

Version 2.00

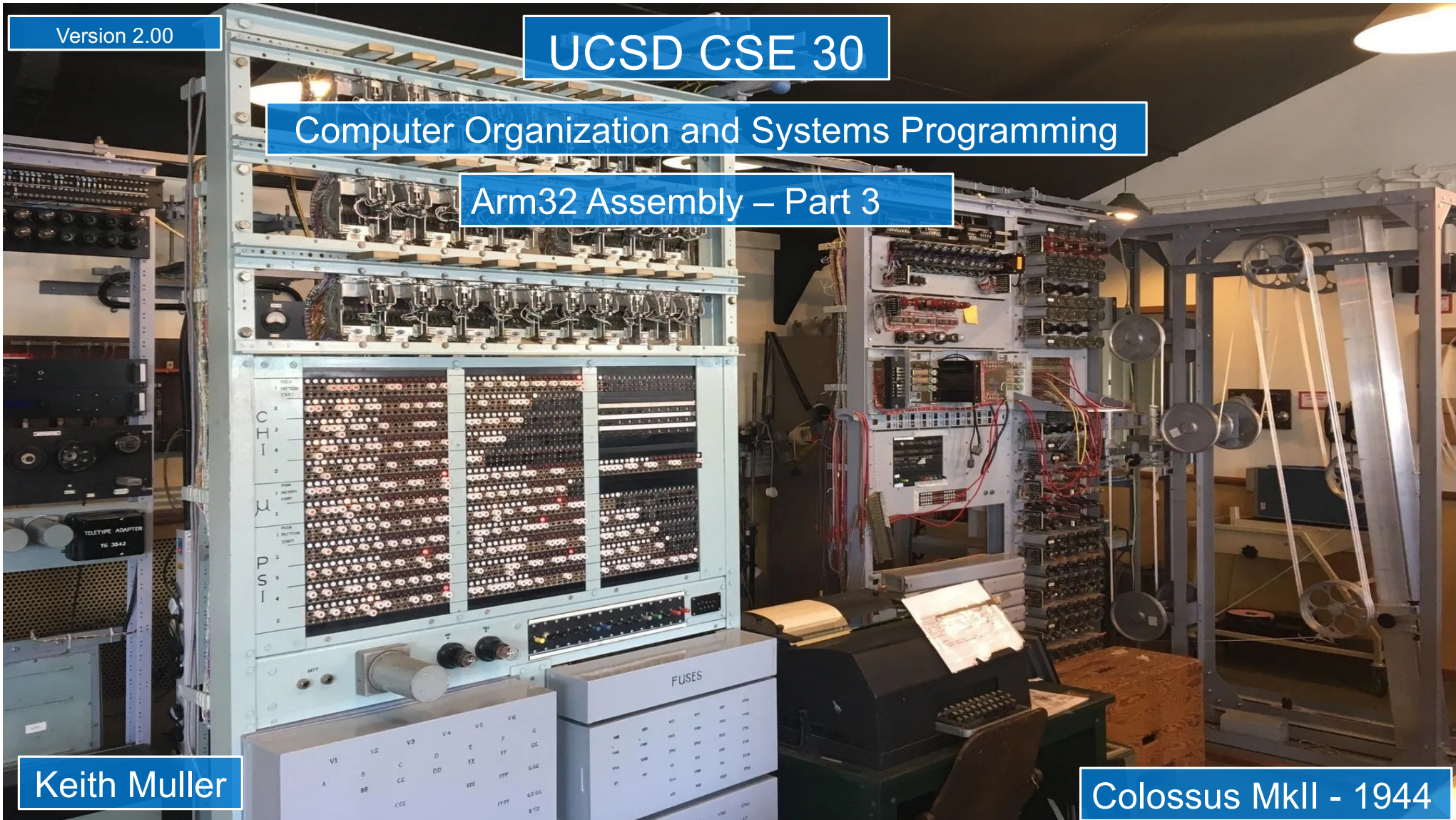
UCSD CSE 30

Computer Organization and Systems Programming

Arm32 Assembly – Part 3

Keith Muller

Colossus MkII - 1944



Reference For PA7/8: C Stream Functions Opening Files

```
FILE *fopen(char filename[], const char mode[]);
```

- Opens a stream to the specified file in specified file access mode
 - returns NULL on failure – **always check the return value; make sure the open succeeded!**
- Mode is a string that describes the actions that can be performed on the stream:

"r" Open for reading.

The stream is positioned at the beginning of the file. Fail if the file does not exist.

"w" Open for writing.

The stream is positioned at the beginning of the file. Create the file if it does not exist.

"a" Open for writing.

The stream is positioned at the end of the file. Create the file if it does not exist.

Subsequent writes to the file will always be at current end of file.

- An optional "+" following "r", "w", or "a" opens the file for both reading and writing

Reference: C Stream Functions Closing Files and Usage

```
int fclose(FILE *stream) ;
```

- Closes the specified stream, forcing output to complete (eventually)
 - returns EOF on failure (often ignored as no easy recovery other than a message)
- Usage template for `fopen()` and `fclose()`
 1. Open a file with `fopen()` **always** checking the return value
 2. do i/o – keep calling stdio io routines
 3. close the file with `fclose()` when done with that I/O stream

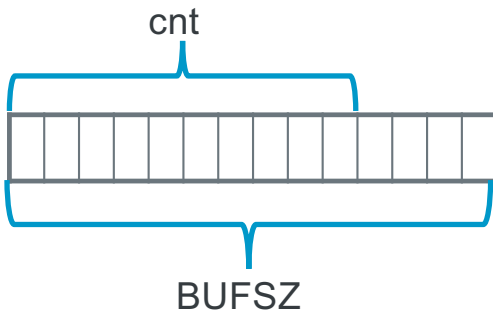
C Stream Functions Array/block read/write

- These do not process contents they simply **transfer** a fixed number of bytes to and from a buffer passed to them
- `size_t fwrite(void *ptr, size_t size, size_t count, FILE *stream);`
 - Writes an array of *count elements* of *size* bytes from *stream*
 - *Updates the write file pointer forward by the number of bytes written*
 - returns number of elements written
 - error is short element count or 0
- `size_t fread(void *ptr, size_t size, size_t count, FILE *stream);`
 - Reads an array of *count elements* of *size* bytes from *stream*
 - *Updates the read file pointer forward by the number of bytes read*
 - returns number of elements read, **EOF is a return of 0**
 - error is short element count or 0
- **I almost always set size to 1 to return bytes read/written**

C fread() and fwrite()

element size of 1 with a char buffer is byte I/O
Capture bytes read so you know how many bytes to write

unless the **input file length is an exact multiple of BUFSZ**, last fread() will always read less than BUFSZ which is why you write cnt



Jargon: the last record is often called the "runt"

```
size_t
fread(void *ptr, size_t size, size_t count, FILE *stream)
    • Reads an array of count elements of size bytes from stream

size_t
fwrite(void *ptr, size_t size, size_t count, FILE *stream)
    • Writes an array of count elements of size bytes to stream
```

```
#define BUFSZ 128
```

```
int copy(FILE *infp, FILE *outfp) {
    unsigned char buf[BUFSZ];
    size_t cnt;

    while ((cnt = fread(buf, 1, BUFSZ, infp)) > 0) {
        fprintf(stderr, "bytes: %u\n", cnt);

        if (fwrite(buf, 1, cnt, outfp) != cnt)
            return -1;
    }
    return 0;
}
```

```
% ls -l a
4 -rw-r--r-- 1 kmuller 1104 May 15 09:45 a
% ./a.out a b
bytes: 128
bytes: 128
bytes: 128
bytes: 128
bytes: 128
bytes: 128
bytes: 128
bytes: 128
bytes: 80
```

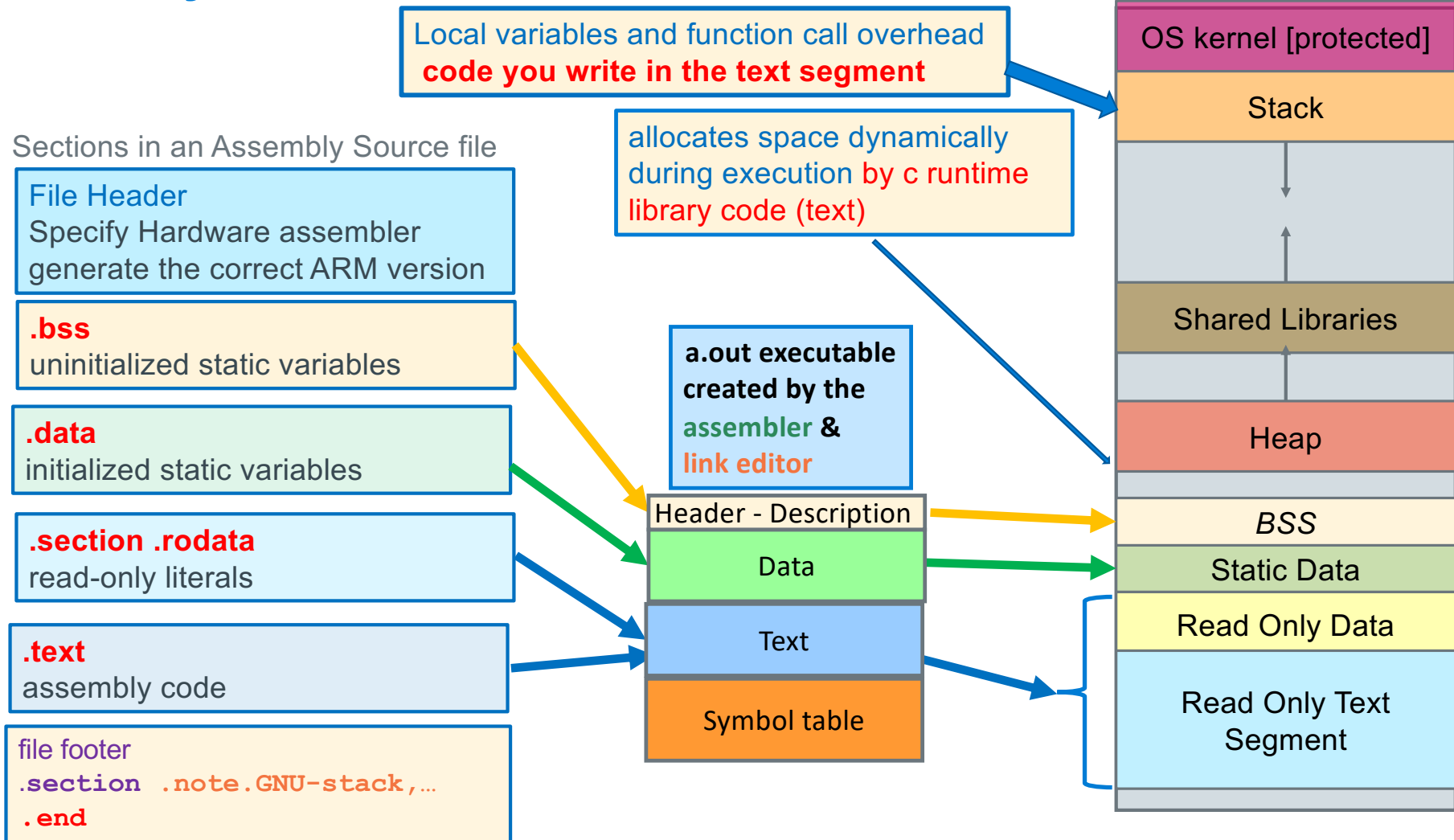
$$128 * 8 + 80 = 1104$$

Using fopen() and fclose()

```
int main(int argc, char **argv)
{
    FILE *infp;
    FILE *outfp;
    int reslt;

    if (argc != 3) {
        fprintf(stderr, "%s requires two args\n", argv[0]);
        return EXIT_FAILURE;
    }
    // Open the input file for read
    if ((infp = fopen(*(argv+1), "r")) == NULL) {
        fprintf(stderr, "fopen for read failed\n");
        return EXIT_FAILURE;
    }
    // Open the output file for write
    if ((outfp = fopen(*(argv+2), "w")) == NULL) {
        fprintf(stderr, "fopen for write failed\n");
        fclose(infp);
        return EXIT_FAILURE;
    }
    reslt = copy(infp, outfp);
    fclose(infp);
    fclose(outfp);
    if (reslt != 0) {
        fprintf(stderr, "copy %s to %s failed\n", *(argv+1), *(argv+2));
        return EXIT_FAILURE;
    }
    return EXIT_SUCCESS;
}
```

Assembly Source File to Executable to Linux Memory



Creating Segments, Definitions In Assembly Source

- The following assembler directives indicate the **start** of a **memory segment specification**
 - **Remains in effect** until the next segment directive is seen

```
.bss
    // start uninitialized static segment variables definitions
    // does not consume any space in the executable file
.data
    // start initialized static segment variables definitions
.section .rodata
    // start read-only data segment variables definitions
.text
    // start read-only text segment (code)
```


Assembly Source File Template

```
// File Header
.arch armv6                // armv6 architecture instructions
.arm                      // arm 32-bit instruction set
.fpu vfp                  // floating point co-processor
.syntax unified           // modern syntax

// BSS Segment (only when you have initialized globals)
.bss

// Data Segment (only when you have uninitialized globals)
.data

// Read-Only Data (only when you have literals)
.section .rodata

// Text Segment - your code
.text

// Function Header
.type main, %function      // define main to be a function
.global main              // export function name
main:
// function prologue        // stack frame setup
    // your code for this function here
// function epilogue        //stack frame teardown

// function footer
.size main, (. - main)

// File Footer
.section .note.GNU-stack,"",%progbits // stack/data non-exec
.end
```

- assembly programs end in **.S**
 - That is a **capital .S**
 - **example:** test.S
- Always use gcc to assemble
 - **_start()** and C runtime
- File has a complete program
gcc file.S
- File has a partial program
gcc -c file.S
- Link files together
gcc file.o cprog.o

ARM Assembly Source File: Header and Footer

File Header

At the top of every
ARM source file

```
.arch    armv6           // armv6 architecture
.arm     // arm 32-bit instruction set
.fpu     vfp             // floating point co-processor
.syntax  unified         // modern syntax
```

```
// Contents of the other memory segment include .text (your code)
```

File Footer

At the bottom of every
ARM source file

```
.section .note.GNU-stack,"",%progbits // set stack/data non-exec
.end

// everything past the .end is ignored!
// Debugging notes etc
```

`.syntax unified`

- use the standard ARM assembly language syntax called *Unified Assembler Language (UAL)*

`.section .note.GNU-stack,"",%progbits`

- tells the linker to **make the stack and all data segments not-executable** (no instructions in those sections) – security measure

`.end`

- at the end of the source file, everything written after the `.end` is ignored

Assembler Directives: .equ and .equiv

```
.equ    BLKSZ, 10240    // buffer size in bytes
.equ    BUFCNT, 100*4   // buffer for 100 ints
.equ    BLKSZ, STRSZ * 4 // redefine BLKSZ from here
```

`.equ <symbol>, <expression>`

- Defines and sets the value of a **symbol** to the **evaluation** of the **expression**
- Used for specifying constants, like a **#define** in C
- You can **(re)set** a symbol many times in the file, **last one seen applies**

```
.equ    BLKSZ, 10240    // buffer size in bytes
// other lines
.equ    BLKSZ, 1024     // buffer size in bytes
```

Example: Assembler Directive and Instructions

assembler directive `.equ` does not allocate any memory (NULL = 0)

Regular label `main` is associated with memory location 0x3000

Local label `.Lloop` is associated with memory location 0x3004

space.S

```
10  .equ NULL, 0
11 main:
12 3000 0310A0E1      mov     r1, r3
13 .Lloop:
14 3004 043083E2      add     r3, r3, 4
15 3008 001093E5      ldr     r1, [r3]
16 300c 000051E3      cmp     r1, NULL
17 3010 FBFFFF1A      bne     .Lloop
```

output generated with
`gcc -c -Wa,-ahlns space.S`
partial output is shown

Memory Contents

Warning contents shown in "reverse" byte order: Lsb – Msb

Instruction Memory Addresses (lowest 2-bits are always 00)
Notice alignment and how addresses increase by 4 (32-bit instructions)

Function Header and Footer Assembler Directives

function entry point
address of the first
instruction in the function
Must not be a local label
(does not start with .L)

```
Function Header {  
    .text  
    .global myfunc           // make myfunc global for linking  
    .type    myfunc, %function // define myfunc to be a function  
    .equ     FP_OFF, 4       // fp offset in main stack frame  
myfunc:  
    // function prologue, stack frame setup  
    // your code  
    // function epilogue, stack frame teardown  
Function Footer {  
    .size myfunc, (. - myfunc)
```

.global function_name

- Exports the function name to other files. Required for main function, optional for others

.type name, %function

- The **.type** directive sets the **type of a symbol/label name**
- %function** specifies that **name** is a function (name is the address of the first instruction)

equ FP_OFF, 4

- Used for basic stack frame setup; the number 4 will change – later slides

.size name, bytes

- The **.size** directive is used to **set the size associated with a symbol**
- Used by the linker to exclude unneeded code and/or data when creating an executable file
- It is also used by the **debugger** gdb
- bytes is best calculated as an expression: (period is the current address in a memory segment)**

.size name, (. - name)

Function Prologue and Epilogue: Stack Frame Management

Minimum Sized stack frame shown

```
.text
.global myfunc          // make myfunc global for linking
.type    myfunc, %function // define myfunc to be a function
.equ     FP_OFF, 4       // fp offset in main stack frame

myfunc:
    // function prologue, stack frame setup - (later slides)
    push    {fp, lr}
    add     fp, sp, FP_OFF
    // your code

    // function epilogue, stack frame teardown, return - (later slides)
    sub     sp, fp, FP_OFF
    pop     {fp, lr}
    bx      lr
.size myfunc, (. - myfunc)
```

Function Prologue

Function Epilogue

Preview: Return Value and Passing Parameters to Functions

(Four parameters or less)

Register	Function Call Use
r0	1 st parameter
r1	2 nd parameter
r2	3 rd parameter
r3	4 th parameter

Register	Function Return Value Use
r0	8, 16 or 32-bit result, 32-bit address or least-significant half of a 64-bit result
r1	most-significant half of a 64-bit result

- Where **r0**, **r1**, **r2**, **r3** are arm registers, the function declaration is (first four arguments):

```
r0 = function(r0, r1, r2, r3)           // 32-bit return
```

```
r0, r1 = function(r0, r1, r2, r3)      // 64-bit return - long long
```
- Each **parameter** and **return value** is limited to data that **can fit in 4 bytes or less**
- You receive **up to the first four parameters** in these four registers
- You copy up to the first four parameters into these four registers before calling a function
- For parameter values using more than 4 bytes, a pointer to the parameter is passed (we will cover this later)
- You MUST ALWAYS assume** that the called function will **alter the contents of all four registers: r0-r3**
 - In terms of C runtime support, these registers contain the copies given to the called function
 - C allows the copies to be changed in any way by the called function

Assembler Directives: Label Scope Control (Normal Labels only)

```
.extern printf
.extern fgets
.extern strcpy
.global fbuf
```

.extern <label>

- **Imports** label (function name, symbol or a static variable name);
- An address associated with the label from another file can be used by code in this file

.global <label>

- **Exports** label (or symbol) to be visible outside the source file boundary (other assembly or c source)
 - label is either a function name or a global variable name
 - Only use with function names or static variables
- **Without** .global, labels are usually **local to the file** from the point where they are defined

Preview: Writing an ARM32 function

```
#include <stdlib.h>
#include <stdio.h>
int sum4(int, int, int, int);
int main()
{
    int reslt;

    reslt = sum4(1,2,3,4);

    printf("%d\n", reslt);
    return EXIT_SUCCESS;
}
```

```
#ifndef SUM4_H
#define SUM4_H

#ifdef __ASSEMBLER__
int sum4(int, int, int, int);
#else
.extern sum4
#endif

#endif
```

```
#include "sum4.h"
.arch armv6
.arm
.fpu vfp
.syntax unified
.global sum4
.type sum4, %function
.equ FP_OFF, 28
// r0 = sum4(r0, r1, r2, r3)
sum4:
    push    {r4-r9, fp, lr}
    add     fp, sp, FP_OFF

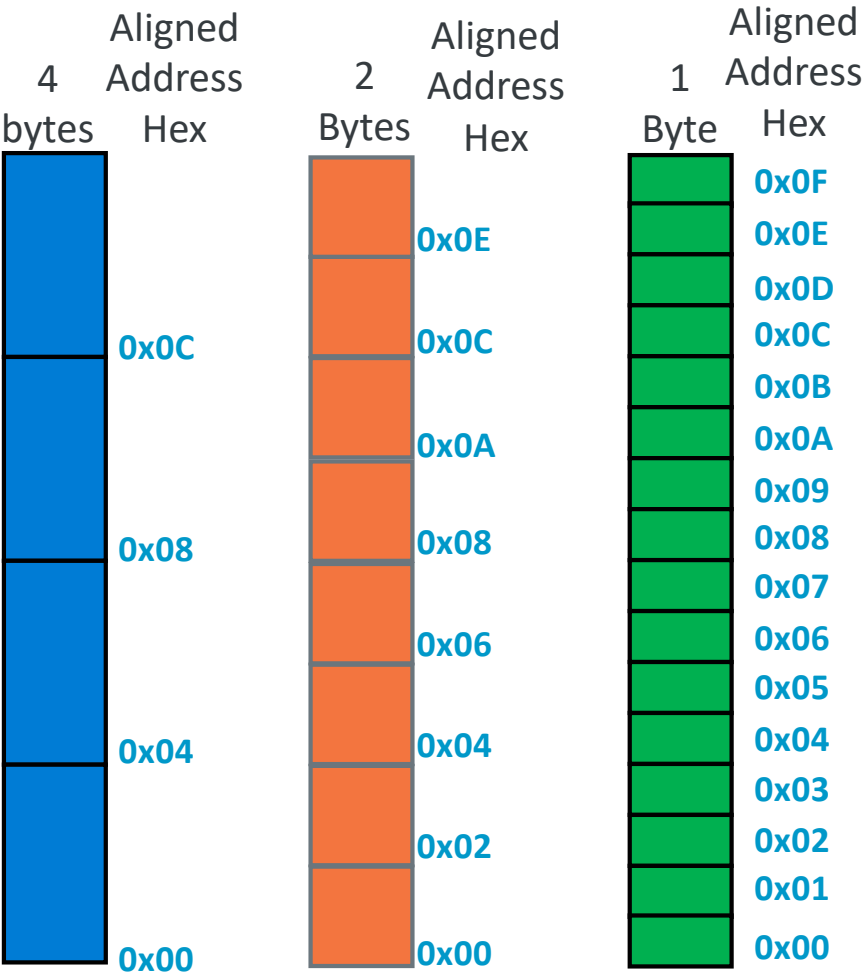
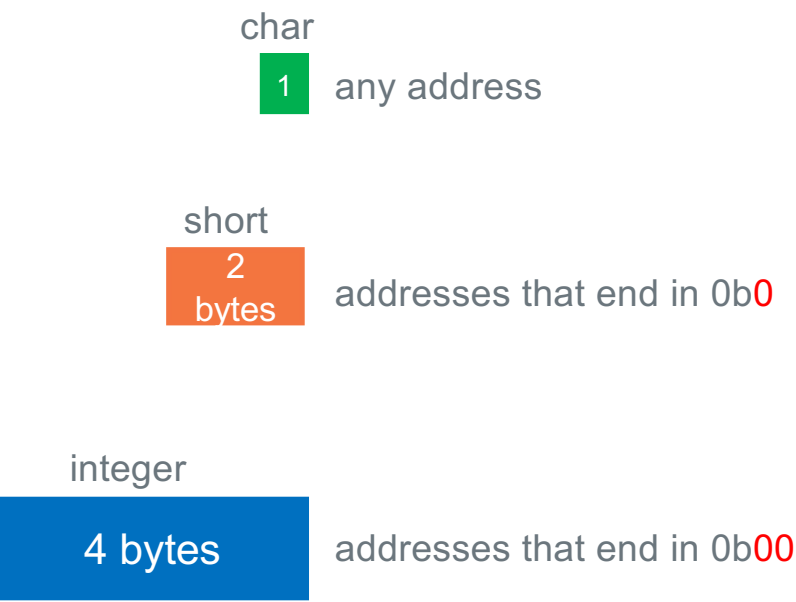
    add     r0, r0, r1
    add     r0, r0, r2
    add     r0, r0, r3

    sub     sp, fp, FP_OFF
    pop     {r4-r9, fp, lr}
    bx      lr
    .size sum4, (. - sum4)
    .section .note.GNU-stack,"",%progbits
.end
```

```
$ gcc -Wall -Wextra -c main.c
$ gcc -c sum4.S
$ gcc sum4.o main.o
$ ./a.out
10
```

