**Topic-Pointers:Introduction, declaration, applications, Introduction to dynamic memory allocation (malloc, calloc, realloc, free), Use of pointers in self-referential structures, notion of linked list (no implementation)**

**File handling:File I/O functions, Standard C preprocessors, defining and calling macros, command-line arguments.**

# C Pointers

The pointer in C language is a variable which stores the address of another variable. This variable can be of type int, char, array, function, or any other pointer. The size of the pointer depends on the architecture. However, in 32-bit architecture the size of a pointer is 2 byte.

Consider the following example to define a pointer which stores the address of an integer.

1. int n = 10;
2. int\* p = &n; // Variable p of type pointer is pointing to the address of the variable n of type integer.



## Declaring a pointer

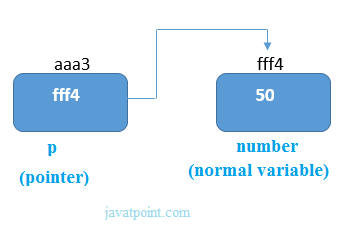
The pointer in c language can be declared using \* (asterisk symbol). It is also known as indirection pointer used to dereference a pointer.

1. int \*a;//pointer to int
2. char \*c;//pointer to char



### Pointer Example

An example of using pointers to print the address and value is given below.



As you can see in the above figure, pointer variable stores the address of number variable, i.e., fff4. The value of number variable is 50. But the address of pointer variable p is aaa3.

By the help of \* (**indirection operator**), we can print the value of pointer variable p.

Let's see the pointer example as explained for the above figure.

1. #include<stdio.h>
2. int main(){
3. int number=50;
4. int \*p;
5. p=&number;//stores the address of number variable
6. printf("Address of p variable is %x \n",p); // p contains the address of the number therefore printing p gives the address of number.
7. printf("Value of p variable is %d \n",\*p); // As we know that \* is used to dereference a pointer therefore if we print \*p, we will get the value stored at the address contained by p.
8. return 0;
9. }



**Output**

Address of number variable is fff4

Address of p variable is fff4

Value of p variable is 50

### Pointer to array

1. int arr[10];
2. int \*p[10]=&arr; // Variable p of type pointer is pointing to the address of an integer array arr.



### Pointer to a function

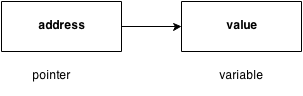
1. void show (int);
2. void(\*p)(int) = &display; // Pointer p is pointing to the address of a function



### Pointer to structure

1. struct st {
2. int i;
3. float f;
4. }ref;
5. struct st \*p = &ref;





## Advantage of pointer

1) Pointer **reduces the code** and **improves the performance**, it is used to retrieving strings, trees, etc. and used with arrays, structures, and functions.

2) We can **return multiple values from a function** using the pointer.

3) It makes you able to **access any memory location** in the computer's memory.

## Usage of pointer

There are many applications of pointers in c language.

**1) Dynamic memory allocation**

In c language, we can dynamically allocate memory using malloc() and calloc() functions where the pointer is used.

**2) Arrays, Functions, and Structures**

Pointers in c language are widely used in arrays, functions, and structures. It reduces the code and improves the performance.

## Address Of (&) Operator

The address of operator '&' returns the address of a variable. But, we need to use %u to display the address of a variable.

1. #include<stdio.h>
2. int main(){
3. int number=50;
4. printf("value of number is %d, address of number is %u",number,&number);
5. return 0;
6. }



**Output**

value of number is 50, address of number is fff4

## NULL Pointer

A pointer that is not assigned any value but NULL is known as the NULL pointer. If you don't have any address to be specified in the pointer at the time of declaration, you can assign NULL value. It will provide a better approach.

int \*p=NULL;

In the most libraries, the value of the pointer is 0 (zero).

