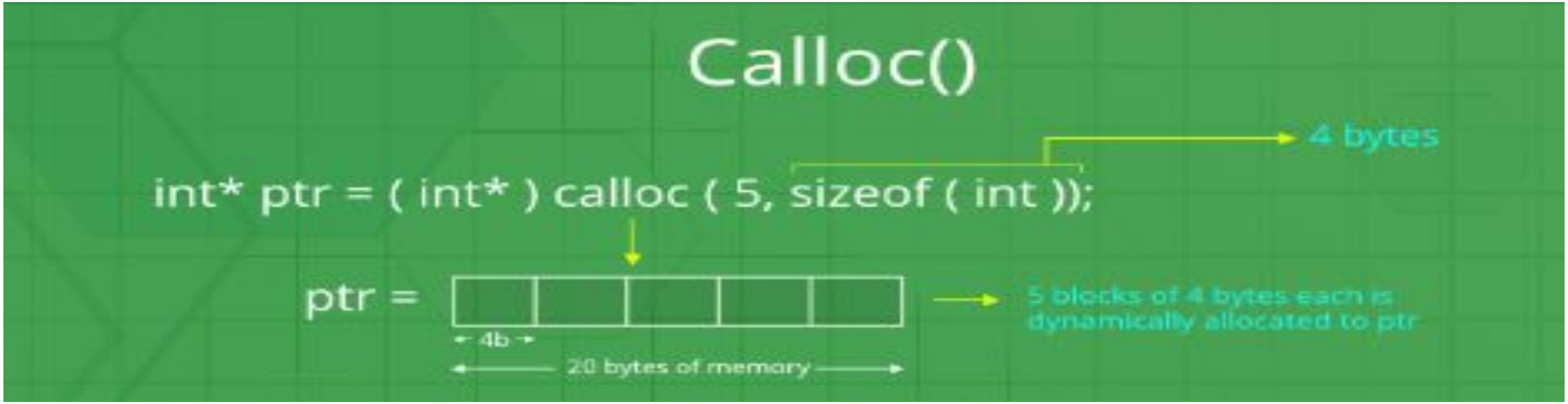
```
ptr = (cast-type*)calloc(n, element-size);
```

- ***Example:***

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

- *This statement allocates contiguous space in memory for 25 elements each with the size of the float.*

Example:



Important Note: If space is insufficient, allocation fails and returns a NULL pointer.

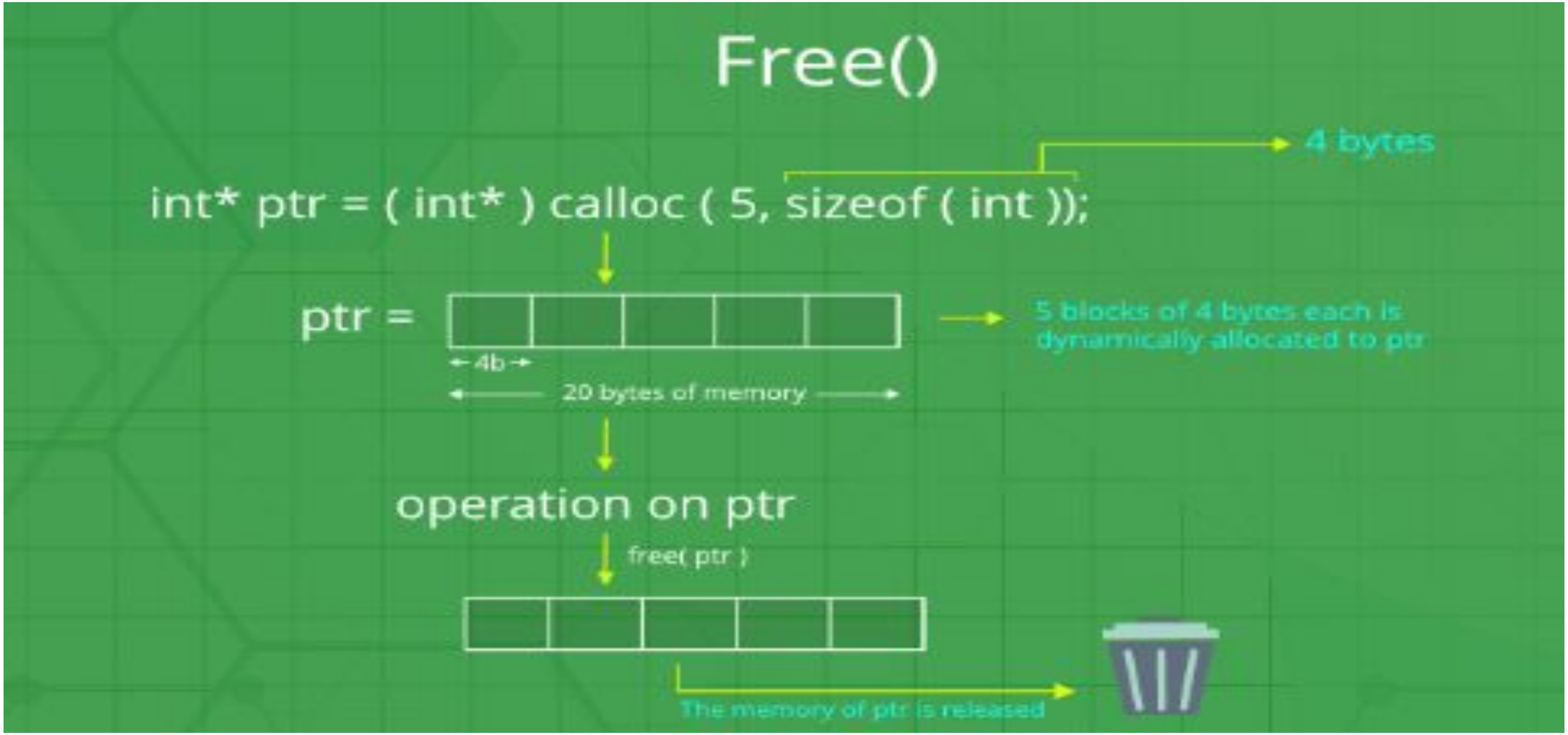
free() function:

- **“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(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

realloc() function:

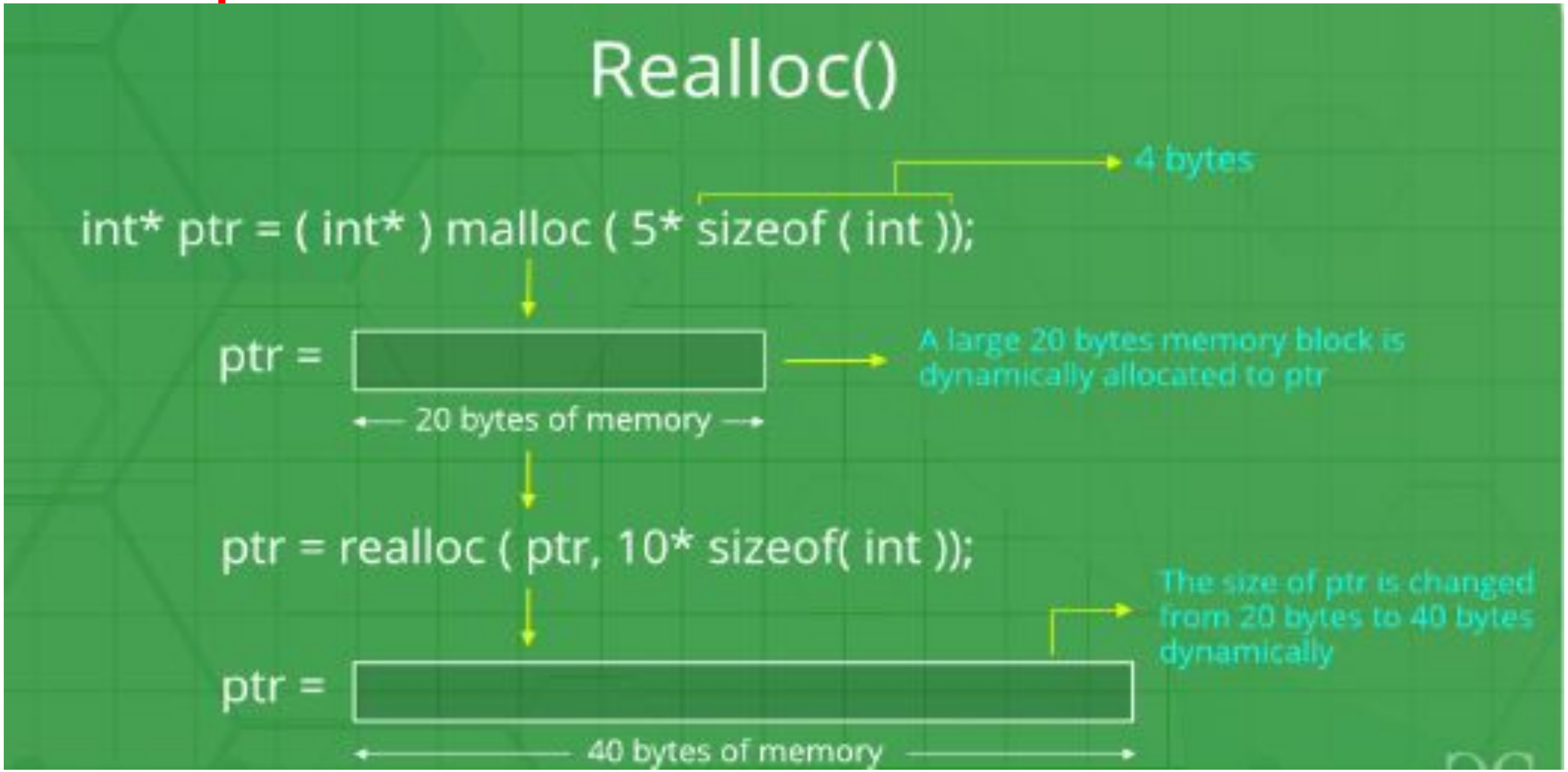
- **“realloc”** or **“re-allocation”** method in C is used to dynamically change the memory allocation of a previously allocated memory. In other words, if the memory previously allocated with the help of malloc or calloc is insufficient, realloc can be used to **dynamically re-allocate memory**. re-allocation of memory maintains the already present value and new blocks will be initialized with default garbage value.