Variable Alignment In Memory and Performance

Accessing **address aligned** memory on many systems **based on data type** has **the best performance** (due to hardware implementation)



Load/Store: Register Base Addressing

ldr r0, [r1]

Copies a 32-bit word from the memory location whose address is contained in r1 (r1 is a pointer) into register r0

32-bit memory



register r0

register r1 (address)



r1 is being used as a pointer to a location in memory

ldr requires the use of a pointer operand

str r0, [r1]

Copies all 32 bits of the value held in register r0 to the 32-bit memory location contained in register r1 (r1 pointer)

register r0



32-bit memory

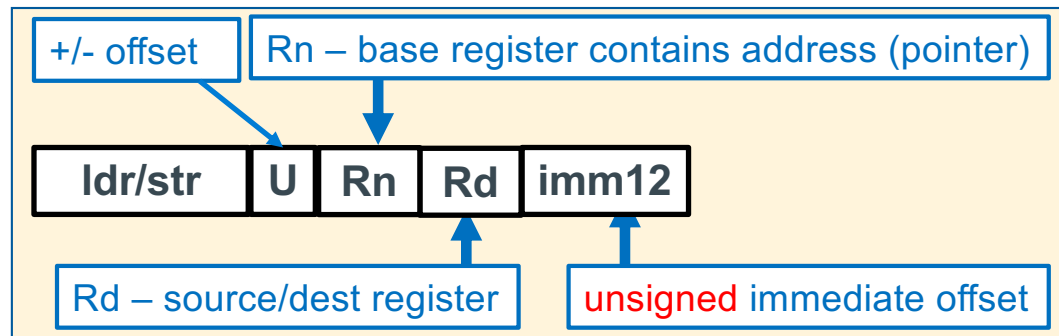
r1 is being used as a pointer to a location in memory

str requires the use of a pointer operand

register r1 (address)

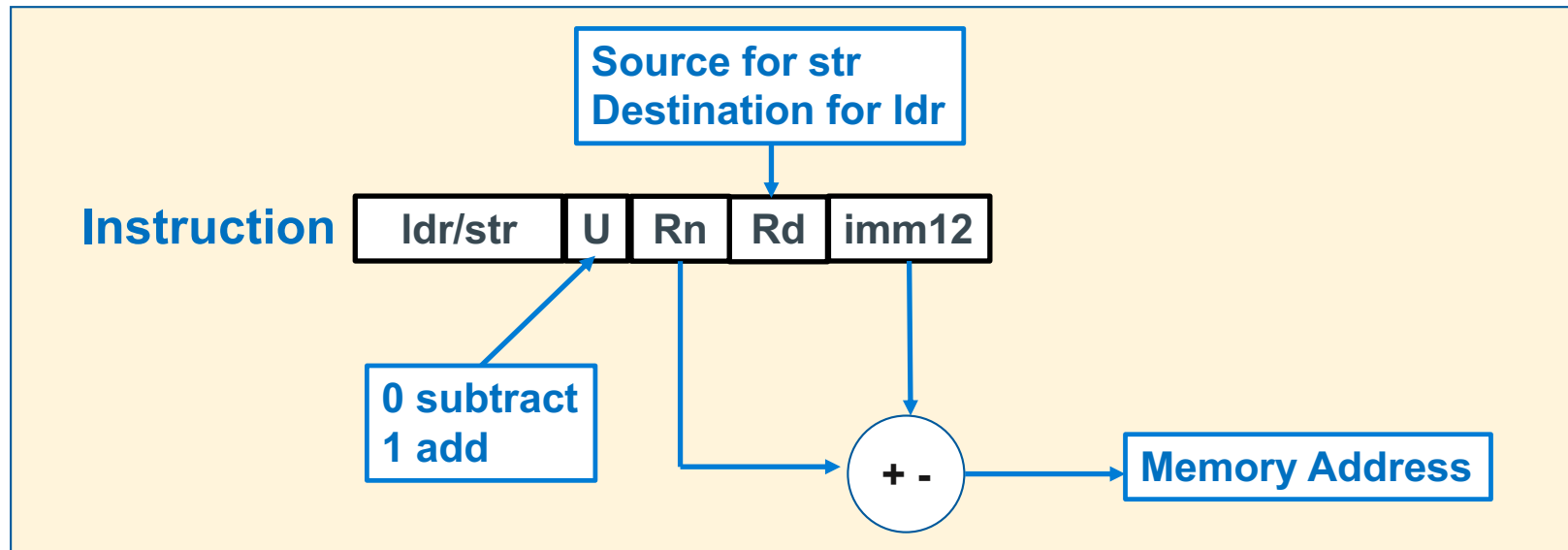


LDR/STR – Base Register + Immediate Offset Addressing



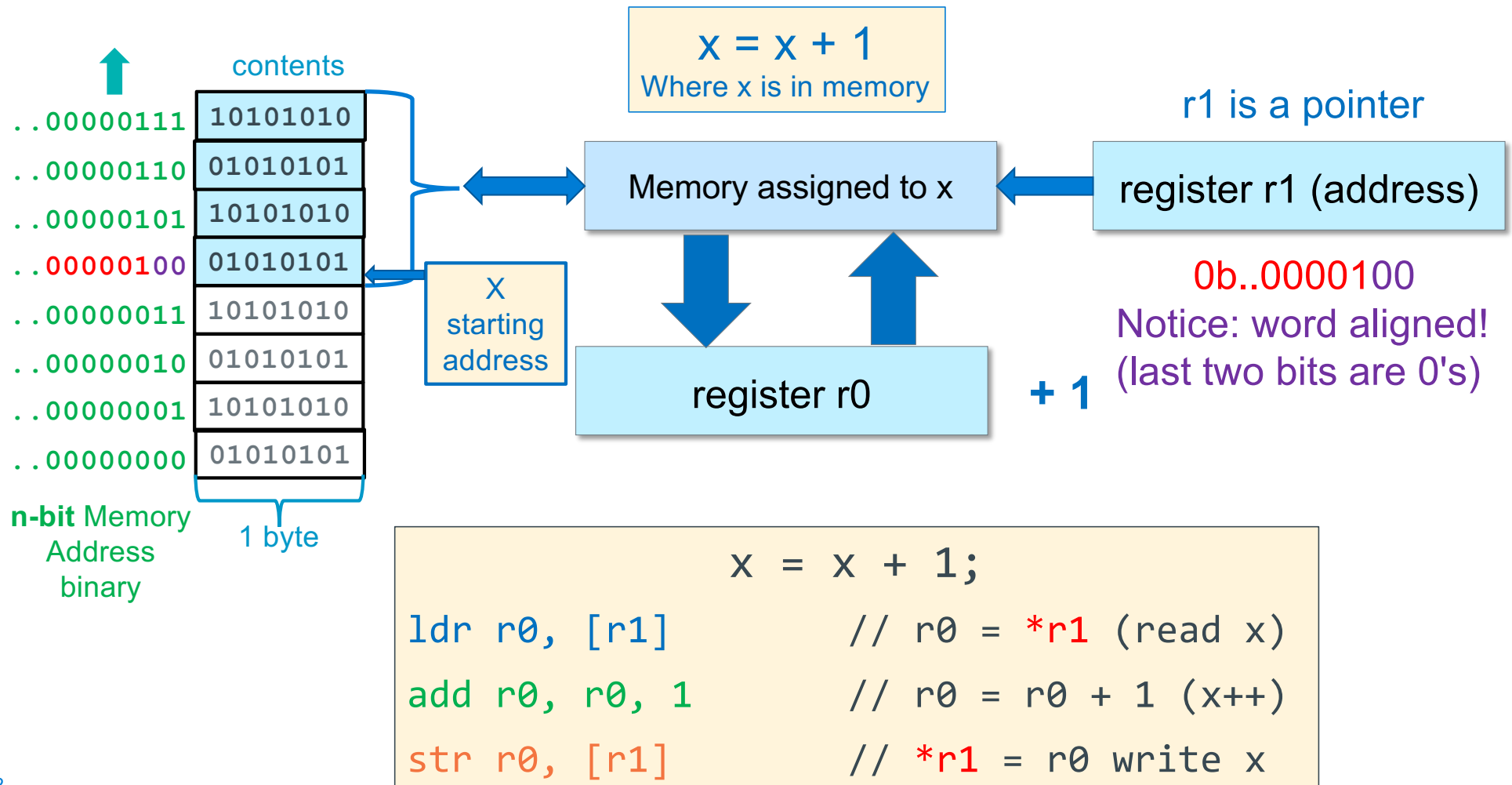
- **Register Base Addressing:**
 - Pointer Address: **Rn**; **source/destination data**: **Rd**
 - **Unsigned pointer address** is stored in the **base register**
- **Register Base + immediate offset Addressing:**
 - Pointer Address = register content + immediate offset $-4095 \leq \text{imm12} \leq 4095$ (bytes)
 - Unsigned offset integer **immediate value (bytes)** is added or subtracted (**U bit above says to add or subtract**) from the **pointer address** in the **base register**

ldr/str Register Base + Immediate Offset Addressing



Syntax	Address	Examples
<code>ldr/str Rd, [Rn, +/- constant]</code> constant is in bytes <code>ldr/str Rd, [Rn]</code>	<code>Rn + or - constant</code> same \longrightarrow	<code>ldr r0, [r5,100]</code> <code>str r1, [r5, 0]</code> <code>str r1, [r5]</code>

Example Base Register Addressing Load – Modify – Store

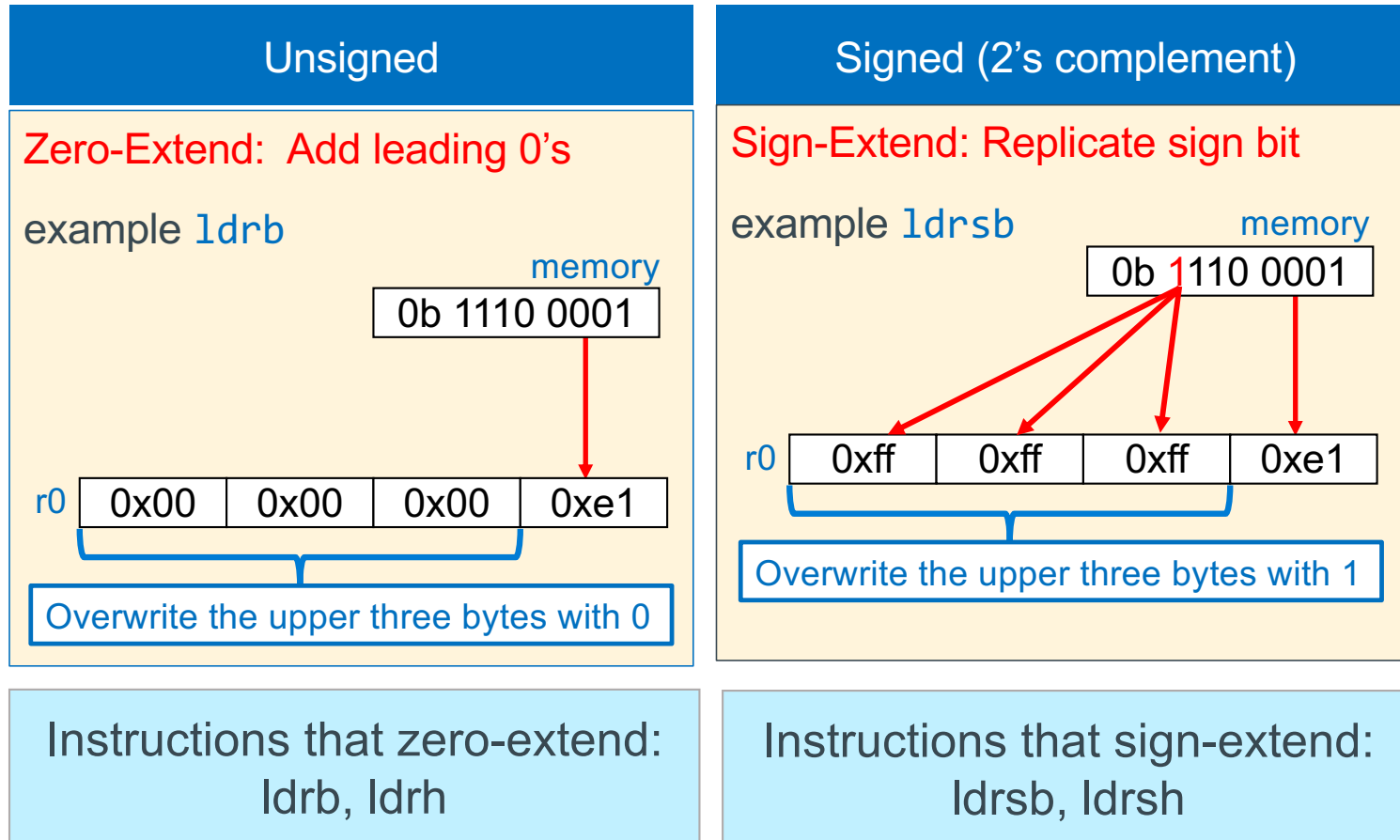


Loading and Storing: Variations List

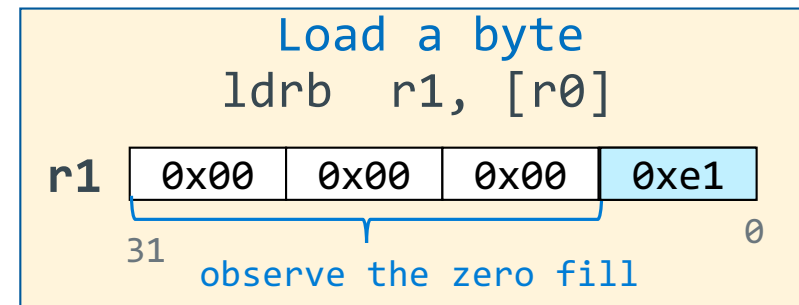
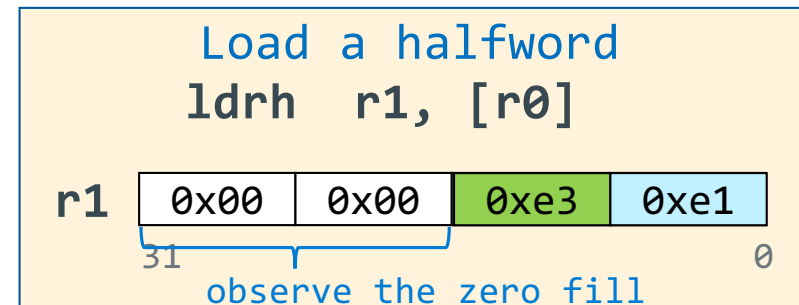
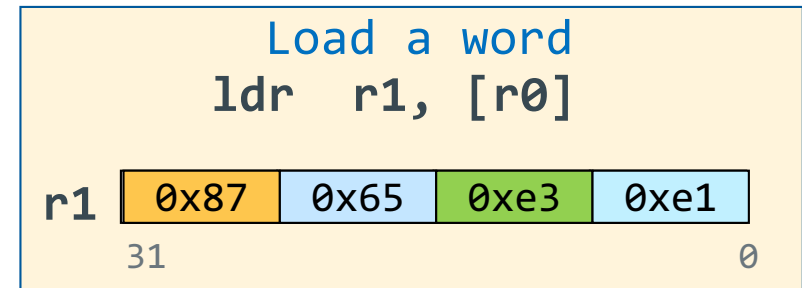
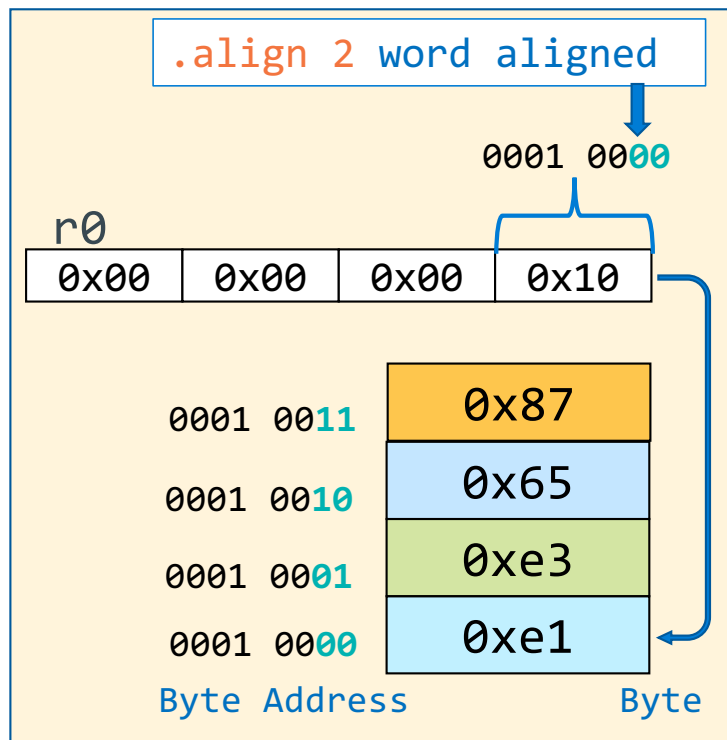
- Load and store have **variations** that move 8-bits, 16-bits and 32-bits
- Load into a register with less than 32-bits will **set the upper bits not filled from memory differently** depending on which **variation of the load instruction** is used
- Store will only select the lower 8-bit, lower 16-bits or all 32-bits of the register to copy to memory, **register contents are not altered**

Instruction	Meaning	Sign Extension	Memory Address Requirement
ldrsb	load signed byte	sign extension	none (any byte)
ldrb	load unsigned byte	zero fill (extension)	none (any byte)
ldrsh	load signed halfword	sign extension	halfword (2-byte aligned)
ldrh	load unsigned halfword	zero fill (extension)	halfword (2-byte aligned)
ldr	load word	---	word (4-byte aligned)
strb	store low byte (bits 0-7)	---	none (any byte)
strh	store halfword (bits 0-15)	---	halfword (2-byte aligned)
str	store word (bits 0-31)	---	word (4-byte aligned)

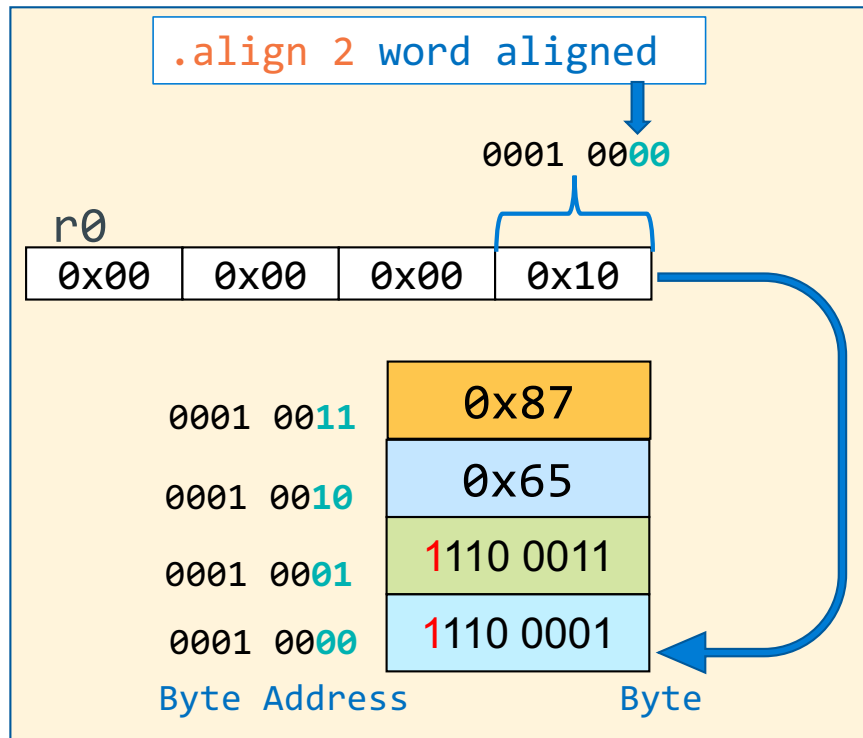
Loading 32-bit Registers From Memory Variables < 32-Bits Wide



Load a Byte, Half-word, Word

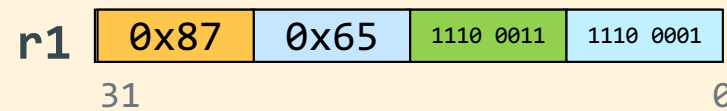


Signed Load a Byte, Half-word, Word



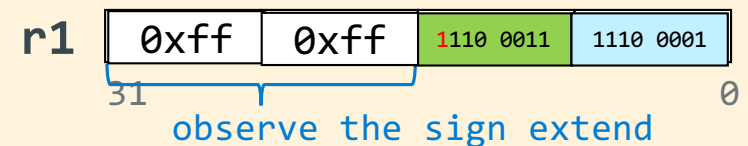
Load a word (no change)

ldr r1, [r0]



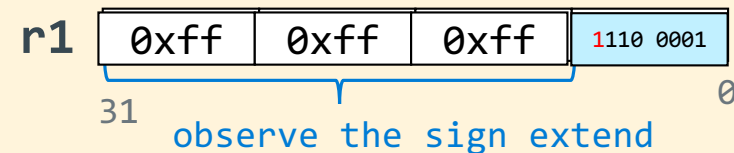
Load a halfword

ldrsh r1, [r0]