## Pointer Program to swap two numbers without using the 3rd variable.

1. #include<stdio.h>
2. int main(){
3. int a=10,b=20,\*p1=&a,\*p2=&b;
5. printf("Before swap: \*p1=%d \*p2=%d",\*p1,\*p2);
6. \*p1=\*p1+\*p2;
7. \*p2=\*p1-\*p2;
8. \*p1=\*p1-\*p2;
9. printf("\nAfter swap: \*p1=%d \*p2=%d",\*p1,\*p2);
11. return 0;
12. }



**Output**

Before swap: \*p1=10 \*p2=20

After swap: \*p1=20 \*p2=10

## Reading complex pointers

There are several things which must be taken into the consideration while reading the complex pointers in C. Lets see the precedence and associativity of the operators which are used regarding pointers.

|  |  |  |
| --- | --- | --- |
| **Operator** | **Precedence** | **Associativity** |
| (), [] | 1 | Left to right |
| \*, identifier | 2 | Right to left |
| Data type | 3 | - |

Here,we must notice that,

* (): This operator is a bracket operator used to declare and define the function.
* []: This operator is an array subscript operator
* \* : This operator is a pointer operator.
* Identifier: It is the name of the pointer. The priority will always be assigned to this.
* Data type: Data type is the type of the variable to which the pointer is intended to point. It also includes the modifier like signed int, long, etc).

**How to read the pointer: int (\*p)[10].**

To read the pointer, we must see that () and [] have the equal precedence. Therefore, their associativity must be considered here. The associativity is left to right, so the priority goes to ().

Inside the bracket (), pointer operator \* and pointer name (identifier) p have the same precedence. Therefore, their associativity must be considered here which is right to left, so the priority goes to p, and the second priority goes to \*.

Assign the 3rd priority to [] since the data type has the last precedence. Therefore the pointer will look like following.

* char -> 4
* \* -> 2
* p -> 1
* [10] -> 3

The pointer will be read as p is a pointer to an array of integers of size 10.

**Example**

How to read the following pointer?

1. int (\*p)(int (\*)[2], int (\*)void))



### Explanation

This pointer will be read as p is a pointer to such function which accepts the first parameter as the pointer to a one-dimensional array of integers of size two and the second parameter as the pointer to a function which parameter is void and return type is the integer.

# Dynamic memory allocation in C

The concept of **dynamic memory allocation in c language** enables the C programmer to allocate memory at runtime. Dynamic memory allocation in c language is possible by 4 functions of stdlib.h header file.

1. malloc()
2. calloc()
3. realloc()
4. free()

Before learning above functions, let's understand the difference between static memory allocation and dynamic memory allocation.

|  |  |
| --- | --- |
| **static memory allocation** | **dynamic memory allocation** |
| memory is allocated at compile time. | memory is allocated at run time. |
| memory can't be increased while executing program. | memory can be increased while executing program. |
| used in array. | used in linked list. |

Now let's have a quick look at the methods used for dynamic memory allocation.

|  |  |
| --- | --- |
| **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. |

## malloc() function in C

The malloc() function allocates single block of requested memory.

It doesn't initialize memory at execution time, so it has garbage value initially.

It returns NULL if memory is not sufficient.

The syntax of malloc() function is given below:

1. ptr=(cast-type\*)malloc(byte-size)



Let's see the example of malloc() function.

1. #include<stdio.h>
2. #include<stdlib.h>
3. int main(){
4. int n,i,\*ptr,sum=0;
5. printf("Enter number of elements: ");
6. scanf("%d",&n);
7. ptr=(int\*)malloc(n\*sizeof(int));  //memory allocated using malloc
8. if(ptr==NULL)
9. {
10. printf("Sorry! unable to allocate memory");
11. exit(0);
12. }
13. printf("Enter elements of array: ");
14. for(i=0;i<n;++i)
15. {
16. scanf("%d",ptr+i);
17. sum+=\*(ptr+i);
18. }
19. printf("Sum=%d",sum);
20. free(ptr);
21. return 0;
22. }



**Output**

Enter elements of array: 3

Enter elements of array: 10

10

10

Sum=30

## calloc() function in C

The calloc() function allocates multiple block of requested memory.

It initially initialize all bytes to zero.

It returns NULL if memory is not sufficient.

