# Recursion in ARM

## Factorial Function

The register R0 is always used to pass the first arguement in a function, however since we are using recursion, we are reusing R0 to pass n*fact(n-1). Thus the compiler moves R0 to R4. Thus contents of R4 is also pushed onto stack

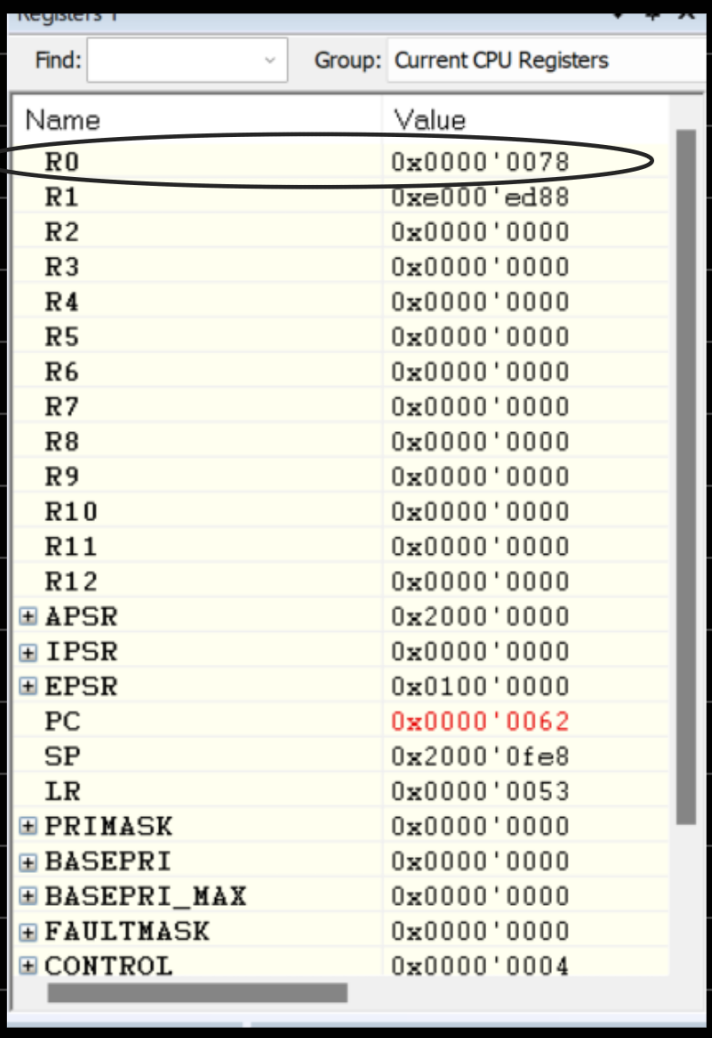
```
                0x5c: 0xb51c          PUSH      {R2-R4, LR}
  x = fact(0);
        0x5e: 0x2000          MOVS      R0, #0
        0x60: 0xf7ff 0xffee   BL        fact
```

