**Functions in C**

# Function

* A function is a set of statements that take inputs, do some specific computation and produces output.
* The idea is to put some commonly or repeatedly done task together and make a function, so that instead of writing the same code again and again for different inputs, we can call the function.

#include <stdio.h>

// function definition

int max(int x, int y) { // x, y formal parameters

if (x > y) return x;

else return y;

}

// main function

int main(void) {

int a = 10, b = 20;

// Calling function max

int m = max(a, b); // a, b actual parameters

printf("m is %d", m);

return 0;

}

Output:

m is 20

# Function Declaration

* Function declaration tells compiler about number of parameters function takes, data-types of parameters and return type of function.
* Putting parameter names in function declaration is optional in function declaration, but it is necessary to put them in definition.

int fun(char, int); (parameter names are not there in below declarations)

* In C, we can do both declaration and definition at same place.
* C also allows to declare and define functions separately, this is especially needed in case of library functions.
* The library functions are declared in header files and defined in library files.

# Parameter passing to functions

Let us assume that a function B() is called from another function A(). In this case A is called the “caller function” and B is called the “called function or callee function”.

Also, the arguments which A sends to B are called actual arguments and the parameters of B are called formal arguments.

* **Actual parameters**: parameters passed to function appears in the function or method call in the calling environment.

For example, in the above program 10 and 20 are actual parameters.

* **Formal parameters**: parameters received by function appear in the prototype of the function or method.

For example, in the above program x and y are formal parameters.

Modes:

**In**: Passes info from caller to calle.

**Out**: Callee writes values in caller.

**In/Out**: Caller tells callee value of variable, which may be updated by callee.

There are two most popular ways to pass parameters.

## **Pass by Value in C**

In this parameter passing method, a function copies the actual value of an argument into the formal parameter of the function and the two types of parameters are stored in different memory locations. So any changes made inside functions are not reflected in actual parameters of caller.

**Shortcomings**:

* Inefficiency in storage allocation
* For objects and arrays, the copy semantics are costly

## **Pass by Reference (or Pass by Pointer) in C**

Both actual and formal parameters refer to same locations, so any changes made inside the function are actually reflected in actual parameters of caller.

In C, parameters are always passed by value.

However, in C, we can use pointers to get the effect of pass by reference.

**Shortcomings**:

* Many potential scenarios can occur
* Programs are difficult to understand sometimes

// pass by value

#include <stdio.h>

void fun(int x) {

x = 30;

printf("Inside function fun() x = %d\n", x);

}

int main(void) {

int x = 20;

printf("Inside main before fun(x) x = %d\n", x);

fun(x);

printf("Inside main after fun(x) x = %d\n", x);

return 0;

}

Output:

Inside main before fun(x) x = 20

Inside function fun() x = 30

Inside main after fun(x) x = 20

// pass by reference

#include <stdio.h>

void fun(int \* x) {

\*x = 30;

printf("Inside function fun() \*x = %d\n", \*x);

}

int main(void) {

int x = 20;

printf("Inside main before fun(x) x = %d\n", x);

fun(&x);

printf("Inside main after fun(x) x = %d\n", x);

return 0;

}

Output:

Inside main before fun(x) x = 20

Inside function fun() \*x = 30

Inside main after fun(x) x = 30

# Important points about functions in C

1. Every C program has a function called main() that is called by operating system when a user runs the program.
2. Every function has a return type. If a function doesn’t return any value, then void is used as return type.
3. In C, functions can return any type except arrays and functions.

We can get around this limitation by returning pointer to array or pointer to function.

1. In C, empty parameter list mean that the parameter list is not specified and function can be called with any parameters.

In C, it is not a good idea to declare a function like **~~fun()~~**.

To declare a function that can only be called without any parameter, we should use **“void fun(void)”.**

In C++, empty list means function can only be called without any parameter.

In C++, both void fun() and void fun(void) are same.

# Importance of function prototype in C

Function prototype tells compiler about

* return type of the data that the function will return.
* the number of arguments passed to the function.
* the data types of the each of the passed arguments.
* the order in which the arguments are passed to the function.

By using this information, compiler cross checks function parameters and their data-type with function definition and function call.

function prototype specifies the input/output interlace to the function i.e. what to give to the function and what to expect from the function.

**Prototype of a function is also called signature of the function**.

If we ignore function prototype, program may compile with warning, and may work properly.

