Embedded Systems Programming

Lecture 8 – Memory model and management

Jarno Tuominen

Lecture 8 – Memory model and management

- Review
- Memory model of a C program
- Memory management

Review of last lecture

- Functions (cont)
- The C preprocessor

Review: Functions returning pointers

- C allows function to **return a pointer** to a
  - local variable (bad idea!) Why?
  - static variable
  - dynamically allocated memory
  - Function
- Syntax: add ”*” in front of the function name to indicate that return value is a pointer (of type int, in the example)
- In this example the function returns an array – which is a pointer, remember?

```
int *getrandom(void) {
    static int r[5]; //must be static!
    for (int i = 0; i<5; i++) {
        r[i] = rand();
    }
    //Note: r is same as &r[0]
    return r;
}

int main() {
    int * rnds_p;
    rnds_p = getrandom();

    for (int i = 0; i<5; i++) {
      printf("*(rnds_p+[%d]) : %d\n",i,*(rnds_p+i));
    }
  return 0;
}
```

```
*(rnds_p+[0]) : 1804289383
*(rnds_p+[1]) : 846930886
*(rnds_p+[2]) : 1681692777
*(rnds_p+[3]) : 1714636915
*(rnds_p+[4]) : 1957747793
```

Review: Function pointers

- A **function pointer** is a pointer variable that contains an **address of a function**, instead of a data object
- The syntax of declaration is similar to the syntax of declaring a function – but instead of using a function name, you use a pointer name inside the parenthesis
- Like with normal pointer variables, before using function pointer you need to assign it a value, i.e. the address of a function
- It is a good practice to type define declaration of function pointers as it will make your code much nicer
- Note: Function pointers are potentially very dangerous, as a loose pointer does not code access to wrong data, but if will cause your program to branch to a random address!

<return_type> (*<pointer_name>) (function_arguments);

```
typedef int (*fpComparer)(int x,int y);

int compare(int x,int y) {
  ...
}

int main() {
  int result;
  ...
  fpComparer fpcomp = &compare;

  result = fpcomp(a,b);
  //result = (*fpComparer)(a,b);
  ...
}
```

Declare the function pointer and assign address of compare-function to it

If not typedef'd

Review: Function pointers packed in a struct

```c
typedef uint8_t (*sensor_fp)(uint8_t SensorID, uint8_t param);
```

Type define: sensor_fp

```c
//Struct for generic sensor instance
typedef struct sensor_t {
          char* name;
          bool enabled;
          sensor_fp init;
          sensor_fp power_ctrl;
} sensor_t;
```

Sensor-related data and its functions packed in a single ”instance” (close to object-oriented thinking but still plain C!)

```c
sensor_t sensors[MAX_SENSOR_COUNT];
```

Create an array of sensor instances

```c
sensors[0].name = "BMI_1";
sensors[0].enabled = true;
sensors[0].init = bmi160_initialize_sensor;
sensors[0].power_ctrl = bmi160_power_ctrl;
```

set-up the sensor instance

```c
sensors[1].name = “ECG";
sensors[1].enabled = true;
sensors[1].init = ads1293_init;
sensors[1].power_ctrl = ads1293_power_ctrl;
```

and another one

Init all the sensors! Beatiful code <3

```c
void init_sensors(uint8_t count) {
  for (uint32_t i=0; i < count; i++)
  {
    e = sensors[i].init(i, NULL);
  }
}
```

Review: functions with variable argument list

```c
#include <stdio.h>
#include <stdarg.h>

double avg(int num,...) {
   va_list valist;
   double sum = 0.0;
   int i;
   /* initialize valist for num number of arguments */
   va_start(valist, num);
   /* access all the arguments assigned to valist */
   for (i = 0; i < num; i++) {
      sum += va_arg(valist, int);
   }
   va_end(valist); /* clean memory reserved for valist */
   return sum/num;
}

int main() {
   printf("Avg = %f\n", avg(4, 2,3,4,5));
   printf("Avg = %f\n", avg(3, 5,10,15));
}
```

num is number of arguments

va_list is a data type that can hold list of arguments. Used by macros **va_start**, **va_arg** and **va_end**

