Call Stack :

- a stack data structure that stores information about the active **subroutines** of a computer program.
- Also known as an **execution stack**, **program stack**, **control stack**, **run-time stack**, or **machine stack**, and is often shortened to just "**the stack**".

- A subroutine is a sequence of program instructions that perform a specific task, packaged as a unit and can be used in programs wherever the task should be performed.
- Subroutine can be a method or a function.

The Call Stack

Case 1	Case 2
Main()	Main()
X =10	X =10
Print X	Print X
f1();	f1();
X = f2();	print “over”
f3(X);	
print X	

Void f1()	Void f1()
Print “f1”	X=5
Return	Print “f1”
	F2();
	Return

Void f2()	Void f2()
Return 30	X=20
	F3(X)
	Return 30

Void f3(X)	Void f3(X)
Print “f3”, X	Print “f3”, X
X = X+10	X = X+10
Return	Return

```
3   X= f3()
          3.1 Call f3()
          3.2 Store X

4   Print (X)
```

- Demands for the Stack type storage structure to maintain call and return information…. Call Stack is a solution

- The main purpose of the Call Stack is to **keep track of the point to which each active subroutine should return** control when it finishes executing.
- Call Stack are usually hidden from the programmer.
  - They are given access only to a set of functions, and not the memory on the stack itself.
  - This is an example of abstraction.
  - **Most assembly languages**, on the other hand, require programmers to be involved with manipulating the stack.

- How does the Stack used for “Call and Return”
  - Since the Call Stack is organized as a stack, the **caller pushes the return address onto the stack,**
  - and the **called subroutine, when it finishes, pulls or pops** the return address off the Call Stack and transfers control to that address.
  - If a called subroutine calls on yet another subroutine, it will push another return address onto the Call Stack, and so on, with the information stacking up and unstacking as the program dictates.
  - The Call Stack only fix space in as the pushing consumes allocated space within the Call Stack, a stack overflow can occur causing the program to crash.

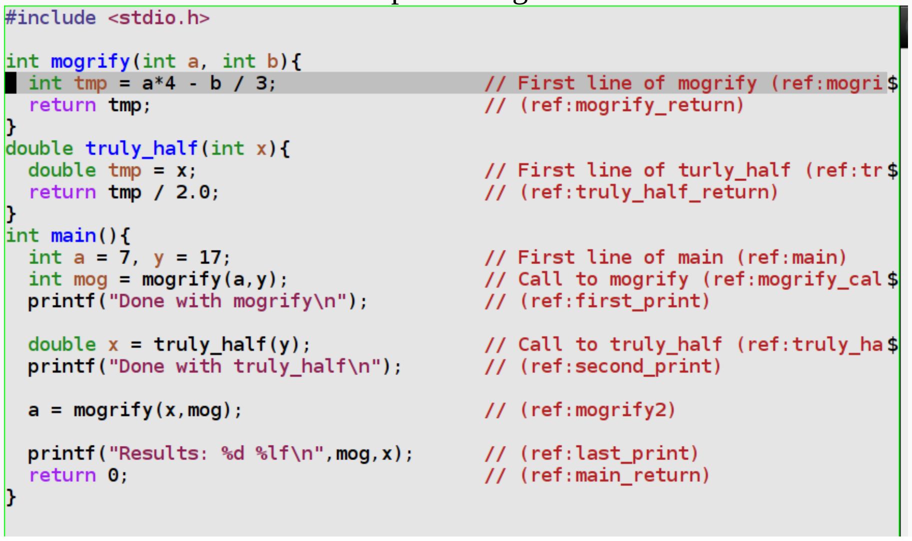
Function call stack in C.
The function call stack (often referred to just as the *call stack* or *the stack*) is responsible for maintaining the local variables and parameters during function execution.
Understanding how the call stack works explains how functions actually "work" including the following.

- The difference between function arguments and variables used to call functions
- Why several variables with the same name but residing in different function can co-exist
- Why there are certain limitations on what functions can do
- Why local variables that are not initialized might have any value contained in them.

Assumptions

- `ints` are 4 bytes (32-bits) and `doubles` are 8 bytes (64-bits). This is the case on many modern platforms but may vary somewhat and is not guaranteed by the C standard.
- The starting address of the first stack frame is chosen arbitrarily. In most cases, it is not important where it starts, only to understand the mechanics of how it grows and shrinks.

Sample C Program



```c
#include <stdio.h>

int mogrify(int a, int b){
  int tmp = a*4 - b / 3;                  // First line of mogrify (ref:mogri$
  return tmp;                             // (ref:mogrify_return)
}
double truly_half(int x){
  double tmp = x;                         // First line of turly_half (ref:tr$
  return tmp / 2.0;                       // (ref:truly_half_return)
}
int main(){
  int a = 7, y = 17;                      // First line of main (ref:main)
  int mog = mogrify(a,y);                 // Call to mogrify (ref:mogrify_cal$
  printf("Done with mogrify\n");          // (ref:first_print)

  double x = truly_half(y);               // Call to truly_half (ref:truly_ha$
  printf("Done with truly_half\n");       // (ref:second_print)

  a = mogrify(x,mog);                     // (ref:mogrify2)

  printf("Results: %d %lf\n",mog,x);      // (ref:last_print)
  return 0;                               // (ref:main_return)
}
```

C Program with Line Number

```
1: #include <stdio.h>
 2: // int mogrify(int,int);
 3: int mogrify(int a, int b){
 4:   int tmp = a*4 - b / 3;                    // First line of
mogrify (mogrify)
 5:   return tmp;                            // (mogrify_return)
 6: }
 7: double truly_half(int x){
 8:   double tmp = x / 2.0;                     // First line of
turly_half (truly_half)
 9:    return  tmp;                                          //
(truly_half_return)
10: }
11: int main(){
12:   int a = 7, y = 17;                        // First line of
main (main)
13:   int mog =mogrify(a,y);                  // Call to mogrify
(mogrify_call)
//13.a Call
//13.b  mog = return val
14:   printf("Done with mogrify\n");          // (first_print)
15:
16:    double x = truly_half(y);                      // Call to
truly_half (truly_half_call)
17:   printf("Done with truly_half\n");      // (second_print)
18:
19:   a = mogrify(x,mog);                     // (mogrify2)
20:
21:   printf("Results: %d %lf\n",mog,x);      // (last_print)
22:   return 0;                               // (main_return)
23: }
```