The syntax of calloc() function is given below:

1. ptr=(cast-type\*)calloc(number, byte-size)



Let's see the example of calloc() function.

1. #include<stdio.h>
2. #include<stdlib.h>
3. int main(){
4. int n,i,\*ptr,sum=0;
5. printf("Enter number of elements: ");
6. scanf("%d",&n);
7. ptr=(int\*)calloc(n,sizeof(int));  //memory allocated using calloc
8. if(ptr==NULL)
9. {
10. printf("Sorry! unable to allocate memory");
11. exit(0);
12. }
13. printf("Enter elements of array: ");
14. for(i=0;i<n;++i)
15. {
16. scanf("%d",ptr+i);
17. sum+=\*(ptr+i);
18. }
19. printf("Sum=%d",sum);
20. free(ptr);
21. return 0;
22. }



**Output**

Enter elements of array: 3

Enter elements of array: 10

10

10

Sum=30

## realloc() function in C

If memory is not sufficient for malloc() or calloc(), you can reallocate the memory by realloc() function. In short, it changes the memory size.

Let's see the syntax of realloc() function.

1. ptr=realloc(ptr, new-size)



## free() function in C

The memory occupied by malloc() or calloc() functions must be released by calling free() function. Otherwise, it will consume memory until program exit.

Let's see the syntax of free() function.

1. free(ptr)

# Command Line Arguments in C

The arguments passed from command line are called command line arguments. These arguments are handled by main() function.

To support command line argument, you need to change the structure of main() function as given below.

1. **int** main(**int** argc, **char** \*argv[] )

Here, **argc** counts the number of arguments. It counts the file name as the first argument.

The **argv[]** contains the total number of arguments. The first argument is the file name always.

## Example

Let's see the example of command line arguments where we are passing one argument with file name.

1. #include <stdio.h>
2. **void** main(**int** argc, **char** \*argv[] )  {
4. printf("Program name is: %s\n", argv[0]);
6. **if**(argc < 2){
7. printf("No argument passed through command line.\n");
8. }
9. **else**{
10. printf("First argument is: %s\n", argv[1]);
11. }
12. }

Run this program as follows in Linux:

1. ./program hello

Run this program as follows in Windows from command line:

1. program.exe hello

**Output:**

Program name is: program

First argument is: hello

If you pass many arguments, it will print only one.

1. ./program hello c how r u

**Output:**

Program name is: program

First argument is: hello

But if you pass many arguments within double quote, all arguments will be treated as a single argument only.

1. ./program "hello c how r u"

**Output:**

Program name is: program

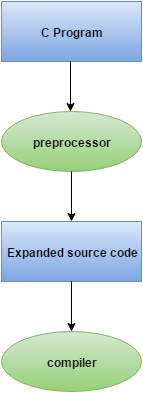
First argument is: hello c how r u

You can write your program to print all the arguments. In this program, we are printing only argv[1], that is why it is printing only one argument.

# C Preprocessor Directives

The C preprocessor is a micro processor that is used by compiler to transform your code before compilation. It is called micro preprocessor because it allows us to add macros.

#### Note: Proprocessor direcives are executed before compilation.



All preprocessor directives starts with hash # symbol.

Let's see a list of preprocessor directives.

* #include
* #define
* #undef
* #ifdef
* #ifndef
* #if
* #else
* #elif
* #endif
* #error
* #pragma

## What is Macro

# C Macros

A macro is a segment of code which is replaced by the value of macro. Macro is defined by #define directive. There are two types of macros:

1. Object-like Macros
2. Function-like Macros

## Object-like Macros

The object-like macro is an identifier that is replaced by value. It is widely used to represent numeric constants. For example:

1. #define PI 3.14

Here, PI is the macro name which will be replaced by the value 3.14.

## Function-like Macros

The function-like macro looks like function call. For example:

1. #define MIN(a,b) ((a)<(b)?(a):(b))

Here, MIN is the macro name.

Visit [#define](https://www.javatpoint.com/c-preprocessor-define) to see the full example of object-like and function-like macros.