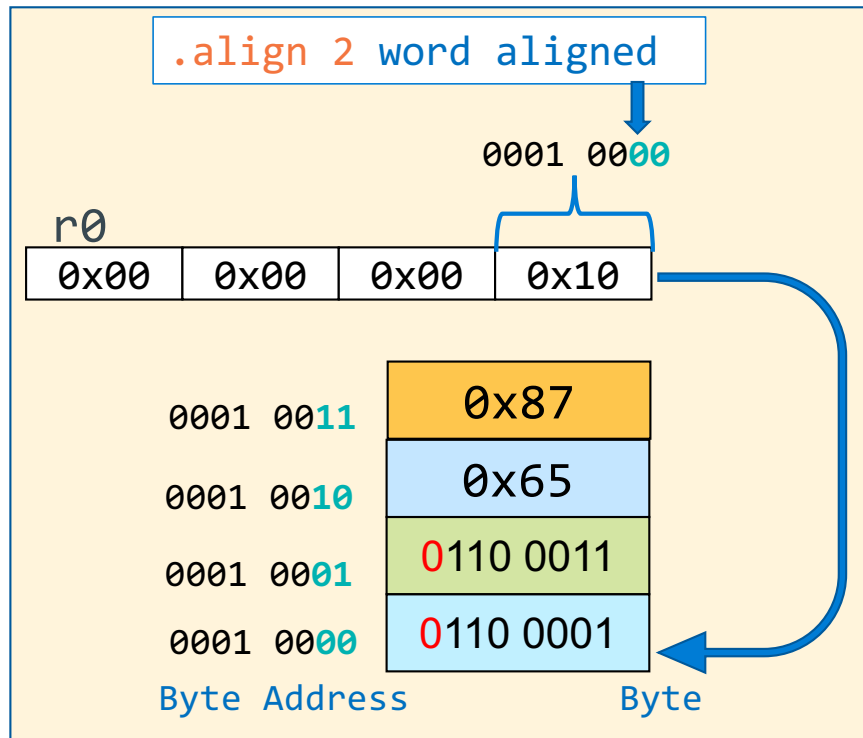


Load a byte

ldrsb r1, [r0]

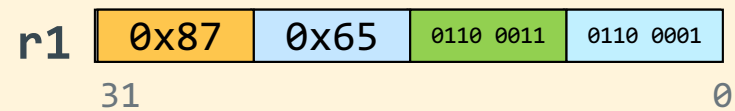


Signed Load a Byte, Half-word, Word



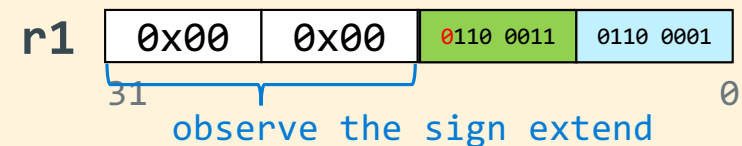
Load a word (no change)

ldr r1, [r0]



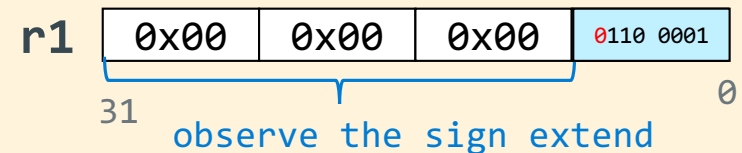
Load a halfword

ldrsh r1, [r0]

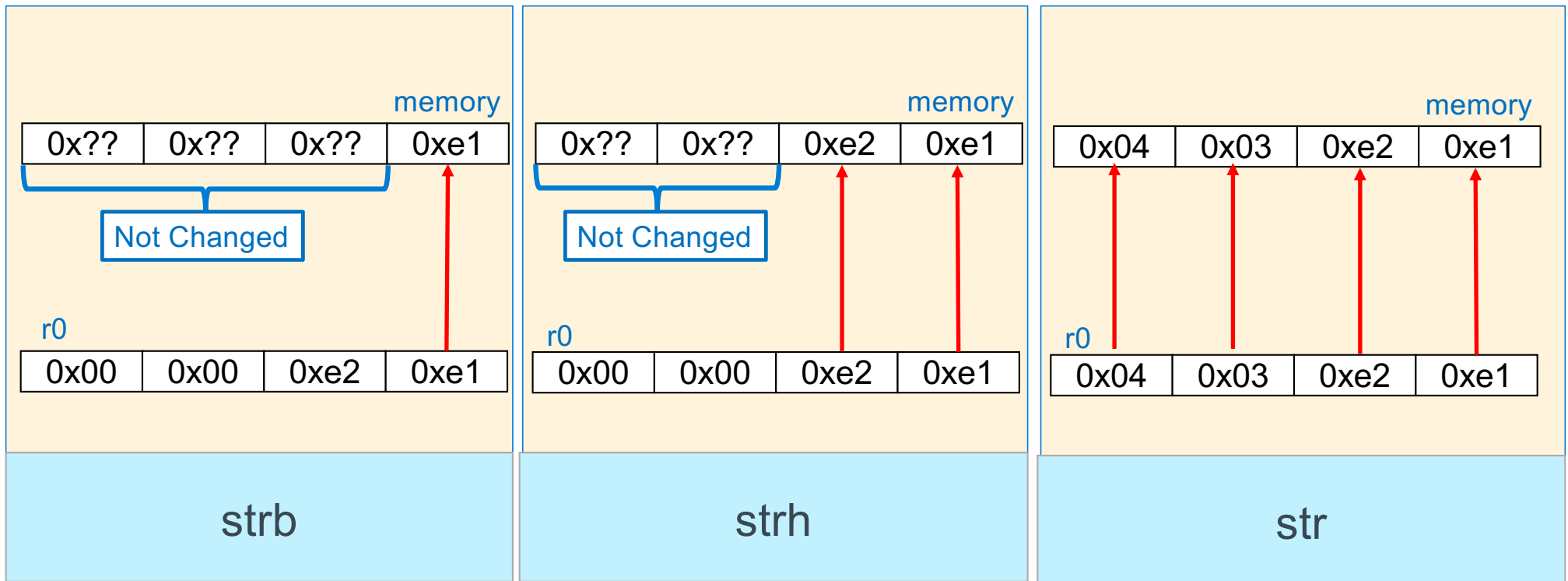


Load a byte

ldrsb r1, [r0]



Storing 32-bit Registers To Memory 8-bit, 16-bit, 32-bit



Store a Byte, Half-word, Word

initial value in r0

0x20	0x00	0x00	0x00
------	------	------	------

Store a byte
`strb r1, [r0]`

r1: 31 | 0x87 | 0x65 | 0xe3 | 0xe1 | 0

Byte Address | Byte

0x20000003	0x33	observe other bytes NOT altered
0x20000002	0x22	
0x20000001	0x11	
0x20000000	0xe1	

Store a halfword
`strh r1, [r0]`

r1: 31 | 0x87 | 0x65 | 0xe3 | 0xe1 | 0

Byte Address | Byte

0x20000003	0x33
0x20000002	0x22
0x20000001	0xe3
0x20000000	0xe1

Store a word
`str r1, [r0]`

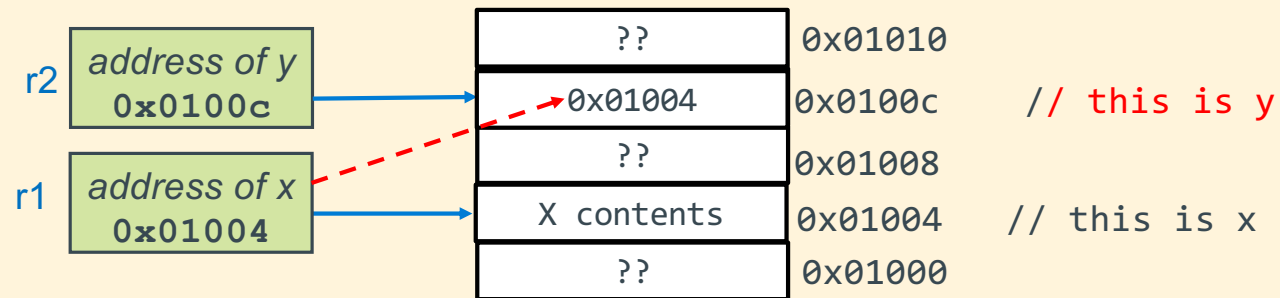
r1: 31 | 0x87 | 0x65 | 0xe3 | 0xe1 | 0

Byte Address | Byte

0x20000003	0x87
0x20000002	0x65
0x20000001	0xe3
0x20000000	0xe1

ldr/str practice - 1

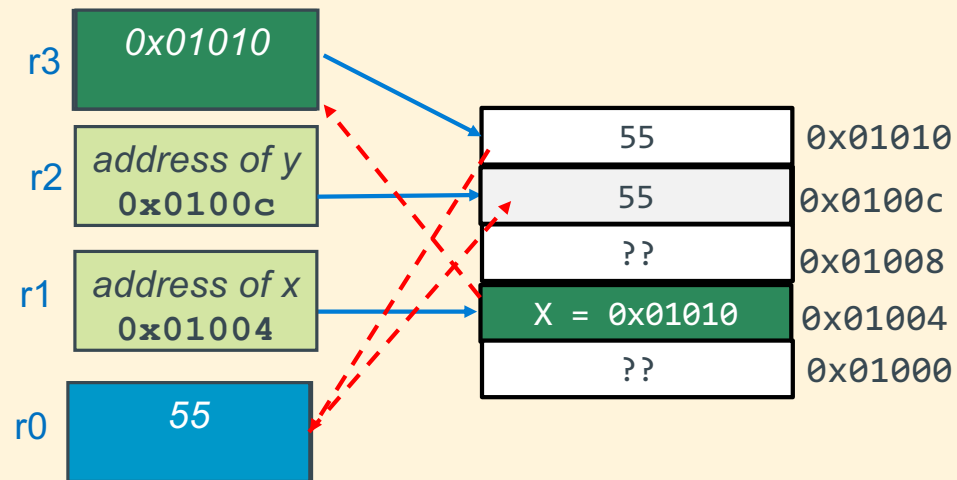
r1 contains the Address of X (defined as int X) in memory; r1 points at X
r2 contains the Address of Y (defined as int *Y) in memory; r2 points at Y
write Y = &X;



```
str    r1, [r2]    // y ← &x
```

ldr/str practice - 2

r1 contains the Address of X (defined as `int *X`) in memory r1 points at X
r2 contains the Address of Y (defined as `int Y`) in memory; r2 points at Y
write `Y = *X;`



```
ldr    r3, [r1]    // r3 ← x (read 1)
ldr    r0, [r3]    // r0 ← *x (read 2)
str    r0, [r2]    // y ← *x
```

using ldr/str: array copy

```
#include <stdio.h>
#include <stdlib.h>
#define SZ 6

void icpy(int *, int *, int);

int main(void)
{
    int  src[SZ] = {1, 2, 3, 4, 5, 6};
    int  dst[SZ];

    icpy(src, dst, SZ);
    for (int i = 0; i < SZ; i++)
        printf("%d\n", *(dst + i));

    return EXIT_SUCCESS;
}
```

```
void icpy(int *src, int *dst, int cnt)
{
    for (int i = 0; i < cnt; i++)
        *dst++ = *src++;

    return;
}
```


Base Register version

```
.arch armv6
.arm
.fpu vfp
.syntax unified
.text
.global icpy
.type icpy, %function
.equ FP_OFF, 12

// r0 contains int *src
// r1 contains int *dst
// r2 contains int cnt
// r3 use as loop term pointer
// r4 use as temp

icpy:
    push    {r4, r5, fp, lr}
    add     fp, sp, FP_OFF
    // see right ->
    sub     sp, fp, FP_OFF
    pop     {r4, r5, fp, lr}
    bx      lr
.size icpy, (. - icpy)
.end
```

```
    cmp     r2, 0
    ble     .Ldone      pre loop guard

    lsl     r2, r2, 2    //convert cnt to int size
    add     r3, r0, r2  // loop term pointer

.Ldo:
    ldr     r4, [r0]     // load from src
    str     r4, [r1]     // store to dest

    add     r0, r0, 4    // src++
    add     r1, r1, 4    // dst++

    cmp     r0, r3       // src < term pointer?
    blt     .Ldo
                                loop guard

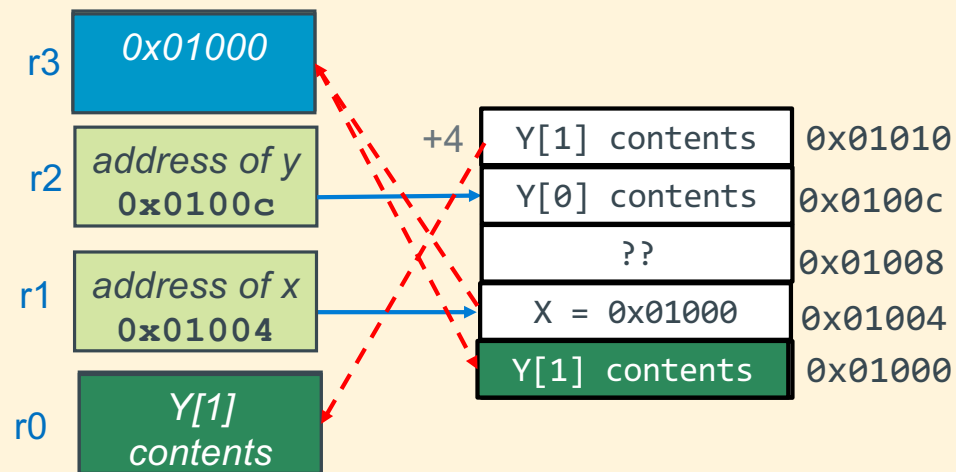
.Ldone:
```

ldr/str practice - 3

r1 contains Address of X (defined as `int *X`) in memory; r1 points at X

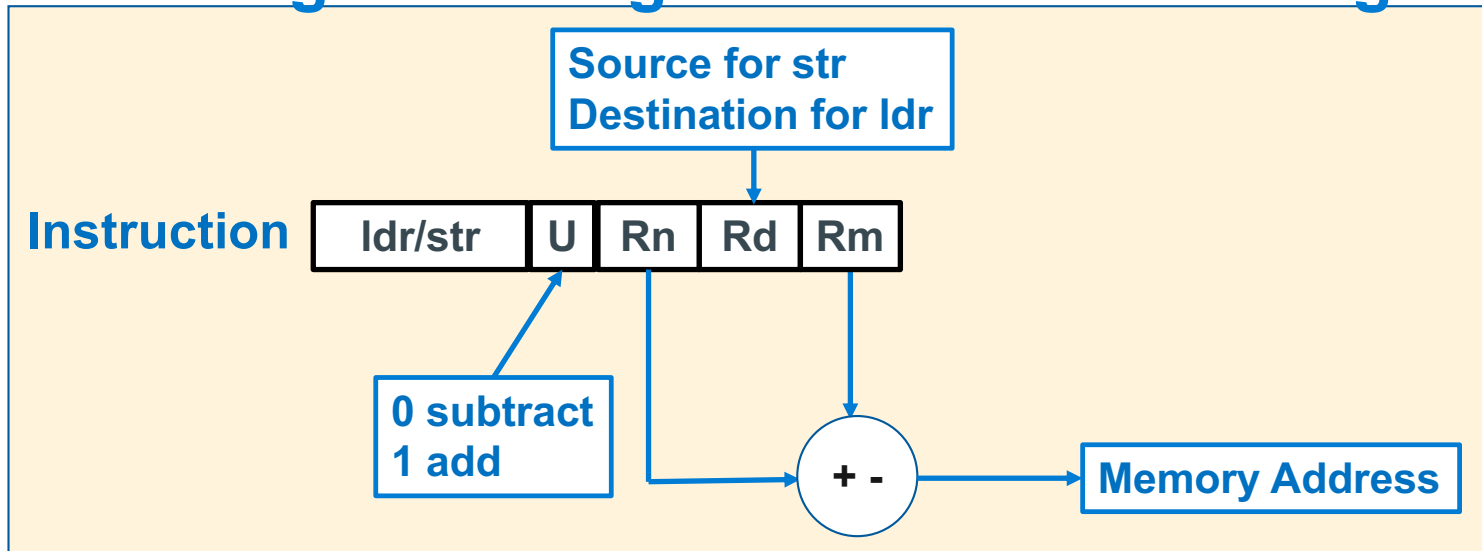
r2 contains Address of Y (defined as `int Y[2]`) in memory; r2 points at `&(Y[0])`

`write *X = Y[1];`



```
ldr    r0, [r2, 4]    // r0 ← y[1]
ldr    r3, [r1]        // r3 ← x
str     r0, [r3]       // *x ← y[1]
```

ldr/str Base Register + Register Offset Addressing



Pointer Address = Base Register + Register Offset

- **Unsigned** offset integer **in a register (bytes)** is either added/subtracted from the **pointer address** in the **base register**

Syntax	Address	Examples
<code>ldr/str Rd, [Rn +/- Rm]</code>	$Rn + \text{or} - Rm$	<code>ldr r0, [r5, r4]</code> <code>str r1, [r5, r4]</code>

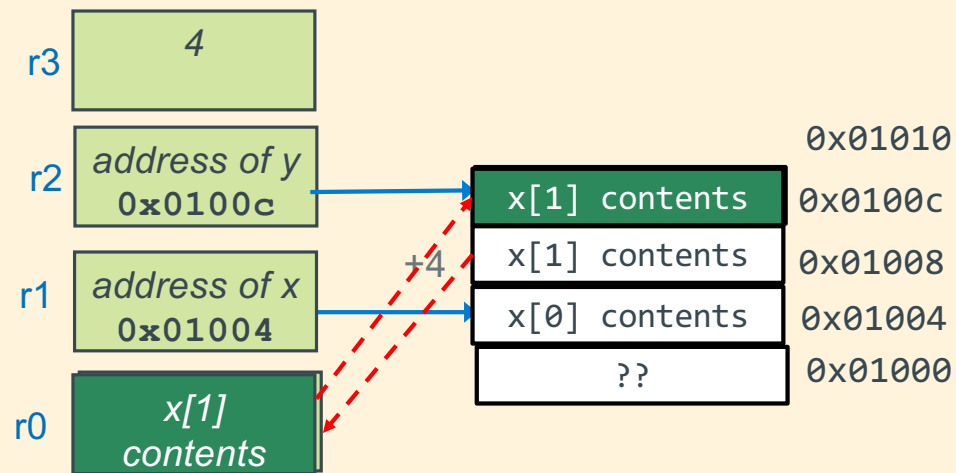
ldr/str practice - 4

r1 contains Address of X (defined as `int X[2]`) in memory; r1 points at `&(x[0])`

r2 contains Address of Y (defined as `int Y`) in memory; r2 points at Y

r3 contains a 4

write `Y = X[1];`



```
ldr    r0, [r1, r3]  // r0 ← x[1]
```

```
str    r0, [r2]      // y ← x[1]
```

Base Register + Register Offset Version

```
.arch armv6
.arm
.fpu vfp
.syntax unified
.text
.global icpy
.type icpy, %function
.equ FP_OFF, 12
// r0 contains int *src
// r1 contains int *dst
// r2 contains int cnt
// r3 use as loop counter
// r4 use as temp
```

```
icpy:
    push    {r4, r5, fp, lr}
    add     fp, sp, FP_OFF
    // see right ->
    sub     sp, fp, FP_OFF
    pop     {r4, r5, fp, lr}
    bx     lr
    .size icpy, (. - cpy)
.end
```

```
    cmp     r2, 0
    ble     .Ldone
    lsl     r2, r2, 2
    mov     r3, 0
.Ldo:
    ldr     r4, [r0, r3]
    str     r4, [r1, r3]
    add     r3, r3, 4
    cmp     r3, r2
    blt     .Ldo
```

pre loop guard

loop guard

.Ldone:

one increment
covers both arrays

Base Register + Register Offset With chars

```
#include <stdio.h>
#include <stdlib.h>
#define SZ 6
void cpy(char *, char *, int);
int main(void)
{
    char src[SZ] =
        {'a', 'b', 'c', 'd', 'e', '\0'};
    char dst[SZ];

    cpy(src, dst, SZ);
    printf("%s\n", dst);
    return EXIT_SUCCESS;
}
```

```
    cmp    r2, 0
    ble    .Ldone

    mov    r3, 0           // initialize counter
.Ldo:
    ldrb   r4, [r0, r3]    // load from src
    strb   r4, [r1, r3]    // store to dest
    add    r3, r3, 1       // counter++
    cmp    r3, r2          // count < r3
    blt    .Ldo

.Ldone:
```

Reference: Addressing Mode Summary for use in CSE30

index Type	Example	Description
Pre-index immediate	<code>ldr r1, [r0]</code>	$r1 \leftarrow \text{memory}[r0]$ $r0$ is unchanged
Pre-index immediate	<code>ldr r1, [r0, 4]</code>	$r1 \leftarrow \text{memory}[r0 + 4]$ $r0$ is unchanged
Pre-index immediate	<code>str r1, [r0]</code>	$\text{memory}[r0] \leftarrow r1$ $r0$ is unchanged
Pre-index immediate	<code>str r1, [r0, 4]</code>	$\text{memory}[r0 + 4] \leftarrow r1$ $r0$ is unchanged
Pre-index register	<code>ldr r1, [r0, +-r2]</code>	$r1 \leftarrow \text{memory}[r0 \pm r2]$ $r0$ is unchanged
Pre-index register	<code>str r1, [r0, +-r2]</code>	$\text{memory}[r0 \pm r2] \leftarrow r1$ $r0$ is unchanged

Base Register Addressing + Offset register

```
#include <stdio.h>
#include <stdlib.h>
int count(char *, int);
int main(void)
{
    char msg[] = "Hello CSE30! We Are CountinG UpPER cASe letters!";

    printf("%d\n", count(msg, sizeof(msg)/sizeof(*msg)));
    return EXIT_SUCCESS;
}
```

```
int count(char *ptr, int len)
{
    int cnt = 0;
    int i;

    for (i = 0; i < len; i++) {
        if ((ptr[i] >= 'A') && (ptr[i] <= 'Z'))
            cnt++;
    }
    return cnt;
}
```