Call Stack behavior

Like all programs, control for the program starts in the `main()` function with the first line. `main()` has 3 local variables: `a,y` which are `ints` and `x` which is a `double`. The initial state of the stack is as follows.

```
11: int main(){
12:   int a = 7, y = 17;                        // First line of
main (main)
13:   int mog =mogrify(a,y);                   // Call to mogrify
(mogrify_call)
//13.a Call
//13.b  mog = return val
14:   printf("Done with mogrify\n");          // (first_print)
15:
16:    double x = truly_half(y);                     // Call to
truly_half (truly_half_call)
17:   printf("Done with truly_half\n");      // (second_print)
18:
19:   a = mogrify(x,mog);                       // (mogrify2)
20:
21:   printf("Results: %d %lf\n",mog,x);       // (last_print)
22:   return 0;                                 // (main_return)
23: }
```

main() called (static variable allocation)

Method	Line	Var	Value	Addr	Notes
main()	12	A	?	1024	
		Y	?	1028	
		Mog	?	1032	
		X	?	1036	

Notice that at the first executable statement of `main is at` line 12,

- stack space is allocated for all of its local variables but none of them have defined values yet.
- C programs do not guarantee local variables are initialized to anything and it is a mistake to use a variable value without first ensuring it has a well-defined value.

After moving ahead one line in `main`, locals `a,y` have defined values.

- This probably constitutes several low-level machine/assembly instructions.

One line of main() executed

Method	Line	Var	Value	Addr	Notes
main()	13	A	7	1024	
		Y	17	1028	

		Mog	?	1032	
		X	?	1036	

```
13:   int mog = mogrify(a,y);                          // Call to mogrify
(mogrify_call)
```

- At line 13 a different function is invoked.
- Control is suspended in `main` until the function `mogrify` completes.
- Calling a function **pushes another stack frame** onto the call stack which has enough space for the arguments to the function any local variables as shown below.

```
2: // int mogrify(int,int);
 3: int mogrify(int a, int b){
 4:   int tmp = a*4 - b / 3;                        // First line
of mogrify (mogrify)
 5:   return tmp;                                  // (mogrify_return)
 6: }
```

mogrify() called

Method	Line	Var	Value	Addr	Notes
main()	13	a	7	1024	
		y	17	1028	
		mog	?	1032	
		x	?	1036	double variable
mogrify()	4	a	7	1044	
		b	17	1048	
		tmp	?	1052	

Notice that the new stack frame starts almost immediately after the frame for `main`. There may be a small amount of additional space required to deal with return values or register saves at the low level, but in our high-level view this doesn't matter too much and we will always start the stack frames as close to each other as possible.

- Notice the allocation of `x` in `main` is a `double` so takes 8 bytes of space
- Control now starts at the beginning of `mogrify` but will eventually return to `main` at which point it will resume execution on line 13 completing that line and moving forward.
- Finally, notice that the parameters `a,b` to `mogrify` have values defined. This is as a result of `main` calling the function and passing actual values in for those parameters.
- The local variable `tmp` does not yet have known value associated with it as the first line of `mogrify` has not yet executed.

mogrify() first line executed

Method	Line	Var	Value	Addr	Notes

main()	13	a	7	1024	
		y	17	1028	
		mog	?	1032	
		x	?	1036	double variable
mogrify()	5	a	7	1044	
		b	17	1048	
		tmp	23	1052	

```
5:    return tmp;                                // (mogrify_return)
```

mogrify() second line (return) executed

Method	Line	Var	Value	Addr	Notes
main()	13	a	7	1024	
		y	17	1028	
		mog	23	1032	
		x	?	1036	double variable

- The second line of `mogrify` is to return a value to the calling function. This has two effects.
- The return value is stored wherever it was intended from the calling function: line 13 of `main` stores the value in variable `mog`.
- Returning **pops the stack frame** for `mogrify` off the call stack leaving it in the state above.
- Control now resumes in `main` by advancing one line forward.

Metho d	Line	Var	Value	Addr	Notes
main()	14	a	7	1024	
		y	17	1028	
		mo g	23	1032	
		x	?	1036	double variable

Executing a `printf`

Metho d	Line	Var	Value	Addr	Notes
main()	14	a	7	1024	
		y	17	1028	
		mo g	23	1032	
		x	?	1036	double variable
printf()	library call	fmt	??	1044	4-byte pointer
		??	7	1048	

- `printf` is like other functions in that it will push another stack frame onto the stack with space for its arguments and local variables.
- `printf` is a little unique in that it is a *variadic* function:
  - it can take any number of arguments so long as the arguments coincide with the format string.
  - We shall skip description for execution of printf()

```
OutPut : Done with mogrify
```

and returns which pops its stack frame. Control resumes in `main` at the next relevant line of code.

```
11: int main(){
12:   int a = 7, y = 17;                         // First line of
main (main)
13:    int mog = mogrify(a,y);                         // Call to
mogrify (mogrify_call)
14:   printf("Done with mogrify\n");          // (first_print)
15:
16:    double x = truly_half(y);                       // Call to
truly_half (truly_half_call)
17:   printf("Done with truly_half\n");      // (second_print)
```

Second function call

Method	Line	Var	Value	Addr	Notes
main()	16	a	7	1024	
		y	17	1028	
		mog	23	1032	
		x	?	1036	double variable

At line 16, another function is called which pushes another stack frame onto the call stack.

```
7: double truly_half(int x){
 8:   double tmp = x / 2.0;                      // First line of
turly_half (truly_half)
 9:     return  tmp;                                             //
(truly_half_return)
10: }
```

`truly_half()` called

Method	Line	Var	Value	Addr	Notes
main()	16	a	7	1024	
		y	17	1028	
		mog	23	1032	
		x	?	1036	double variable
`truly_half()`	8	x	17	1044	
		tmp	?	1048	double variable

- When `truly_half` is called, its stack frame is pushed on below `main`; notice that the memory addresses for its local variables are identical to previous function calls to `mogrify` and `printf`.

Space on the stack is re-used by subsequent function calls. This has implications for the values of variables that are not assigned values which we may discuss later.