But sometimes, it will give strange output and it is very hard to find such programming mistakes

**Up to C90 standard, when we don’t provide prototype of function, the compiler assumes that function returns an integer.**

Later C99 standard specified that compilers can no longer assume return type as int. Therefore, **C99 became more restrict in type checking of function prototype**. But to make C99 standard backward compatible, in practice, compilers throw the warning saying that the return type is assumed as int. But they go ahead with compilation.

#include <stdio.h>

int main(void) {

int \*p = malloc(sizeof(int));

if (p == NULL) {

perror("malloc()");

return -1;

}

\*p = 10;

free(p);

printf("Fun Prototype.\n");

return 0;

}

linux$

linux$ gcc -Wall -g fun\_prototype.c -o fun\_prototype

fun\_prototype.c: In function ‘main’:

fun\_prototype.c:6:5: warning: implicit declaration of function ‘malloc’ [-Wimplicit-function-declaration]

int \*p = malloc(sizeof(int));

^

fun\_prototype.c:6:14: warning: incompatible implicit declaration of built-in function ‘malloc’ [enabled by default]

int \*p = malloc(sizeof(int));

^

fun\_prototype.c:14:5: warning: implicit declaration of function ‘free’ [-Wimplicit-function-declaration]

free(p);

^

fun\_prototype.c:14:5: warning: incompatible implicit declaration of built-in function ‘free’ [enabled by default]

linux$

linux$

linux$ ./fun\_prototype

Fun Prototype.

linux$

Above code will work fine on IA-32 model, but will fail on IA-64 model.

Reason for failure of this code is we haven’t included prototype of malloc() function and returned value is truncated in IA-64 model.

**What happens when a function is called before its declaration in C?**

In C, if a function is called before its declaration, the compiler assumes return type of the function as int.

**What about parameters?**

compiler assumes nothing about parameters. Therefore, the compiler will not be able to perform compile-time checking of argument types and arity when the function is applied to some arguments. This can cause problems.

There is this misconception that the compiler assumes input parameters also int. Had compiler assumed input parameters int, the above program would have failed in compilation.

// example a fun is called before its declaration

#include <stdio.h>

int main(void) {

// Note that fun() is not declared

printf("%d\n", fun());

return 0;

}

//char fun() **// In C, CE: conflicting types for ‘fun’**

int fun() {

return 'G';

}

linux$ ./fun\_prototype

71

In C++, CE: ‘fun’ was not declared in this scope

If a function is called before its declaration

#include <stdio.h>

int main (void) {

printf("%d\n", sum(10, 5));

return 0;

}

int sum (int b, int c, int a) {

return (a+b+c);

}

linux$ ./fun\_prototype

-520028569 // garbage value as output

linux$ gcc -Wall -g fun\_prototype.c -o fun\_prototype

fun\_prototype.c: In function ‘main’:

fun\_prototype.c:5:5: warning: implicit declaration of function ‘sum’ [-Wimplicit-function-declaration]

printf("%d\n", sum(10, 5));

^

linux$

# Functions that are executed before and after main() in C

#include<stdio.h>

/\* Apply the constructor attribute to myStartupFun() so that it

is executed before main() \*/

void myStartupFun (void) \_\_attribute\_\_ ((constructor));

/\* Apply the destructor attribute to myCleanupFun() so that it

is executed after main() \*/

void myCleanupFun (void) \_\_attribute\_\_ ((destructor));

/\* implementation of myStartupFun \*/

void myStartupFun (void) { printf ("startup code before main()\n"); }

/\* implementation of myCleanupFun \*/

void myCleanupFun (void) { printf ("cleanup code after main()\n"); }

int main (void) {

printf ("hello\n"); return 0;

}

linux$ gcc -Wall -g before\_after\_main.c -o before\_after\_main

linux$ ./before\_after\_main

startup code before main()

hello

cleanup code after main()

linux$

# \_\_attribute\_\_((constructor)) and \_\_attribute\_\_((destructor)) syntaxes in C

GCC specific syntaxes :

1. **\_\_attribute\_\_((constructor)) syntax**: This particular GCC syntax, when used with a function, executes the same function at the start-up of the program, i.e. before main() function.
2. **\_\_attribute\_\_((destructor)) syntax**: This particular GCC syntax, when used with a function, executes the same function just before the program terminates through \_exit, i.e. after main() function.