Example:

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

- `where ptr is reallocated with new size 'newSize'.`

Example:



Important Note: If space is insufficient, allocation fails and returns a NULL pointer.

Structure in C

What is a structure in C ?

- In C programming, a structure is a collection of variables called as fields or members of different types under a single name.
- Structures are used to store related information under a single name.
- Examples:
    - Information about student
    - Information about employee
    - Information about a commodity in a super market

How to define structures?

- Structure must be defined before you create structure variables
- Once the structure is defined a User Defined Datatype(UDT) gets created
- This UDT can be used to create variables
- ***struct*** keyword is used to define structure

Syntax of struct

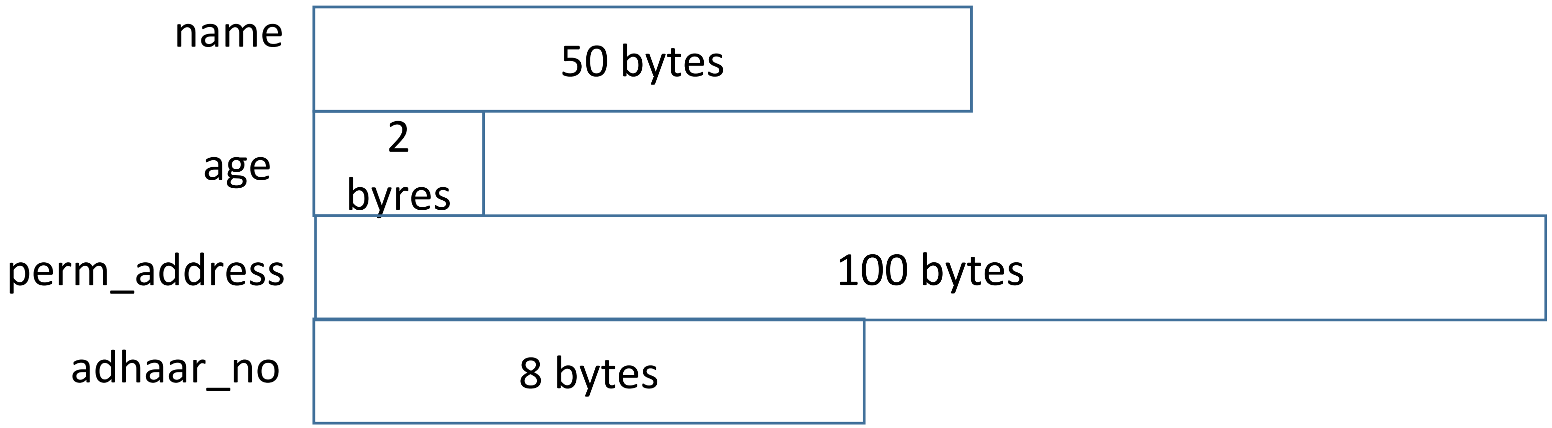
```
struct tag_name {
        dataType member1;
        dataType member2;
        .....
        ......
};
```

