Unit=4

File management in c

Random access file function

fseek(),

ftell()

rewind()

- fseek(): It is used to move the reading control to different positions using fseek function.

Following is the declaration for fseek() function.

```
int fseek(FILE *pointer, long int offset, current position)
```

Parameters

- **file pointer** − This is the pointer to a FILE object that identifies the stream.
- **offset** − This is the number of bytes to offset from whence.
- **Current_possition** − This is the position from where offset is added. It is specified by one of the following constants −

Sr.No.	Constant & Description
1	**SEEK_SET** Beginning of file
2	**SEEK_CUR** Current position of the file pointer
3	**SEEK_END** End of file

Example

The following example shows the usage of fseek() function.

```
#include <stdio.h>

int main () {
   FILE *fp;

   fp = fopen("file.txt","w+");
   fputs("This is tutorialspoint.com", fp);

   fseek( fp, 7, SEEK_SET );
   fputs(" C Programming Language", fp);
   fclose(fp);

   return(0);
}
```

-
- ftell(): It tells the byte location of current position of cursor in file pointer

Following is the declaration for ftell() function.

```
long int ftell(FILE *stream)
```

Parameters

- **stream** − This is the pointer to a FILE object that identifies the stream.

Return Value

This function returns the current value of the position indicator. If an error occurs, -1L is returned, and the global variable errno is set to a positive value.

Example

The following example shows the usage of ftell() function.

```
#include <stdio.h>

void main ()
{
   FILE *fp;
   int len;
```

```
   fp = fopen("file.txt", "r");
   if( fp == NULL )  {
      perror ("Error opening file");
      return(-1);
   }
   fseek(fp, 0, SEEK_END);

   len = ftell(fp);
   fclose(fp);

   printf("Total size of file.txt = %d bytes\n", len);

   getch();
}
```

- .
- rewind(): It moves the control to beginning of the file.

Following is the declaration for rewind() function.

```
void rewind(FILE *stream)
```

Parameters

- **stream** − This is the pointer to a FILE object that identifies the stream.

Return Value

This function does not return any value.

Example

The following example shows the usage of rewind() function.

Live Demo

```
#include <stdio.h>

int main () {
   char str[] = "This is tutorialspoint.com";
   FILE *fp;
   int ch;

   /* First let's write some content in the file */
   fp = fopen( "file.txt" , "w" );
   fwrite(str , 1 , sizeof(str) , fp );
```

```
   fclose(fp);

   fp = fopen( "file.txt" , "r" );
   while(1) {
      ch = fgetc(fp);
      if( feof(fp) ) {
         break ;
      }
      printf("%c", ch);
   }
   rewind(fp);
   printf("\n");
   while(1) {
      ch = fgetc(fp);
      if( feof(fp) ) {
         break ;
      }
      printf("%c", ch);

   }
   fclose(fp);

   return(0);
}
```

Why files are needed?

When the program is terminated, the entire data is lost in C progr High level file I/O functions can be categorized as:

1. Text file
2. Binary file

File Operations

1. Creating a new file
2. Opening an existing file
3. Reading from and writing information to a file
4. Closing a file

Working with file

While working with file, you need to declare a pointer of type file. This declaration is needed for communication between file and program.

FILE *ptr;

fopen() and fclose() functions

Opening a file

Opening a file is performed using library function fopen(). The syntax for opening a file in standard I/O is:

ptr=fopen("fileopen","mode")

For Example:
SYNTAX:-

```
fopen("E:\\cprogram\program.txt","w");

/* ---------------------------------------------------------- */
 E:\\cprogram\program.txt is the location to create file.
 "w" represents the mode for writing.
/* ---------------------------------------------------------- */
```

Here, the program.txt file is opened for writing mode.

Opening Modes in Standard I/O

File Mode	Meaning of Mode	During Inexistence of file
r	Open for reading.	If the file does not exist, fopen() returns NULL.
w	Open for writing.	If the file exists, its contents are overwritten. If the file does not exist, it will

Opening Modes in Standard I/O

File Mode	Meaning of Mode	During Inexistence of file
		be created.
a	Open for append. i.e, Data is added to end of file.	If the file does not exists, it will be created.
r+	Open for both reading and writing.	If the file does not exist, fopen() returns NULL.
w+	Open for both reading and writing.	If the file exists, its contents are overwritten. If the file does not exist, it will be created.
a+	Open for both reading and appending.	If the file does not exists, it will be created.

Closing a File

The file should be closed after reading/writing of a file. Closing a file is performed using library function fclose().

SYNTAX:-

fclose(ptr); //ptr is the file pointer associated with file to be closed.

The Functions fprintf() and fscanf() functions.

Important is that the functions fprintf() and fscanf() are the file version of printf() and fscanf().

The only difference while using fprintf() and fscanf() is that, the first argument is a pointer to the structure FILE

Character input /output functions ,String input and output functions

Examples:-

Writing to a file

```
#include <stdio.h>
int main()
{
   int n;
   FILE *fptr;
   fptr=fopen("C:\\program.txt","w");
   if(fptr==NULL){
      printf("Error!");
      exit(1);
   }
   printf("Enter n: ");
   scanf("%d",&n);
   fprintf(fptr,"%d",n);
   fclose(fptr);
   return 0;
}
```

This program takes the number from user and stores in file.

After you compile and run this program, you can see a text file program.txt created in C drive of your computer.

When you open that file, you can see the integer you entered.

Similarly, fscanf() can be used to read data from file.