C Predefined Macros

ANSI C defines many predefined macros that can be used in c program.

|  |  |  |
| --- | --- | --- |
| **No.** | **Macro** | **Description** |
| 1 | \_DATE\_ | represents current date in "MMM DD YYYY" format. |
| 2 | \_TIME\_ | represents current time in "HH:MM:SS" format. |
| 3 | \_FILE\_ | represents current file name. |
| 4 | \_LINE\_ | represents current line number. |
| 5 | \_STDC\_ | It is defined as 1 when compiler complies with the ANSI standard. |

C predefined macros example

*File: simple.c*

1. #include<stdio.h>
2. **int** main(){
3. printf("File :%s\n", \_\_FILE\_\_ );
4. printf("Date :%s\n", \_\_DATE\_\_ );
5. printf("Time :%s\n", \_\_TIME\_\_ );
6. printf("Line :%d\n", \_\_LINE\_\_ );
7. printf("STDC :%d\n", \_\_STDC\_\_ );
8. **return** 0;
9. }

Output:

File :simple.c

Date :Dec 6 2015

Time :12:28:46

Line :6

STDC :1

C - File I/O

The last chapter explained the standard input and output devices handled by C programming language. This chapter cover how C programmers can create, open, close text or binary files for their data storage.

A file represents a sequence of bytes, regardless of it being a text file or a binary file. C programming language provides access on high level functions as well as low level (OS level) calls to handle file on your storage devices. This chapter will take you through the important calls for file management.

## Opening Files

You can use the **fopen( )** function to create a new file or to open an existing file. This call will initialize an object of the type **FILE**, which contains all the information necessary to control the stream. The prototype of this function call is as follows −

FILE \*fopen( const char \* filename, const char \* mode );

Here, **filename** is a string literal, which you will use to name your file, and access **mode** can have one of the following values −

|  |  |
| --- | --- |
| **Sr.No.** | **Mode & Description** |
| 1 | **r**  Opens an existing text file for reading purpose. |
| 2 | **w**  Opens a text file for writing. If it does not exist, then a new file is created. Here your program will start writing content from the beginning of the file. |
| 3 | **a**  Opens a text file for writing in appending mode. If it does not exist, then a new file is created. Here your program will start appending content in the existing file content. |
| 4 | **r+**  Opens a text file for both reading and writing. |
| 5 | **w+**  Opens a text file for both reading and writing. It first truncates the file to zero length if it exists, otherwise creates a file if it does not exist. |
| 6 | **a+**  Opens a text file for both reading and writing. It creates the file if it does not exist. The reading will start from the beginning but writing can only be appended. |

If you are going to handle binary files, then you will use following access modes instead of the above mentioned ones −

"rb", "wb", "ab", "rb+", "r+b", "wb+", "w+b", "ab+", "a+b"

## Closing a File

To close a file, use the fclose( ) function. The prototype of this function is −

int fclose( FILE \*fp );

The **fclose(-)** function returns zero on success, or **EOF** if there is an error in closing the file. This function actually flushes any data still pending in the buffer to the file, closes the file, and releases any memory used for the file. The EOF is a constant defined in the header file **stdio.h**.

There are various functions provided by C standard library to read and write a file, character by character, or in the form of a fixed length string.

## Writing a File

Following is the simplest function to write individual characters to a stream −

int fputc( int c, FILE \*fp );

The function **fputc()** writes the character value of the argument c to the output stream referenced by fp. It returns the written character written on success otherwise **EOF** if there is an error. You can use the following functions to write a null-terminated string to a stream −

int fputs( const char \*s, FILE \*fp );

The function **fputs()** writes the string **s** to the output stream referenced by fp. It returns a non-negative value on success, otherwise **EOF** is returned in case of any error. You can use **int fprintf(FILE \*fp,const char \*format, ...)** function as well to write a string into a file. Try the following example.

Make sure you have **/tmp** directory available. If it is not, then before proceeding, you must create this directory on your machine.