Explanation:

The way constructors and destructors work is that the shared object file contains special sections (.ctors and .dtors on ELF) which contain references to the functions marked with the constructor and destructor attributes, respectively. When the library is loaded/unloaded, the dynamic loader program checks whether such sections exist, and if so, calls the functions referenced therein.

Few points regarding these are worth noting:

1. **\_\_attribute\_\_((constructor))** runs when a shared library is loaded, typically during program startup.
2. **\_\_attribute\_\_((destructor))** runs when the shared library is unloaded, typically at program exit.
3. The two parentheses are presumably to distinguish them from function calls.
4. **\_\_attribute\_\_** is a GCC specific syntax; not a function or a macro.

**constructor, destructor, constructor (priority), destructor (priority)**

The constructor attribute causes the function to be called automatically before execution enters main ().

The destructor attribute causes the function to be called automatically after main () has completed or exit () has been called.

Functions with these attributes are useful for initializing data that will be used implicitly during the execution of the program.

You may provide an optional integer priority to control the order in which constructor and destructor functions are run.

A constructor with a smaller priority number runs before a constructor with a larger priority number; the opposite relationship holds for destructors.

So, if you have a constructor that allocates a resource and a destructor that deallocates the same resource, both functions typically have the same priority.

The priorities for constructor and destructor functions are the same as those specified for namespace-scope C++ objects (see [C++ Attributes](https://gcc.gnu.org/onlinedocs/gcc-4.7.2/gcc/C_002b_002b-Attributes.html#C_002b_002b-Attributes))

Check:

<http://www.geeksforgeeks.org/__attribute__constructor__attribute__destructor-syntaxes-c/>

<https://gcc.gnu.org/onlinedocs/gcc-4.7.2/gcc/Function-Attributes.html>

# Return statement vs exit() in main()

Check In C++

# How to Count Variable Numbers of Arguments in C?

Check variable arguments topic

# What is evaluation order of function parameters in C?

It is compiler dependent in C.

It is never safe to depend on the order of evaluation of side effects.

For example, a function call like below may very well behave differently from one compiler to another:

void func (int, int);

int i = 2;

func (i++, i++);

There is no guarantee (in either the C or the C++ standard language definitions) that the increments will be evaluated in any particular order.

Either increment might happen first. func might get the arguments `2, 3′, or it might get `3, 2′, or even `2, 2′.

# Does C support function overloading?

C doesn’t support this feature.

However, one can achieve the similar functionality in C indirectly.

One of the approach is as follows.

Have a void \* type of pointer as an argument to the function and another argument telling the actual data type of the first argument that is being passed.

int foo(void \* arg1, int arg2);

Suppose, arg2 can be interpreted as follows.

0 = Struct1 type variable,

1 = Struct2 type variable etc.

Here Struct1 and Struct2 are user defined struct types.

foo(arg1, 0); /\*Here, arg1 is pointer to struct type Struct1 variable\*/

foo(arg1, 1); /\*Here, arg1 is pointer to struct type Struct2 variable\*/

#include <stdio.h>

struct st1 {

int num;

};

struct st2 {

char ar[16];

};

int fun(void \* arg\_ptr, int arg\_dtype);

int main(void) {

struct st1 st1Var = {1};

struct st2 st2Var = {"Ratnesh"};

printf("%d\n", st1Var.num);

printf("%s\n", st2Var.ar);

printf("\n");

fun(&st1Var, 0);

fun(&st2Var, 1);

return 0;

}

int fun(void \* arg\_ptr, int arg\_dtype) {

if(0 == arg\_dtype) {

struct st1 \* ptr;

ptr = (struct st1 \*)arg\_ptr;

printf("%d\n", ptr->num);

}

else if(1 == arg\_dtype) {

struct st2 \* ptr;

ptr = (struct st2 \*)arg\_ptr;

printf("%s\n", ptr->ar);

}

else {

printf("Invalid Data Type.\n");

}

}

Output:

1

Ratnesh

1

Ratnesh

There can be several other ways of implementing function overloading in C.

But all of them will have to use pointers – the most powerful feature of C.

# How to return multiple values from a function?

Even though a function can return only one value but that value can be of pointer type.

**to return multiple values of**

**same data types**, we could return the pointer to array of that data types.

**different data types**, We can declare the function such that, it returns a structure type user defined variable or a pointer to it.