Reading from file

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

```
int main()
{
   int n;
   FILE *fptr;
   if ((fptr=fopen("C:\\program.txt","r"))==NULL){
       printf("Error! opening file");
       exit(1);         /* Program exits if file pointer returns NULL. */
   }
   fscanf(fptr,"%d",&n);
   printf("Value of n=%d",n);
   fclose(fptr);
   return 0;
}
```

fgetchar(), fputc() etc. can be used in similar way.

The C library function **int fputc(int char, FILE *stream)** writes a character (an unsigned char) specified by the argument **char** to the specified stream and advances the position indicator for the stream.

Declaration

Following is the declaration for fputc() function.

SYNTAX:-

```
int fputc(int char, FILE *stream)
```

Parameters

- **char** -- This is character to be written. This is passed as its int promotion.
- **stream** -- This is the pointer to a FILE object that identifies the stream where the character is to be written.

Return Value

If there are no errors, the same character that has been written is returned. If an error occurs, EOF is returned and the error indicator is set.

Example

The following example shows the usage of **fputc() function.**

```
#include <stdio.h>


int main ()
{
   FILE *fp;
   int ch;


   fp = fopen("file.txt", "w+");
   for( ch = 33 ; ch <= 100; ch++ )
   {
      fputc(ch, fp);
   }
   fclose(fp);


   return(0);
}
```

Let us compile and run the above program, this will create a file **file.txt** in the current directory which will have following content:

```
!"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcd
```

Now let's the content of the above file using the following program:

```
#include <stdio.h>


int main ()
{
   FILE *fp;
   int c;


   fp = fopen("file.txt","r");
   while(1)
   {
      c = fgetc(fp);
      if( feof(fp) )
      {
         break ;
      }
      printf("%c", c);
   }
   fclose(fp);
   return(0);
}
```

Binary Files

Depending upon the way file is opened for processing, a file is classified into text file and binary file.

If a large amount of numerical data it to be stored, text mode will be insufficient. In such case binary file is used.

Working of binary files is similar to text files with few differences in opening modes, reading from file and writing to file.

Opening modes of binary files

Opening modes of binary files are rb, rb+, wb, wb+,ab and ab+.

The only difference between opening modes of text and binary files is that, b is appended to indicate that, it is binary file.

Reading and writing of a binary file.

Functions **fread() and fwrite()** are used for reading from and writing to a file on the disk respectively in case of binary files.

Function fwrite() takes four arguments, address of data to be written in disk, size of data to be written in disk, number of such type of data and pointer to the file where you want

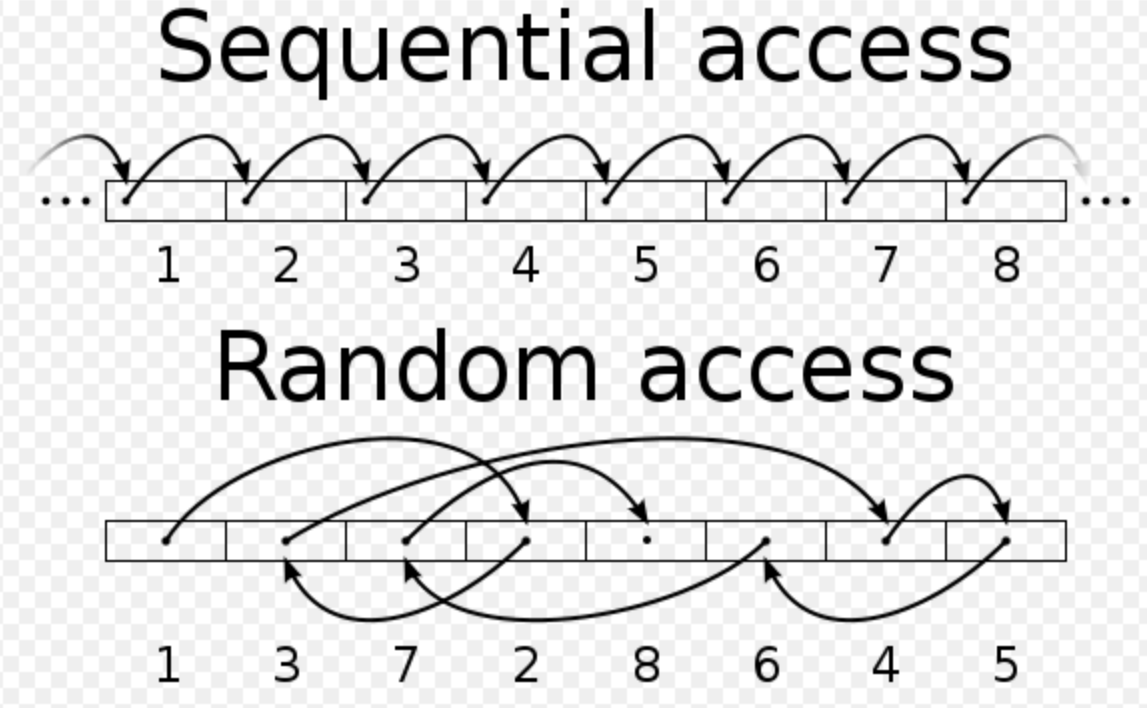
to write.

Syntax:-

fwrite(address_data,size_data,numbers_data,pointer_to_file);

Function fread() also take 4 arguments similar to fwrite() function as above.