#include <stdio.h>

main() {

FILE \*fp;

fp = fopen("/tmp/test.txt", "w+");

fprintf(fp, "This is testing for fprintf...\n");

fputs("This is testing for fputs...\n", fp);

fclose(fp);

}

When the above code is compiled and executed, it creates a new file **test.txt** in /tmp directory and writes two lines using two different functions. Let us read this file in the next section.

## Reading a File

Given below is the simplest function to read a single character from a file −

int fgetc( FILE \* fp );

The **fgetc()** function reads a character from the input file referenced by fp. The return value is the character read, or in case of any error, it returns **EOF**. The following function allows to read a string from a stream −

char \*fgets( char \*buf, int n, FILE \*fp );

The functions **fgets()** reads up to n-1 characters from the input stream referenced by fp. It copies the read string into the buffer **buf**, appending a **null** character to terminate the string.

If this function encounters a newline character '\n' or the end of the file EOF before they have read the maximum number of characters, then it returns only the characters read up to that point including the new line character. You can also use **int fscanf(FILE \*fp, const char \*format, ...)** function to read strings from a file, but it stops reading after encountering the first space character.

#include <stdio.h>

main() {

FILE \*fp;

char buff[255];

fp = fopen("/tmp/test.txt", "r");

fscanf(fp, "%s", buff);

printf("1 : %s\n", buff );

fgets(buff, 255, (FILE\*)fp);

printf("2: %s\n", buff );

fgets(buff, 255, (FILE\*)fp);

printf("3: %s\n", buff );

fclose(fp);

}

When the above code is compiled and executed, it reads the file created in the previous section and produces the following result −

1 : This

2: is testing for fprintf...

3: This is testing for fputs...

Let's see a little more in detail about what happened here. First, **fscanf()** read just **This** because after that, it encountered a space, second call is for **fgets()** which reads the remaining line till it encountered end of line. Finally, the last call **fgets()** reads the second line completely.

## Binary I/O Functions

There are two functions, that can be used for binary input and output −

size\_t fread(void \*ptr, size\_t size\_of\_elements, size\_t number\_of\_elements, FILE \*a\_file);

size\_t fwrite(const void \*ptr, size\_t size\_of\_elements, size\_t number\_of\_elements, FILE \*a\_file);

Both of these functions should be used to read or write blocks of memories - usually arrays or structures.

## What is a Storage Class?

A storage class represents the visibility and a location of a variable. It tells from what part of code we can access a variable. A storage class is used to describe the following things:

* The variable scope.
* The location where the variable will be stored.
* The initialized value of a variable.
* A lifetime of a variable.
* Who can access a variable?

Thus a storage class is used to represent the information about a variable.

NOTE: A variable is not only associated with a data type, its value but also a storage class.

There are total four types of standard storage classes. The table below represents the storage classes in 'C'.

|  |  |
| --- | --- |
| **Storage class** | **Purpose** |
| **auto** | It is a default storage class. |
| **extern** | It is a global variable. |
| **static** | It is a local variable which is capable of returning a value even when control is transferred to the function call. |
| **register** | It is a variable which is stored inside a Register. |

In this tutorial, you will learn-

* [Auto storage class](https://www.guru99.com/c-storage-classes.html#1)
* [Extern storage class](https://www.guru99.com/c-storage-classes.html#2) 
  + [First File: main.c](https://www.guru99.com/c-storage-classes.html#3)
  + [Second File: original.c](https://www.guru99.com/c-storage-classes.html#4)
* [Static storage class](https://www.guru99.com/c-storage-classes.html#5)
* [Register storage class](https://www.guru99.com/c-storage-classes.html#6)

## Auto storage class

The variables defined using auto storage class are called as local variables. Auto stands for automatic storage class. A variable is in auto storage class by default if it is not explicitly specified.

The scope of an auto variable is limited with the particular block only. Once the control goes out of the block, the access is destroyed. This means only the block in which the auto variable is declared can access it.

A keyword auto is used to define an auto storage class. By default, an auto variable contains a garbage value.