populate *valist* using macro **va_start**

Get arguments of type int using macro **va_arg**

va_end releases memory

Review: The C Preprocessor

- **C preprocessor** is the macro preprocessor, which provides the ability for the
  - inclusion of header files
  - macro expansions
  - conditional compilation
  - line control
  - Handling of pragma operators (in C99)
- invoked by the compiler as the first part of code translation
- Preprocessor macros begin with **#**

https://en.wikipedia.org/wiki/C_preprocessor

Review: Preprocessor macros and directives

- **#include**
  - Inclusion of header files
  - Remember include guards
- **#define, #undef**
  - Defining/undefining macros
- **#if, #elif, #endif, #ifdef, #ifndef**
  - For conditional compilation
- **#error, #warning**
  - For custom errors/warnings

```
#ifndef GRANDPARENT_H
#define GRANDPARENT_H
  #include “child.h”
  ...
#endif /* GRANDPARENT_H */
```

```
#define PI 3.14159
#define RADTODEG(x) ((x) * 57.29578)
#undef PI
```

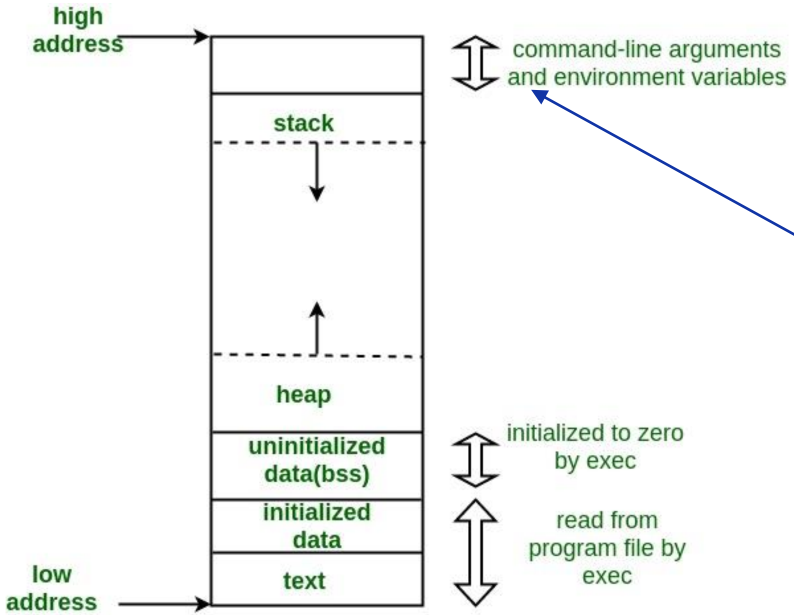
```
#if VERBOSE >= 2
printf(“lots of trace messages\n");
#elif VERBOSE >1
printf(“some trace messages\n");
#endif
```

```
#ifdef DEBUG
some_debug_function();
#endif
```