Sequential and Random Access File Handling in C



Basic C- Files Examples:-

1. **Write a C program to read name and marks of n number of students from user and store them in a file.**

```
#include <stdio.h>
#include<conio.h>
int main(){
   char name[50];
   int marks,i,n;
   clrscr();
   printf("Enter number of students: ");
   scanf("%d",&n);
   FILE *fptr;
   fptr=(fopen("C:\\student.txt","w"));
   if(fptr==NULL){
      printf("Error!");
      exit(1);
```

```
    }
    for(i=0;i<n;++i)
    {
      printf("For student%d\nEnter name: ",i+1);
      scanf("%s",name);
      printf("Enter marks: ");
      scanf("%d",&marks);
      fprintf(fptr,"\nName: %s \nMarks=%d \n",name,marks);
    }
    fclose(fptr);
    getch();
    return 0;
}
```

2. Write a C program to read name and marks of n number of students from user and store them in a file. If the file previously exits, add the information of n students.

```
#include <stdio.h>
#include<conio.h>
int main(){
   char name[50];
   int marks,i,n;
   clrscr();
   printf("Enter number of students: ");
   scanf("%d",&n);
   FILE *fptr;
   fptr=(fopen("C:\\student.txt","a"));
   if(fptr==NULL){
       printf("Error!");
       exit(1);
   }
   for(i=0;i<n;++i)
   {
      printf("For student%d\nEnter name: ",i+1);
      scanf("%s",name);
      printf("Enter marks: ");
      scanf("%d",&marks);
      fprintf(fptr,"\nName: %s \nMarks=%d \n",name,marks);
```

```
    }
    fclose(fptr);
    getch();
    return 0;
}
```

3. **Write a C program to write all the members of an array of strcures to a file using fwrite(). Read the array from the file and display on the screen.**

```
#include<stdio.h>
#include<conio.h>
struct s
{
char name[50];
int height;
};
int main(){
    struct s a[5],b[5];
    FILE *fptr;
    int i;
clrscr();
    fptr=fopen("file.txt","wb");
    for(i=0;i<5;++i)
    {
        fflush(stdin);
        printf("Enter name: ");
        gets(a[i].name);
        printf("Enter height: ");
        scanf("%d",&a[i].height);
    }
    fwrite(a,sizeof(a),1,fptr);
    fclose(fptr);
    fptr=fopen("file.txt","rb");
    fread(b,sizeof(b),1,fptr);
    for(i=0;i<5;++i)
    {    printf("Name: %s\nHeight: %d",b[i].name,b[i].height);
    }
```

```
    fclose(fptr);
getch();
}
```

preprocessor

a C Preprocessor is instructs the compiler to do required pre-processing before the actual compilation.

All preprocessor commands begin with a hash symbol (#).

a preprocessor directive should begin in the first column.

Preprocessor is a program which source code preprocessed.

The following section lists down all the important preprocessor directives −

Sr.No.	Directive & Description
1	**#define** Substitutes a preprocessor macro.
2	**#include** Inserts a particular header from another file.
3	**#undef** Undefines a preprocessor macro.
4	**#ifdef** Returns true if this macro is defined.
5	**#ifndef** Returns true if this macro is not defined.
6	**#if** Tests if a compile time condition is true.
7	**#else**

	The alternative for #if.
8	**#elif** #else and #if in one statement.
9	**#endif** Ends preprocessor conditional.
10	**#error** Prints error message on stderr.
11	Issues special commands to the compiler, using a standardized method.

Preprocessors Examples

Analyze the following examples to understand various directives.

```
#define MAX_ARRAY_LENGTH 20
```

This directive tells the CPP to replace instances of MAX_ARRAY_LENGTH with 20. Use *#define* for constants to increase readability.

Types of preprocessor directives:

C language supports different preprocessor statements like

- Macro substitution
- File inclusions
- Conditional inclusion/Compilation
- operators

Rules in writing preprocessor directives

We must follow certain rules while writing preprocessor statement, Some of these rules are

- All preprocessor directives must be started with # symbol
- Every preprocessor statement may be started from the first column (Optional)
- There shouldn't be any space between # and directive
- A preprocessor statement must not be terminated with a semicolon
- Multiple preprocessor statements must not be written in a single line

- Preprocessor statements can be written any where within the block, function or outside any function

Macro substitution

Single instruction that expands automatically into a set of instructions to perform particular task.

Macro substitution has a name and replacement text, defined with #define directive.

The preprocessor simply replaces the name of macro with replacement text from the place where the macro is defined in the source code.

Syntax:

```
#define name replacement text
```

```
/* Prog.c */

#include<stdio.h>

#define NUM int

#define OUT printf

NUM main()

{

 NUM a,b,c;

 a=45;

 b=25;

 c=a+b;

 OUT("Sum %d",c);

 return 0;

}
```

There are three types

Simple macro

Nested macro

```
Argument macro

Simple macro

#define VALUE 10

#include<stdio.h>

#define LIKES 100

#define MSG “hello”

Void main()

{

Int n=10;

If(LIKES>50)

{

    Printf(MSG);

}

Getch();

}
```

Nested macro

```
#include<stdio.h>

#define SQUARE(X) ((X)*(X))

#define CUBE(X) (SQUARE(X)*(X))

Void main()

{

   Int ans;

   Ans=CUBE(2);

   Printf(“ans is=%d”,ans);

Getch();


}
```

File Inclusive Directives

1. File inclusive Directories are **used to include user define header file** inside C Program.
2. File inclusive directory checks included header file inside same directory (if path is not mentioned).
3. File inclusive directives begins with **#include**
4. If Path is mentioned then it will include that **header file into current scope**.
5. Instead of using triangular brackets we use "**Double Quote**" for inclusion of user defined header file.
6. It instructs the compiler to include **all specified files.**

Ways of including header file

way 1 : Including Standard Header Files

```
#include<filename>
```

- Search File in Standard Library

way 2 :User Defined Header Files are written in this way

```
#include"FILENAME"
```

- Search File in Current Library
- If not found , Search File in Standard Library
- User defined header files are written in this format

