CS100
Introduction to Programming

Lecture 11. Memory in C

1. Global Variable

- Variables declared outside a function
- "file scope," meaning they are visible within the file
- Global Scope and Global Lifetime
- Can be used in any functions in the files

1. Global Variable

```c
#include<stdio.h>

int g_a = 10;

int f(void) {
        printf("in %s g_a = %d \n", __func__, g_a);
        g_a += 2;
        printf("again in %s g_a = %d \n", __func__, g_a);
        return g_a;
}

int main(void) {
        printf("in %s g_a = %d \n", __func__, g_a);
        f();
        printf("again in %s g_a = %d \n", __func__, g_a);
        return 0;
}
```

1. Global Variable Initialization

- Zero value without initialization
- NULL for Pointer without initialization
- Use the known value during compiling
- Initialization before main function

2. Static Local Variable

- Local variable with the Static key-word
- It will be maintained when leaving the local function
- Initialization: happen when first entering the local function
- Maintaining the same value next time when re-entering the local function

2. Static Local Variable

```
#include<stdio.h>

int f(void) {
        static int s_a = 1;
        printf("in %s s_a = %d \n", __func__, s_a);
        s_a += 2;
        printf("again in %s s_a = %d \n", __func__, s_a);
        return s_a;
}

int main(void) {
        f();
        f();
        f();
        return 0;
}
```

2. Static Local Variable

- Special global variable
- Located in the same memory region
- Local Scope and Global Lifetime
- Static means Local Scope

2. Static Local Variable

```
#include<stdio.h>

int g_a = 10;
int f(void) {
int tmp = 0;
        static int s_a = 1;
        printf("&g_a = %p \n", &g_a);
        printf("&s_a = %p \n", &s_a);
        printf("&tmp = %p \n", &tmp);
        return s_a;
}

int main(void) {
        f();
        return 0;
}
```