The first line of `truly_half` assigns `tmp` as follows.

`truly_half()` first line executed

Method	Line	Var	Value	Addr	Notes
main()	16	a	7	1024	
		y	17	1028	
		mog	23	1032	
		x	?	1036	double variable
`truly_half()`	9	x	17	1044	
		tmp	8.5	1048	double variable

- Executing the next line of `truly_half` returns this value to assign the local variable `x` in `main` and pops the stack frame of `truly_half` off the call stack.

Control returns to main

Method	Line	Var	Value	Addr	Notes
main()	17	a	7	1024	
		y	17	1028	
		mog	23	1032	
		x	8.5	1036	double variable

In `main` another result is printed

```
Done with truly_half
```

and control advances. Line 19 calls `mogrify` again with different argument

```
11: int main(){
12:   int a = 7, y = 17;                         // First line of
main (main)
13:   int mog = mogrify(a,y);                   // Call to mogrify
(mogrify_call)
14:   printf("Done with mogrify\n");            // (first_print)
15:
16:   double x = truly_half(y);                          // Call to
truly_half (truly_half_call)
17:   printf("Done with truly_half\n");         // (second_print)
18:
```

```
19:   a = mogrify(x,mog);                        // (mogrify2)
20:
21:   printf("Results: %d %lf\n",mog,x);       // (last_print)
22:   return 0;                                  // (main_return)
23: }
```

```
3: int mogrify(int a, int b){
 4:   int tmp = a*4 - b / 3;                     // First line of
mogrify (mogrify)
 5:   return tmp;                              // (mogrify_return)
 6: }
```

mogrify called again

Method	Line	Var	Value	Addr	Notes
main()	19	a	7	1024	
		y	17	1028	
		mog	23	1032	
		x	8.5	1036	double variable
mogrify()	4	a	8	1044	caste to int
		b	23	1048	
		tmp	?	1052	

- Control in `main` is suspended at a different location than the first call to `mogrify` (line 19 this time) but control begins again in `mogrify` at its first line (line 4).
- Notice that the first argument to `mogrify` in this call is a little interesting:
  - it was passed as a `double` with value 8.5 (x=8.5) but appears as an `int` with value 8.
  - The compiler has automatically inserted low-level instructions to caste the 8-byte floating point value to a 4-byte integer value.
  - Most of the time this is nice convenience which saves programmers the trouble of writing such code but it can create subtle bugs if the programmer did not intend for such a conversion to happen.

mogrify first line (tmp =....) executed again

Method	Line	Var	Value	Addr	Notes
main()	19	a	7	1024	
		y	17	1028	
		mog	23	1032	

		x	8.5	1036	double variable
mogrify()	5	a	8	1044	caste to int
		b	23	1048	
		tmp	25	1052	

mogrify second line (return) executed again

Method	Line	Var	Value	Addr	Notes
main()	19	a	25	1024	
		y	17	1028	
		mog	23	1032	
		x	8.5	1036	double variable

`mogrify` completes assigning its result to local variable `a` in `main` and popping its stack frame off.

```
11: int main(){
12:   int a = 7, y = 17;                        // First line of
main (main)
13:   int mog = mogrify(a,y);                  // Call to mogrify
(mogrify_call)
14:   printf("Done with mogrify\n");           // (first_print)
15:
16:    double x = truly_half(y);                     // Call to
truly_half (truly_half_call)
17:   printf("Done with truly_half\n");        // (second_print)
18:
19:   a = mogrify(x,mog);                      // (mogrify2)
20:
21:   printf("Results: %d %lf\n",mog,x);       // (last_print)
22:   return 0;                                // (main_return)
23: }
```

The program completes by printing the final results

```
Results: 23 8.500000
```

and returning 0. Two things to note here:

- `main` is also a function which returns meaning its stack frame will be popped off and control will be returned to the mysterious and powerful **C runtime system** which is responsible for setting up `main` to run in the first place.
- The integer return value from `main` is passed back to whatever entity ran the program in the first place.

Example 2: Source Code

```
 1: #include <stdio.h>
 2:
 3: double f1(double x){
 4:   return x+1.0;                  // (f1_1)
 5: }
 6:
 7: double f2(double x){
 8:   double tmp = f1(x);            // (f2_1)
 9:   double z = f1(x+1);            // (f2_2)
10:   return (z+ tmp) / 2;           // (f2_3)
11: }
12:
13: double f3(double x, double y){
14:   double z = f1(1);              // (f3_1)
15:   double tmp1 = x*z;             // (f3_2)
16:   double tmp2 = f2(y);           // (f3_3)
17:   return tmp1+tmp2;              // (f3_4)
18: }
19:
20: int main(){
21:   double x = 2;                  // (main_1)
22:   double y = f3(x, x+3);         // (main_2)
23:   printf("%.3lf\n",y);           // (main_3)
24:   return 0;                      // (main_4)
25: }
```

Call stack behavior

This is a somewhat more involved example. The computation doesn't do anything particularly interesting but is a good demonstration of how the stack can grow and shrink as one function calls another function. Little explanation is given in each step so pay careful attention to which line is being executed and remember that after a `return` is executed, control returns to the previous function and a stack frame is popped off.
Two minor notes

- All of the variables in this example are `double` so are assumed to be 8 bytes which means the size of memory cells will differ by 8
- The stack starts at memory address 2048 this time. In examples it is fairly arbitrary where the frame for `main` starts in the stack but after specifying its location, the behavior of the program follows a well-defined patter.

```
20: int main(){
21:   double x = 2;                  // (main_1)
22:   double y = f3(x, x+3);         // (main_2)
23:   printf("%.3lf\n",y);           // (main_3)
24:   return 0;                      // (main_4)
25: }
```

main() called

Method	Line	Var	Value	Addr	Notes
main()	21	x	?	2048	
		y	?	2056	

main() line 1 finished

Method	Line	Var	Value	Addr	Notes
main()	22	x	2.0	2048	
		y	?	2056	

f3 called

Method	Line	Var	Value	Addr	Notes
main()	22	x	2.0	2048	
		y	?	2056	
f3()	14	x	2.0	2064	
		y	5.0	2072	
		z	?	2080	
		tmp1	?	2088	
		tmp2	?	2096	

```
13: double f3(double x, double y){
14:   double z = f1(1);              // (f3_1)
15:   double tmp1 = x*z;             // (f3_2)
16:   double tmp2 = f2(y);           // (f3_3)
17:   return tmp1+tmp2;              // (f3_4)
18: }
19:
```