Base Register + Offset register

```
.arch armv6
.arm
.fpu vfp
.syntax unified
.text
.global count
.type    count, %function
.equ     FP_OFF, 12
// r0 contains char *ptr
// r1 contains int len
// r2 contains int cnt
// r3 contains int i
// r4 contains char

count:
    push    {r4, r5, fp, lr}
    add     fp, sp, FP_OFF
    // see right ->
    sub     sp, fp, FP_OFF
    pop     {r4, r5, fp, lr}
    bx     lr
    .size count, (. - count)
.end
```

byte array
Also use ldrb here
offsets are 0,1,2,...

```
count:
    push    {r4, r5, fp, lr}
    add     fp, sp, FP_OFF

    mov     r2, 0
    cmp     r1, 0
    ble     .Ldone
    mov     r3, 0

.Lfor:
    cmp     r3, r1
    bge     .Ldone

    ldrb     r4, [r0, r3]
    cmp     r4, 'A'
    blt     .Lendif
    cmp     r4, 'Z'
    bgt     .Lendif
    add     r2, r2, 1

.Lendif:
    add     r3, r3, 1
    b       .Lfor

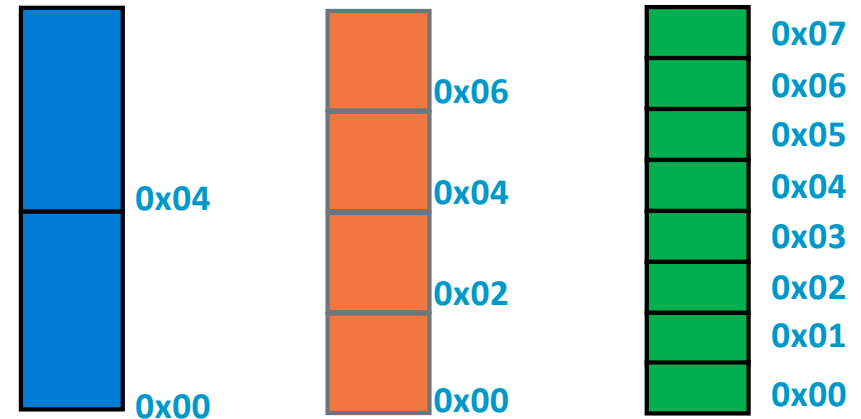
.Ldone:
    mov     r0, r2
```

loop guard

Defining Static Variables: Allocation and Initialization

Variable SIZE	Directive	.align	C static variable Definition	Assembler static variable Definition
8-bit char (1 byte)	.byte		char chx = 'A'; char string[] = {'A','B','C', 0};	chx: .byte 'A' string: .byte 'A','B',0x42,0
16-bit int (2 bytes)	.short	1	short length = 0x55aa;	length: .short 0x55aa
32-bit int (4 bytes)	.word .long	2	int dist = 5; int *distptr = &dist; unsigned int mask = 0xaa55aa55; int array[] = {12,~0x1,0xCD,-1};	dist: .word 5 distptr: .word dist mask: .word 0xff array: .word 12,~0x1,0xCD,-3
string with '\0'	.string		char class[] = "cse30";	class: .string "cse30"

SIZE	Address ends in	Align
8-bit char -1 byte	0b..0 or 0b..1	
16-bit int -2 bytes	0b.. 0	.align 1
32-bit int -4 bytes	0b.. 00	.align 2



Defining Static Variables: Allocation and Initialization

SIZE	Address ends in	Align
8-bit char -1 byte	0b..0 or 0b..1	
16-bit int -2 bytes	0b..0	.align 1
32-bit int -4 bytes	0b..00	.align 2

```
int num;           //4 bytes
int *ptr = &num;    //4 bytes
char *lit = "456"; //4 bytes, "456" string literal
char msg[] = "123"; //4 bytes - array
```

read-only
string literal

```
.bss
    .align 2
num:    .word 0

.data
    .align 2
ptr:    .word num
    .align 2
lit:    .word .Lmsg

msg:    .string "123"

.section .rodata
.Lmsg:  .string "456"
```

initializes
a pointer

Defining Static Array Variables

Label: .size_directive expression, ... expression

```
In C:      int int_buf[100];
           int array[] = {1, 2, 3, 4, 5};
           char buffer[100];

.bss
           .align 2
int_buf:   .space 400    // convert 100 to 400 bytes
char_buf:  .space 100

.data
           .align 2
array:     .word 1, 2, 3, 4, 5
           .align 2
one_buf:   .space 100, 1    // 100 bytes each byte filled with 1
```

.space size, fill

- Allocates **size** bytes, each of which contain the value **fill**
- Both **size** and **fill** are absolute expressions
- If the comma and **fill** are **omitted**, **fill** is assumed to be **zero**
- **.bss section**: Must be used **without a specified fill**

Loading Static variable address into a register

- Tell the assembler load the address (Lvalue) of a label into a register:

`ldr/str Rd, =Label // Rd = address`

- Example to the right: $y = x$;*

two step to **load** a **memory** variable

- load the pointer to the memory
- read (load) from *pointer

two steps **store** to a **memory** variable

- load the pointer to the memory
- write (store) to *pointer

```
.bss
y: .space 4
```

```
.data
x: .word 200
```

```
.text
// function header
main:

// load the address, then contents
// using r2
ldr r2, =x      // int *r2 = &x
ldr r2, [r2]    // r2 = *r2;

// &x was only needed once above
// Note: r2 was a pointer then an int
// no "type" checking in assembly!

// store the contents of r2
ldr r1, =y      // int *r1 = &y
str r2, [r1]    // *r1 = r2
...
```

Loading large constants into a register:

Error: invalid constant (3ff) after fixup

- In data processing instructions, the field **imm8 + rotate 4 bits** is too small to store the immediate value, how do you get larger immediate values into a register?



fails



```
mov    r0, 1023
```

xxx.s:24: Error: invalid constant (3ff) after fixup

replacement

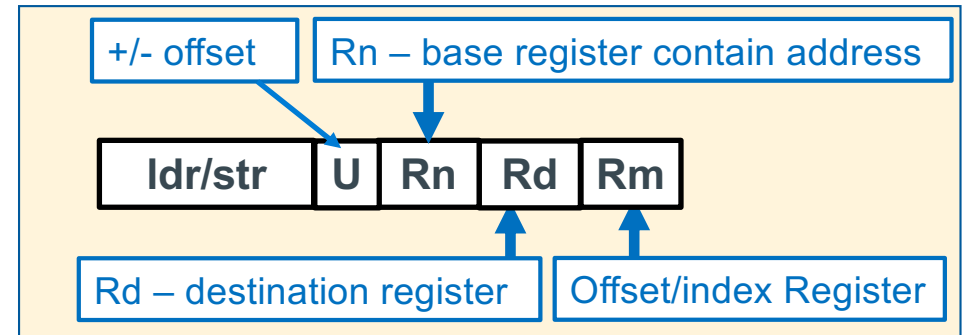
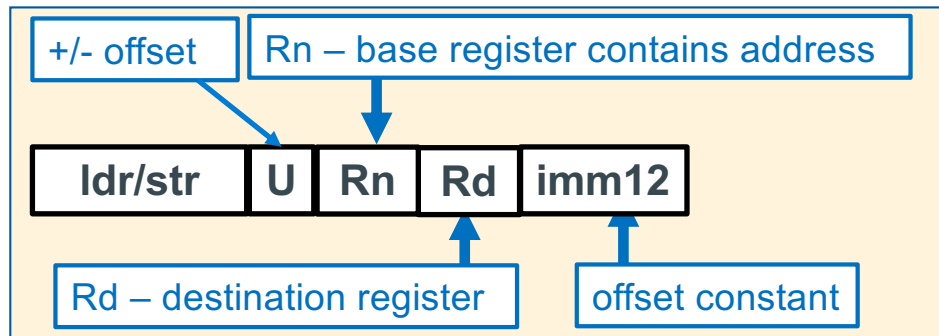


```
ldr    r0, =1023
```

- Answer: use **ldr** instruction with the constant as an operand: **=constant**
- Assembler creates a **literal table entry** with the **constant**

```
ldr    Rd, =constant    // =constant
ldr    r1, =0x2468abcd   // loads the constant 0x246abcd into r1
```

LDR/STR – Register To/From Memory Copy



```
ldr/str Rd, [Rn, +/- imm12] // base register pointer + offset  imm12 in bytes
                             -4095 <= imm12 <= 4095 (bytes)
ldr/str Rd, [Rn]             // base register pointer + 0 (imm12 is 0)
ldr/str Rd, [Rn, +/- Rm]     // base register pointer +/- offset register
```

```
ldr      r1, =var_x           // r1 = &var_x
str      r1, =mylabel+4       // *(mylabel+4) = r1
ldr      r1, =0x246abcd       // load an immediate into r1
ldr      r1, [r3]             // y = *r3 (4 bytes)
str      r1, [r0]             // *r0 = r1
ldr      r1, [r3, -4]         // y = *(r3 - 4) (4 bytes)
str      r1, [r0, r2]         // *(r0 + r2) = r1
```

Function Calls, Parameters and Locals: Requirements

```
int
main(int argc, char *argv[])
{
    int x, z = 4;

    x = a(z);
    z = b(z); ←
    return EXIT_SUCCESS;
}

int
a(int n)
{
    int i = 0;
    if (n == 1)
        i = b(n); ←
    return i;
}

int
b(int m)
{
    return m+1; ←
    /* the return cannot be done with a
    branch */
}
```

- Since **b()** is called both by main and a() how does the **return m+1** statement in b() know where to return to? (Obviously, it cannot be a branch)
- Where are the parameters (args) to a function stored so the function has a copy that it can alter?
- Where is the return value from a function call stored?
- How are Automatic variables *lifetime* and *scope* implemented?
 - When you enter a variables scope: memory is allocated for the variables
 - When you leave a variable scope: memory lifetime is ended (memory can be reused -- deallocated) – contents are **no longer valid**

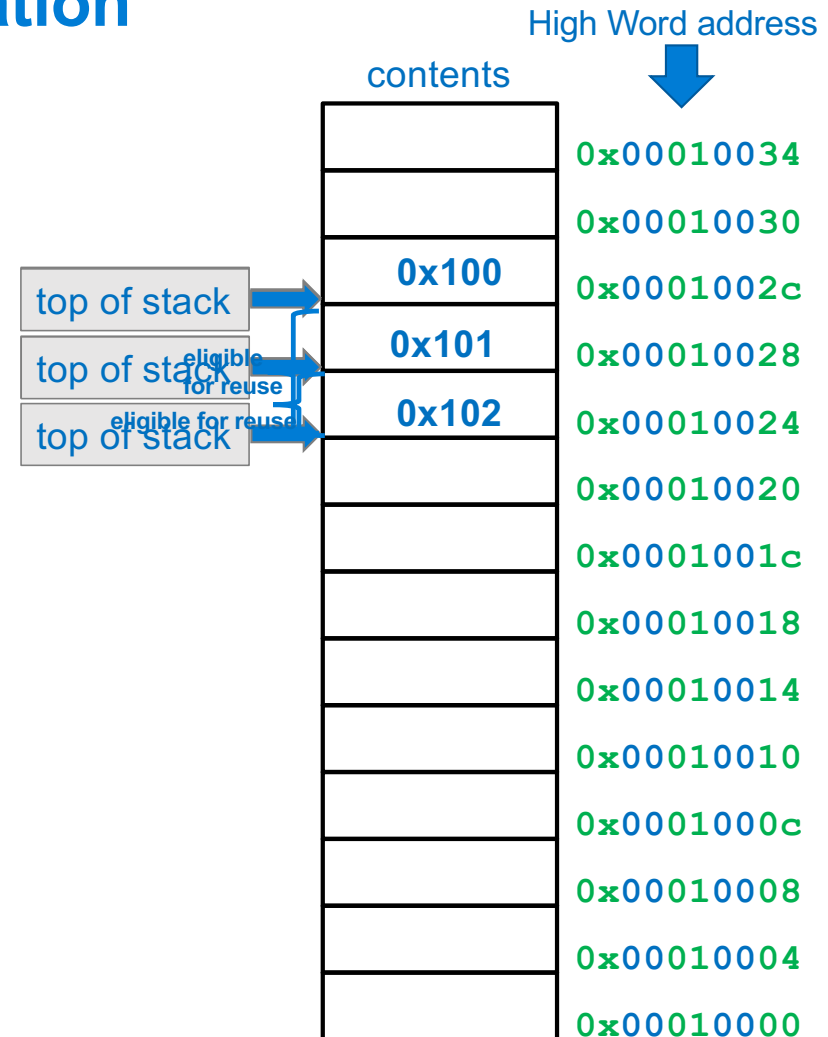
Data Structure Review: Stack Operation

- A Stack Implements a **last-in first-out** (LIFO) protocol
- **Stacks** are expandable and grow downward from high memory address towards low memory address
- **Stack pointer** always points at the **top of stack**
 - contains the starting address of the top element
- New items are **pushed** (*added*) onto the **top of the stack** by **subtracting from the stack pointer the size of the element** and then writing the element

push (sp - element size) & write

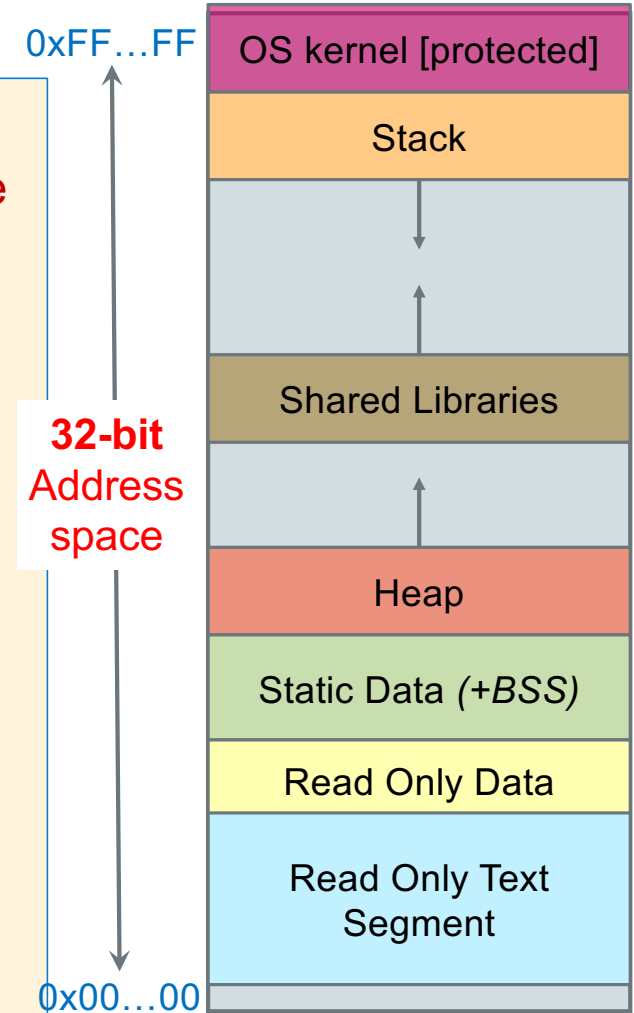
- Existing items are **popped** (*removed*) from the top of the stack by **adding to the stack pointer the size of the element** (leaving the **old contents unchanged**)

pop (sp + element size)



Stack Segment: Support of Functions

- The stack consists of a series of "*stack frames*" or "*activation frames*", one is **created** each time a function is called at runtime
- Each frame represents a function that is currently being executed and has not yet completed (why activation frame)
- A function's stack "frame" goes away when the function returns
- Specifically, a new stack frame is
 - allocated (**pushed** on the stack) for each function call (**contents are not implicitly zeroed**)
 - deallocated (**popped** from the stack) on function return
- **Stack frame** contains:
 - Local variables, parameters of function called
 - Where to return to which caller when the function completes (the return address)

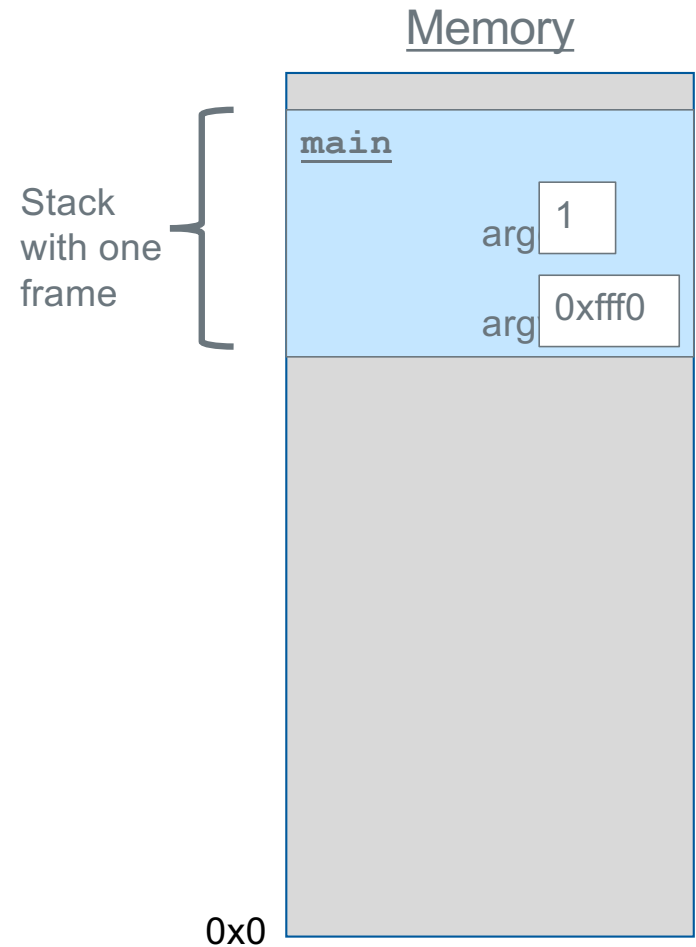


The Stack

```
void func2() {  
    int d = 0;  
}
```

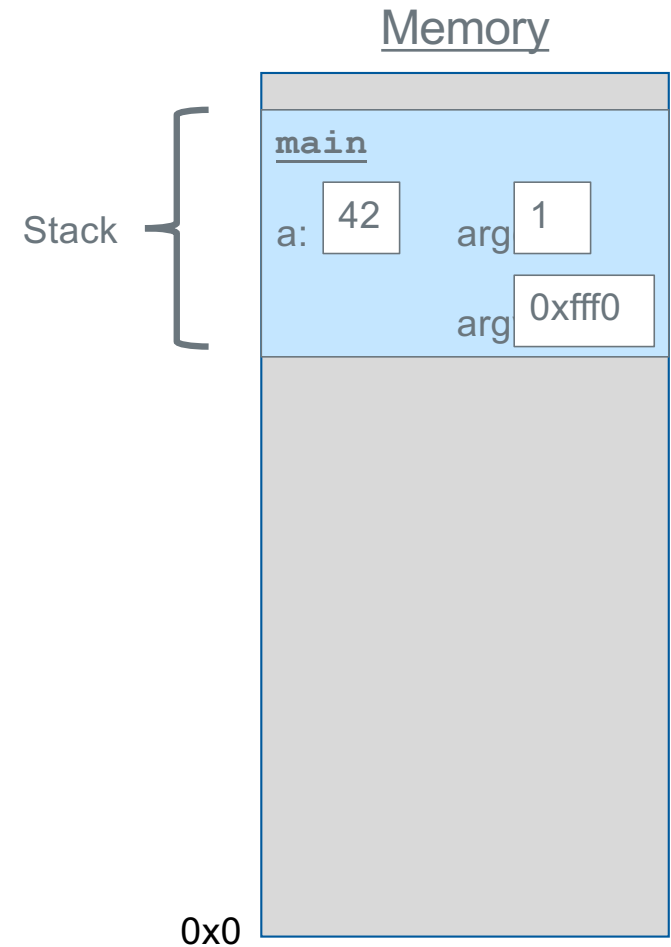
```
void func1() {  
    int c = 99;  
    func2();  
}
```

```
int main(int argc, char *argv[]) {  
    int a = 42;  
    int b = 17;  
    func1();  
    printf("Done.");  
    return 0;  
}
```



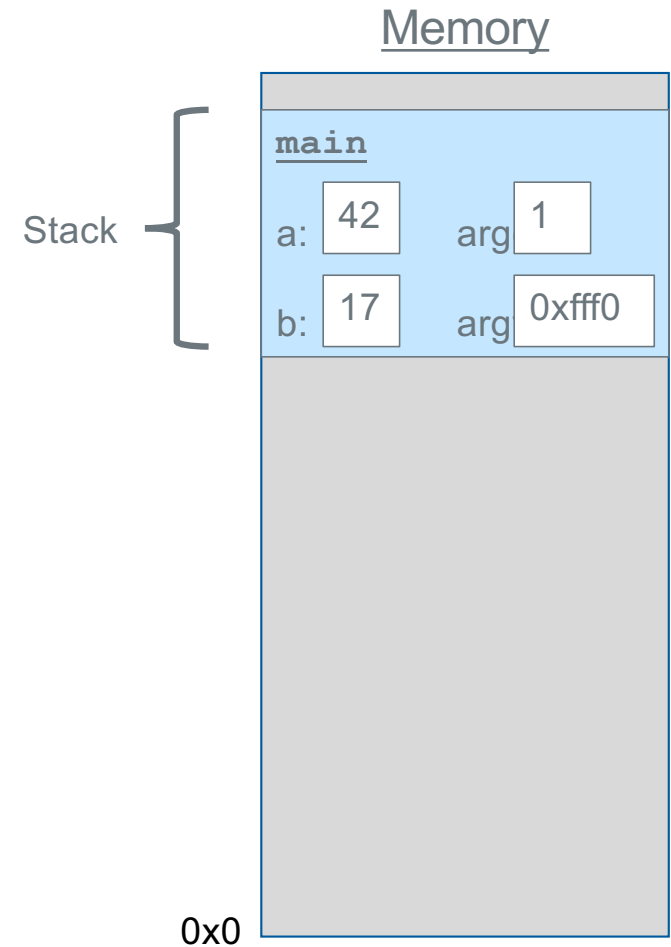
The Stack

```
void func2() {  
    int d = 0;  
}  
  
void func1() {  
    int c = 99;  
    func2();  
}  
  
int main(int argc, char *argv[]) {  
    int a = 42;  
    int b = 17;  
    func1();  
    printf("Done.");  
    return 0;  
}
```



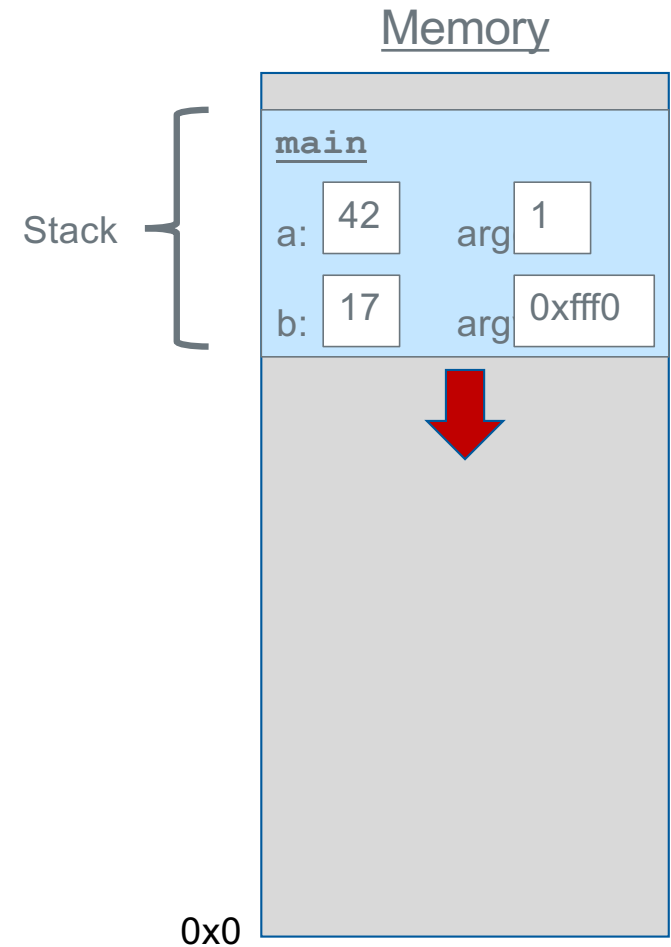
The Stack

```
void func2() {  
    int d = 0;  
}  
  
void func1() {  
    int c = 99;  
    func2();  
}  
  
int main(int argc, char *argv[]) {  
    int a = 42;  
    int b = 17;  
    func1();  
    printf("Done.");  
    return 0;  
}
```