Live Example :

```
#include<stdio.h>     // Standard Header File
#include<conio.h>     // Standard Header File
#include"myfunc.h"    // User Defined Header File
```

Explanation :

1. In order to include user defined header file inside C Program , we must have to **create one user defined header file**. [Learn How to Create User Defined Header File]

2. Using double quotes include user defined header file inside Current C Program.
3. “myfunc.h” is user defined header file .
4. **We can combine all our user defined functions** inside header file and can include header file whenever require.

```
Factorial.c

#include<stdio.h>

Int factorial(int a)

{

Int f=1,I;

For(i=1;i<a;i++)

F=f*I;

Return f;

}

Fact.h

#include<stdio.h>

#include”fact.h”

Void main()

{

Int num,fact;

Scanf(“%d”,&num);

Fact=factorial(num);

Printf(“\nfactorial of %d is %d”,num,fact);

Getch();

}
```

Chain of Pointers

a pointer to point another pointer, thus creating a chain of pointers.

For example, in the following program, the pointer variable **'ptr2'** contains the address of the pointer variable **'ptr1'**, which points to the location that contains the desired value.

This is known as multiple indirections.

A variable that is a pointer to a pointer must be declared using additional indirection operator symbols in front of the name.
The declaration **'int **ptr2'** tells the compiler that **'ptr2'** is a pointer to a pointer of **int** type.

the pointer **'ptr2'** is not a pointer to an integer, but rather a pointer to an integer pointer.

We can access the target value indirectly pointed to by pointer to a pointer by applying the indirection operator twice.

Program

```c
#include <stdio.h>

int main()
{
    int x = 50;
    int *ptr1 = &x;
    int **ptr2 = &ptr1;

    printf("*ptr1  = %d\n",*ptr1);
    printf("**ptr2 = %d",**ptr2);

    return 0;
}
```

Program Output

```
*ptr1  = 50
**ptr2 = 50
```

Return statement.

The **return statement** returns the flow of the execution to the function from where it is called.

The return statement may or may not return anything for a void function, but for a non-void function, a return value is must be returned.

The return statement is used to terminate the execution of a function and transfer program

Control back to the calling function

Exit() function

The C library function void exit(int status) terminates the calling process immediately.

Call by value and call by reference

There are two ways to pass arguments/parameters to function calls –
call by value
and *call by reference.*
The major difference between call by value and call by reference is that in call by value a copy of actual arguments is passed to respective formal arguments.
in call by reference the location (address) of actual arguments is passed to formal arguments, hence any change made to formal arguments will also reflect in actual arguments.

In C, the calling and called functions do not share any memory -- they have their own copy and the called function cannot directly alter a variable in the calling function; it can only alter its private, temporary copy.

Difference between Call by Value and Call by Reference

Difference between *call by value* and *call by reference*

call by value	call by reference
In *call by value*, a copy of actual arguments is passed to formal arguments of the called function and any change made to the formal arguments in the called function have no effect on the values of actual arguments in the calling function.	In *call by reference*, the location (address) of actual arguments is passed to formal arguments of the called function. This means by accessing the addresses of actual arguments we can alter them within from the called function.
In call by value, actual arguments will remain safe, they cannot be modified accidentally.	In *call by reference*, alteration to actual arguments is possible within from called function; therefore the code must handle arguments carefully else you get unexpected results.

Call by Value

call by value a copy of actual arguments is passed to respective formal arguments. S

The classic example of wanting to modify the caller's memory is a `swapByValue()` function which exchanges two values.

```
#include <stdio.h>

void swapByValue(int, int);

int main()
{
  int n1 = 10, n2 = 20;


  swapByValue(n1, n2);
  printf("n1: %d, n2: %d\n", n1, n2);
}
```

```
void swapByValue(int a, int b)
{
  int t;
  t = a;
  a = b;
  b = t;
}

OUTPUT
======
n1: 10, n2: 20
```

The `swapByValue()` does not affect the arguments `n1` and `n2` in the calling function it only operates on `a` and `b` local to `swapByValue()` itself.

Call by Reference

In call by reference, to pass a variable `n` as a reference parameter, the programmer must pass a pointer to `n` instead of `n` itself.

The calling function will need to use `&` to compute the pointer of actual parameter.
The called function will need to dereference the pointer with `*` where appropriate to access the value of interest
. Here is an example of a correct *swap* `swapByReference()` function.
So, now you got the difference between call by value and call by reference!

```
#include <stdio.h>

void swapByReference(int*, int*);

int main() /* Main function */
{
  int n1 = 10, n2 = 20;

  /* actual arguments will be altered */
  swapByReference(&n1, &n2);
  printf("n1: %d, n2: %d\n", n1, n2);
}

void swapByReference(int *a, int *b)
{
```

```
  int t;
  t = *a;
*a = *b;
 *b = t;
}

OUTPUT
======
n1: 20, n2: 10
```

malloc()	Allocates single block of requested memory.
calloc()	Allocates multiple block of requested memory.
realloc()	Reallocates the memory occupied by malloc() or calloc() functions.
free()	Frees the dynamically allocated memory.

BLOG

TUTORIALS

Command line argument

```
int main() { /* ... */ }
```

We can also give command-line arguments in C and C++.

Command-line arguments are given after the name of the program in command-line shell of Operating Systems.
To pass command line arguments, we typically define main() with two arguments :

first argument is the number of command line arguments

and second is list of command-line arguments.

```
int main(int argc, char *argv[]) { /* ... */ }
```

or

```
int main(int argc, char **argv) { /* ... */ }
```

- **argc (ARGument Count)** is int and stores number of command-line arguments passed by the user including the name of the program.