f3 calls f1 on line 1

Method	Line	Var	Value	Addr	Notes
main()	22	x	2.0	2048	
		y	?	2056	
f3()	14	x	2.0	2064	
		y	5.0	2072	
		z	?	2080	
		tmp1	?	2088	
		tmp2	?	2096	
f1	4	x	1.0	2104	

```
3: double f1(double x){
 4:   return x+1.0;                    // (f1_1)
 5: }

13: double f3(double x, double y){
14:   double z = f1(1);                // (f3_1)
15:   double tmp1 = x*z;               // (f3_2)
16:   double tmp2 = f2(y);             // (f3_3)
17:   return tmp1+tmp2;                // (f3_4)
18: }
```

f1 returns, stack frame popped, f3 advances to next line

Method	Line	Var	Value	Addr	Notes
main()	22	x	2.0	2048	
		y	?	2056	
f3()	15	x	2.0	2064	
		y	5.0	2072	
		z	2.0	2080	
		tmp1	?	2088	
		tmp2	?	2096	

f3 executes second line

Method	Line	Var	Value	Addr	Notes
main()	22	x	2.0	2048	
		y	?	2056	
f3()	16	x	2.0	2064	
		y	5.0	2072	
		z	2.0	2080	
		tmp1	4.0	2088	
		tmp2	?	2096	

```
3: double f1(double x){
 4:   return x+1.0;                    // (f1_1)
 5: }
 6:
 7: double f2(double x){
 8:   double tmp = f1(x);              // (f2_1)
 9:   double z = f1(x+1);              // (f2_2)
10:   return (z+ tmp) / 2;             // (f2_3)
11: }
12:
13: double f3(double x, double y){
14:   double z = f1(1);                // (f3_1)
15:   double tmp1 = x*z;               // (f3_2)
16:   double tmp2 = f2(y);             // (f3_3)
17:   return tmp1+tmp2;                // (f3_4)
18: }
```

f3 calls f2 on third line

Method	Line	Var	Value	Addr	Notes
main()	22	x	2.0	2048	
		y	?	2056	
f3()	16	x	2.0	2064	
		y	5.0	2072	
		z	2.0	2080	
		tmp1	4.0	2088	
		tmp2	?	2096	

f2()	8	x	5.0	2104	
		tmp	?	2112	
		z	?	2120	

```
3: double f1(double x){
 4:   return x+1.0;                     // (f1_1)
 5: }
 6:
 7: double f2(double x){
 8:   double tmp = f1(x);               // (f2_1)
 9:   double z = f1(x+1);               // (f2_2)
10:   return (z+ tmp) / 2;              // (f2_3)
11: }
12:
```

f2 calls f1 on first line

Method	Line	Var	Value	Addr	Notes
main()	22	x	2.0	2048	
		y	?	2056	
f3()	16	x	2.0	2064	
		y	5.0	2072	
		z	2.0	2080	
		tmp1	4.0	2088	
		tmp2	?	2096	
f2()	8	x	5.0	2104	
		tmp	?	2112	
		z	?	2120	
f1()	4	x	5.0	2128	

f1 returns to f2, f2 advances to next line

Method	Line	Var	Value	Addr	Notes
main()	22	x	2.0	2048	
		y	?	2056	
f3()	16	x	2.0	2064	
		y	5.0	2072	
		z	2.0	2080	

		tmp1	4.0	2088	
		tmp2	?	2096	
f2()	9	x	5.0	2104	
		tmp	6.0	2112	
		z	?	2120	

```
3: double f1(double x){
 4:   return x+1.0;                  // (f1_1)
 5: }
 6:
 7: double f2(double x){
 8:   double tmp = f1(x);            // (f2_1)
 9:   double z = f1(x+1);            // (f2_2)
10:   return (z+ tmp) / 2;           // (f2_3)
11: }
12:
```

f2 calls f1 on second line

Method	Line	Var	Value	Addr	Notes
main()	22	x	2.0	2048	
		y	?	2056	
f3()	16	x	2.0	2064	
		y	5.0	2072	
		z	2.0	2080	
		tmp1	4.0	2088	
		tmp2	?	2096	
f2()	9	x	5.0	2104	
		tmp	6.0	2112	
		z	?	2120	
f1()	4	x	6.0	2128	

f1 returns to f2, f2 advances to next line

Method	Line	Var	Value	Addr	Notes
main()	22	x	2.0	2048	
		y	?	2056	

f3()	16	x	2.0	2064	
		y	5.0	2072	
		z	2.0	2080	
		tmp 1	4.0	2088	
		tmp 2	?	2096	
f2()	10	x	5.0	2104	
		tmp	6.0	2112	
		z	7.0	2120	

f2 returns control to f3, f3 advances to next line

Method	Line	Var	Value	Addr	Notes
main()	22	x	2.0	2048	
		y	?	2056	
f3()	17	x	2.0	2064	
		y	5.0	2072	
		z	2.0	2080	
		tmp1	4.0	2088	
		tmp2	6.5	2096	

f3 returns control to main, main advances to next

Method	Line	Var	Value	Addr	Notes
main()	23	x	2.0	2048	
		y	10.5	2056	

main calls printf on line 3

Method	Line	Var	Value	Addr	Notes
main()	23	X	2.0	2048	
		Y	10.5	2056	
printf()	library call	fmt	?	2064	4-byte pointer
		?	10.5	2068	

Output

```
10.500
```

printf returns, main advances to next line

Method	Line	Var	Value	Addr	Notes
main()	24	x	2.0	2048	
		y	10.5	2056	

main returns 0, program ends

Terminology of Function Call Stack

An area of program memory that is used to manage function calls. Program memory can be roughly divided into 4 parts:

- the function call stack which supports functions
- the heap (or store) which supports dynamic memory allocation,
- the global variable area which stores values for global variables
- the code area which stores the actual instructions for the running program which are usually not changed during execution

stack frame
A portion of memory in the function call stack associated with a single running function. Functions that have many parameters and local variables will have larger stack frames than those with few parameters and local variables. The compiler is able to determine during compilation the stack frame size for a function. There are usually as many stack frames on the stack as there are functions that have yet to return.

pushing a frame
When a function is called, a new frame is pushed onto the "top" of the function call stack. The frame will be big enough to hold all of the parameters to the function being called plus the space required for all local variables in a function.

popping a frame
When a function finishes executing, it returns control to the function it which called/invoked it. The frame associated with the returning function is removed from the top of the stack.

active function / frame
In serial programs (non-parallel programs), there is only one active function at any given moment in the program. This function is usually the one at the "top" of the function call stack and the active frame refers to the frame at the top of the call stack. Pushing and popping happen only at the top of the stack and change the active frame.

stack overflow
Memory is a finite resource and if too many functions are called before any returns, a program can run out of space on the stack. This usually happens only in the case of recursive functions that are misbehaving but for programs with very tight memory constraints it may happen in some kinds of programs.

Author: Chris Kauffman (kauffman@cs.gmu.edu)
Date: 2015-06-17 Wed 16:26

Recursion

Some computer programming languages allow a module or function to call itself. This technique is known as recursion. In recursion, a function **α** either calls itself directly or calls a function **β** that in turn calls the original function **α**. The function **α** is called recursive function.

Example − a function calling itself.

```
int function(int value) {
  if(value > max)
    return;
  function(value + 1);

  printf("%d ",value);
}
```

Example − a function that calls another function which in turn calls it again.

```
int function1(int value1) {
  if(value1 < 1)
    return;
  function2(value1 - 1);
  printf("%d ",value1);
}
int function2(int value2) {
  function1(value2);
}
```

Properties

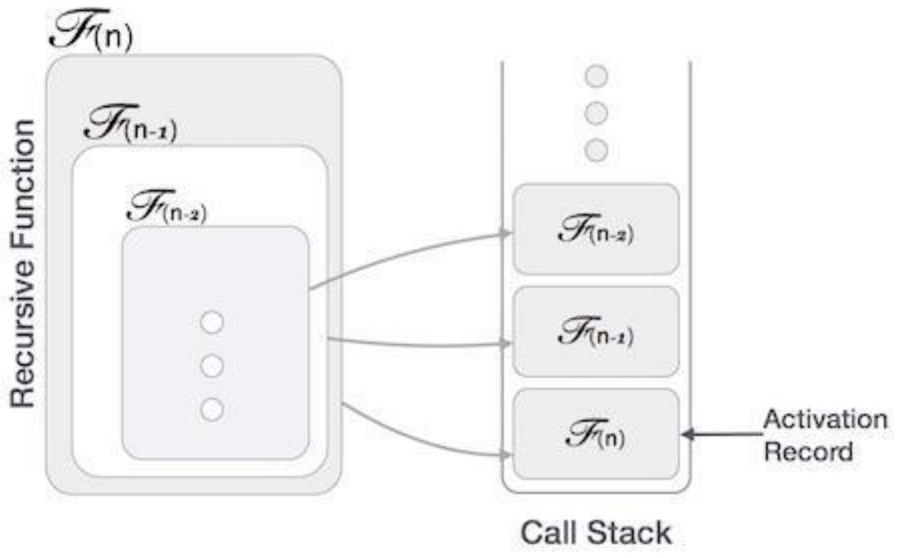
A recursive function can go infinite like a loop. To avoid infinite running of recursive function, there are two properties that a recursive function must have −

- **Base criteria** − There must be at least one base criteria or condition, such that, when this condition is met the function stops calling itself recursively.
- **Progressive approach** − The recursive calls should progress in such a way that each time a recursive call is made it comes closer to the base criteria.

Implementation

Many programming languages implement recursion by means of **stacks**. Generally, whenever a function (**caller**) calls another function (**callee**) or itself as callee, the caller function transfers execution control to the callee. This transfer process may also involve some data to be passed from the caller to the callee.

This implies, the caller function has to suspend its execution temporarily and resume later when the execution control returns from the callee function. Here, the caller function needs to start exactly from the point of execution where it puts itself on hold. It also needs the exact same data values it was working on. For this purpose, an activation record (or stack frame) is created for the caller function.



This activation record keeps the information about local variables, formal parameters, return address and all information passed to the caller function.

Analysis of Recursion

One may argue why to use recursion, as the same task can be done with iteration.

The first reason is, recursion makes a program more readable and because of latest enhanced CPU systems, recursion is more efficient than iterations.

Time Complexity

- In case of iterations, we take number of **iterations to count the time complexity**.
- Likewise, in case of recursion, assuming everything is constant, we try to figure out the number of times a recursive call is being made.
- A call made to a function is O(1), hence the (n) number of times a recursive call is made makes the recursive function O(n).

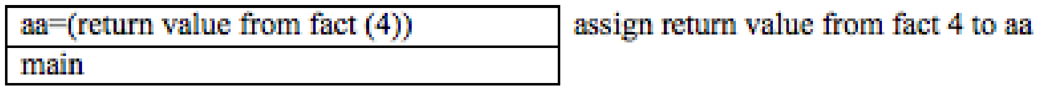
Space Complexity

- Space complexity is counted as what amount of extra space is required for a module to execute.
- In case of iterations, the compiler hardly requires any extra space. The compiler keeps updating the values of variables used in the iterations.
- But in case of recursion, the system needs to store activation record each time a recursive call is made.
- Hence, it is considered that space complexity of recursive function may go higher than that of a function with iteration.