&g_a = 0058301C

&s_a = 00583018

&tmp = 010FFF18

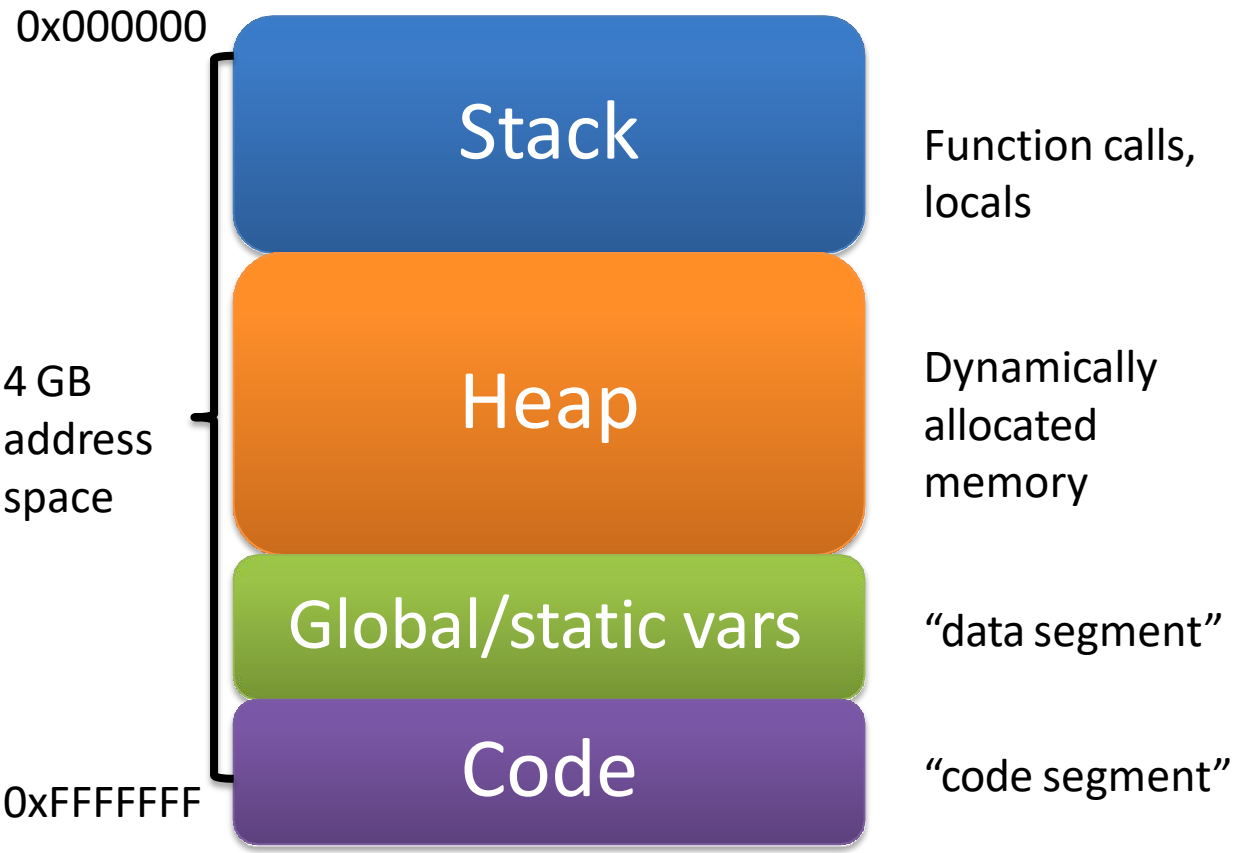
2. Function returning a Pointer

- Return the address of Local variable → Dangerous
- Return the address of Global/Static variable → Safe
- Return the address of dynamic memory (malloc)

  → safe but easily get into trouble
- Return the input Pointer instead

3. C Memory Layout

- each process gets its own memory chunk, or ***address space***



Code/Text Segment

- memory allocated by the program as it runs
  - Executable code
  - Constant variables
  - Read only
- fixed at pre-compile time

Data Segment

- memory allocated by the program as it runs
  - Initialized global variables
  - Initialized local static variables
  - BSS: Block Start by Symbol (uninitialized)

- fixed at compile time

Global/static vars

Stack Allocation

- memory allocated by the program as it runs
  - local variables
  - function calls
- fixed at compile time

Heap Allocation

- dynamic memory allocation
  - memory allocated at run-time

- Function for allocating memory:
  - **`malloc()`**
    - Requires `#include <stdlib.h>` to work

malloc()

```
void* malloc ( <size to be allocated> )
```

```
char *letters;
letters = (char*) malloc(userVariable *
                         sizeof(char));
```

- malloc returns a pointer to a ***contiguous*** block memory of the size requested

Casting Allocated Memory

- **`malloc()`** return a pointer of type **`void`**, so you must cast the memory to match the given type

```
letters = (char*) malloc(userVariable *
                         sizeof(char));
```

Handling Allocated Memory

- **IMPORTANT**: before using allocated memory make sure it's *actually been allocated*

- if memory wasn't correctly allocated, the address that is returned will be **`null`**
  - this means there isn't a contiguous block of memory large enough to handle request

Exiting in Case of NULL

- if the address returned is **`null`**, your program should exit
  - **`exit()`** takes an integer value
  - non-zero values are used as error codes

```
if (grades == NULL) {
   printf("Memory not allocated,
              exiting.\n");
   exit(-1);
}
```

Managing Your Memory

- ***stack*** allocated memory is automatically freed when functions **`return`**
  - including **`main()`**

- memory on the ***heap*** was allocated by you – so it must also be freed by you

Freeing Memory

- done using the **`free()`** function
  - free takes a pointer as an argument: **`free(grades);`**
    `free(letters);`

- **`free()`** does not work recursively
  - for each individual allocation, there must be an individual call to free that allocated memory
  - called in a sensible order

4. More about #define

- #define → macro
- #define <name> <value>
- No ``;'' in the end,
- <name> should be word,
- <value> could be anything
- Text-level replace during pre-compiling

```
#include<stdio.h>

#define _PI_ 3.1415926
#define _FORMAT_ "%f...\n"
#define _PI2_ 2*_PI_

int main(void) {
    printf(_FORMAT_, _PI2_);
    return 0;
}
```

4. More about #define

- Macro without value

```
#define _DEBUG_

#define _RELEASE_
```

```c
#include<stdio.h>
int main(void) {
    printf("%s: %d \n", __FILE__, __LINE__);
    printf("%s: %s \n", __DATE__, __TIME__);
    return 0;
}
```

- Pre-defined Macro

__LINE__, __FILE__, __DATE__, __TIME__, __STDC__ ……

- Macro with variables: brackets anywhere!

```
#define MIN(a,b) ( (a) > (b)? (b): (a))
```