Example, auto int age;

The program below defines a function with has two local variables

int add(void) {

int a=13;

auto int b=48;

return a+b;}

We take another program which shows the scope level "visibility level" for auto variables in each block code which are independently to each other:

#include <stdio.h>

int main( )

{

auto int j = 1;

{

auto int j= 2;

{

auto int j = 3;

printf ( " %d ", j);

}

printf ( "\t %d ",j);

}

printf( "%d\n", j);}

OUTPUT:

3 2 1

## Extern storage class

Extern stands for external storage class. Extern storage class is used when we have global functions or variables which are shared between two or more files.

Keyword **extern** is used to declaring a global variable or function in another file to provide the reference of variable or function which have been already defined in the original file.

The variables defined using an extern keyword are called as global variables. These variables are accessible throughout the program. Notice that the extern variable cannot be initialized it has already been defined in the original file

Example, extern void display();

### First File: main.c

#include <stdio.h>

extern i;

main() {

printf("value of the external integer is = %d\n", i);

return 0;}

### Second File: original.c

#include <stdio.h>

i=48;

Result:

value of the external integer is = 48

In order to compile and run the above code, follow the below steps

## Static storage class

The static variables are used within function/ file as local static variables. They can also be used as a global variable

* Static local variable is a local variable that retains and stores its value between function calls or block and remains visible only to the function or block in which it is defined.
* Static global variables are global variables visible **only to the file in which it is declared.**

Example: static int count = 10;

Keep in mind that static variable has a default initial value zero and is initialized only once in its lifetime.

#include <stdio.h> /\* function declaration \*/

void next(void);

static int counter = 7; /\* global variable \*/

main() {

while(counter<10) {

next();

counter++; }

return 0;}

void next( void ) { /\* function definition \*/

static int iteration = 13; /\* local static variable \*/

iteration ++;

printf("iteration=%d and counter= %d\n", iteration, counter);}

Result:

iteration=14 and counter= 7

iteration=15 and counter= 8

iteration=16 and counter= 9

Global variables are accessible throughout the file whereas static variables are accessible only to the particular part of a code.

The lifespan of a static variable is in the entire program code. A variable which is declared or initialized using static keyword always contains zero as a default value.

## Register storage class

You can use the register storage class when you want to store local variables within functions or blocks in CPU registers instead of RAM to have quick access to these variables. For example, "counters" are a good candidate to be stored in the register.

Example: register int age;

The keyword **register** is used to declare a register storage class. The variables declared using register storage class has lifespan throughout the program.

It is similar to the auto storage class. The variable is limited to the particular block. The only difference is that the variables declared using register storage class are stored inside CPU registers instead of a memory. Register has faster access than that of the main memory.

The variables declared using register storage class has no default value. These variables are often declared at the beginning of a program.

#include <stdio.h> /\* function declaration \*/

main() {

{register int weight;

int \*ptr=&weight ;/\*it produces an error when the compilation occurs ,we cannot get a memory location when dealing with CPU register\*/}

}

OUTPUT:

error: address of register variable 'weight' requested

The next table summarizes the principal features of each storage class which are commonly used in C programming

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Storage Class** | **Declaration** | **Storage** | **Default Initial Value** | **Scope** | **Lifetime** |
| **auto** | Inside a function/block | Memory | Unpredictable | Within the function/block | Within the function/block |
| **register** | Inside a function/block | CPU Registers | Garbage | Within the function/block | Within the function/block |
| **extern** | Outside all functions | Memory | Zero | Entire the file and other files where the variable is declared as extern | program runtime |
| **Static (local)** | Inside a function/block | Memory | Zero | Within the function/block | program runtime |
| **Static (global)** | Outside all functions | Memory | Zero | Global | program runtime |

**Summary**

In this tutorial we have discussed storage classes in C, to sum up:

* A storage class is used to represent additional information about a variable.
* Storage class represents the scope and lifespan of a variable.
* It also tells who can access a variable and from where?
* Auto, extern, register, static are the four storage classes in 'C'.
* auto is used for a local variable defined within a block or function
* register is used to store the variable in CPU registers rather memory location for quick access.
* Static is used for both global and local variables. Each one has its use case within a C program.
* Extern is used for data sharing between C project files.