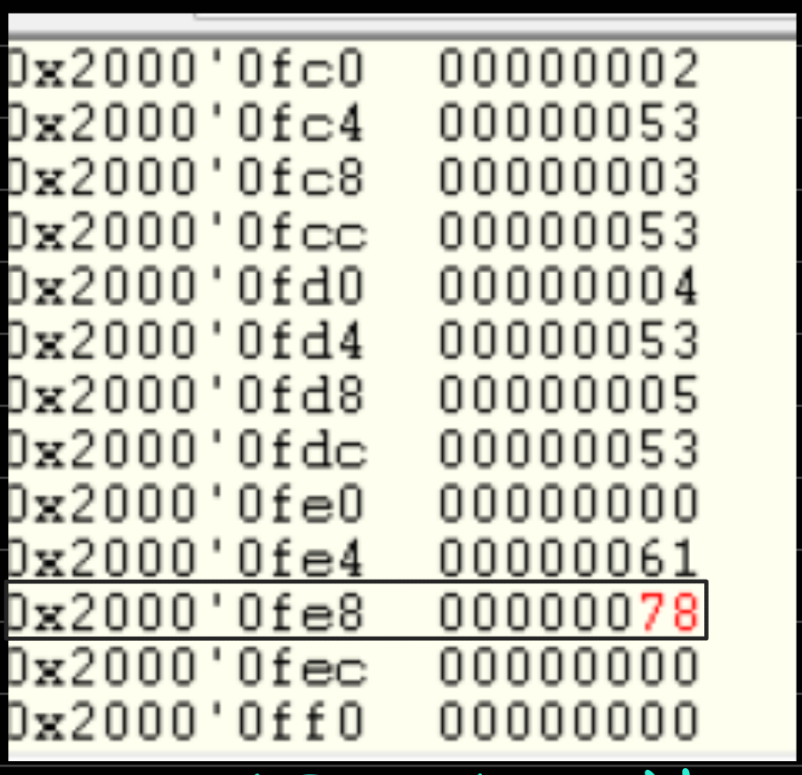
0 is passed via R0

```
                fact:
        0x40: 0xb510          PUSH      {R4, LR}
        0x42: 0x0004          MOVS      R4, R0
  if(n == 1 || n == 0)
```

content of R4 is saved onto stack before R0 content is passed to R4

After the recursion call is finished the growing stack shrinks down to the base address which now only contains the arguement to be returned (0x78 or 120 which is 5 factorial). This arguement is returned via R0:

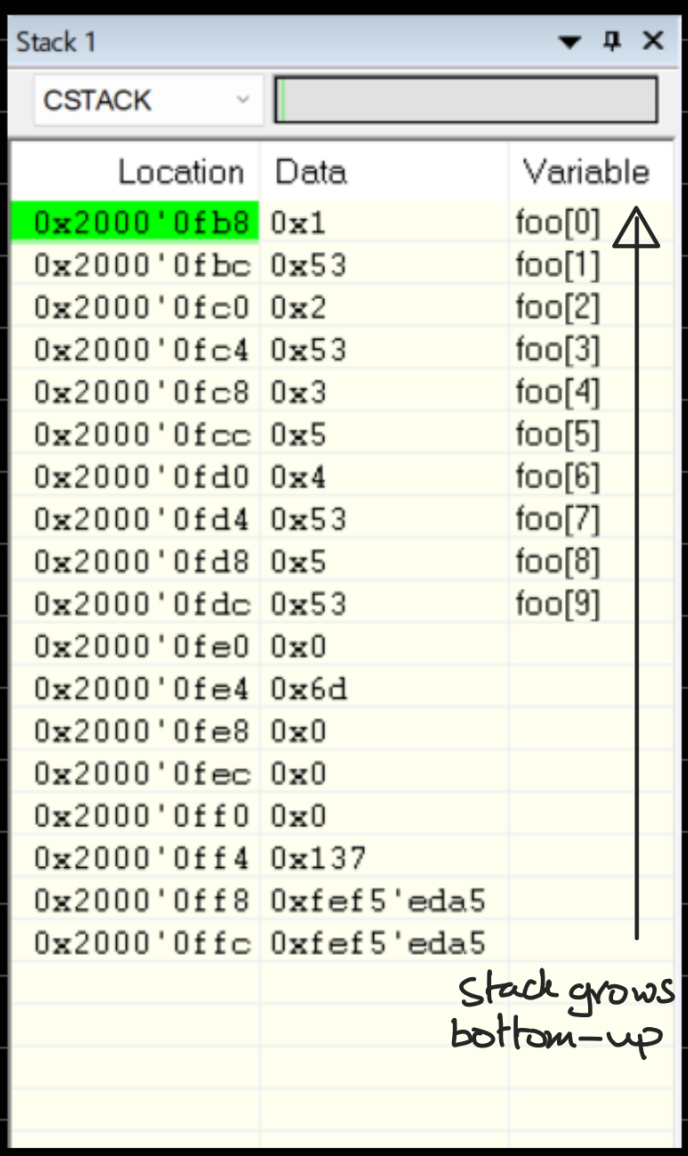



0x20000fe8 was base address of stack pointer. The LR address is also popped to the PC register to get back to the code just after the first function call.