- So if we pass a value to a program, value of argc would be 2 (one for argument and one for program name)
- The value of argc should be non negative.
- **argv(ARGument Vector)** is array of character pointers listing all the arguments.
- If argc is greater than zero,the array elements from argv[0] to argv[argc-1] will contain pointers to strings.
- Argv[0] is the name of the program , After that till argv[argc-1] every element is command -line arguments.

```
// Name of program mainreturn.cpp
#include <iostream>
using namespace std;

int main(int argc, char** argv)
{
    cout << "You have entered " << argc
         << " arguments:" << "\n";

    for (int i = 0; i < argc; ++i)
        cout << argv[i] << "\n";

    return 0;
}
```


```
// Name of program mainreturn.cpp
#include <iostream>
using namespace std;

int main(int argc, char** argv)
{
    cout << "You have entered " << argc
         << " arguments:" << "\n";

    for (int i = 0; i < argc; ++i)
        cout << argv[i] << "\n";

    return 0;
}
```


Input:

```
$ g++ mainreturn.cpp -o main
$ ./main geeks for geeks
```

Output:

```
You have entered 4 arguments:
./main
geeks
```

```
for
geeks
```

Dynamic Memory Allocation can be defined as a procedure in which the size of a data structure (like Array) is changed during the runtime.

C provides some functions to achieve these tasks.

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()

1. C malloc() method

"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.

Syntax:

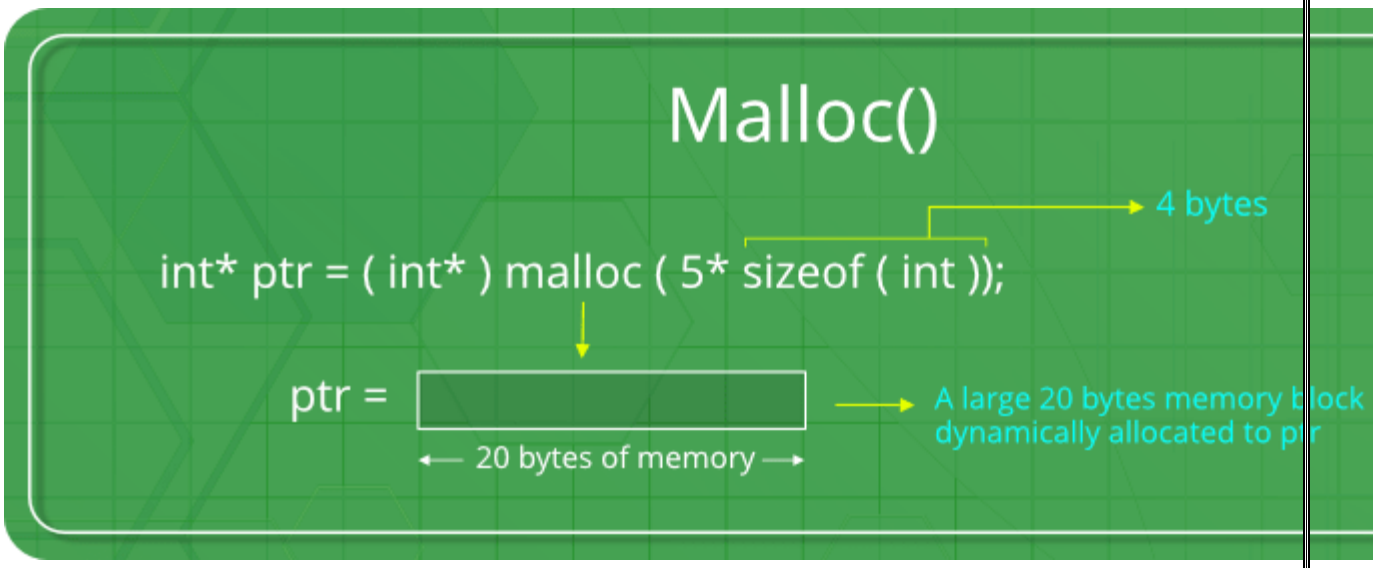
```
ptr = (cast-type*) malloc(byte-size)
```

For 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.



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

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

int main()
{

    // This pointer will hold the
    // base address of the block created
    int* ptr;
    int n, i;

    // Get the number of elements for the array
    n = 5;
    printf("Enter number of elements: %d\n", n);

    // Dynamically allocate memory using malloc()
```

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

    // Check if the memory has been successfully
    // allocated by malloc or not
    if (ptr == NULL) {
        printf("Memory not allocated.\n");
        exit(0);
    }
    else
{

        // Memory has been successfully allocated
        printf("Memory successfully allocated using malloc.\n");

        // Get the elements of the array
        for (i = 0; i < n; ++i) {
            ptr[i] = i + 1;
        }

        // 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;
```

```
    }
 // Check if the memory has been successfully
    // allocated by malloc or not
    if (ptr == NULL) {
        printf("Memory not allocated.\n");
        exit(0);
    }
    else
{


        // Memory has been successfully allocated
        printf("Memory successfully allocated using malloc.\n");


        // Get the elements of the array
        for (i = 0; i < n; ++i) {
            ptr[i] = i + 1;
        }


        // Print the elements of the array
        printf("The elements of the array are: ");
        for (i = 0; i < n; ++i) {
            printf("%d, ", ptr[i]);
        }
    }


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

```
int main()
{

    // This pointer will hold the
    // base address of the block created
    int* ptr;
    int n, i;

    // Get the number of elements for the array
    n = 5;
    printf("Enter number of elements: %d\n", n);

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

    // Check if the memory has been successfully
    // allocated by malloc or not
    if (ptr == NULL) {
        printf("Memory not allocated.\n");
        exit(0);
    }
    else
{

