Week 5

Start at 8: 05, Scan the QR code

P1: Head Guards

Double inclusion

File "grandparent.h"

```
struct foo {
    int member;
};
```

File "parent.h"

```
#include "grandparent.h"
```

File "child.c"

```
#include "grandparent.h"
#include "parent.h"
```

Result

```c
struct foo {
    int member;
};
struct foo {
    int member;
};
```

compilation error, since the structure type `foo` will thus be defined twice

#include guards

File "grandparent.h"

```c
#ifndef GRANDPARENT_H
#define GRANDPARENT_H

struct foo {
    int member;
};

#endif /* GRANDPARENT_H */
```

File "parent.h"

```c
#include "grandparent.h"
```

File "child.c"

```c
#include "grandparent.h"
#include "parent.h"
```

```c
#ifndef GRANDPARENT_H  // => no exits, start if statement
#define GRANDPARENT_H  // => define
```

```
struct foo {
    int member;
};

#endif /* GRANDPARENT_H */ // => end if

#ifndef GRANDPARENT_H     // => already exist? skip the next until
endif
#define GRANDPARENT_H

struct foo {
    int member;
};

#endif /* GRANDPARENT_H */
```

Result

```
struct foo {
    int member;
};
```

#ifndef test returns false, the preprocessor skips down to the #endif

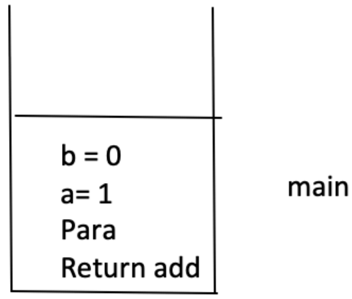
P2: Stack and Heap

Stack

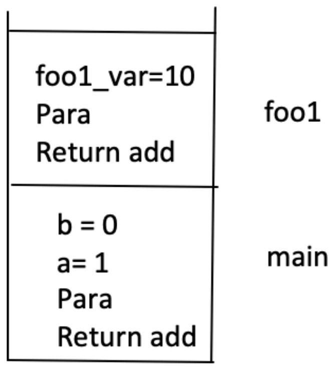
To keep track of the current memory place, there is a special processor register called **Stack Pointer**. Every time you need to save something — like a variable or the return address from a function — it pushes and moves the stack pointer up. Every time you exit from a function, it pops everything from the stack pointer until the saved return address from the function.

```c
void foo1(char c){
  int foo1_var = 10;
  return;
}

int main(){
  int a = 1;
  int b = 0;
  foo1('x');
}
```