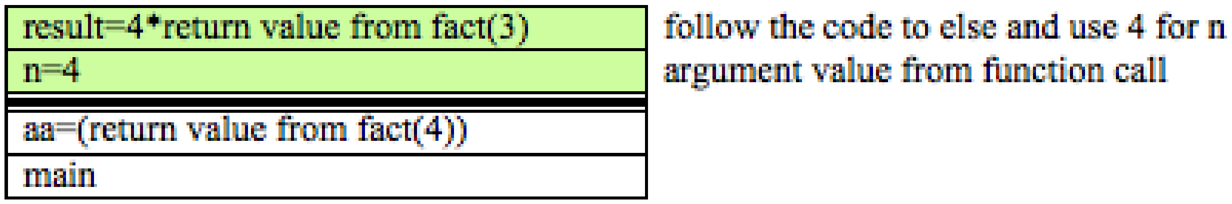
1. Factorial of N

```
fact(n)
1. if(n==1 || n==0)……[can write n<=1 also….check for base case]
        rc=1;             //if n is 1 or 0 rc is 1
   else{                 //else it is recursive case
        rc=n * fact(n-1);  //rc is n * (n-1)!
 2. return rc;
}

main()

1 aa = fact(4);
2. Print aa
```

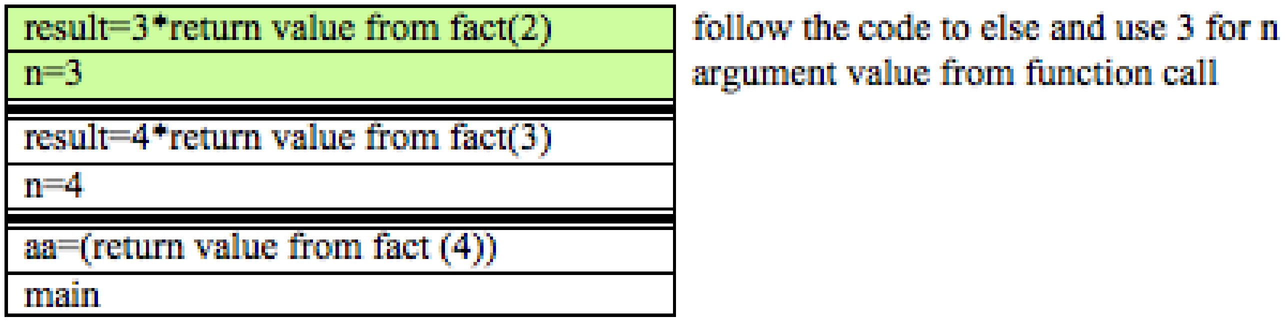
When the program starts this is our run time stack:


aa=(return value from fact (4))
main
assign return value from fact 4 to aa

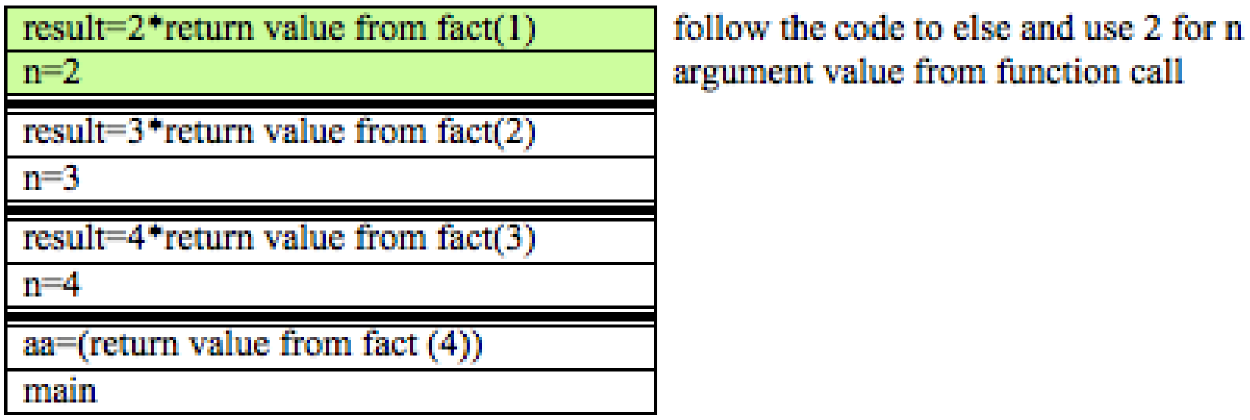
fact(4) means that we call function fact() with 4 as argument so push that information onto the stack


result=4*return value from fact(3)
n=4
aa=(return value from fact(4))
main
follow the code to else and use 4 for n argument value from function call

Since we make a function call to fact(3) we must now push that information onto the stack also. Note, return statement not reached before calling fact(3) so stack isn't popped at this point


result=3*return value from fact(2)
n=3
result=4*return value from fact(3)
n=4
aa=(return value from fact (4))
main
follow the code to else and use 3 for n argument value from function call

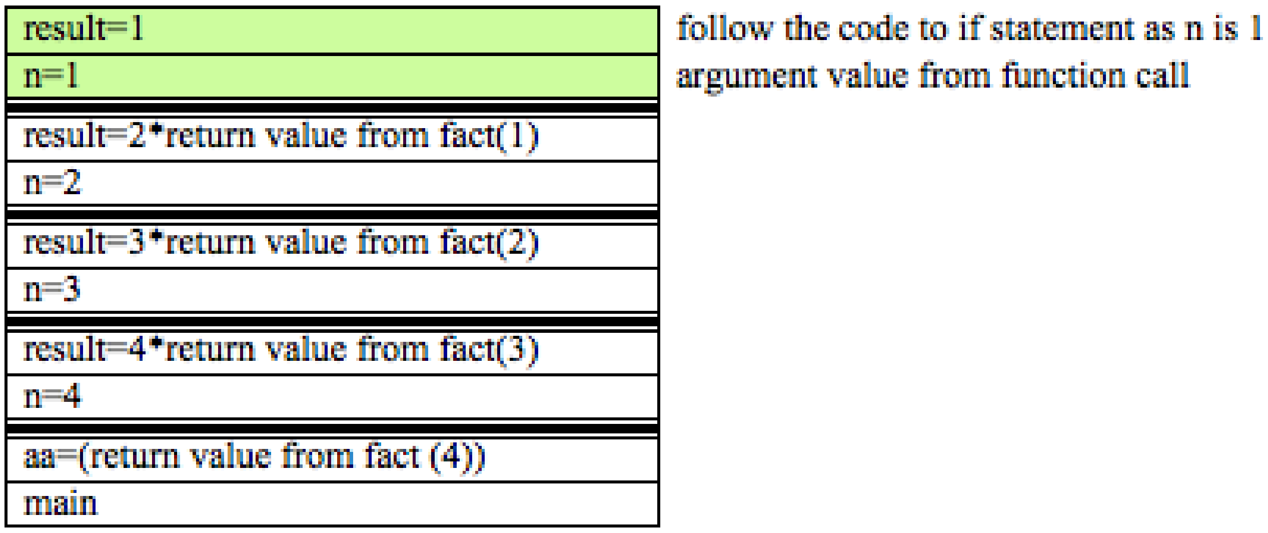
Again, because we are calling the function with fact(2) , we must also push this onto the stack