The Stack

```
void func2() {  
    int d = 0;  
}  
  
void func1() {  
    int c = 99;  
    func2();  
}  
  
int main(int argc, char *argv[]) {  
    int a = 42;  
    int b = 17;  
    func1();  
    printf("Done.");  
    return 0;  
}
```

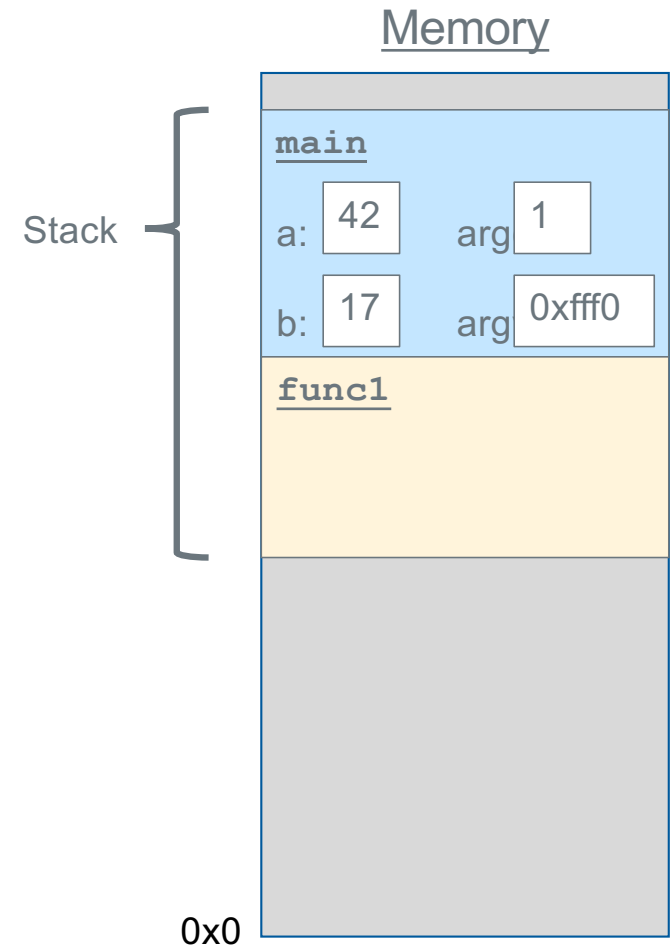


The Stack

```
void func2() {  
    int d = 0;  
}
```

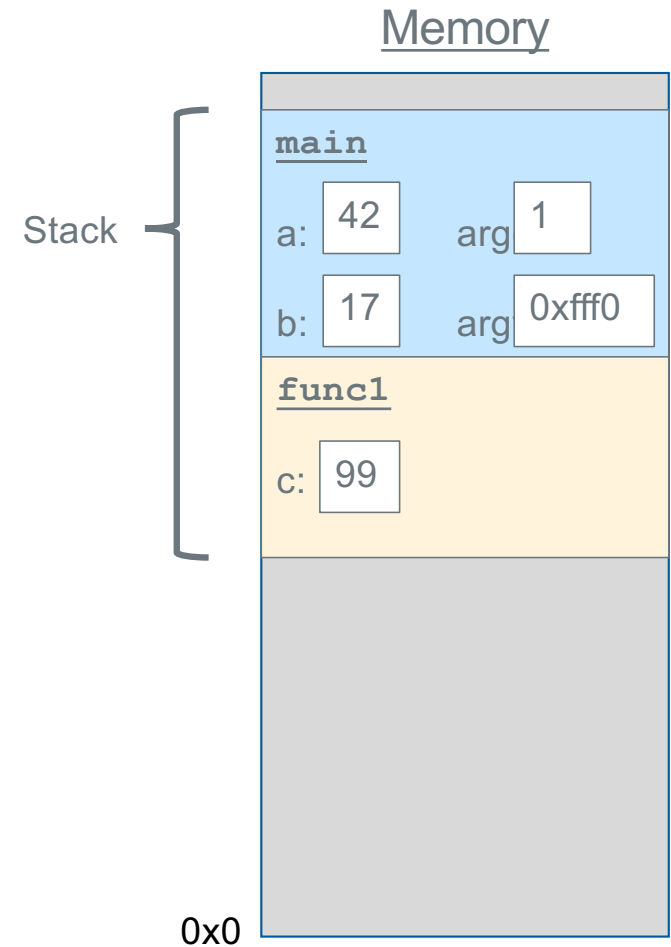
```
void func1() {  
    int c = 99;  
    func2();  
}
```

```
int main(int argc, char *argv[]) {  
    int a = 42;  
    int b = 17;  
    func1();  
    printf("Done.");  
    return 0;  
}
```



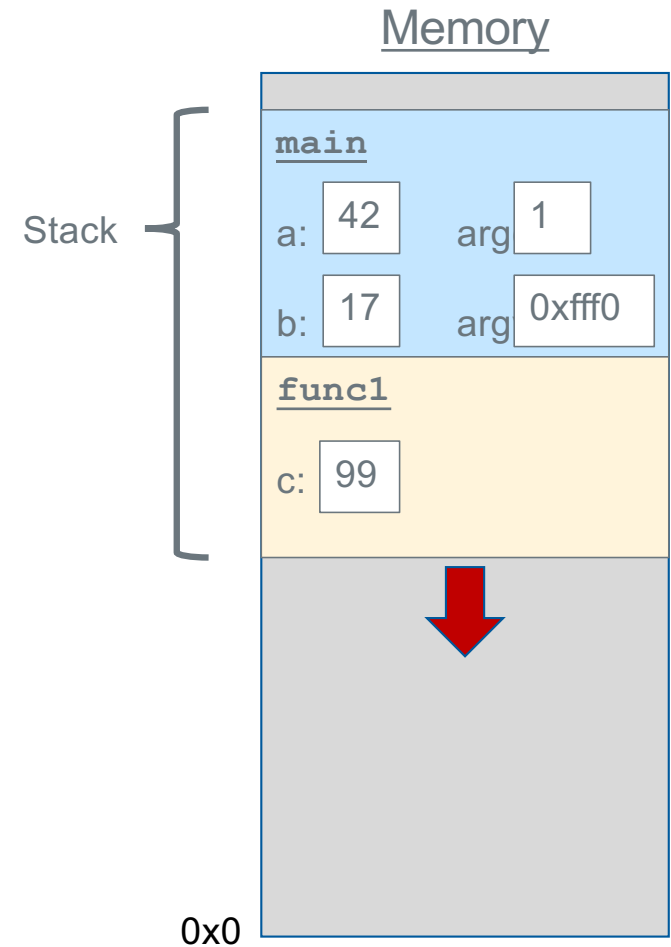
The Stack

```
void func2() {  
    int d = 0;  
}  
  
void func1() {  
    int c = 99;  
    func2();  
}  
  
int main(int argc, char *argv[]) {  
    int a = 42;  
    int b = 17;  
    func1();  
    printf("Done.");  
    return 0;  
}
```



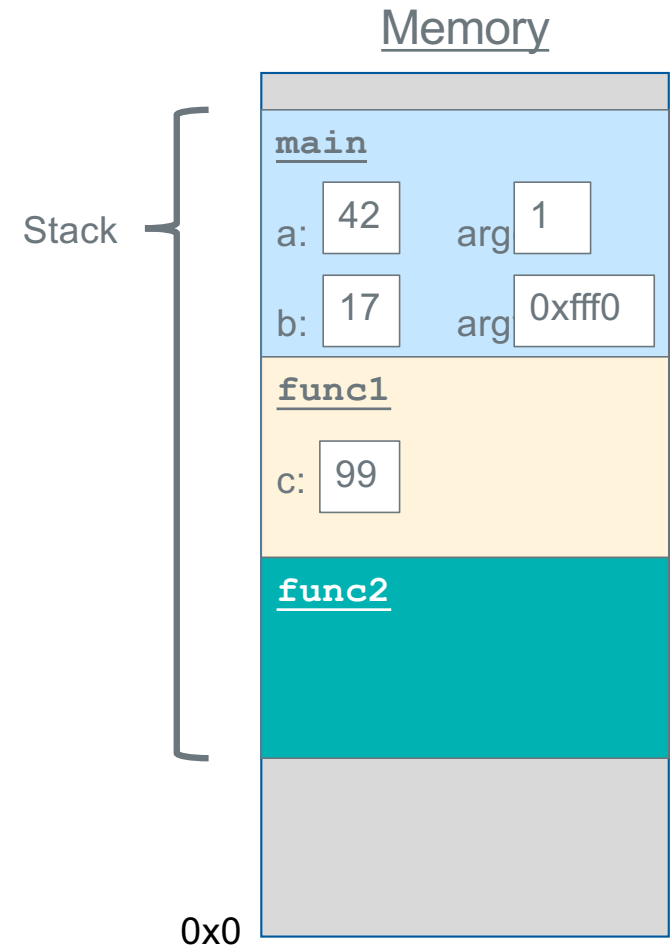
The Stack

```
void func2() {  
    int d = 0;  
}  
  
void func1() {  
    int c = 99;  
    func2();  
}  
  
int main(int argc, char *argv[]) {  
    int a = 42;  
    int b = 17;  
    func1();  
    printf("Done.");  
    return 0;  
}
```



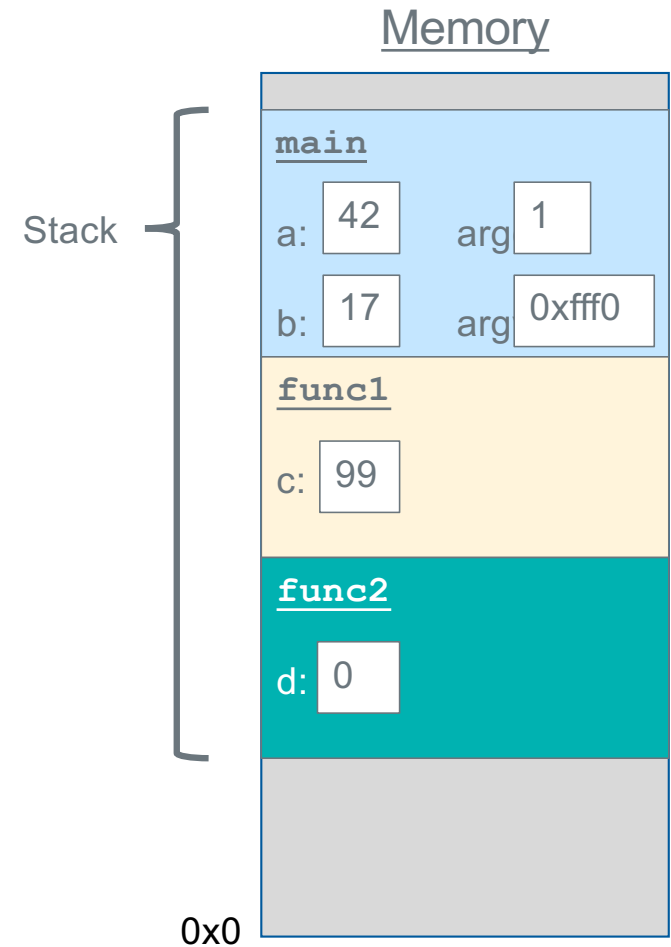
The Stack

```
void func2() {  
    int d = 0;  
}  
  
void func1() {  
    int c = 99;  
    func2();  
}  
  
int main(int argc, char *argv[]) {  
    int a = 42;  
    int b = 17;  
    func1();  
    printf("Done.");  
    return 0;  
}
```



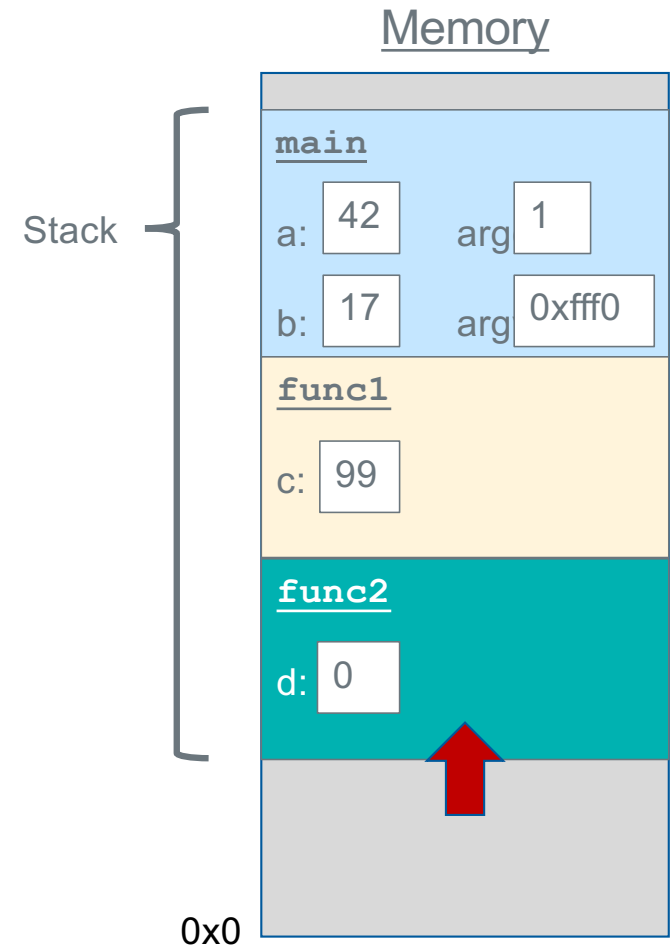
The Stack

```
void func2() {  
    int d = 0;  
}  
  
void func1() {  
    int c = 99;  
    func2();  
}  
  
int main(int argc, char *argv[]) {  
    int a = 42;  
    int b = 17;  
    func1();  
    printf("Done.");  
    return 0;  
}
```



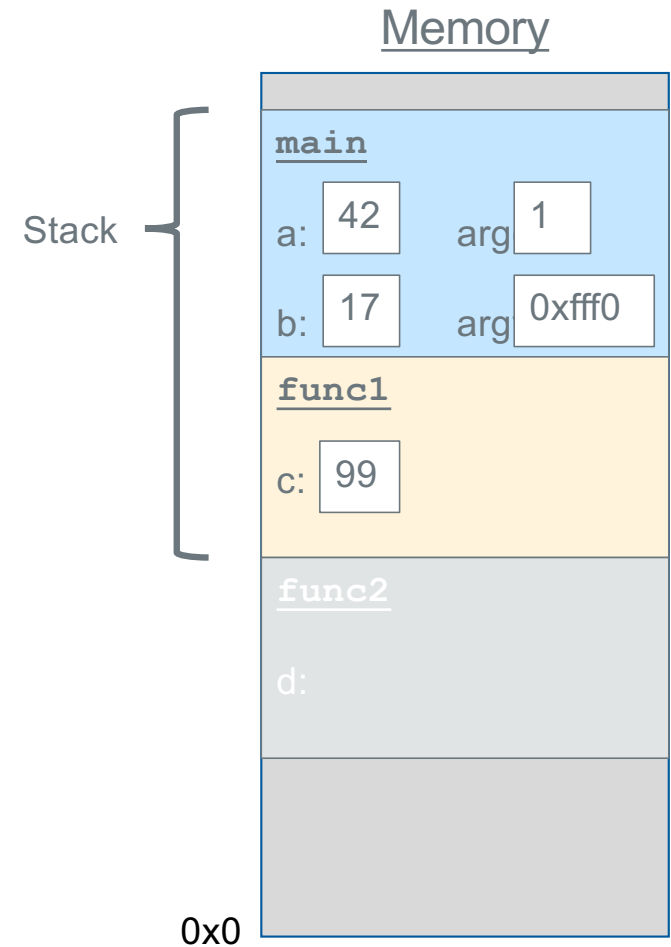
The Stack

```
void func2() {  
    int d = 0;  
}  
  
void func1() {  
    int c = 99;  
    func2();  
}  
  
int main(int argc, char *argv[]) {  
    int a = 42;  
    int b = 17;  
    func1();  
    printf("Done.");  
    return 0;  
}
```



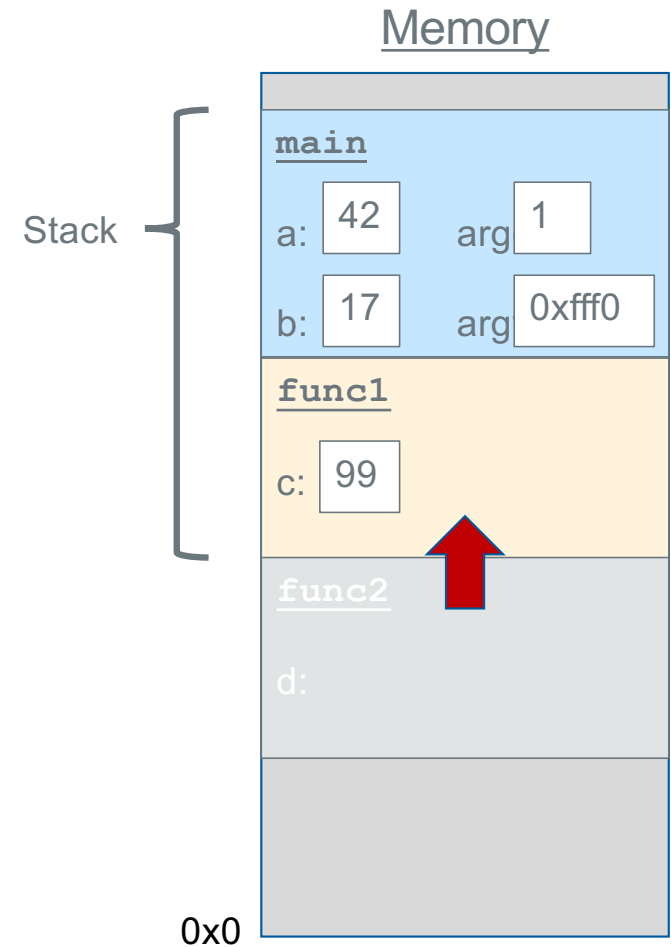
The Stack

```
void func2() {  
    int d = 0;  
}  
  
void func1() {  
    int c = 99;  
    func2();  
}  
  
int main(int argc, char *argv[]) {  
    int a = 42;  
    int b = 17;  
    func1();  
    printf("Done.");  
    return 0;  
}
```



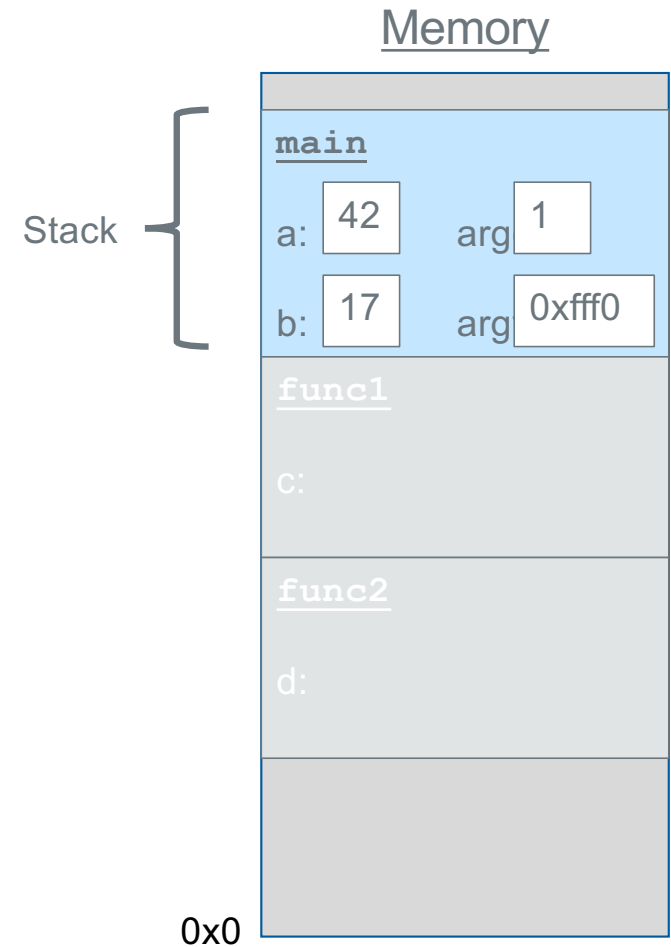
The Stack

```
void func2() {  
    int d = 0;  
}  
  
void func1() {  
    int c = 99;  
    func2();  
}  
  
int main(int argc, char *argv[]) {  
    int a = 42;  
    int b = 17;  
    func1();  
    printf("Done.");  
    return 0;  
}
```



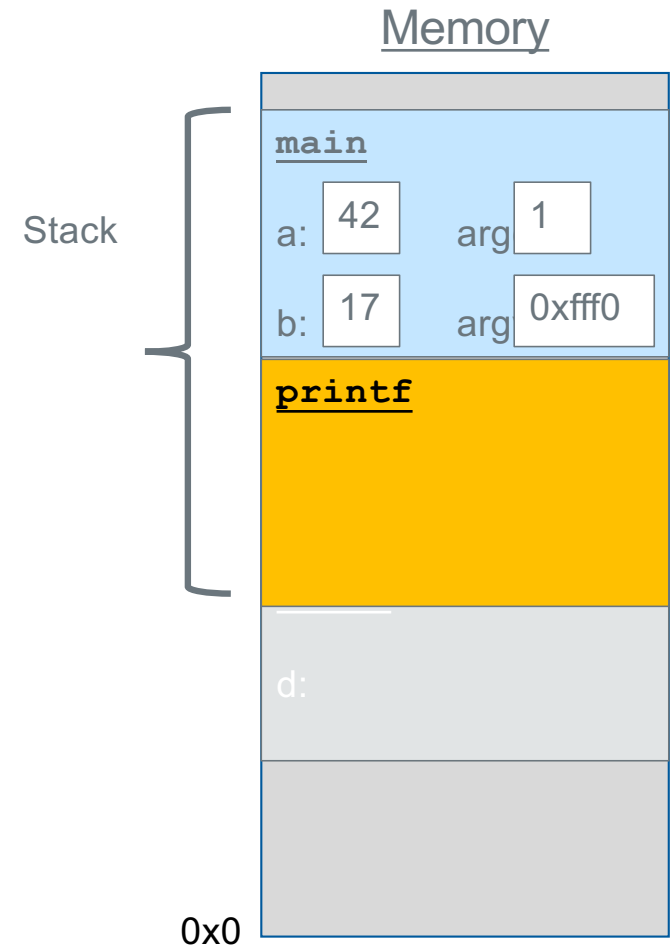
The Stack

```
void func2() {  
    int d = 0;  
}  
  
void func1() {  
    int c = 99;  
    func2();  
}  
  
int main(int argc, char *argv[]) {  
    int a = 42;  
    int b = 17;  
    func1();  
    printf("Done.");  
    return 0;  
}
```