Example:

```
struct person{
    char name[50];
    int age;
    char  perm_address[100];
    long int adhaar_no;
};
```

Note: This definition is used to create a template but not a variable of person type.

Representation in memory



Total memory requirement for variable of person type would be 160 Bytes (minimum) excluding padding bytes if any

Important points to note while defining struct

- The template is terminated with a semicolon
- Entire definition of structure is considered as single statement
- Each member must be declared with name and datatype independently
- Tag name used for defining structure must follow rules for writing a valid variable name in C
- Tag name must be used for declaring structure variables in the program

Arrays Vs Structures

Arrays

1. An array is collection of similar data elements.
2. An array is derived data type.
3. Array behaves as built in data type.
4. Array elements are stored in continuous memory locations

Structures

1. Structure is collection of different type of data elements.
2. Structure is Programmer/User defined data type.
3. First structure is to be defined before it can be used to create variable of structure type.
4. Structure members may not be stored in continuous memory locations because of *padding bytes*

Declaring Structure variables

- Following elements must be used while declaring struct variables

1. The keyword struct
2. The structure tag name
3. List of variable names separated by a comma
4. Terminating semicolon

Example: Declaring Structure variables (style 1)

```
struct   person p1,p2,p3,p4;
```

Here p1,p2,p3 and p4 are variables of struct person type

Declaring Structure variables (style 2)

```
struct person{
    char name[50];
    int age;
    char  perm_address[100];
    long int adhaar_no;
}p1,p2,p3,p4;
```

Note: in both styles 4 variables are created and each variable will have 4 independent fields to store data elements.

Declaring Structure variables (style 3)

```
struct {
    char name[50];
    int age;
    char  perm_address[100];
    long int adhaar_no;
}p1,p2,p3,p4;
```

- In this style of declaration tag name is not used
- This style is not recommended, because without tag name it is not possible to create variables in future declarations.
- Variables can be created only at one place i.e. at the end of structure decleration.

typedef

- typedef keyword can be used to declare structure variables in more easy way.

```
typedef  struct
        {
            type member1;
            type member2;
            …
        }type_name;
type_name variable1, variable2,……;
```

type_name is type definition name and not a variable

It is not possible to create variables using typedef declaration

Example:

```
typedef struct {
     char name[50];
     int age;
     char  perm_address[100];
     long int adhaar_no;
}person;
person p1,p2,p3,p4; //creates four variables of person type
person p[100];//creates array of 100 elements of person type
```

Accessing structure members:

- There are two types of operators used for accessing members of a structure.

1. **.** (dot) Member operator
2. -> (arrow) Structure pointer operator

Example:

If it is required to access persons age from variable p1 and assign 60 to it then following statement is written:

p1.age=60

Example

Initialization of structure members:

```
struct Point
{
int x = 0; // COMPILER ERROR: cannot initialize members here
int y = 0; // COMPILER ERROR: cannot initialize members here
};
```

The reason for above error is simple, when a datatype is declared, no memory is allocated for it. Memory is allocated only when variables are created.

Structure members **can be** initialized using curly braces ‘{}’. For example, following is a valid initialization.

```
struct Point
{
    int x, y;
};

int main()
{
// A valid initialization. member x gets value 0 and y
// gets value 1. The order of declaration is followed.
struct Point p1 = {0, 1};
}
```

Rules for initialization of structure variables:

- Initialization of individual members within the structure template is not allowed
- The order of values enclosed in braces must match the order of members in the structure definition
- It is permitted to have partial initialization. In such cases initialization of members is done from left to right. Uninitialized members should be only at the end of the list.
- The uninitialized members will be assigned default values as follows:
  - Zeros for integer and floating point members
  - ‘\0’ for character and string type members

Array of structures :

Like other primitive data types, we can create an array of structures.

```
#include<stdio.h>



struct Point
{
int x, y;
};
```

Thank You!!!