result=2*return value from fact(1)
n=2
result=3*return value from fact(2)
n=3
result=4*return value from fact(3)
n=4
aa=(return value from fact (4))
main

follow the code to else and use 2 for n argument value from function call

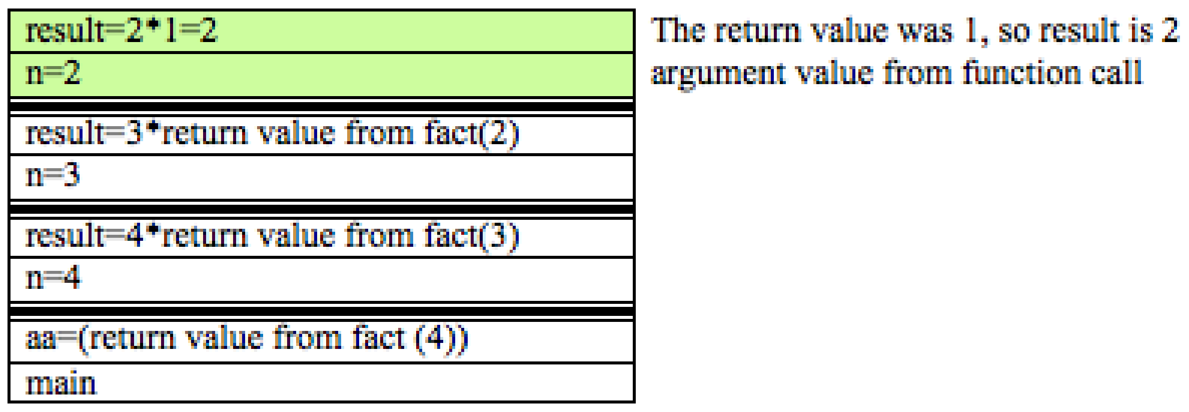
Once again, because we are calling the function with fact(1) we must also push this onto the stack



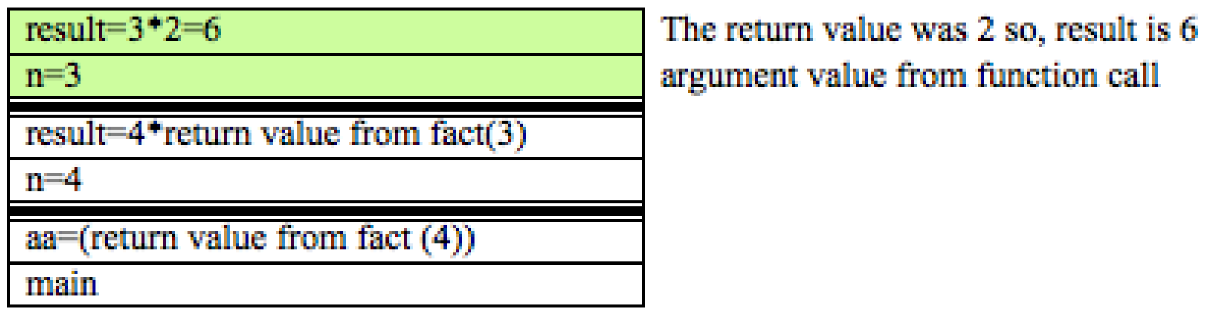
result=1
n=1
result=2*return value from fact(1)
n=2
result=3*return value from fact(2)
n=3
result=4*return value from fact(3)
n=4
aa=(return value from fact (4))
main

follow the code to if statement as n is 1 argument value from function call

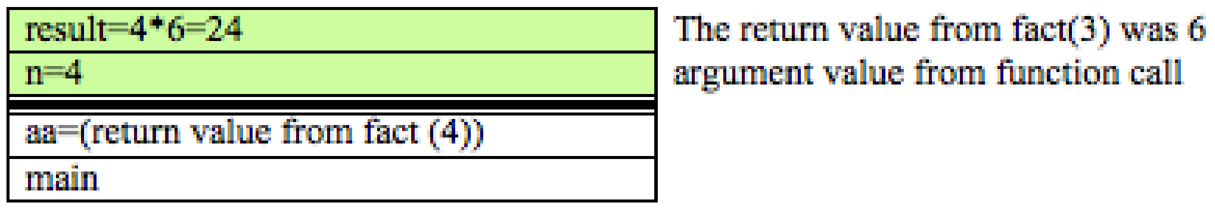
If we follow the code at this point we can see that we are able to reach the return statement without further function calls. Thus, we can now pop the stack.


result=2*1=2
n=2
result=3*return value from fact(2)
n=3
result=4*return value from fact(3)
n=4
aa=(return value from fact (4))
main
The return value was 1, so result is 2
argument value from function call

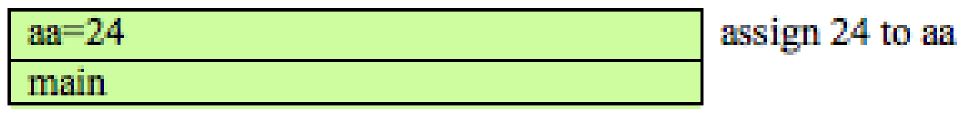
This will lead to the completion of the topmost function call and again, it can be removed from the stack. This time, the return value is used to solve the problem one more layer above:


result=3*2=6
n=3
result=4*return value from fact(3)
n=4
aa=(return value from fact (4))
main
The return value was 2 so, result is 6
argument value from function call

This will lead to the completion of the topmost function call and again, it can be removed from the stack. This time, the return value is used to solve the problem one more layer above:


result=4*6=24
n=4
aa=(return value from fact (4))
main
The return value from fact(3) was 6
argument value from function call