Lecture 8 – Memory model and management

- Review
- Memory model of a C program
- Memory management

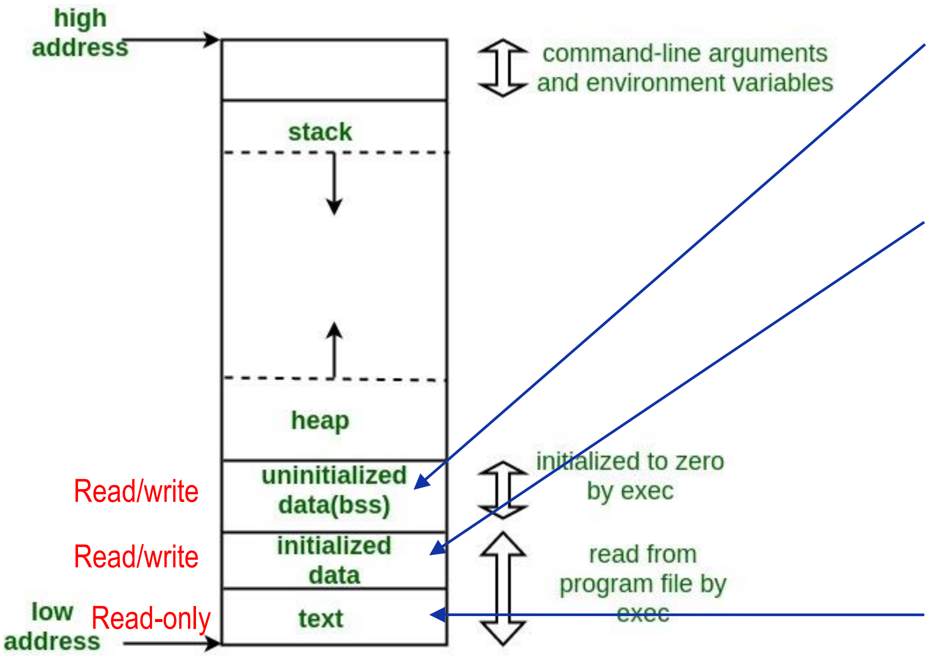
Memory layout of C programs



- When a C program is running under an OS, it has it’s own memory space
  - OS takes care that the program does not exceed its memory boundaries
- In (bare metal) embedded systems, with no OS, the situation is similar, except
  - the segment for command line arguments and environment variables is not present

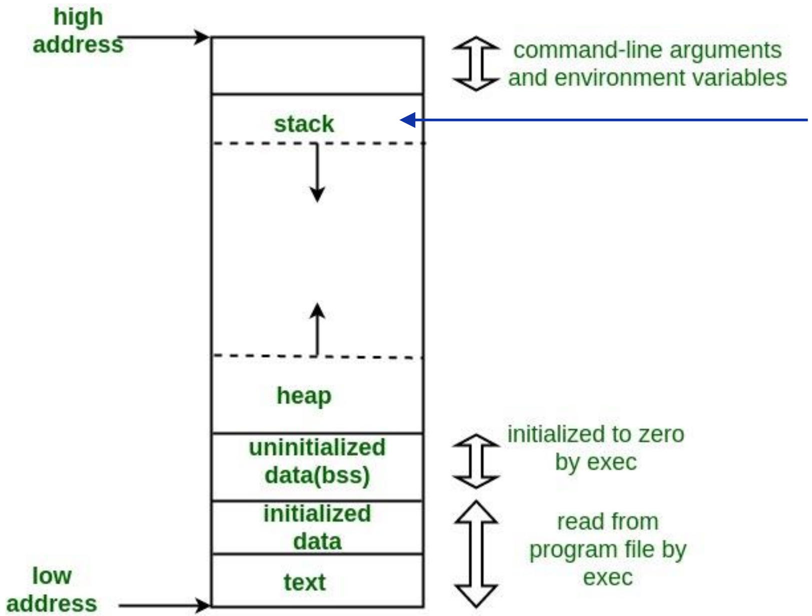
https://www.geeksforgeeks.org/memory-layout-of-c-program/

Memory layout of C programs – static regions



- Uninitialized data segment (bss)
    - Contains global and static variables which are not initialized, or are initialized to zero
- Initialized data segment contains **initialized static variables**, that is, global variables and static local variables which have a predefined value
    - The size of this segment is determined by the size of the values in the program's source code, and does not change at run time
- Your program code is located in the ”text” area of the memory
    - Read-only segment

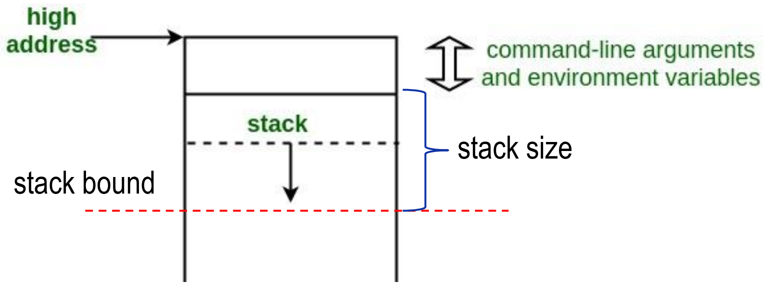
Memory layout of C programs - stack



What is a stack?