We can also make the function return multiple values by using the arguments of the function. How?

By providing the pointers as arguments

Usually, when a function needs to return several values, we use one pointer in return instead of several pointers as arguments.

#include <stdio.h>

struct st1 {

int num1;

char ar[16];

};

struct st1 fun(void);

int main(void) {

struct st1 st1Var;

st1Var = fun();

printf("num1 : %d, num2 : %s", st1Var.num1, st1Var.ar);

return 0;

}

struct st1 fun(void) {

struct st1 val = {5, "Ratnesh"};

return val;

}

Output:

num1 : 5, num2 : Ratnesh

# Static functions in C

In C, functions are global by default.

The “static” keyword before a function name makes it static.

static int fun1(void) {

printf("Inside static fun1");

}

Unlike global functions in C, access to static functions is restricted to the file where they are declared.

Therefore, when we want to restrict access to functions, we make them static.

Another reason for making functions static can be reuse of the same function name in other files.

//cmain.c

#include <stdio.h>

static int fun1(void);

int main(void) {

printf("Ratnesh\n");

fun1();

return 0;

}

//cfun1.c

#include <stdio.h>

static int fun1(void) {

printf("inside fun1\n");

return 0;

}

linux$

linux$gcc -Wall cmain.c cfun1.c -o cout

cmain.c:3:12: warning: 'fun1' used but never defined

static int fun1(void);

^

cfun1.c:3:12: warning: 'fun1' defined but not used [-Wunused-function]

static int fun1(void)

^

C:\Users\Admin\AppData\Local\Temp\cceMyfXI.o:cmain.c:(.text+0x1b): undefined ref

erence to `fun1'

collect2.exe: error: ld returned 1 exit status

To use static function defined in different file, use function pointer

//cmain.c

#include <stdio.h>

extern int (\*funptr1)(void);

extern int (\*funptr2)(void);

int main(void) {

printf("Ratnesh\n");

(funptr1)();

(funptr2)();

return 0;

}

//cfun1.c

#include <stdio.h>

static int fun1(void) {

printf("inside fun1\n");

return 0;

}

int (\*funptr1)(void) = &fun1;

//cfun2.c

#include <stdio.h>

static int fun1(void) {

printf("inside fun2\n");

return 0;

}

int (\*funptr2)(void) = &fun1;

linux$gcc -Wall cmain.c cfun1.c cfun2.c -o cout

linux$./cout

Ratnesh

inside fun1

inside fun2

linux$

# Callback function in C

Functions used as arguments to another function are sometimes called callback functions.

A callback is any executable code that is passed as an argument to other code, which is expected to call back (execute) the argument at a given time.

In simple language, If a reference of a function is passed to another function as an argument to call it, then it it will be called as a Callback function.

In C, a callback function is a function that is called through a function pointer.

#include <stdio.h>

void A() {

printf("Inside function A\n");

}

void B(void (\*ptr)()) {

printf("Inside function B\n");

(\*ptr) (); // callback to A

}

int main() {

void (\*ptr)() = &A;

B(ptr);

return 0;

}

Output:

Inside function B

Inside function A

In C++ STL, functors are also used for this purpose.

# Nested functions in C

Some programmer thinks that defining a function inside an another function is known as “nested function”.

But the reality is that it is not a nested function, it is treated as lexical scoping.

**Lexical scoping is not valid in C** because the compiler can’t reach/find the correct memory location of the inner function.

**Nested function is not supported by C** because we cannot define a function within another function in C.

We can declare a function inside a function, but it’s not a nested function.

Because **nested functions** definitions cannot access local variables of the surrounding blocks, they **can access only global variables of the containing module.**

This is done so that lookup of global variables doesn’t have to go through the directory.

As in C, there are two nested scopes: local and global (and beyond this, built-ins).

Therefore, nested functions have only a limited use.

If we try to approach nested function in C, then we will get compile time error.

// C program to illustrate the concept of Nested function.

#include <stdio.h>

int main(void) {

printf("Main");

int fun() {

printf("fun");

// defining view() function inside fun() function.

int view() {

printf("view");

}

return 1;

}

view();

}

// CE: undefined reference to `view'

An extension of the GNU C Compiler allows the declarations of nested functions. The declarations of nested functions under GCC’s extension need to be prefix/start with the auto keyword.