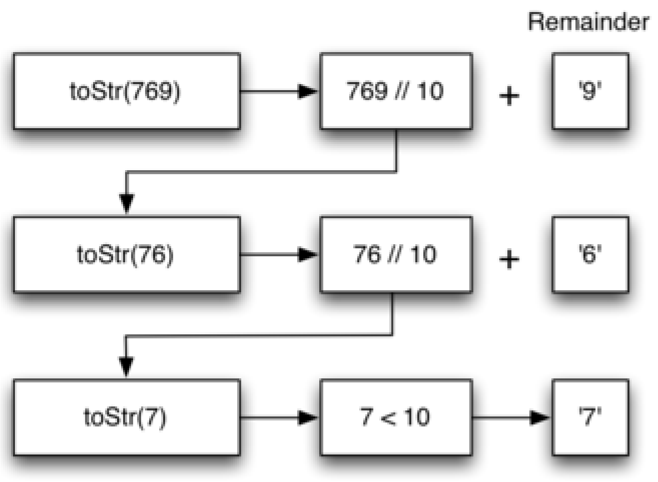
Finally, we can go back to main, as we have reached the final result:



Practice Problem:

1. Sum of digits

- ❖ **Write Algorithm**



Initially S = “ “
N = Number
B = Base

Display digits considering base = 10

```
Convert((N)
1. if (N<10) print (N) return
 2. Convert(N/10)
3. print (N%10)
4. Return
```

Display digit considering base = 2 (N is a binary number)
N/base
N%base

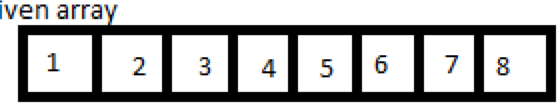
//**Sum of all digits**
Sum_digit = n%10 + sum_digit(n/10)

Sample table for simulation or “dry run”

N	O/P	Stack	Line
789			

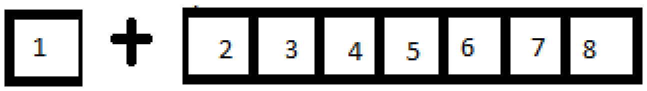
2. Sum of first n elements using recusion :

Given array



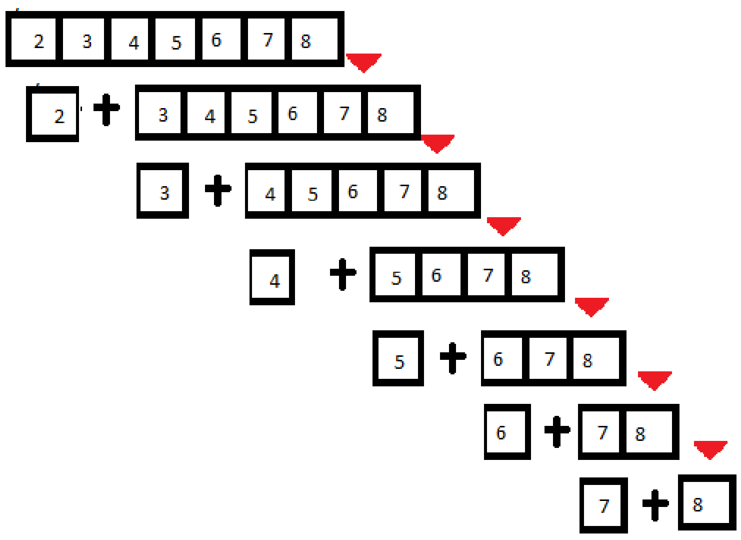
We can easily deduce that sum of array is equal to :

First element + sum of the remaining elements of the



array i.e

Here we get two parts: **first element+ an array of rest of elements**.now the second part itself is an array whose sum is again **its first element +sum of the remaining elements of the array**. and so on.



//For Sum of n elements of an array

Int S[10]

For(i=0;i<10;i++)
Sum+=s[i]
S= 1020
S +1 1024
S+2 1028

S+3 1032

Sum = *s + sum(s+1)

Sum of first n numbers
Sum(n)
{if   n <= 1 return 0
S = n + sum(n-1)
	Sum of any n elements in an array
	Sum (s)
	{if s[0] == -1
	      Retun 0
	S = S[0] + sum(s++)
		Input Empty array s[0] = -1

All recursive algorithms must obey three important laws:

1. A recursive algorithm must have a **base case**.
2. A recursive algorithm must change its state and move toward the base case.
3. A recursive algorithm must call itself, recursively.

Issues:

- Issue of Infinite loop if improper base condition may result in stack overflow
- Or with insufficient memory it may end up with stack overflow

Convert Recursion into iteration :

If we wanted to convert an algorithm from a recursive form to an iterative form, we could simulate this process with our own stacks.

Any recursive algorithm can be converted to an iterative form by using a stack to capture the "history" of

- actual parameters
- local variables

that would have been placed on the activation stack.

The general idea is:

- recursive calls get replaced by push
    - depending on details, may push new values, old values, or both
- returns from recursive calls get replaced by pop
- main calculation of recursive routine gets put inside a loop
    - at start of loop, set variables from stack top and pop the stack

Example Iterative procedure for Factorial using User defined stack

```
push(s, top, ele)

1    if (top >= MAX) print “Stack Overflow"
2    else  s[++top] = ele;
 3   return top;
```

```
pop(a, top)

 1  if (top >= 0)  top = top - 1;
 2     return a[(*top) + 1];
3    else
                 print "Stack underflow”
 4     return 0;
```

```
main()
```

```
1 TOP = -1; // stack variables that mainntain stack's top
2 Input n
3 if(n<=0) print “The number can not be less than 0", return
4  I = n
5  repeat step 6 while i >0 , step -1 //Push all the elements in
the stack
6  TOP = push(s,TOP, i);
  // now pop all the elements one by one
  // multiply them with the answer variable
7     Repeat step 8  while(TOP>=0)
8.                  ans = ans * pop(s,TOP);
 9. Print ,ans;
```

Self Study

Tower of Hanoi
https://www.freecodecamp.org/news/analyzing-the-algorithm-to-solve-the-tower-of-hanoi-problem-686685f032e3/

https://www.tutorialspoint.com/data_structures_algorithms/tower_of_hanoi.htm#:~:text=Tower%20of%20Hanoi%20puzzle%20with%20n%20disks%20can%20be%20solved,3%20%2D%201%20%3D%207%20steps.