- At each function call, your program stores the values of local variables of a calling function to stack (offen referred as ”stack push”), after which the function is executed. When the function returns, the old values are ”popped” from the stack (LIFO structure)
- Stack grows downwards
- The set of variables pushed/popped to/from stack is called a ”stack frame”

Memory layout of C programs – stack(2)

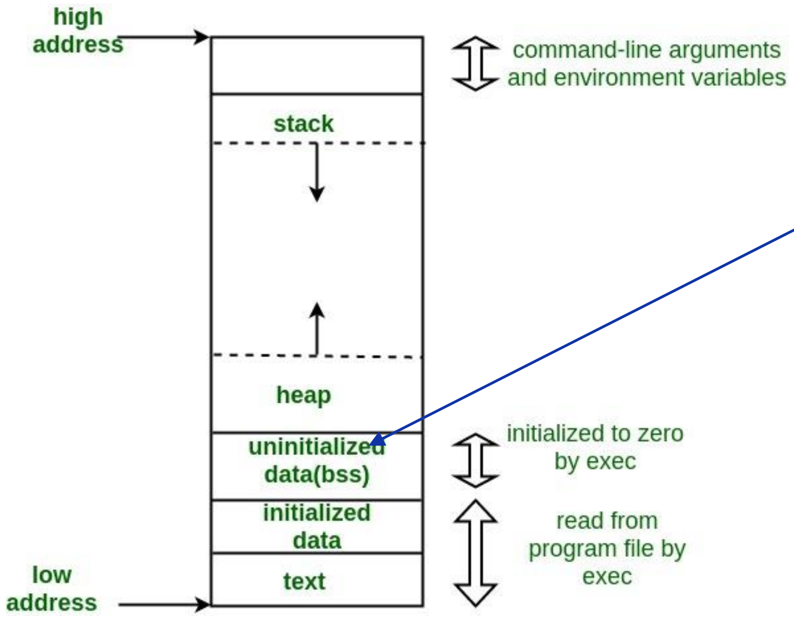



```
                IF :DEF: __STARTUP_CONFIG
#include "startup_config.h"
                ENDIF

                IF :DEF: __STARTUP_CONFIG
Stack_Size      EQU __STARTUP_CONFIG_STACK_SIZE
                ELIF :DEF: __STACK_SIZE
Stack_Size      EQU __STACK_SIZE
                ELSE
Stack_Size      EQU     2048
                ENDIF

                AREA    STACK, NOINIT, READWRITE, ALIGN=3
Stack_Mem       SPACE   Stack_Size
__initial_sp
```

What is a <u>stack overflow</u>?

- A fatal error which occurs when the call stack pointer exceeds the stack bound. It is essentially a buffer overflow which causes your program to crash (A ”segmentation fault”)
- Typically caused by
  - infinite or very deep recursion
  - very large local variables
- Stack size is set at the beginning of your program – in the routines that are called before main(), often in assembler

Memory layout of C programs - heap



What is a heap?

Like stack, heap is a fixed-size memory area, from where you allocate memory dynamically with malloc()

Grows upwards

What is ”out of heap space?”

Your system has ran out of memory – when the heap pointer has hit the stack pointer (or the heap boundary, if set)

Often caused by badly behaving processes/threads which allocate memory, but do not free it after use (memory leakage)

Lecture 8 – Memory model and management

- Review
- Memory model of a C program
- Memory management

Dynamic memory allocation

- Two ways to allocate memory dynamically:
- From **stack**
  - Named variables in functions
  - Allocated for you when you call a function
  - Deallocated for you when function returns
- From **heap**
  - ”Memory on demand”
  - You are responsible for all allocation and deallocation

```
int x[10];
```

```
int* x = malloc(10*sizeof(int));
```

- The size of an array must be declared/defined before it can be used
- Hence, the array may be insufficient or more than required to hold the data
- To solve this issue, memory can be allocated dynamically from the heap

Using heap memory

- There are four functions, defined in <stdlib.h> for dynamic memory operations

malloc()	Allocates requested size of bytes and returns a pointer first byte of allocated space
calloc()	Allocates space for an array of elements, initializes to zero and then returns a pointer to memory
free()	deallocate the previously allocated space
realloc()	Change the size of previously allocated space

malloc()

- The name malloc stands for "memory allocation“
- Reserves a block of memory of specified size and return a pointer of type void which can be casted into pointer of any form.
- If the space is insufficient, allocation fails and returns NULL pointer
  - This is ”out of heap space” = ”out of memory”
  - But, not a segmentation fault
- Warning: malloc() takes time, so be very careful in applications with hard real-time requirements!

```
ptr = (cast-type*) malloc(byte-size);

ptr = (int*) malloc(100 * sizeof(int));
```