// C program of nested function with the help of gcc extension

#include <stdio.h>

int main(void) {

auto int view(); // declare function with auto keyword

view(); // calling function

printf("Main\n");

int view() {

printf("View\n");

return 1;

}

printf("GEEKS");

return 0;

}

Output:

view

Main

GEEKS

# Predefined Identifier \_\_func\_\_ in C

C language standard (i.e. C99 and C11) defines a predefined identifier as follows in clause 6.4.2.2:

“The identifier \_\_func\_\_ shall be implicitly declared by the translator as if, immediately following the opening brace of each function definition, the declaration

**static const char \_\_func\_\_[] = “function-name”;**

appeared, where function-name is the name of the lexically-enclosing function.”

C compiler implicitly adds \_\_func\_\_ in every function so that it can be used in that function to get the function name.

#include <stdio.h>

void foo(void) { printf("%s\n",\_\_func\_\_); }

void bar(void) { printf("%s\n",\_\_func\_\_); }

int main() {

foo();

bar();

printf("In file:%s, function:%s() line:%d date:%s time:%s",\_\_FILE\_\_,\_\_func\_\_,\_\_LINE\_\_,\_\_DATE\_\_,\_\_TIME\_\_);

return 0;

}

Output:

foo

bar

In file:cmain.c, function:main() line:15 date:Dec 18 2017 time:01:32:13

**You might get error but C standard says “undefined behaviour” if someone explicitly defines \_\_func\_\_**

#include <stdio.h>

int \_\_func\_\_ = 10;

int main() {

printf("%d",\_\_func\_\_);

return 0;

}

linux$gcc -Wall -g cmain.c -o cout

cmain.c:2:5: error: expected identifier or '(' before '\_\_func\_\_'

int \_\_func\_\_ = 10;

^

cmain.c: In function 'main':

cmain.c:5:9: warning: format '%d' expects argument of type 'int', but argument 2 has type 'const char \*' [-Wformat=]

printf("%d",\_\_func\_\_);

^

linux$

# \_Noreturn function specifier in C

After the removal of “noreturn” keyword, C11 standard (known as final draft) of C programming language introduce a new “\_Noreturn” function specifier that specify that the function does not return to the function that it was called from.

If the programmer tries to return any value from that function which is declared as \_Noreturn type, then compiler automatically generate run time error.

#include <stdio.h>

#include <stdlib.h>

\_Noreturn void view() {

printf("Insid fun view");

//return 10; //(2) when return 10

}

int main(void) {

printf("Ready to begin...\n");

view();

printf("NOT over till now\n");

return 0;

}

linux$gcc -Wall -g cmain.c -o cout

cmain.c: In function 'view':

cmain.c:8:1: warning: 'noreturn' function does return

}

^

linux$./cout

Ready to begin...

Insid fun view

linux$

//(2) when return 10

linux$gcc -Wall -g cmain.c -o cout

cmain.c: In function 'view':

cmain.c:7:9: warning: function declared 'noreturn' has a 'return' statement

return 10;

^

cmain.c:7:9: warning: 'return' with a value, in function returning void

cmain.c:7:9: warning: 'noreturn' function does return

return 10;

^

linux$

# Implicit return type int in C

In C, if we do not specify a return type, **compiler assumes an implicit return type as int**. However, C99 standard doesn’t allow return type to be omitted even if return type is int. This was allowed in older C standard C89.

In C++, the below program is not valid except few old C++ compilers like Turbo C++. Every function should specify the return type in C++.

#include <stdio.h>

fun(int x) { return x\*x; }

int main(void) {

printf("%d", fun(10));

return 0;

}

linux$

linux$gcc -Wall -g -std=c99 cmain.c -o cout

cmain.c:2:1: warning: return type defaults to 'int' [-Wimplicit-int]

fun(int x)

^

linux$

linux$gcc -Wall -g -std=c11 cmain.c -o cout

cmain.c:2:1: warning: return type defaults to 'int' [-Wimplicit-int]

fun(int x)

^

linux$gcc -Wall -g cmain.c -o cout

cmain.c:2:1: warning: return type defaults to 'int' [-Wimplicit-int]

fun(int x)

^

linux$

linux$g++ -Wall -g -std=c++11 cppmain.cpp -o cppout

error: ISO C++ forbids declaration of 'fun' with no type [-fpermissive]

fun(int x)

^

**NOTE: On Windows different behaviour observed**

G:\coding\test>

G:\coding\test>g++ -Wall -g -std=c++11 cppmain.cpp -o cppout

G:\coding\test>g++ -Wall -g cppmain.cpp -o cppout

G:\coding\test>cppout

100

G:\coding\test>

G:\coding\test>g++ --version

g++ (GCC) 5.3.0

Copyright (C) 2015 Free Software Foundation, Inc.