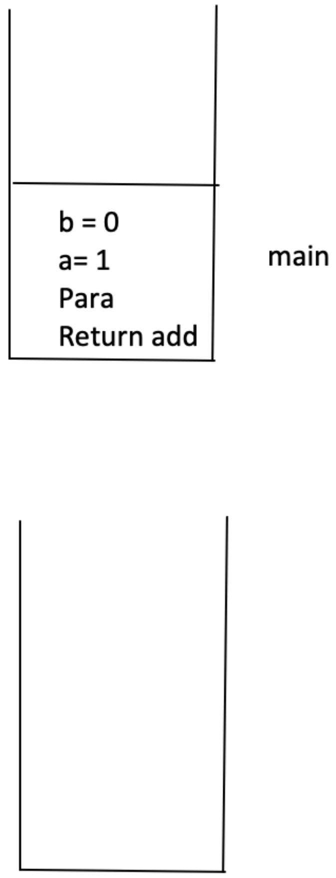
The memory for stack



The memory for stack



The memory for stack



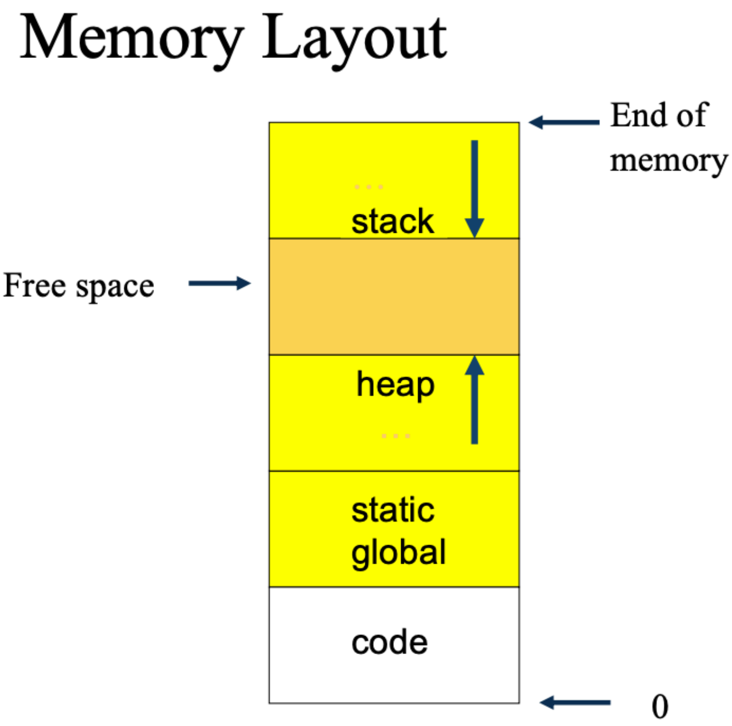
A bad example:

```
#include <stdio.h>

int* createArray(int size){
  int arr[size];
  return arr;
}

int main(){
  int s = 10;
  int* arr = createArray(s);
```

```
  for (int i = 0; i < s; i++){
    arr[i] = i;
  }
  return 0;
}
```

Heap

Exist independently of functions and scopes




```
#include <stdlib.h>

/*
  Returns a pointer to the allocated memory, if
  successful, or a NULL pointer if unsuccessful

  int* arr = (int*) malloc(sizeof(int) * 10);
*/
void* malloc(size_t size);
```

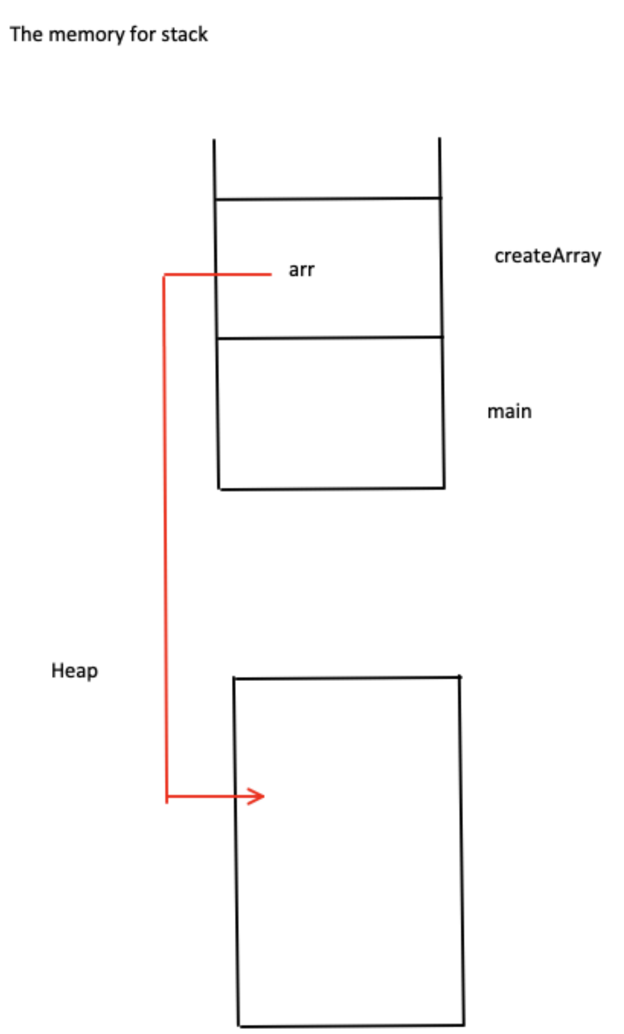
```
/*
  It has two arguments:
  — num specifies the number of “blocks” of contiguous
  memory
  — size specifies the size of each block

  • The allocated memory is cleared (set to ‘0’).
  // assume you are create an array with num elements and each of them
has size
*/
void* calloc(size_t num, size_t size);



/*
  This takes previously-allocated memory and
  attempts to resize it. ==> will free the original memory

  This may require a new block of memory to be
  found, so it returns a new void pointer to memory.

  Contents are preserved

*/
void* realloc(void *ptr, size_t size);




void free(void *ptr);
```