        // Memory has been successfully allocated
        printf("Memory successfully allocated using malloc.\n");
```

```
        // Get the elements of the array
        for (i = 0; i < n; ++i) {
            ptr[i] = i + 1;
        }

        // 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
Memory successfully allocated using malloc.
The elements of the array are: 1, 2, 3, 4, 5,
```

2. C calloc() method

“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’.

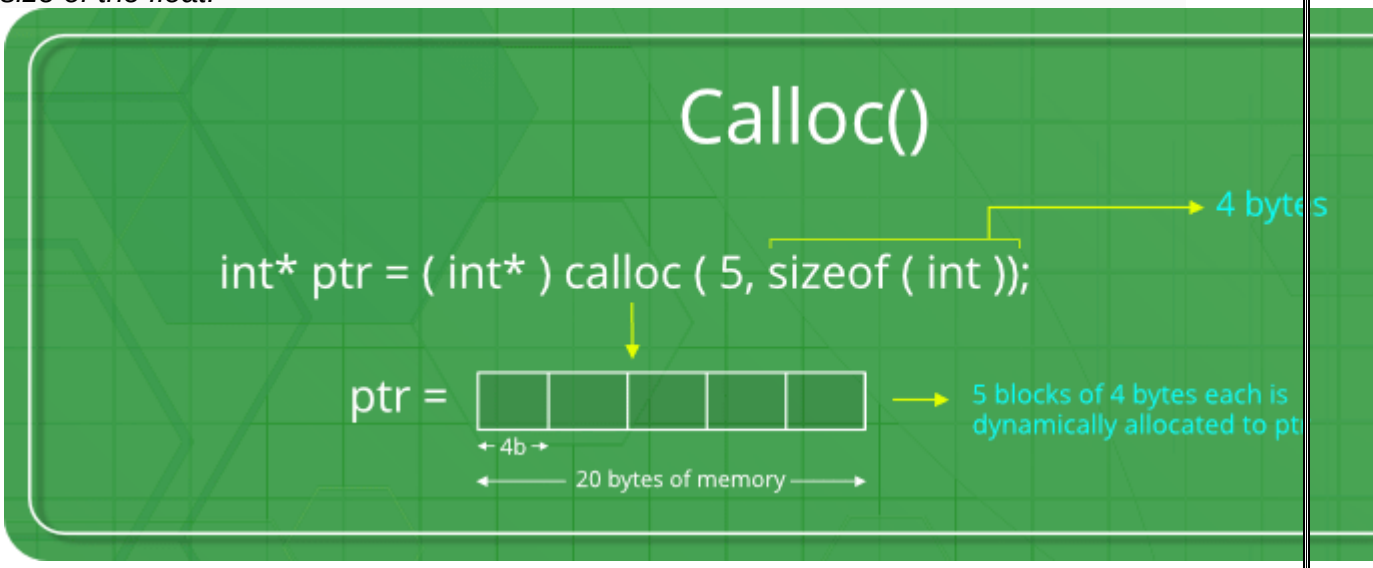
Syntax:

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

For Example:

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

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



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

```
#include <stdio.h>

#include <stdlib.h>

int main()

{

    // This pointer will hold the

    // base address of the block created

    int* ptr;

    int n, i;


    // Get the number of elements for the array

    n = 5;

    printf("Enter number of elements: %d\n", n);
```

```
// Dynamically allocate memory using calloc()
ptr = (int*)calloc(n, sizeof(int));

// Check if the memory has been successfully
// allocated by calloc or not
if (ptr == NULL) {
    printf("Memory not allocated.\n");
    exit(0);
}
else {

    // Memory has been successfully allocated
    printf("Memory successfully allocated using calloc.\n");

    // Get the elements of the array
    for (i = 0; i < n; ++i) {
        ptr[i] = i + 1;
    }

    // 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;
```

```
}
#include <stdio.h>
#include <stdlib.h>
int main()
{
    // This pointer will hold the
    // base address of the block created
    int* ptr;
    int n, i;

    // Get the number of elements for the array
    n = 5;
    printf("Enter number of elements: %d\n", n);

    // Dynamically allocate memory using calloc()
    ptr = (int*)calloc(n, sizeof(int));

    // Check if the memory has been successfully
    // allocated by calloc or not
    if (ptr == NULL) {
        printf("Memory not allocated.\n");
        exit(0);
    }
    else {

        // Memory has been successfully allocated
        printf("Memory successfully allocated using calloc.\n");
```

```
        // Get the elements of the array
        for (i = 0; i < n; ++i) {
            ptr[i] = i + 1;
        }