## The "stack 1" view

Allows to see the entire stack. The bottom address is the begining address of the stack, and it grows bottom up :


# Stack Overflow

```
unsigned fact(unsigned n)
{
 unsigned foo[10];
 foo[n] = n;

 if(n == 0u)
 {
  return 1u;
 }
 return foo[n]*fact(n - 1u)
}
```

→ function being executed. This gets saved onto the stack.

The passed arguement is n=5 and hence foo[5] contains 0x5 while all other foo's contain garbage values.

| Location | Data | Variab |
|---|---|---|
| 0x2000'0fb8 | 0x1 | foo[0] |
| 0x2000'0fbc | 0x53 | foo[1] |
| 0x2000'0fc0 | 0x2 | foo[2] |
| 0x2000'0fc4 | 0x53 | foo[3] |
| 0x2000'0fc8 | 0x3 | foo[4] |
| 0x2000'0fcc | 0x5 | foo[5] |
| 0x2000'0fd0 | 0x4 | foo[6] |
| 0x2000'0fd4 | 0x53 | foo[7] |
| 0x2000'0fd8 | 0x5 | foo[8] |
| 0x2000'0fdc | 0x53 | foo[9] |
| 0x2000'0fe0 | 0x0 | |
| 0x2000'0fe4 | 0x6d | |
| 0x2000'0fe8 | 0x0 | |
| 0x2000'0fec | 0x0 | |
| 0x2000'0ff0 | 0x0 | |
| 0x2000'0ff4 | 0x137 | |
| 0x2000'0ff8 | 0xfef5'eda5 | |
| 0x2000'0ffc | 0xfef5'eda5 | |

If we 'step into' our code again then we see the stack grow again due to foo and only foo[4] contains correct value which is 0x4. The rest of the foo's are again garbage :

| | | |
|---|---|---|
| 0x2000'0f88 | 0xd004'4f60 | foo[0] |
| 0x2000'0f8c | 0xf04f'4658 | foo[1] |
| 0x2000'0f90 | 0xf7ff'4160 | foo[2] |
| 0x2000'0f94 | 0x9b0d'fd27 | foo[3] |
| 0x2000'0f98 | 0x4 | foo[4] |
| 0x2000'0f9c | 0x4650'd059 | foo[5] |
| 0x2000'0fa0 | 0x4160'f04f | foo[6] |
| 0x2000'0fa4 | 0xf1b8'e053 | foo[7] |
| 0x2000'0fa8 | 0xd003'0fef | foo[8] |
| 0x2000'0fac | 0x3fe'f24e | foo[9] |
| 0x2000'0fb0 | 0x5 | |
| 0x2000'0fb4 | 0x5d | |
| 0x2000'0fb8 | 0x1 | foo[0] |
| 0x2000'0fbc | 0x53 | foo[1] |
| 0x2000'0fc0 | 0x2 | foo[2] |
| 0x2000'0fc4 | 0x53 | foo[3] |
| 0x2000'0fc8 | 0x3 | foo[4] |
| 0x2000'0fcc | 0x5 | foo[5] |
| 0x2000'0fd0 | 0x4 | foo[6] |
| 0x2000'0fd4 | 0x53 | foo[7] |
| 0x2000'0fd8 | 0x5 | foo[8] |
| 0x2000'0fdc | 0x53 | foo[9] |
| 0x2000'0fe0 | 0x0 | |
| 0x2000'0fe4 | 0x6d | |

The stack growing this much is an instance of possible stack overflow. We can demonstrate this further by setting foo[] size to 100 (set a breakpoint at return foo[n]*(n-1) statement).