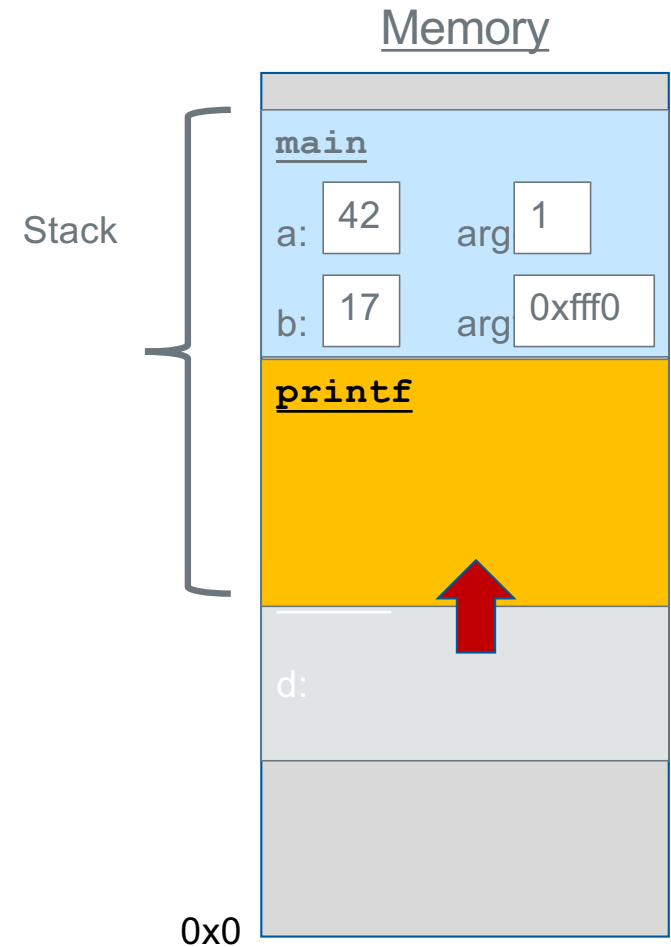
The Stack

```
void func2() {  
    int d = 0;  
}  
  
void func1() {  
    int c = 99;  
    func2();  
}  
  
int main(int argc, char *argv[]) {  
    int a = 42;  
    int b = 17;  
    func1();  
    printf("Done.");  
    return 0;  
}
```



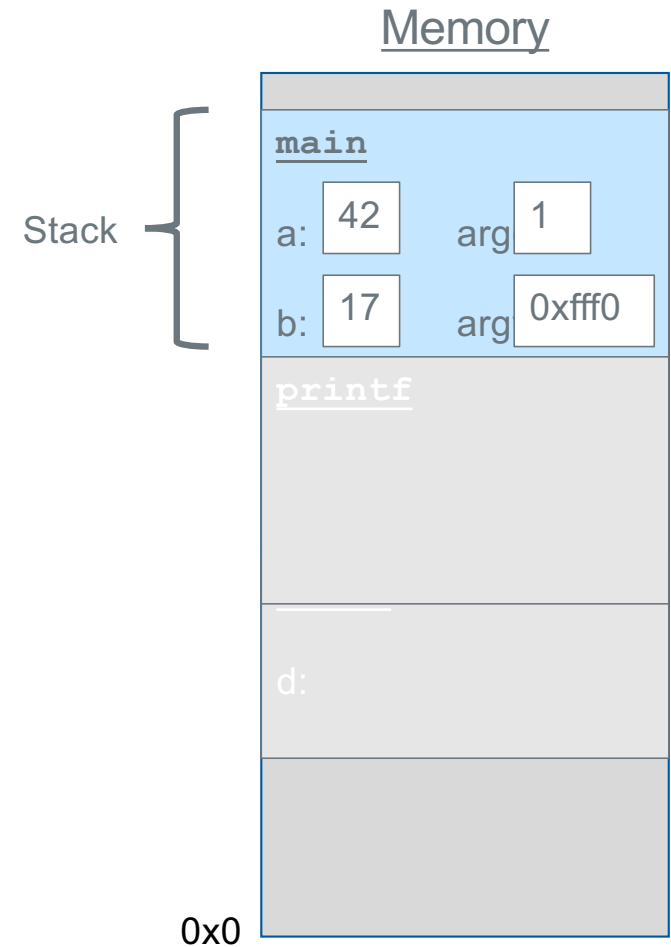
The Stack

```
void func2() {  
    int d = 0;  
}  
  
void func1() {  
    int c = 99;  
    func2();  
}  
  
int main(int argc, char *argv[]) {  
    int a = 42;  
    int b = 17;  
    func1();  
    printf("Done.");  
    return 0;  
}
```



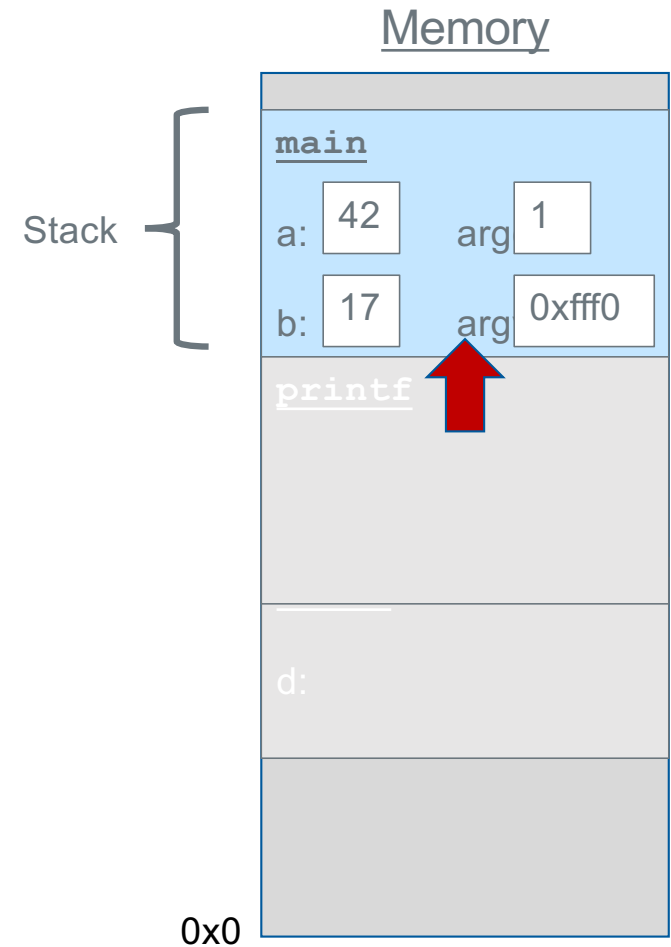
The Stack

```
void func2() {  
    int d = 0;  
}  
  
void func1() {  
    int c = 99;  
    func2();  
}  
  
int main(int argc, char *argv[]) {  
    int a = 42;  
    int b = 17;  
    func1();  
    printf("Done.");  
    return 0;  
}
```



The Stack

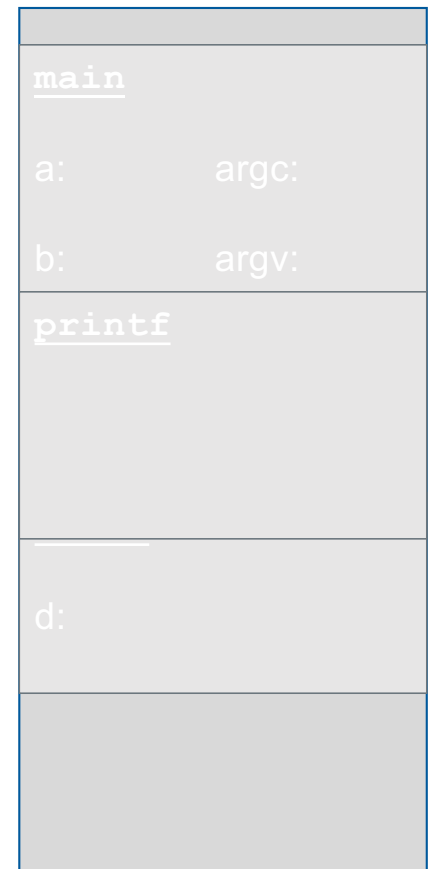
```
void func2() {  
    int d = 0;  
}  
  
void func1() {  
    int c = 99;  
    func2();  
}  
  
int main(int argc, char *argv[]) {  
    int a = 42;  
    int b = 17;  
    func1();  
    printf("Done.");  
    return 0;  
}
```



The Stack

```
void func2() {  
    int d = 0;  
}  
  
void func1() {  
    int c = 99;  
    func2();  
}  
  
int main(int argc, char *argv[]) {  
    int a = 42;  
    int b = 17;  
    func1();  
    printf("Done.");  
    return 0;  
}
```

Memory

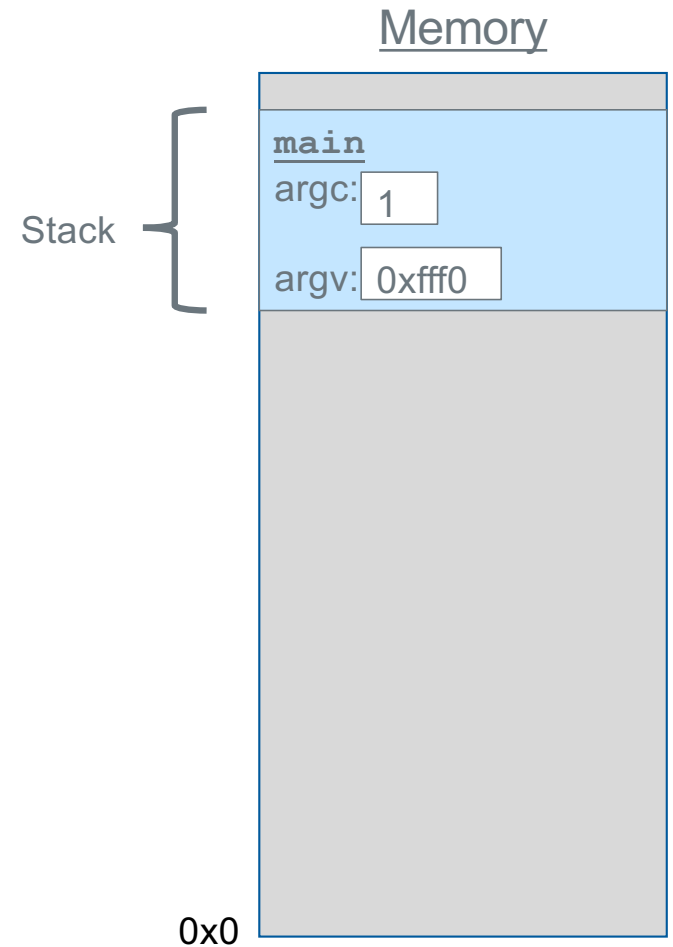


The Stack - Recursion

Each function **call** has its own *stack frame* for its own copy of variables

```
int factorial(int n) {  
    if (n == 1) {  
        return 1;  
    } else {  
        return n * factorial(n - 1);  
    }  
}
```

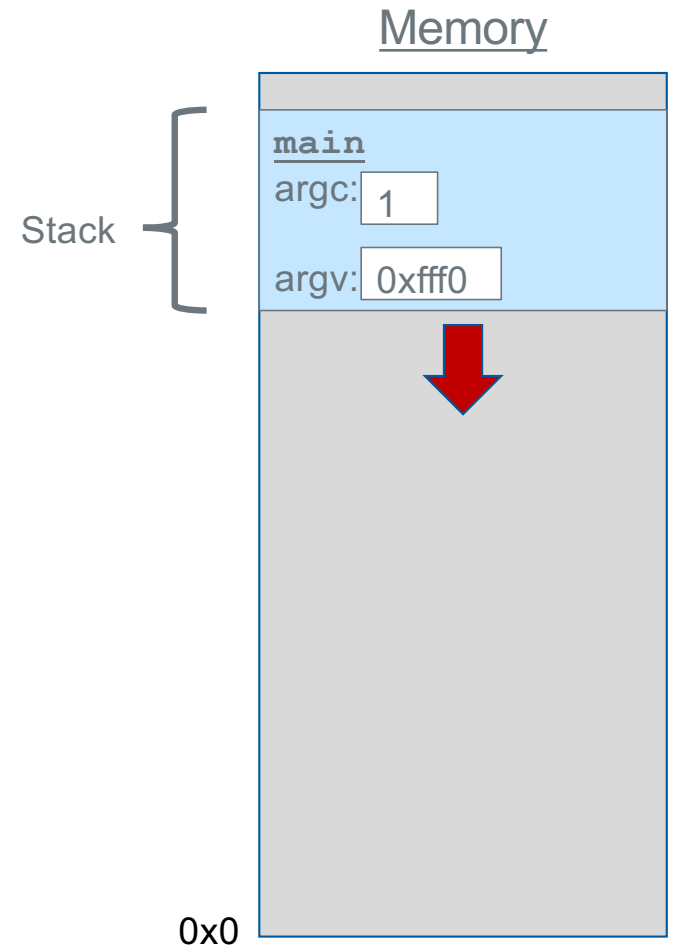
```
int main(int argc, char *argv[]) {  
    printf("%d", factorial(4));  
    return 0;  
}
```



The Stack

Each function **call** has its own *stack frame* for its own copy of variables.

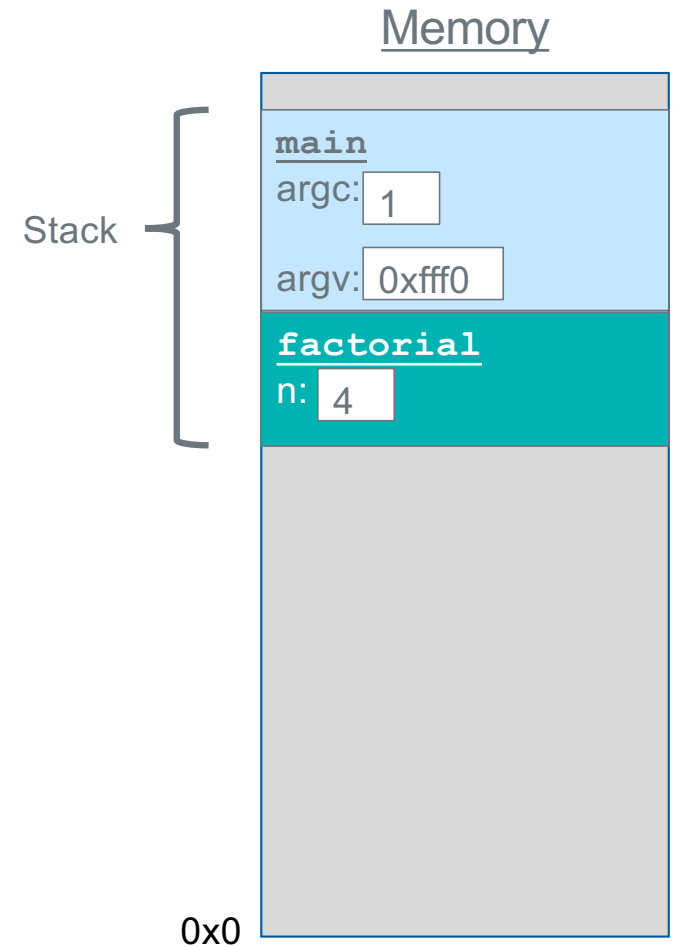
```
int factorial(int n) {  
    if (n == 1) {  
        return 1;  
    } else {  
        return n * factorial(n - 1);  
    }  
}  
  
int main(int argc, char *argv[]) {  
    printf("%d", factorial(4));  
    return 0;  
}
```



The Stack

Each function **call** has its own *stack frame* for its own copy of variables.

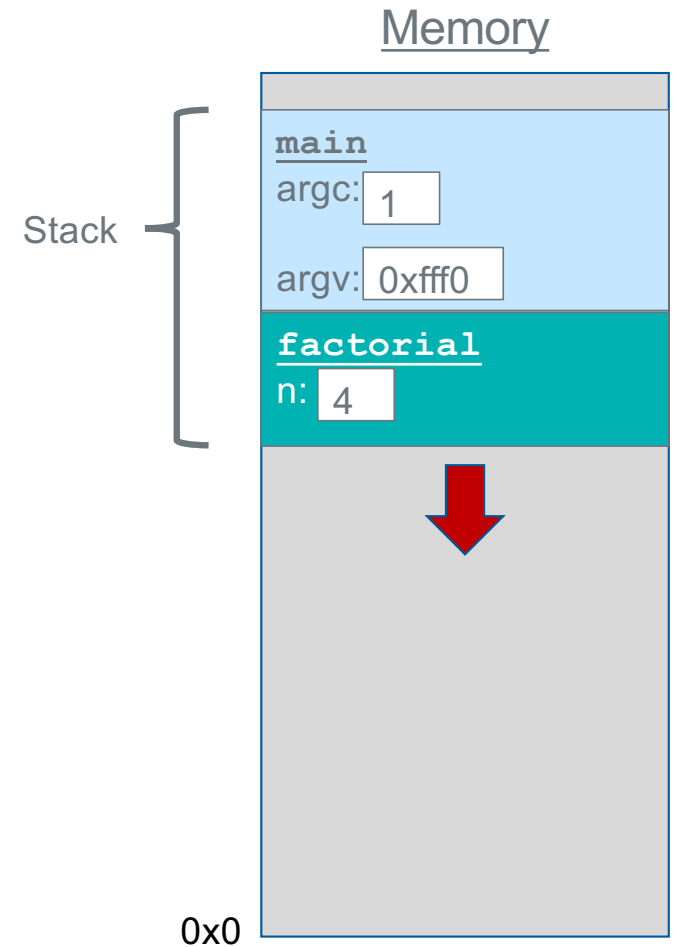
```
int factorial(int n) {  
    if (n == 1) {  
        return 1;  
    } else {  
        return n * factorial(n - 1);  
    }  
}  
  
int main(int argc, char *argv[]) {  
    printf("%d", factorial(4));  
    return 0;  
}
```



The Stack

Each function **call** has its own *stack frame* for its own copy of variables.

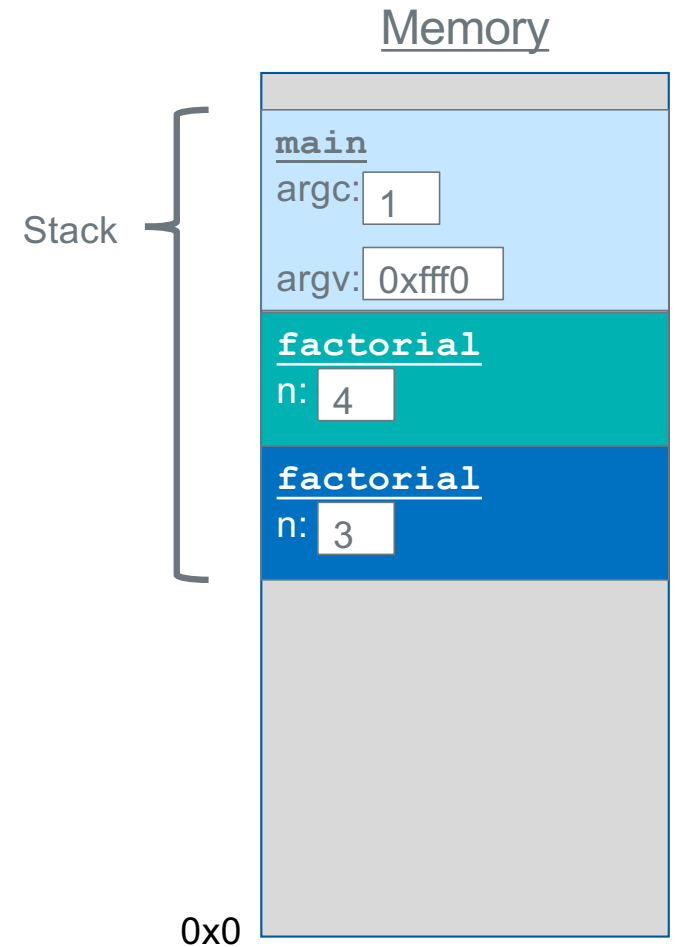
```
int factorial(int n) {  
    if (n == 1) {  
        return 1;  
    } else {  
        return n * factorial(n - 1);  
    }  
}  
  
int main(int argc, char *argv[]) {  
    printf("%d", factorial(4));  
    return 0;  
}
```



The Stack

Each function **call** has its own *stack frame* for its own copy of variables.

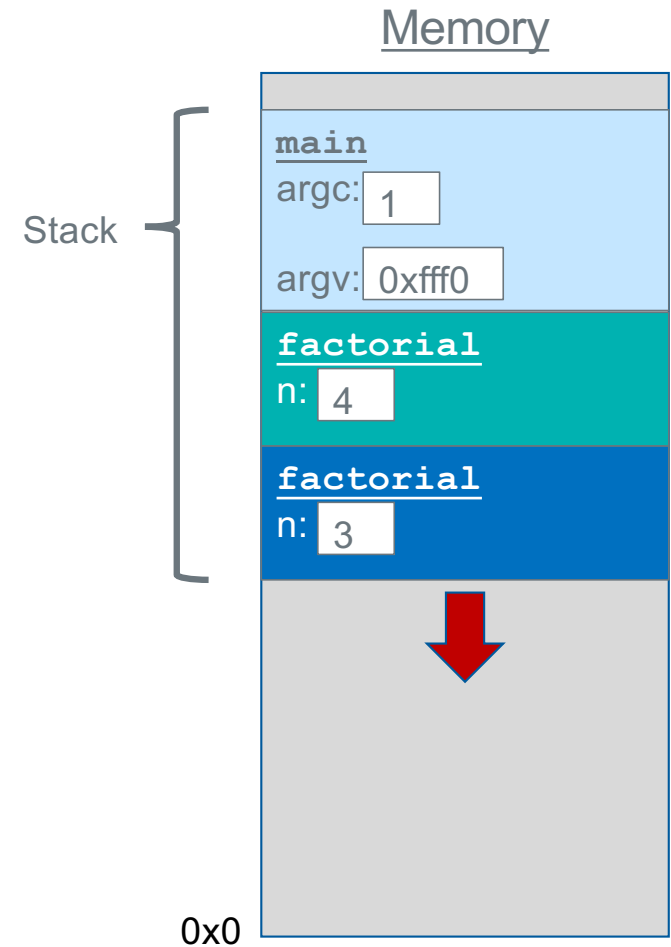
```
int factorial(int n) {  
    if (n == 1) {  
        return 1;  
    } else {  
        return n * factorial(n - 1);  
    }  
}  
  
int main(int argc, char *argv[]) {  
    printf("%d", factorial(4));  
    return 0;  
}
```



The Stack

Each function **call** has its own *stack frame* for its own copy of variables.