        // 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
Memory successfully allocated using calloc.
The elements of the array are: 1, 2, 3, 4, 5,
```

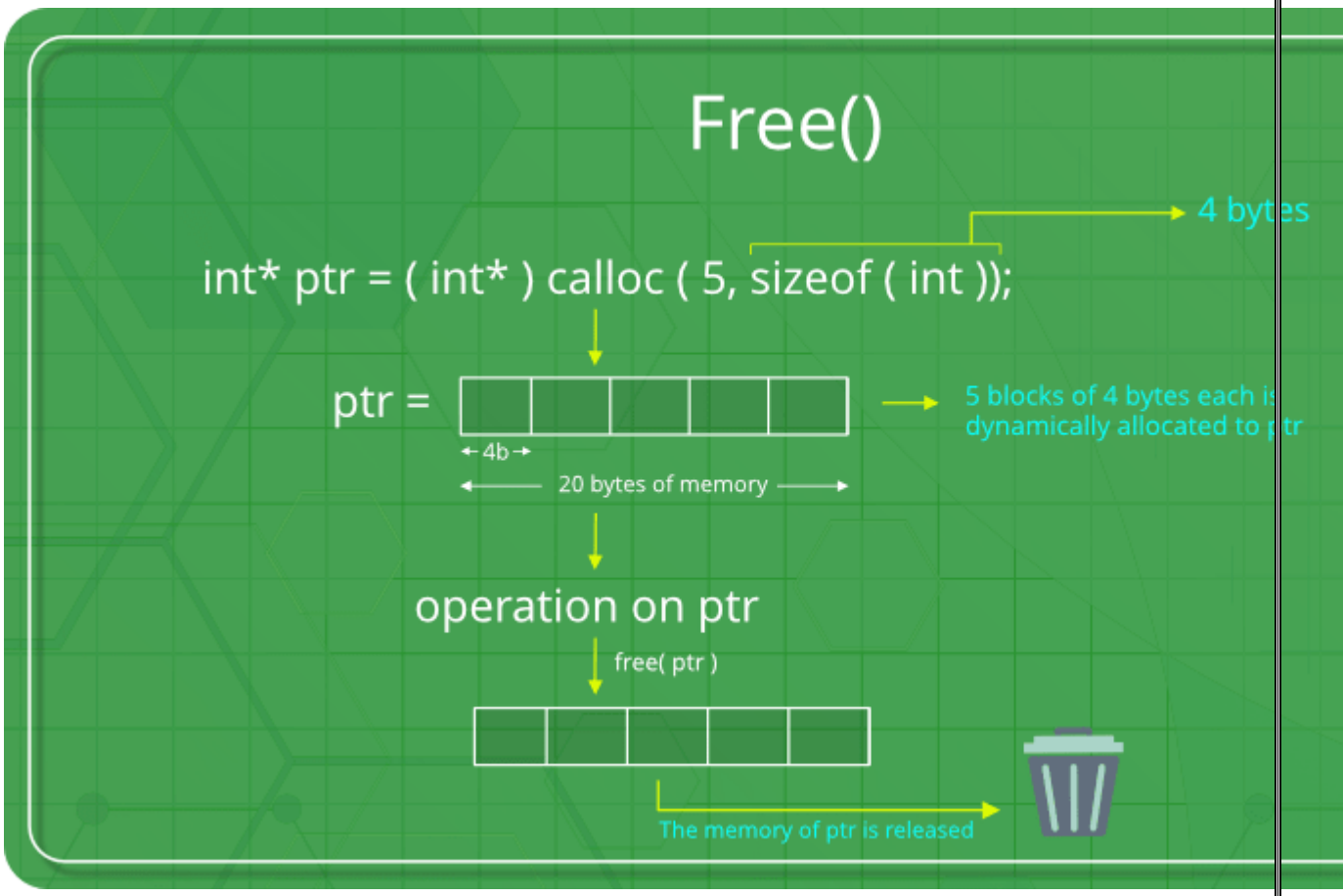
3. C free() method

"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);
```



Example:

filter_none

edit
play_arrow
brightness_4

```
#include <stdio.h>

#include <stdlib.h>


int main()

{

   // This pointer will hold the

    // base address of the block created

    int *ptr, *ptr1;
```

```
int n, i;
// Get the number of elements for the array
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));

// Check if the memory has been successfully
// allocated by malloc or not
if (ptr == NULL || ptr1 == NULL) {
    printf("Memory not allocated.\n");
    exit(0);
}
else {

    // Memory has been successfully allocated
    printf("Memory successfully allocated using malloc.\n");

    // Free the memory
    free(ptr);
    printf("Malloc Memory successfully freed.\n");

    // Memory has been successfully allocated
    printf("\nMemory successfully allocated using calloc.\n");
```

```
        // Free the memory
        free(ptr1);
        printf("Calloc Memory successfully freed.\n");
    }

    return 0;
}
#include <stdio.h>
#include <stdlib.h>

int main()
{
   // This pointer will hold the
    // base address of the block created
    int *ptr, *ptr1;
    int n, i;
    // Get the number of elements for the array
    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));

    // Check if the memory has been successfully
```

```
    // allocated by malloc or not
    if (ptr == NULL || ptr1 == NULL) {
        printf("Memory not allocated.\n");
        exit(0);
    }
    else {

        // Memory has been successfully allocated
        printf("Memory successfully allocated using malloc.\n");

        // Free the memory
        free(ptr);
        printf("Malloc Memory successfully freed.\n");

        // Memory has been successfully allocated
        printf("\nMemory successfully allocated using calloc.\n");

        // Free the memory
        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.
```