This statement will allocate either 200 or 400 bytes, depending on size of int (2 or 4 bytes), respectively. The pointer points to the address of first byte of memory

In C, auto-casting is possible, in C++ casting “(int*)” to integer type is required

calloc()

- The name calloc stands for "contiguous allocation“
- calloc() allocates multiple blocks of memory each of same size and sets all bytes to zero
- As with malloc(), if the space is insufficient, allocation fails and returns NULL pointer.

```
ptr = (cast-type*) calloc(n, element-size);

ptr = (float*) calloc(25, sizeof(float));
```

This statement allocates contiguous space in memory for an array of 25 elements each of size of float, i.e, 4 bytes

In C, auto-casting is possible, in C++ casting “(int*)” to integer type is required

free(), realloc()

- Dynamically allocated memory created with either calloc() or malloc() doesn't get freed on its own
- You must explicitly use **free()** to release the space
- If the previously allocated memory is insufficient or more than required, you can change the previously allocated memory size using **realloc()**

```
free(ptr);
```

This statement frees the space allocated in the memory pointed by ptr

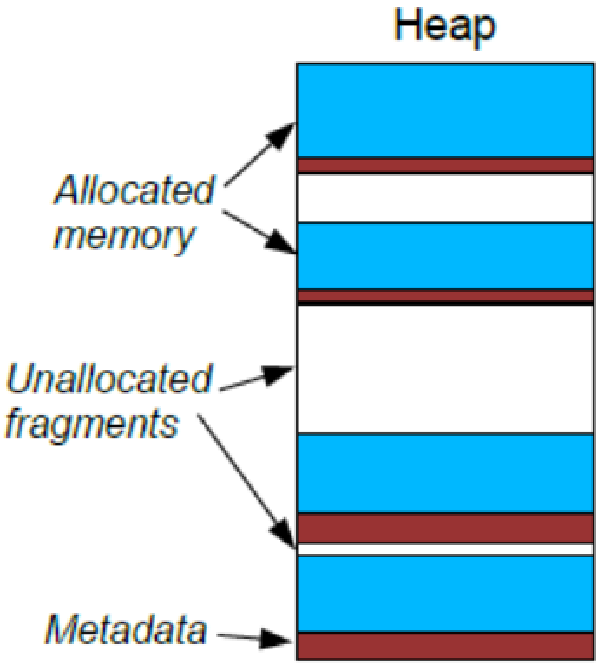
NOTE! After this you still have the pointer variable ”ptr”, but now it is a ”**dangling pointer**”. Attempt to dereference it (with *) is a typical, fatal programming mistake.

```
ptr = realloc(ptr, newsize);
```

Here, ptr is reallocated with size of newsize

A deeper look to malloc()

- You will actually reserve more memory than requested, because
  - Heap memory blocks may have minimum block size
  - Alignment requirements (padding)
  - Overhead of maintaining heap data structures
- malloc() includes an ”allocator”
  - It keeps track where free memory is located – this is uasually implemented using linked lists
  - The linked lists are ”metadata”, which is stored in heap as well, wasting heap memory space
  - Allocator tries to find the optimal memory block – the faster the allocator is, the sloppier work it does
- After heavy malloc() – free() – realloc() usage, there will be multiple chunks of heap memory that are unallocated. This is called fragmentation



Useful memory manipulation functions

- Standard library <string.h> contains a set of useful functions for memory operations

```
void *memcpy(void *str1, const void *str2, size_t n)
```

Copies **n** characters from memory area **str2** to memory area **str1**

```
void *memmove(void *str1, const void *str2, size_t n)
```

Copies **n** characters from **str2** to **str1**, but for overlapping memory blocks, memmove() is a safer approach than memcpy()

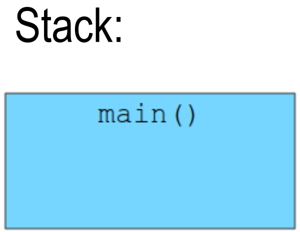
```
void *memset(void *str, int c, size_t n)
```