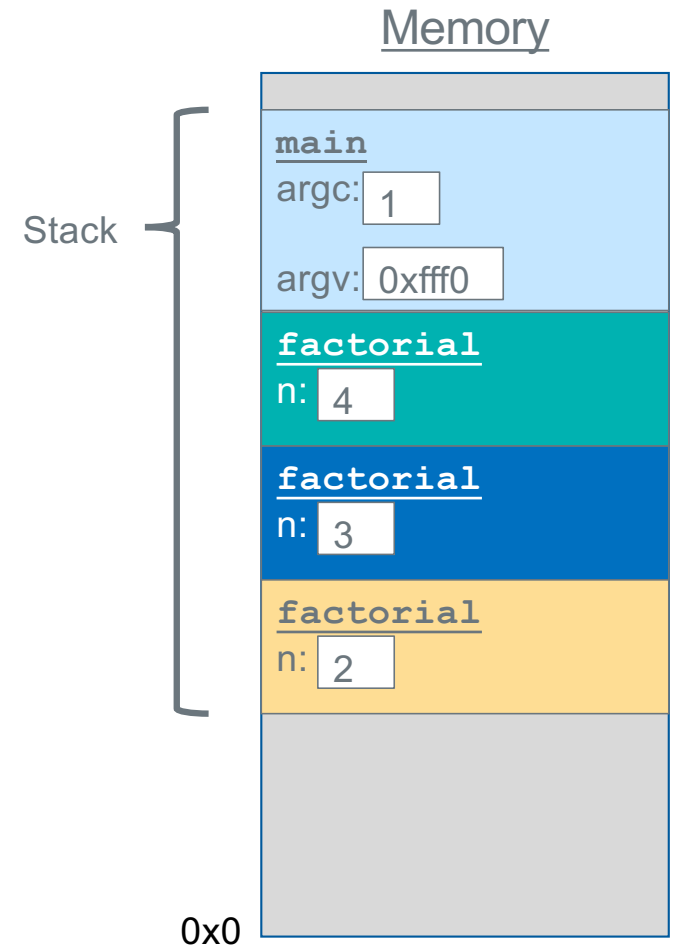
```
int factorial(int n) {  
    if (n == 1) {  
        return 1;  
    } else {  
        return n * factorial(n - 1);  
    }  
}  
  
int main(int argc, char *argv[]) {  
    printf("%d", factorial(4));  
    return 0;  
}
```



The Stack

Each function **call** has its own *stack frame* for its own copy of variables.

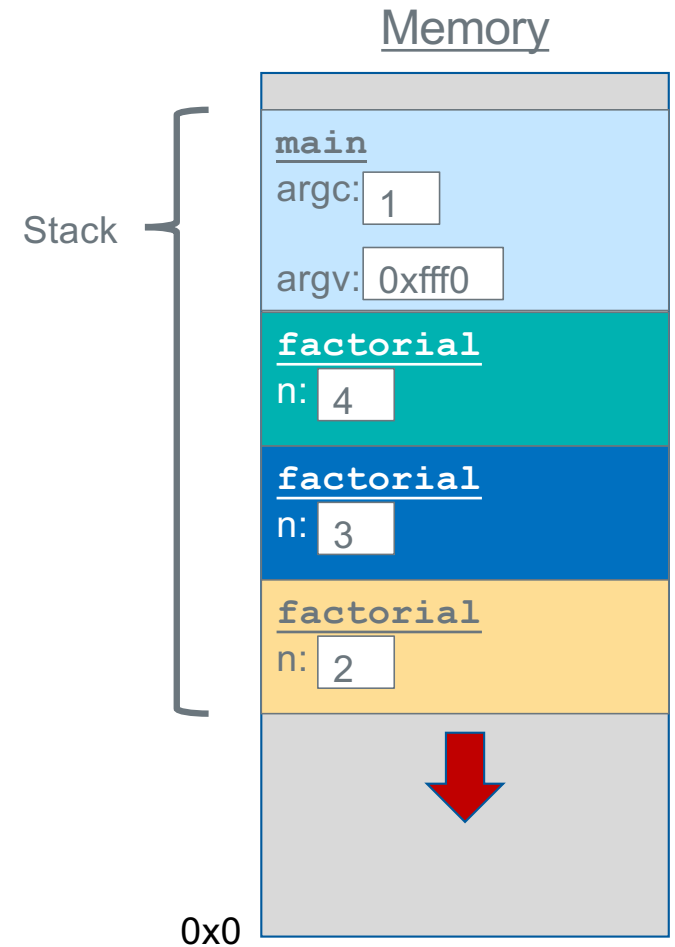
```
int factorial(int n) {  
    if (n == 1) {  
        return 1;  
    } else {  
        return n * factorial(n - 1);  
    }  
}  
  
int main(int argc, char *argv[]) {  
    printf("%d", factorial(4));  
    return 0;  
}
```



The Stack

Each function **call** has its own *stack frame* for its own copy of variables.

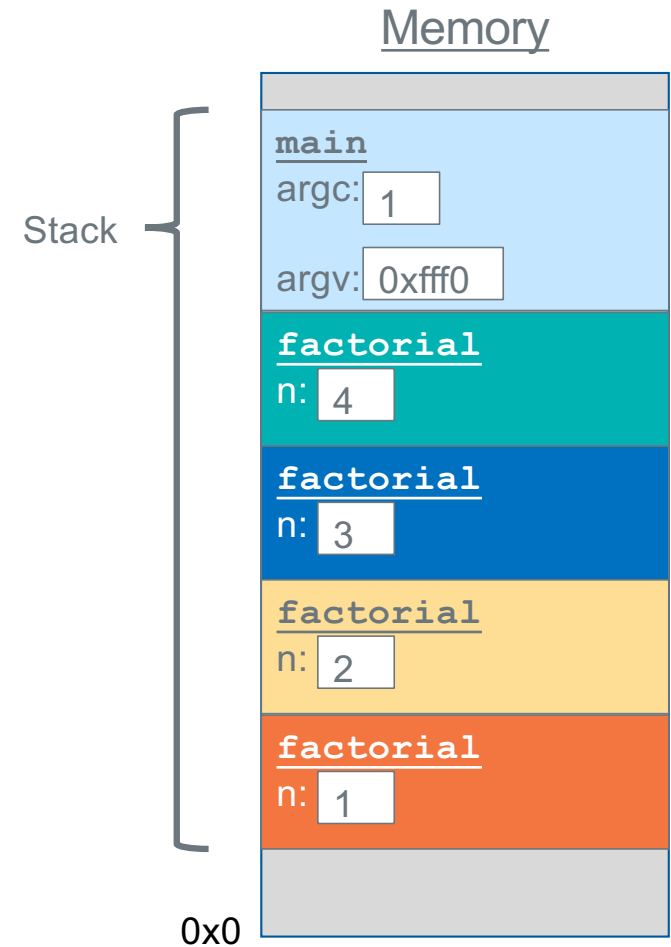
```
int factorial(int n) {  
    if (n == 1) {  
        return 1;  
    } else {  
        return n * factorial(n - 1);  
    }  
}  
  
int main(int argc, char *argv[]) {  
    printf("%d", factorial(4));  
    return 0;  
}
```



The Stack

Each function **call** has its own *stack frame* for its own copy of variables.

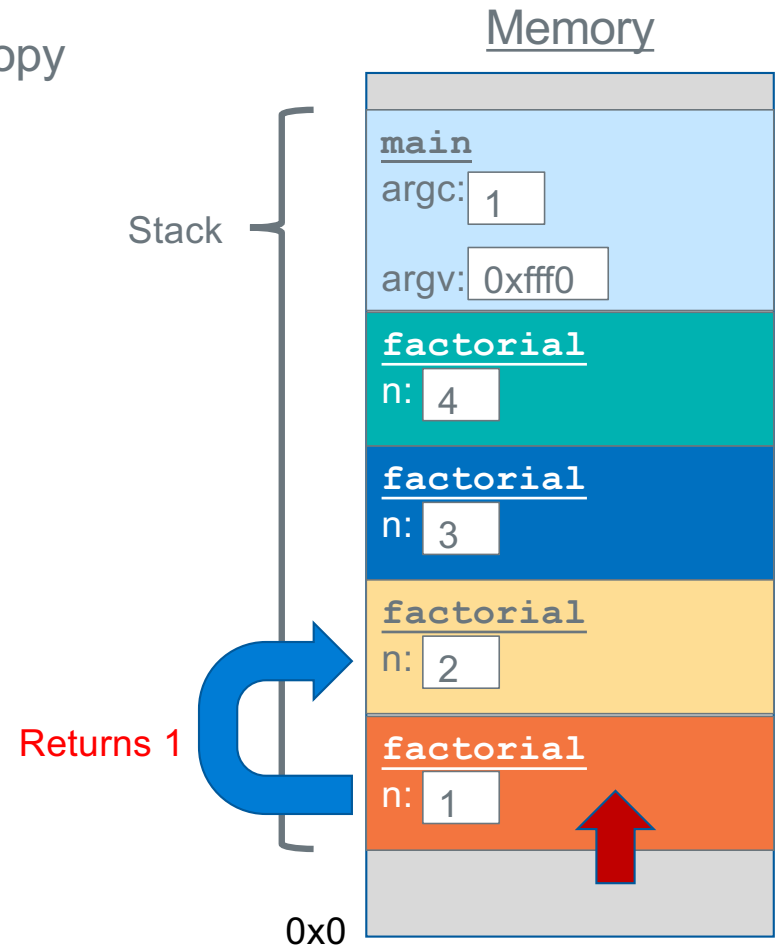
```
int factorial(int n) {  
    if (n == 1) {  
        return 1;  
    } else {  
        return n * factorial(n - 1);  
    }  
}  
  
int main(int argc, char *argv[]) {  
    printf("%d", factorial(4));  
    return 0;  
}
```



The Stack

Each function **call** has its own *stack frame* for its own copy of variables.

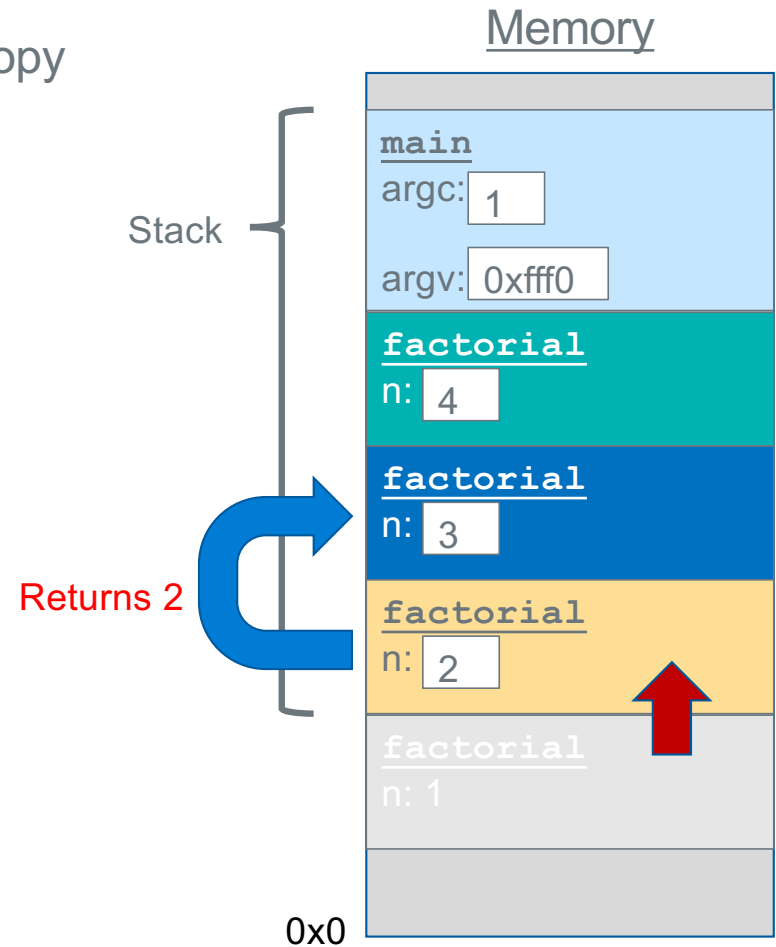
```
int factorial(int n) {  
    if (n == 1) {  
        return 1;  
    } else {  
        return n * factorial(n - 1);  
    }  
}  
  
int main(int argc, char *argv[]) {  
    printf("%d", factorial(4));  
    return 0;  
}
```



The Stack

Each function **call** has its own *stack frame* for its own copy of variables.

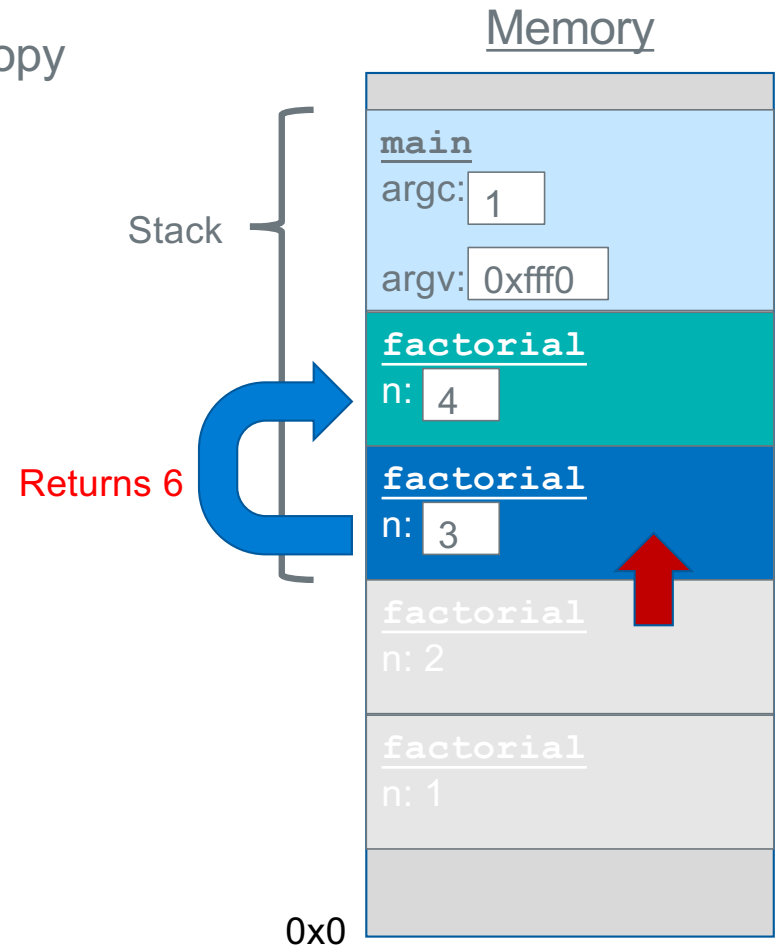
```
int factorial(int n) {  
    if (n == 1) {  
        return 1;  
    } else {  
        return n * factorial(n - 1);  
    }  
}  
  
int main(int argc, char *argv[]) {  
    printf("%d", factorial(4));  
    return 0;  
}
```



The Stack

Each function **call** has its own *stack frame* for its own copy of variables.

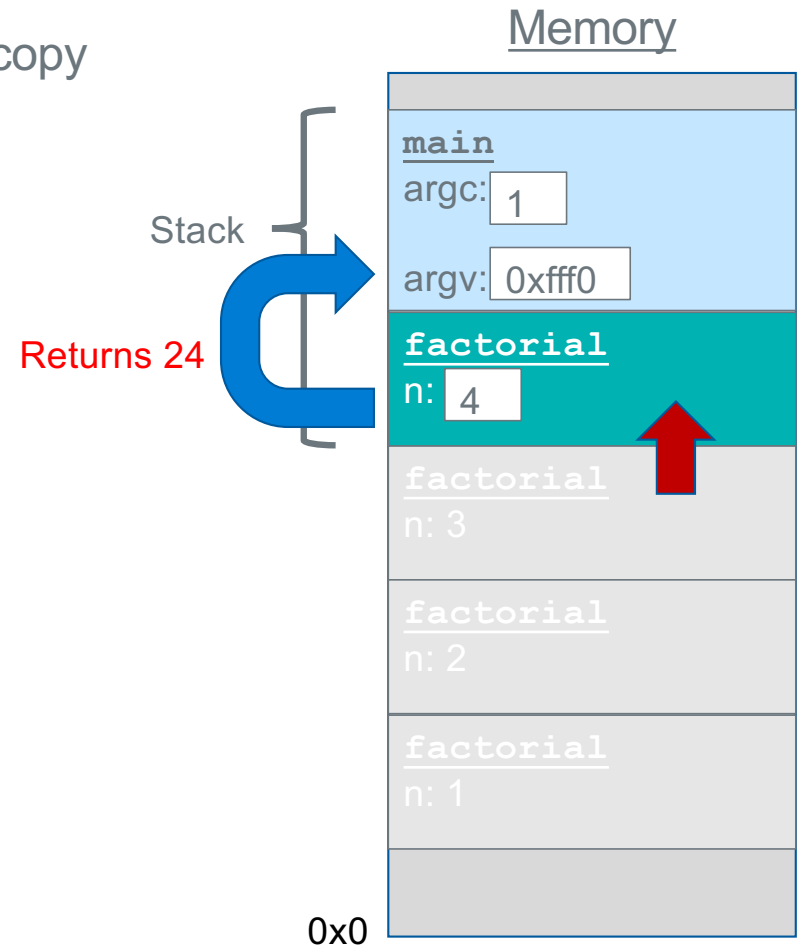
```
int factorial(int n) {  
    if (n == 1) {  
        return 1;  
    } else {  
        return n * factorial(n - 1);  
    }  
}  
  
int main(int argc, char *argv[]) {  
    printf("%d", factorial(4));  
    return 0;  
}
```



The Stack

Each function **call** has its own *stack frame* for its own copy of variables.

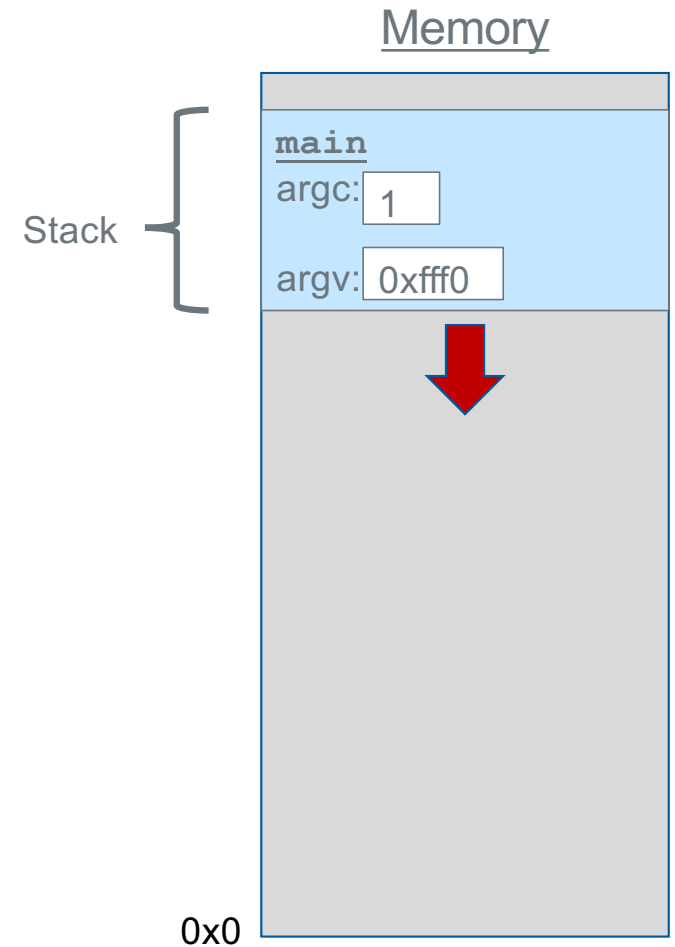
```
int factorial(int n) {  
    if (n == 1) {  
        return 1;  
    } else {  
        return n * factorial(n - 1);  
    }  
}  
  
int main(int argc, char *argv[]) {  
    printf("%d", factorial(4));  
    return 0;  
}
```



The Stack

Each function **call** has its own *stack frame* for its own copy of variables.

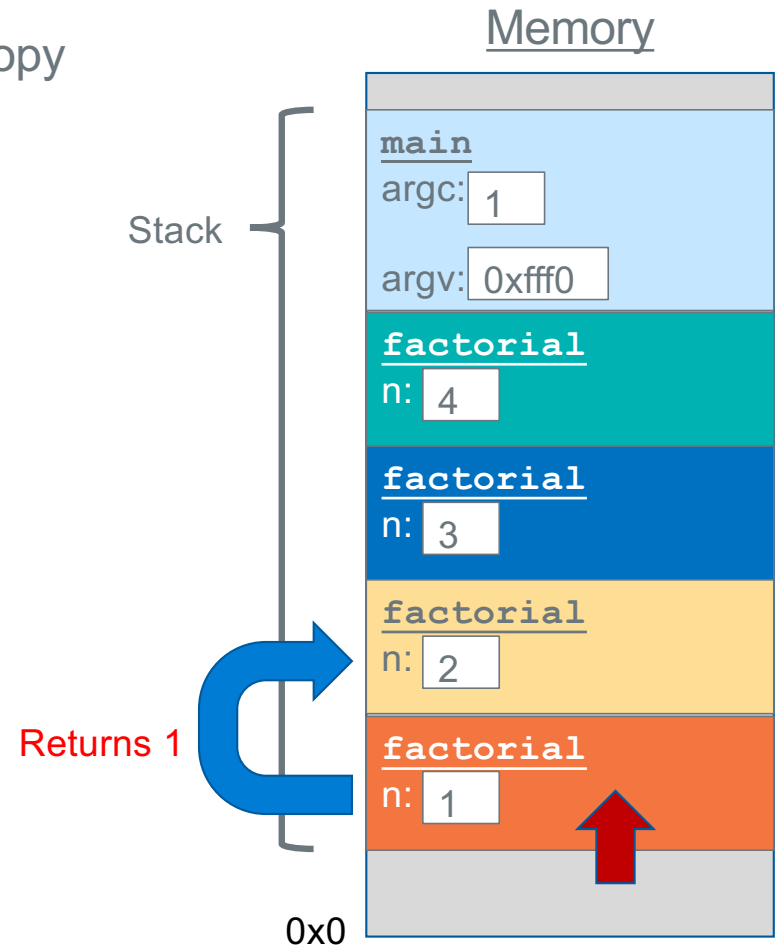
```
int factorial(int n) {  
    if (n == 1) {  
        return 1;  
    } else {  
        return n * factorial(n - 1);  
    }  
}  
  
int main(int argc, char *argv[]) {  
    printf("%d", factorial(4));  
    return 0;  
}
```



The Stack

Each function **call** has its own *stack frame* for its own copy of variables.