Change the bad example

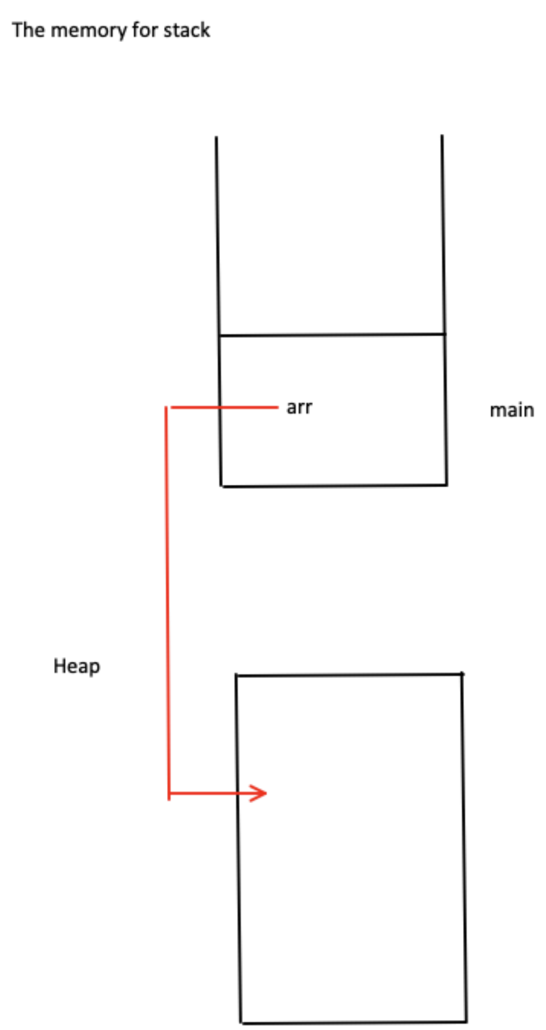
```
#include <stdio.h>
#include <stdlib.h>

int* createArray(int size){
  int* arr = (int*) malloc(sizeof(int) * 10);
```

```c
  return arr;
}

int main(){
  int s = 10;
  int * arr = createArray(s);

  for (int i = 0; i < s; i++){
    arr[i] = i;
  }

  free(arr);

  arr = NULL;

  return 0;
}
```

Before createArray return



After createArray return



Once malloc remember to free once ! `Memory Leak/....`

If we have two pointers points to the same memory, only free once.

2. The order for free

3. Compare with Java

   Java use keyword `new` for dynamic memory, but we do not need to worry about `free` since In *Java*, process of deallocating memory is handled automatically by the *garbage collector*