First we change to the "call stack" view in IAR. This shows all function calls currently nested on stack. "main" itself is called from the stack as shown by [_call_main +0x9], this place contains startup code.

```
main
[_call_main + 0xd]
```

for Fact(10) and foo[100], stack overflow occurs after 7 calls to function fact. Here program freezes and the stack pointer is below the valid RAM in ARM (it is at 0x1ffffff90 while RAM starts at 0x20000000).

Stack 1

CSTACK

| Location | Data | Variable |
|---|---|---|
| 0x1fff'ff90 | – | |
| 0x1fff'ff94 | – | |
| 0x1fff'ff98 | – | |
| 0x1fff'ff9c | – | |
| 0x1fff'ffa0 | – | |
| 0x1fff'ffa4 | – | |
| 0x1fff'ffa8 | – | |
| 0x1fff'ffac | – | |
| 0x1fff'ffb0 | – | |
| 0x1fff'ffb4 | – | |
| 0x1fff'ffb8 | – | |
| 0x1fff'ffbc | – | |
| 0x1fff'ffc0 | – | |

Registers 1

Find: Group: Current CPU Register

| Name | Value |
|---|---|
| R9 | 0x0000'0000 |
| R10 | 0x0000'0000 |
| R11 | 0x0000'0000 |
| R12 | 0x0000'0000 |
| ⊞ APSR | 0x2000'0000 |
| ⊞ IPSR | 0x0000'0003 |
| ⊞ EPSR | 0x0100'0000 |
| PC | 0x0000'011a |
| SP | 0x1fff'ff90 |
| LR | 0xffff'ffe9 |
| ⊞ PRIMASK | 0x0000'0000 |
| ⊞ BASEPRI | 0x0000'0000 |
| ⊞ BASEPRI_MAX | 0x0000'0000 |

A look at dissassembly tells us that the code is at "BusFault_Handler" and this is an interrupt by ARM used to stop program flow when non-existant memory. This is an endless loop.

We can change this stack size in IAR by ticking "overide default" in config tab in project options. We can edit the stack (cstack) by clicking the edit button below and going to stack/heap option. We can also set heap to 0 since it is a bad idea to use in embedded systems.

## Another example for Stack Overflow

Set foo size to foo[6] and call fact with arguement 7 [fact(7)]. Set a breakpoint at fact(7) and single step from there. After the usual push R4, LR to stack command there is a SUB SP, SP, #0x18 performed

```
unsigned fact(unsigned n)
{
                fact:
        0x40: 0xb510        PUSH    {R4, LR}
        0x42: 0xb086        SUB     SP, SP, #0x18
  foo[n] = n;
        0x44: 0x4669        MOV     R1, SP
        0x46: 0xf841 0x0020 STR.W   R0, [R1, R0, LS
  if(n == 0u)
        0x4a: 0x2800        CMP     R0, #0
        0x4c: 0xd101        BNE.N   ??fact_0
    return 1u;
```

This SUB instruction subtracts 0x18 from our stack address and makes stack larger to accomodate for foo[6] array

Stack 1 — CSTACK

| Location | Data | Variable |
|---|---|---|
| 0x2000'03e0 | 0x0 | |
| 0x2000'03e4 | 0x6d | |
| 0x2000'03e8 | 0x0 | |
| 0x2000'03ec | 0x0 | |
| 0x2000'03f0 | 0x0 | |
| 0x2000'03f4 | 0x137 | |
| 0x2000'03f8 | 0xfef5'eda5 | |
| 0x2000'03fc | 0xfef5'eda5 | |

Registers 1 — Group: Current CPU Registers

| Name | Value |
|---|---|
| R12 | 0x0000'0000 |
| APSR | 0x2000'0000 |
| IPSR | 0x0000'0000 |
| EPSR | 0x0100'0000 |
| PC | 0x0000'0042 |
| SP | 0x2000'03e0 |
| LR | 0x0000'006d |
| PRIMASK | 0x0000'0000 |