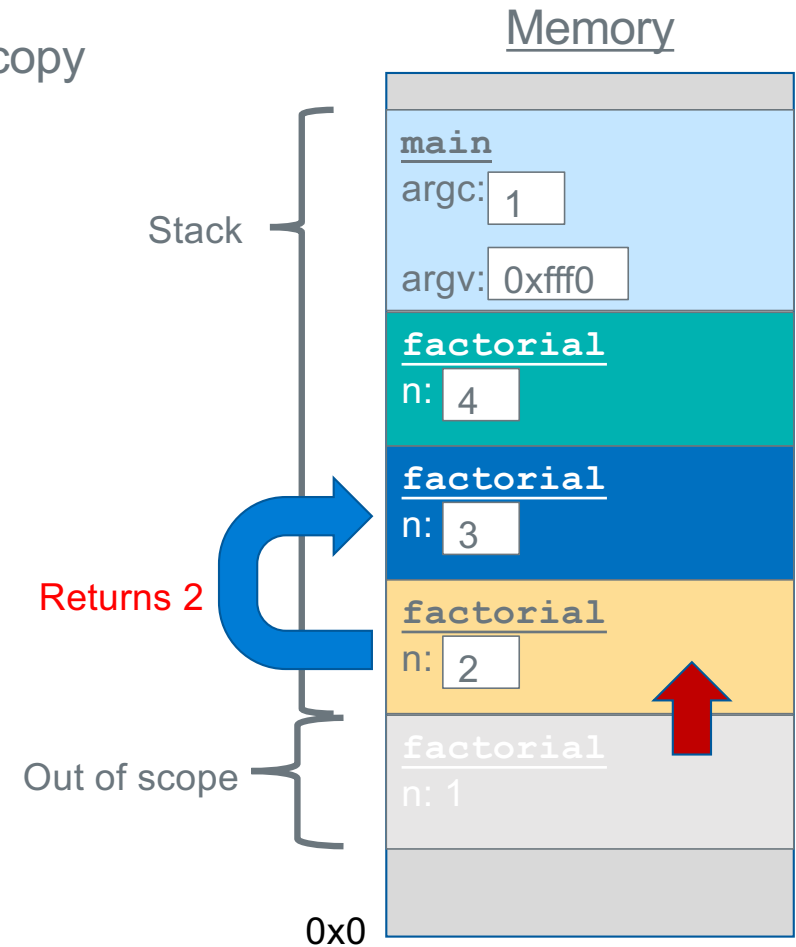
```
int factorial(int n) {  
    if (n == 1) {  
        return 1;  
    } else {  
        return n * factorial(n - 1);  
    }  
}  
  
int main(int argc, char *argv[]) {  
    printf("%d", factorial(4));  
    return 0;  
}
```



The Stack

Each function **call** has its own *stack frame* for its own copy of variables.

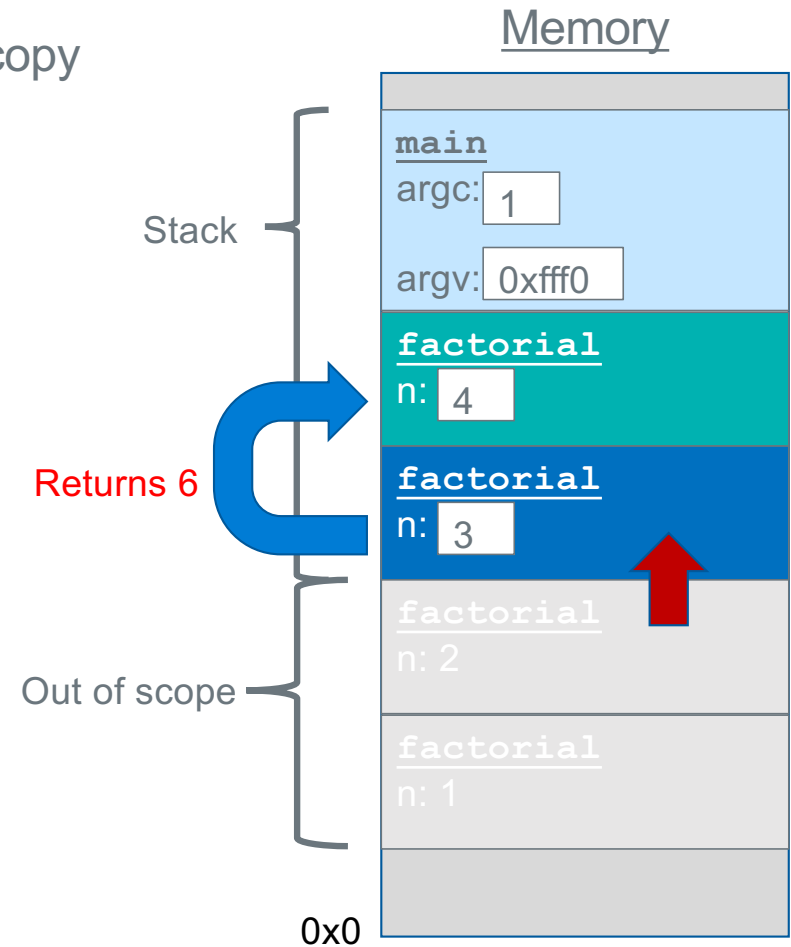
```
int factorial(int n) {  
    if (n == 1) {  
        return 1;  
    } else {  
        return n * factorial(n - 1);  
    }  
}  
  
int main(int argc, char *argv[]) {  
    printf("%d", factorial(4));  
    return 0;  
}
```



The Stack

Each function **call** has its own *stack frame* for its own copy of variables.

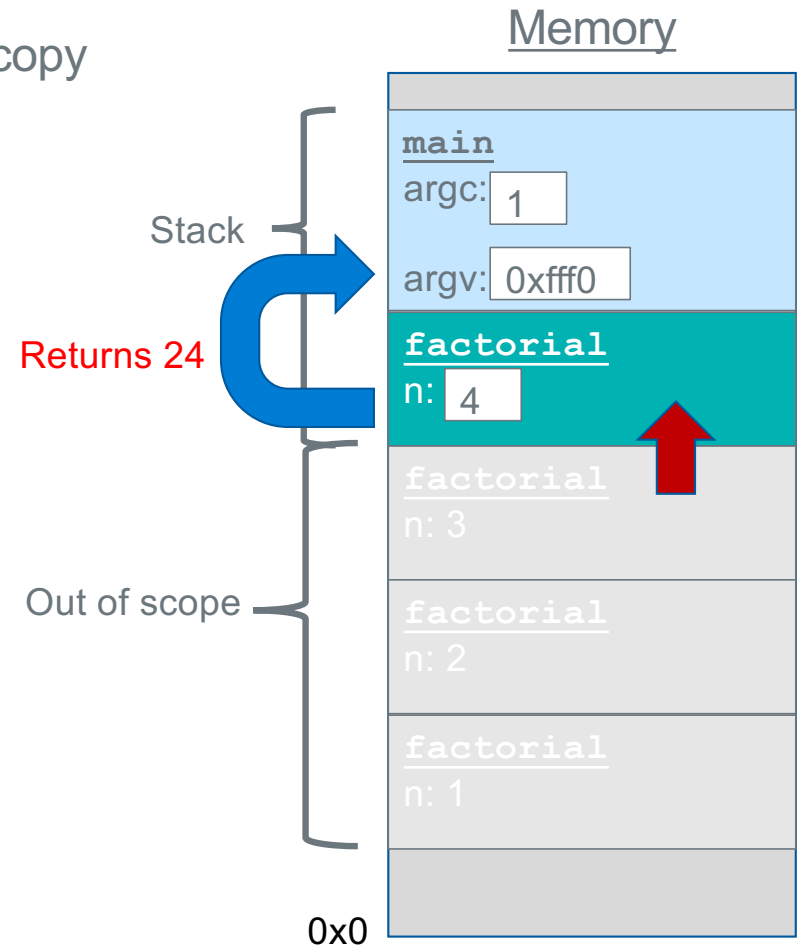
```
int factorial(int n) {  
    if (n == 1) {  
        return 1;  
    } else {  
        return n * factorial(n - 1);  
    }  
}  
  
int main(int argc, char *argv[]) {  
    printf("%d", factorial(4));  
    return 0;  
}
```



The Stack

Each function **call** has its own *stack frame* for its own copy of variables.

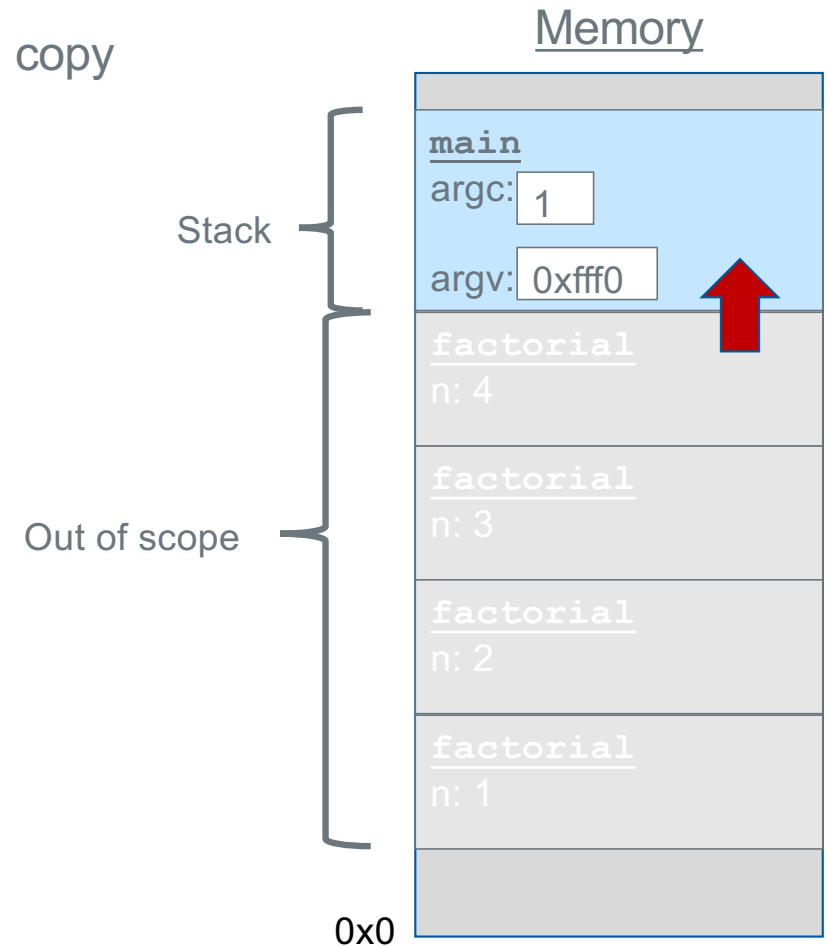
```
int factorial(int n) {  
    if (n == 1) {  
        return 1;  
    } else {  
        return n * factorial(n - 1);  
    }  
}  
  
int main(int argc, char *argv[]) {  
    printf("%d", factorial(4));  
    return 0;  
}
```



The Stack

Each function **call** has its own *stack frame* for its own copy of variables.

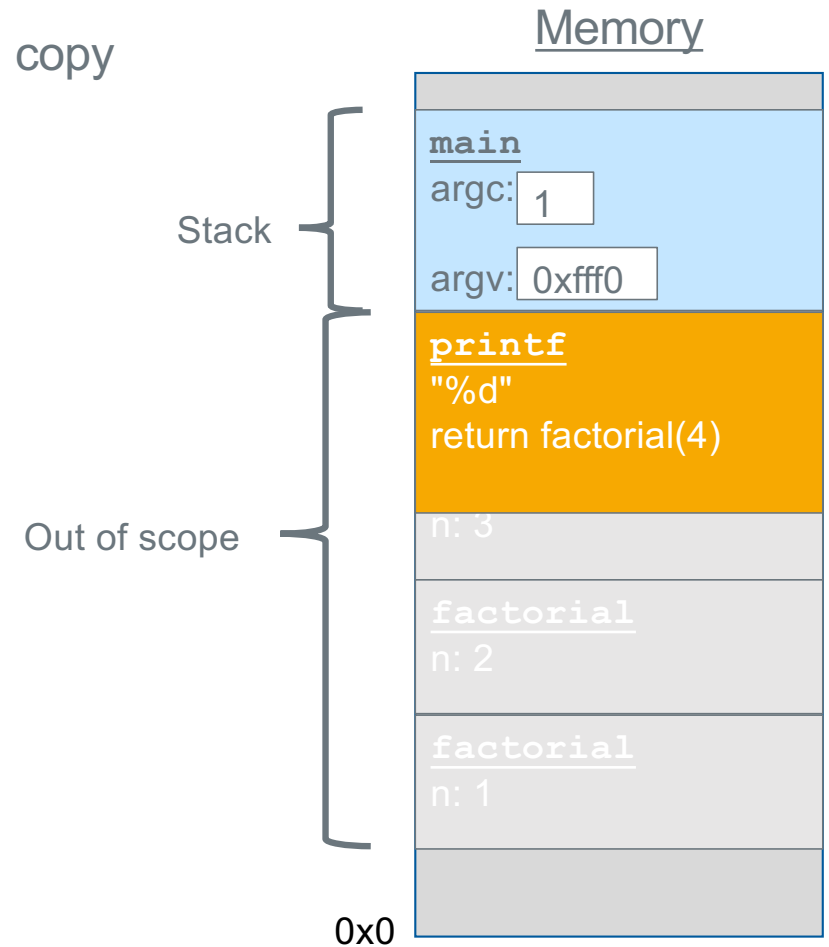
```
int factorial(int n) {  
    if (n == 1) {  
        return 1;  
    } else {  
        return n * factorial(n - 1);  
    }  
}  
  
int main(int argc, char *argv[]) {  
    printf("%d", factorial(4));  
    return 0;  
}
```



The Stack

Each function **call** has its own *stack frame* for its own copy of variables.

```
int factorial(int n) {  
    if (n == 1) {  
        return 1;  
    } else {  
        return n * factorial(n - 1);  
    }  
}  
  
int main(int argc, char *argv[]) {  
    printf("%d", factorial(4));  
    return 0;  
}
```



Ghost of Stack Frames Past.....

same stack frame
variable layout

```
% ./a.out
before ghost: 0 66328
after ghost: 30 300
wraith: 30 300
%
```

See how wraith has the
old values left over
from the prior call to
ghost

```
void ghost(int n)
{
    int x;
    int y;

    printf("before ghost: %d %d\n", x, y);
    x = 10*n;
    y = 100*n;
    printf("after ghost: %d %d\n", x, y);
    return;
}

void wraith (void)
{
    int a;
    int b;

    printf("wraith: %d %d\n", a, b);
    return;
}

int main(void)
{
    ghost(3);
    wraith();
    return EXIT_SUCCESS;
}
```

Extra Slides

Data Segment Variable Alignment

```
.data
ch:    .byte 'A','B','C','D','E'
str:    .string "HIT"
ary:    .hword 0, 1
a:      .byte 'A'
b:      .byte 'B'
xx:     .word 2
```

```
% gcc -c -Wa,-ahlns all.S
1          .data
2 0000 41424344 ch:    .byte 'A','B','C','D','E'
2         45
3 0005 48495400 str:    .string "HIT"
4 0009 00000100 ary:    .hword 0, 1
5 000d 41      a:      .byte 'A'
6 000e 42      b:      .byte 'B'
8 000f 02000000 xx:     .word 2
```

address contents

- Output on the right side is generated by:
- `%gcc -c -Wa,-ahlns all.S`

```
.data
xx:     .word 2
ch:     .byte 'A','B','C','D','E'
        .align 2
str:    .string "HI"
        .align 1
ary:    .hword 0, 1
a:      .byte 'A'
b:      .byte 'B'
```

```
gcc -c -Wa,-ahlns all.S
1          .data
2 0000 02000000 xx:     .word 2
3 0004 41424344 ch:     .byte 'A','B','C','D','E'
3         45
4 0009 00000000        .align 2
5 000c 484900    str:    .string "HI"
6 000f 00        .align 1
7 0010 00000100 ary:    .hword 0, 1
8 0014 41      a:      .byte 'A'
9 0015 42      b:      .byte 'B'
```

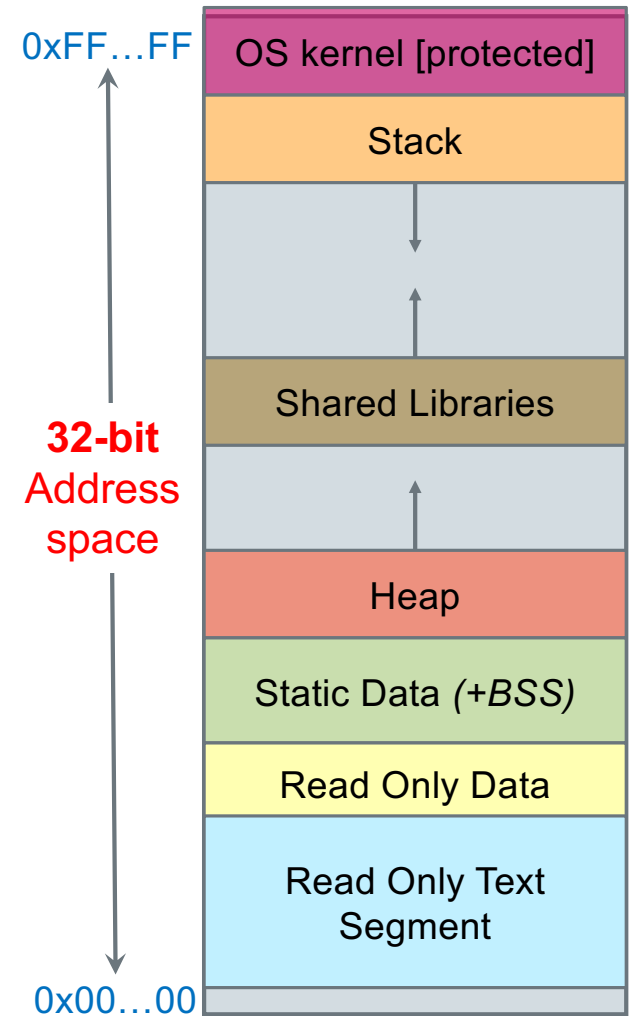
How to get an address into a register?

- Assembler **creates a table of pointers** in the **text segment** called the **literal table**
- For each variable in one of the data segments you reference in a special form of the **ldr** instruction (next slide), **the assembler makes an entry (it does this while assembling, so it is not seen in your source code)** for that variable whose contents is the 32-bit Label address

```
.bss
y:  .space 40

.data
x:  .word 200

.text
// your code
// last line of your code
// below is added by the assembler
.word y      // contents: 32-bit address of y
.word x      // contents: 32-bit address of x
```



Literal Table (Array) each entry is a pointer to a different Label

- Assembler automatically inserts into the text segment an array (table) of pointers
- Each entry contains a 32-bit address of one of the labels
- Uses r15 (PC) as base register to load the entry into a reg
displacement (bytes) - 8

The assembler creates this table before generating the .o file

```
.bss
y: .space 4
.data
x: .word 200

.section .rodata
.Lmsg: .string "Hello World"

.text
main:
(address)ldr r0, [PC, displacement] // replaces: ldr r0, =y
<last line of your assembly, typically a function return>

.word y // entry #1 32-bit address for y
.word x // entry #2 32-bit address for x
.word .Lmsg // entry #3 32-bit address for .Lmsg
```

Literal Table (Array) each entry is a pointer to a different Label

The displacement is different for each use. As the PC is different at each instruction

```
.bss
y: .space 4
.data
x: .word 200
.section .rodata
.Lmsg: .string "Hello World"
.text
main:
(address)ldr r0, [PC, displacement1] // replaces: ldr r0, =y
(address)ldr r0, [PC, displacement2] // replaces: ldr r0, =y
<last line of your assembly, typically a function return>
.word y // entry #1 32-bit address for y
.word x // entry #2 32-bit address for x
.word .Lmsg // entry #3 32-bit address for .Lmsg
```

displacement1 - 8

displacement2 - 8

ARM Assembly Source File: Header

File Header

At the top of every
ARM source file

```
.arch    armv6           // armv6 architecture
.arm     // arm 32-bit instruction set
.fpu     vfp             // floating point co-processor
.syntax  unified         // modern syntax
```

```
// Contents of the other memory segment include .text (your code)
```

.arch <architecture>

- Specifies the target architecture to generate machine code
- Typically specify oldest ARM arch you want the code to run on – most arm CPUs are backwards compatible

.arm

- Use the 32-bit ARM instructions, There is an alternative 16-bit instruction set called thumb that we will not be using


.fpu <version>

- Specify which floating point co-processor instructions to use (OPTIONAL we will not be using floating point)

putchar/getchar Setting up and Usage

```
#include <stdio.h>
#include <stdlib.h>
int
main(void)
{
    int c;
    int count = 0;

    while ((c = getchar()) != EOF) {
        putchar(c);
        count++;
    }
    printf("Echo count: %d\n", count);
    return EXIT_SUCCESS;
}
```



```
.extern getchar
.extern putchar
.section .rodata
.Lfstr: .string "Echo count: %d\n"
.text
.equ    EOF,      -1
.type   main, %function
.global main
.equ    FP_OFF,    12
.equ    EXIT_SUCCESS, 0
main:   push      {r4, r5, fp, lr}
        add       fp, sp, FP_OFF
        mov       r4, 0    //r4 = count

/* while loop code will go here */
.Ldone:
        mov       r1, r4 // count
        ldr       r0, =.Lfstr
        bl        printf
        mov       r0, EXIT_SUCCESS
        sub       sp, fp, FP_OFF
        pop       {r4, r5, fp, lr}
        bx        lr
        .size main, (. - main)
```


Putchar/getchar: The while loop

```
#include <stdio.h>
#include <stdlib.h>
int
main(void)
{
    int c;
    int count = 0;

    while ((c = getchar()) != EOF) {
        putchar(c);
        count++;
    }
    printf("Echo count: %d\n", count);
    return EXIT_SUCCESS;
}
```

initialize count

pre loop test with a call to getchar()
if it returns EOF in r0 we are done

echo the character read with getchar and
then read another and increment count

did getchar() return EOF if not loop

saw EOF, print count

```
mov    r4, 0    //count
bl     getchar
cmp    r0, EOF
beq    .Ldone

.Lloop:
bl     putchar
bl     getchar
add    r4, r4, 1
cmp    r0, EOF
bne    .Lloop

.Ldone:
mov    r1, r4
ldr    r0, =pfstr
bl     printf
```

File header and footers are not shown

printing error messages in assembly

```
.Lmsg0: .string "Read failed\n"
    ldr    r0, =.Lmsg0           // read failed print error
    bl     errmsg
```

```
    // int errmsg(char *errmsg)
    // writes error messages to stderr
.type    errmsg, %function      // define to be a function
.equ     FP_OFF, 4              // fp offset in stack frame
errmsg:
    push   {fp, lr}             // stack frame register save
    add    fp, sp, FP_OFF       // set the frame pointer

    mov     r1, r0
    ldr     r0, =stderr
    ldr     r0, [r0]
    bl      fprintf
    mov     r0, EXIT_FAILURE    // Set return value
    sub     sp, fp, FP_OFF      // restore stack frame top
    pop     {fp, lr}           // remove frame and restore
    bx      lr                 // return to caller
    // function footer
.size     errmsg, (. - errmsg) // set size for function
```