4. C realloc() method

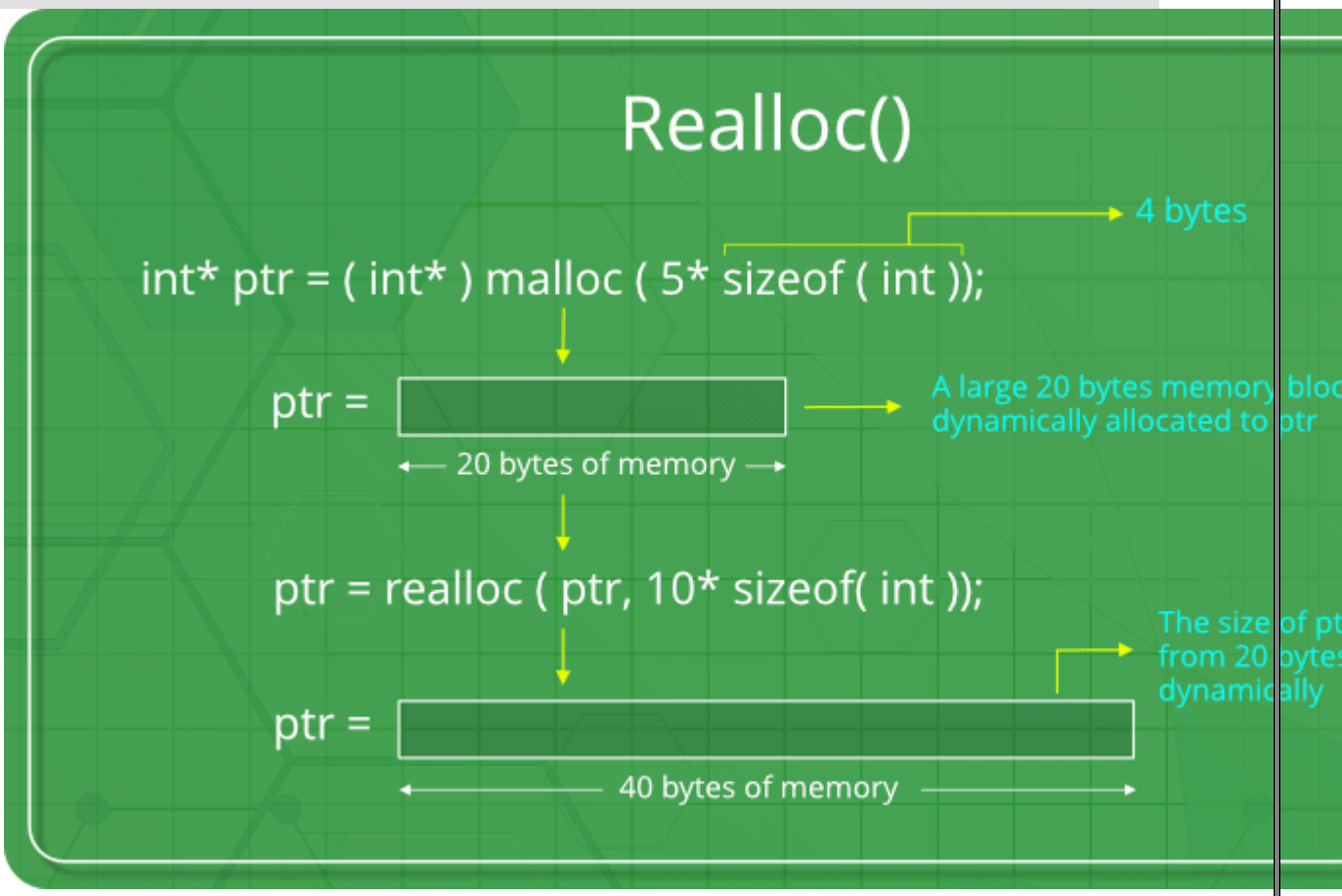
“realloc” or **“re-allocation”** method in C is used to dynamically change the memory allocation of a previously allocated memory.

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**.

Syntax:

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

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



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

Example:

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

int main()
{

    // This pointer will hold the
    // base address of the block created
    int* ptr;
    int n, i;

    // Get the number of elements for the array
    n = 5;
    printf("Enter number of elements: %d\n", n);

    // Dynamically allocate memory using calloc()
    ptr = (int*)calloc(n, sizeof(int));

    // Check if the memory has been successfully
    // allocated by malloc or not
    if (ptr == NULL) {
        printf("Memory not allocated.\n");
        exit(0);
    }
    else {

        // Memory has been successfully allocated
```

```
printf("Memory successfully allocated using calloc.\n");

// Get the elements of the array
for (i = 0; i < n; ++i) {
    ptr[i] = i + 1;
}

// Print the elements of the array
printf("The elements of the array are: ");
for (i = 0; i < n; ++i) {
    printf("%d, ", ptr[i]);
}

// Get the new size for the array
n = 10;
printf("\n\nEnter the new size of the array: %d\n", n);

// Dynamically re-allocate memory using realloc()
ptr = realloc(ptr, n * sizeof(int));

// Memory has been successfully allocated
printf("Memory successfully re-allocated using realloc.\n");

// Get the new elements of the array
for (i = 5; i < n; ++i) {
    ptr[i] = i + 1;
}
```

```
        // Print the elements of the array
        printf("The elements of the array are: ");
        for (i = 0; i < n; ++i) {
            printf("%d, ", ptr[i]);
        }

        free(ptr);
    }

    return 0;
}
// Get the number of elements for the array
n = 5;
printf("Enter number of elements: %d\n", n);

// Dynamically allocate memory using calloc()
ptr = (int*)calloc(n, sizeof(int));

// Check if the memory has been successfully
// allocated by malloc or not
if (ptr == NULL) {
    printf("Memory not allocated.\n");
    exit(0);
}
else {
```

```
    // Memory has been successfully allocated
    printf("Memory successfully allocated using calloc.\n");

    // Get the elements of the array
    for (i = 0; i < n; ++i) {
        ptr[i] = i + 1;
    }

    // Print the elements of the array
    printf("The elements of the array are: ");
for (i = 0; i < n; ++i) {
        printf("%d, ", ptr[i]);
    }

    // Get the new size for the array
    n = 10;
    printf("\n\nEnter the new size of the array: %d\n", n);

    // Dynamically re-allocate memory using realloc()
    ptr = realloc(ptr, n * sizeof(int));

    // Memory has been successfully allocated
    printf("Memory successfully re-allocated using realloc.\n");

    // Get the new elements of the array
    for (i = 5; i < n; ++i) {
        ptr[i] = i + 1;
```

```
        }


        // Print the elements of the array
        printf("The elements of the array are: ");
        for (i = 0; i < n; ++i) {
            printf("%d, ", ptr[i]);
        }


        free(ptr);
    }
```