Disassembly

```
unsigned fact(unsigned n)
{
                fact:
        0x40: 0xb510        PUSH    {R4, LR}
        0x42: 0xb086        SUB     SP, SP, #0x18
  foo[n] = n;
        0x44: 0x4669        MOV     R1, SP
        0x46: 0xf841 0x0020 STR.W   R0, [R1, R0, LSL
```

Stack pointer at 0x2000 03e0 goes to 0x2000 03c8 to make room for foo[6] (-0x018)

Stack 1 — CSTACK

| Location | Data | Variable |
|---|---|---|
| 0x2000'03c8 | 0x1188 | foo[0] |
| 0x2000'03cc | 0x4 | foo[1] |
| 0x2000'03d0 | 0x0 | foo[2] |
| 0x2000'03d4 | 0x7c00'0000 | foo[3] |
| 0x2000'03d8 | 0x5901'8400 | foo[4] |
| 0x2000'03dc | 0xffff | foo[5] |
| 0x2000'03e0 | 0x0 | |
| 0x2000'03e4 | 0x6d | |
| 0x2000'03e8 | 0x0 | |
| 0x2000'03ec | 0x0 | |
| 0x2000'03f0 | 0x0 | |
| 0x2000'03f4 | 0x137 | |
| 0x2000'03f8 | 0xfef5'eda5 | |

Registers 1 — Group: Current CPU Registers

| Name | Value |
|---|---|
| R12 | 0x0000'0000 |
| APSR | 0x2000'0000 |
| IPSR | 0x0000'0000 |
| EPSR | 0x0100'0000 |
| PC | 0x0000'0044 |
| SP | 0x2000'03c8 |
| LR | 0x0000'006d |
| PRIMASK | 0x0000'0000 |
| BASEPRI | 0x0000'0000 |
| BASEPRI_MAX | 0x0000'0000 |
| FAULTMASK | 0x0000'0000 |
| CONTROL | 0x0000'0004 |
| CYCLECOUNTER | 0 |

Disassembly

```
unsigned fact(unsigned n)
{
                fact:
        0x40: 0xb510        PUSH    {R4, LR}
        0x42: 0xb086        SUB     SP, SP, #0x18
  foo[n] = n;
        0x44: 0x4669        MOV     R1, SP
        0x46: 0xf841 0x0020 STR.W   R0, [R1, R0, LSL
  if(n == 0u)
        0x4a: 0x2800        CMP     R0, #0
        0x4c: 0xd101        BNE.N   ??fact_0
    return 1u;
        0x4e: 0x2001        MOVS    R0, #1
```

Next, the MOV instruction moves SP address to R1 and the STR R0, [R1, R0, LSL #2] instruction shifts the arguement passed to our function (and still stored in R0) to the R0 register again. LSL is logical shift left

However our index n=7 is beyond the array bound of foo[5]. Hence foo[n]=n will go two places beyond foo[5]'s address (0x2000 03dc) to 0x2000 03e4 which contains our link register data. This corrupts LR data and the stack.

To troubleshoot this, add a breakpoint at the return address of function (just before POP instruction);

```
Disassembly
        0x42: 0xb086          SUB       SP, SP, #0x18
  foo[n] = n;
        0x44: 0x4669          MOV       R1, SP
        0x46: 0xf841 0x0020   STR.W     R0, [R1, R0, LSI
  if(n == 0u)
        0x4a: 0x2800          CMP       R0, #0
        0x4c: 0xd101          BNE.N     ??fact_0
    return 1u;
        0x4e: 0x2001          MOVS      R0, #1
        0x50: 0xe006          B.N       ??fact_1
  return foo[n]*fact(n - 1u);
                ??fact_0:
        0x52: 0xf851 0x4020   LDR.W     R4, [R1, R0, LSI
        0x56: 0x1e40          SUBS      R0, R0, #1
        0x58: 0xf7ff 0xfff2   BL        fact
        0x5c: 0x4344          MULS      R4, R0, R4
        0x5e: 0x0020          MOVS      R0, R4
                ??fact_1:
        0x60: 0xb006          ADD       SP, SP, #0x18
        0x62: 0xbd10          POP       {R4, PC}
int main()
{
                main:
        0x64: 0xb51c          PUSH      {R2-R4, LR}
```