This is free software; see the source for copying conditions. There is NO

warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

G:\coding\test>

# exit() vs \_Exit() in C and C++

In C, exit() terminates the calling process without executing the rest code which is after the exit() function.

exit() function performs some cleaning before termination of the program like connection termination, buffer flushes etc.

\_exit, \_Exit - terminate the calling process

#include <unistd.h>

**void \_exit(int status);**

#include <stdlib.h>

**void \_Exit(int status);**

The function **\_Exit() is equivalent to \_exit().**

The \_Exit() function in C/C++ gives normal termination of a program without performing any cleanup tasks.

For example it does not execute functions registered with atexit() and on\_exit().

#include <stdio.h>

#include <stdlib.h>

void fun(void) { printf("exiting...\n"); }

int main() {

atexit(fun);

exit(10);

//\_Exit(10); // 2 \_Exit()

}

Output:

exiting...

// 2 \_Exit()

Output:

(no output)

You should use \_exit (or its synonym \_Exit) to abort the child program when the exec fails, because in this situation, the child process may interfere with the parent process' external data (files) by calling its atexit handlers, calling its signal handlers, and/or flushing buffers.

# exit(), abort() and assert()

## **exit()**

#include <stdlib.h>

**void exit(int status);**

exit - cause normal process termination

status: Status value returned to the parent process. Generally, a status value of 0 or EXIT\_SUCCESS indicates success, and any other value or the constant EXIT\_FAILURE is used to indicate an error. exit() performs following operations.

* Flushes unwritten buffered data.
* Closes all open files.
* Removes temporary files.
* Returns an integer exit status to the operating system.

The C standard atexit() function can be used to customize exit() to perform additional actions at program termination.

## **abort()**

#include <stdlib.h>

**void abort(void);**

abort - cause abnormal process termination

This function actually terminates **the process by raising a SIGABRT signal**, and your program can include a handler to intercept this signal (see this).

Unlike exit() function, abort() may not close files that are open. It may also not delete temporary files and may not flush stream buffer. Also, it does not call functions registered with atexit().

So programs like below might not write “Geeks for Geeks” to “myfile.txt”

#include <stdio.h>

#include <stdlib.h>

int main() {

FILE \*fp = fopen("C:\\myfile.txt", "w");

if(fp == NULL) {

printf("\n could not open file ");

getchar();

exit(1);

}

fprintf(fp, "%s", "Geeks for Geeks");

/\* Something went wrong so terminate here \*/

abort();

getchar();

return 0;

}

## **assert()**

#include <assert.h>

**void assert(scalar expression);**

assert - abort the program if assertion is false

If expression evaluates to 0 (false), then the expression, sourcecode filename, and line number are sent to the standard error, and then abort() function is called.

If the identifier NDEBUG (“no debug”) is defined with

**#define NDEBUG**

**then the macro assert does nothing.**

Common error outputting is in the form:

Assertion failed: expression, file filename, line line-number

#include<stdio.h>

#include<stdlib.h>

#include<assert.h>

void open\_record(char \*record\_name) {

assert(record\_name != NULL);

/\* Rest of code \*/

}

int main(void) {

open\_record(NULL);

return 0;

}

G:\coding\test>gcc -Wall -g cmain.c -o cout

G:\coding\test>cout.exe

Assertion failed: record\_name != NULL, file cmain.c, line 7

This application has requested the Runtime to terminate it in an unusual way.

Please contact the application's support team for more information.

G:\coding\test>

# Recursive Function

A recursive function in C++ is a function that calls itself.

**Recursive termination conditions**

A recursive termination is a condition that, when met, will cause the recursive function to stop calling itself.

void countDown(int count) {

std::cout << "push " << count << '\n';

if (count > 1) // termination condition

countDown(count-1);

std::cout << "pop " << count << '\n';

}

# References

<https://www.geeksforgeeks.org/c-programming-language/>

# END