Copies the character **c** (an unsigned char) to the first **n** characters of the string pointed to, by the argument **str**

A deeper look into stack operation

- At the beginning of the program, a stack frame for main() is created
  - OS does this, or in case of no OS, start-up routines

```
int b() {
    /* … */
}

int a() {
    /* … */
    b();
}

int main() {
    /* … */
    a();
}
```

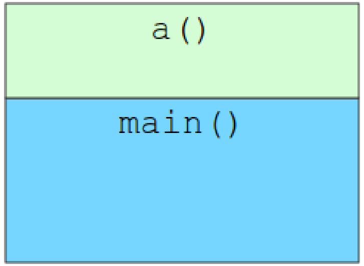


A deeper look into stack operation

- When a() is called, a new stack frame is created for a()

```
int b() {
   /* … */
}

int a() {
   /* … */
   b();
}

int main() {
    /* … */
    a();
}
```

Stack:

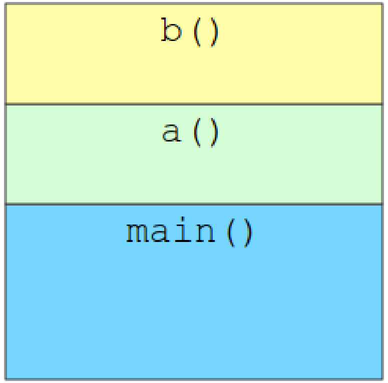


A deeper look into stack operation

- Same for b: when b() is called, a new stack frame is created for b()

```
int b() {
   /* … */
}

int a() {
   /* … */
   b();
}

int main() {
   /* … */
   a();
}
```

Stack:

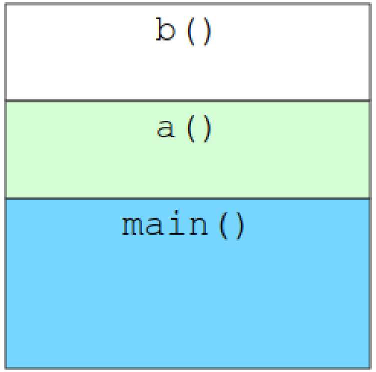


A deeper look into stack operation

- When b() finishes running, its stack frame is removed
- ... and same happens when a() is finished

```
int b() {
   /* … */
}

int a() {
   /* … */
   b();
}

int main() {
   /* … */
    a();
}
```

Stack:



The map file

- A report file, which shows
  - Memory segmentation
  - the data and code size for each module
  - And much more!

```
Image component sizes

      Code (inc. data)   RO Data    RW Data    ZI Data      Debug   Object Name
        20          0          0          8          0       1304   adc.o
      2324        274          0         34          0      31599   app_timer.o
       160         22          0         12          0       6522   app_util_platform.o
        56         18        192          0       4096        636   arm_startup_nrf51.o
      1902         74          0          2          0      18044   diskio.o
       392        104          0          4          0      82783   error_handler.o
      2448        462          0         13        114     115595   main.o
       988        136          0          4      12700      12608   sdcard.o
      1810        346         53          0         80      30380   spi_master_multislave_fast.o
       348         38          0          4          0       2287   system_nrf51.o
       972        226         28          0        320       7225   timers_init.o
             ......
    ----------------------------------------------------------------------
     66208       6378       1592        472      20904    1069396   Object Totals
         0          0         32          0          0          0   (incl. Generated)
       122          0          6         36          2          0   (incl. Padding)

    ----------------------------------------------------------------------
```

```
m_gc        0x20006354   Data        16   fds.o(.bss)
desc        0x20006364   Data        12   fds.o(.bss)
.bss        0x20006370   Section     44   localtime_w.o(.bss)
_tms        0x20006370   Data        44   localtime_w.o(.bss)
HEAP        0x200063a0   Section   2048   arm_startup_nrf51.o(HEAP)
STACK       0x20006ba0   Section   2048   arm_startup_nrf51.o(STACK)
```

```
Total RO  Size (Code + RO Data)              72980 (  71.27kB)
Total RW  Size (RW Data + ZI Data)           21432 (  20.93kB)
Total ROM Size (Code + RO Data + RW Data)    73040 (  71.33kB)
```