at this breakpoint all calls are complete and will recursively unwind from here.

Stack 1

CSTACK

| Location | Data | Variable |
|---|---|---|
| 0x2000'02e8 | 0x0 | foo[0] |
| 0x2000'02ec | 0xb001'dbf8 | foo[1] |
| 0x2000'02f0 | 0xbf00'4770 | foo[2] |
| 0x2000'02f4 | 0xf'4240 | foo[3] |
| 0x2000'02f8 | 0x5188'f64e | foo[4] |
| 0x2000'02fc | 0x100'f2ce | foo[5] |
| 0x2000'0300 | 0x1 | |
| 0x2000'0304 | 0x5d | |
| 0x2000'0308 | 0x8f4f'f3bf | foo[0] |
| 0x2000'030c | 0x1 | foo[1] |
| 0x2000'0310 | 0x7001'f04f | foo[2] |
| 0x2000'0314 | 0xa10'eee1 | foo[3] |
| 0x2000'0318 | 0xe7fe'4770 | foo[4] |
| 0x2000'031c | 0xf80d'f000 | foo[5] |
| 0x2000'0320 | 0x2 | |
| 0x2000'0324 | 0x5d | |
| 0x2000'0328 | 0x8000'f3af | foo[0] |
| 0x2000'032c | 0xf3af'2000 | foo[1] |
| 0x2000'0330 | 0x2 | foo[2] |
| 0x2000'0334 | 0xf000'ff97 | foo[3] |
| 0x2000'0338 | 0x2001'f802 | foo[4] |
| 0x2000'033c | 0xf000'4770 | foo[5] |
| 0x2000'0340 | 0x3 | |
| 0x2000'0344 | 0x5d | |

The Add SP,SP, #0x18 adds 0x18 to stack address and reduces it's size, The POP instruction restores the saved (and now corrupted) link register into the PC. Recall this place hold data 0x07 due to stack corruption when foo[7] crossed foo array bounds into the place where LR was saved on the stack. This causes wrong program flow and puts the cpu into a cycle. Throughout all this stack is not properly ejected, and thus grows till there is stack overflow

```
Disassembly
Go to:                Zone: Memory
Disassembly
        0x106: 0x6008         STR       R0, [R1]
        0x108: 0xf3bf 0x8f4f  DSB
        0x10c: 0xf3bf 0x8f6f  ISB
        0x110: 0xf04f 0x7001  MOV.W     R0, #33816576
        0x114: 0xeee1 0x0a10  VMSR      FPSCR, R0
        0x118: 0x4770         BX        LR
                BusFault_Handler:
                DebugMon_Handler:
                HardFault_Handler:
                MemManage_Handler:
                NMI_Handler... +5 symbols not displayed
        0x11a: 0xe7fe         B.N       BusFault_Handler
                ?main:
                __cmain:
        0x11c: 0xf000 0xf80d  BL        __low_level_init
        0x120: 0x2800         CMP       R0, #0
        0x122: 0xd001         BEQ.N     _call_main
        0x124: 0xf3af 0x8000  NOP.W
                _call_main:
        0x128: 0xf3af 0x8000  NOP.W
        0x12c: 0x2000         MOVS      R0, #0
        0x12e: 0xf3af 0x8000  NOP.W
        0x132: 0xf7ff 0xff97  BL        main
        0x136: 0xf000 0xf802  BL        exit
```

```c
void swap(volatile int *x, volatile int *y)
{
 int tmp;
 tmp = *x;
 *x = *y;
 *y = tmp;
}

int main()
{

 volatile int x;
 volatile int y;

 x = 5u;
 y = 2u;

 swap(&x, &y);
}
```

Since swap is using pointers, all changes in x and y of swap mean x and y of main is getting altered directly

```
int main()
{
                main:
        0x4c: 0xb51c            PUSH    {R2-R4, LR}
  x = 5u;
        0x4e: 0x2005            MOVS    R0, #5
        0x50: 0x9001            STR     R0, [SP, #0x4]
  y = 2u;
        0x52: 0x2002            MOVS    R0, #2
        0x54: 0x9000            STR     R0, [SP]
  swap(&x, &y);
        0x56: 0x4669            MOV     R1, SP
        0x58: 0xa801            ADD     R0, SP, #0x4
        0x5a: 0xf7ff 0xfff1     BL      swap
```

x and y are stored in the stack using STR function.
X is stored at address:
SP address + 0x04 = 0x2000 03ec
and y is stored in top SP address: 0x2000 03e8

Stack 1

CSTACK

| Location | Data | Variable | Va |
|---|---|---|---|
| 0x2000'03e8 | 0x2 | y | 2 |
| 0x2000'03ec | 0x5 | x | 5 |
| 0x2000'03f0 | 0x0 | | |
| 0x2000'03f4 | 0x127 | | |
| 0x2000'03f8 | 0xfef5'eda5 | | |
| 0x2000'03fc | 0xfef5'eda5 | | |

As per ARM standard, address of R0 and R1 in stack is copied to R1 and R0 registers just before branch (BL) instruction is carried out:

| Name | Value |
|---|---|
| R0 | 0x2000'03ec |
| R1 | 0x2000'03e8 |
| R2 | 0x0000'0000 |
| R3 | 0x0000'0000 |
| R4 | 0x0000'0000 |
| R5 | 0x0000'0000 |
| R6 | 0x0000'0000 |
| R7 | 0x0000'0000 |
| R8 | 0x0000'0000 |
| R9 | 0x0000'0000 |
| R10 | 0x0000'0000 |
| R11 | 0x0000'0000 |
| R12 | 0x0000'0000 |

Disassembly

```
  *y = *x;
        0x46: 0x6800            LDR     R0, [R0]
        0x48: 0x6008            STR     R0, [R1]
}
        0x4a: 0x4770            BX      LR
int main()
{
                main:
        0x4c: 0xb51c            PUSH    {R2-R4, LR}
  x = 5u;
        0x4e: 0x2005            MOVS    R0, #5
        0x50: 0x9001            STR     R0, [SP, #0x4]
  y = 2u;
        0x52: 0x2002            MOVS    R0, #2
        0x54: 0x9000            STR     R0, [SP]
  swap(&x, &y);
        0x56: 0x4669            MOV     R1, SP
        0x58: 0xa801            ADD     R0, SP, #0x4
        0x5a: 0xf7ff 0xfff1     BL      swap
```

The next set of instructions show how swap function changes the value of x and y themselves via utilising pointers :

```
swap:
        0x40: 0x6802        LDR     R2, [R0]
  *x = *y;
        0x42: 0x680b        LDR     R3, [R1]
        0x44: 0x6003        STR     R3, [R0]
  *y = tmp;
        0x46: 0x600a        STR     R2, [R1]
}
```

1st LDR moves content of address stored in R0 (value of x) to R2

2nd LDR moves content of address stored in R1 (value of y) to R3

3rd STR stores R3 contents at address stored in R0

4th STR stores R2 contents in address stored in R1. This causes x and y swap.

* Imp for returning local variables in function

Note → stack destroys previous values of local variables from a function, anytime another function call is initiated. If the variables and their values need to be used after returning to main, it is best to declare the function local variables as static