==> Tool

> Tutorial/Week5/Q1, Q4, Q6

Do Question 5

P3: Linked List

Linked list in combined by nodes and we link nodes with pointers

```
#include <stdio.h>
#include <stdlib.h>

typedef struct node node;

struct node{
    node* next;
    int v;
}

node* list_init(int value){
    node* n = malloc(sizeof(struct node));
    // check the return value for malloc
    n->v = value;
    n->next = NULL;
    return n;
```

```
}

void list_add(node* h, int value){
    if (h != NULL){
        node* c = h;
        while (c->next != NULL){
            c = c -> next;
        }

        c -> next = list_init(value);
    }
}

void list_delete(node** h, node* n){
    if (h ! = NULL){
        if (*h != NULL){
            node* prev = NULL;
            node* cur = *h;
            // find the node to be deleted and find the prev
            while (cur != n && cur != NULL){
                prev = cur;
                cur = cur -> next;
            }

            if (cur != NULL){
                // Two cases
                // 1. The node to be deleted is the head
                if (prev == NULL){
                    /*


                        node0 ---> node1 ---> node2 ---> node3
            prev        cur
                        deleted


                    */
                    node* new_head = (*h) -> next;
                    free(*h);
                    // Here is the reason why we need node** head
                    *h = new_head;
```

```c
                }else{

                    /*

                        node0 ---> node1 ---> node2 ---> node3
                                    prev       cur
                                            deleted

                    */
                    // 2. The node to be deleted is not the head
                    prev->next = cur->next;
                    free(cur);
                }

            }
        }
    }

}

node* list_next(node* n){
    node* r = NULL;
    if (n){
        r = n -> next;
    }
    return r;
}

void list_free(node* h){
    node* t = NULL;
    while (h){
        // record the next firstly
        t = h -> next;
        // free the cur
        free(h);
        h = t;
    }
}
```

```
// // #########
// void changeTheValue(int* a){
//     *a = 2;
// }

// int main(){
//     int a = 1;
//     changeTheValue(&a);
// }
// // #########
// void changeTheHead(node** head){
//     *head = new_head;
// }
// int main(){
//     node* head = ...;
// }
// // #########
```

> Tutorial/Week4/Q2, Q3, Q5, Q7

Tool

Check memory leak using gcc flag

```
$ gcc -fsanitize=address -Wall -Werror -std=gnu11
```

```
=================================================================
==37==ERROR: LeakSanitizer: detected memory leaks

Direct leak of 40 byte(s) in 1 object(s) allocated from:
    #0 0x7f22dc1d3279 in __interceptor_malloc /build/gcc/src/gcc/libsanitizer/asan/asan_malloc_linux.cpp:145
    #1 0x556c6f1da17a in main (/home/a.out+0x117a)
    #2 0x7f22dbf77b24 in __libc_start_main (/usr/lib/libc.so.6+0x27b24)
SUMMARY: AddressSanitizer: 40 byte(s) leaked in 1 allocation(s).
```

Other error like: heap buffer overflow

GDB

A debugger for C (and C++)

It allows you to do things like run the program up to a certain point then stop and print out the values of certain variables at that point, or step through the program one line at a time and print out the values of each variable after executing each line.

1. Installed ?

   ```
   $ gdb --version
   ```

2. Compile your program with `-g` or `-g -O0`
3. Some common operations

   ```
   # start up gdb in the directory of executable file
   $ cd path/to/a.out
   $ gdb

   # specify the debug file
   (gdb) file [executable_file]

   # print the file
   (gdb) list

   # set break point
   (gdb) break [Line]

   # run the program
   (gdb) run

   # print variable
   (gdb) print [variable_name]
   ```

```
# contine to run until the next break point
(gdb) continue

# contine to run next line
(gdb) next

# quit
(gdb) quit
```

Cheat Sheet: https://darkdust.net/files/GDB%20Cheat%20Sheet.pdf

Valgrind

`Valgrind` supports lots of tools, like `memcheck`, `addrcheck`, `cachegrind`...

If we do not specify which one, `memcheck` is the default.

`memcheck` detects problems with memory management in programs. It checks all read/write operations to memory and intercepts all malloc/new/free/delete calls. So the memcheck tool is able to detect the following problems:

- use uninitialized memory
- Read/write memory that has been freed
- read/write memory out of bounds
- Read/write inappropriate memory stack space
- memory leak
- Using malloc/new/new[] and free/delete/delete[] do not match.
- Overlap of src and dst

1. Installed ?

   ```
   $ valgrind --version
   ```

2. Comilpe program with debug option `-g`

3. Use `Valgrind`

```
$ valgrind --leak-check=full --show-leak-kinds=all ./a.out
```

   - `--leak-check=full`: "each individual leak will be shown in detail"
   - `--show-leak-kinds=all`: Show all of "definite, indirect, possible, reachable" leak kinds in the "full" report
     - *"indirectly lost" means **your program is leaking memory in a pointer-based structure**. (E.g. if the root node of a binary tree is "definitely lost", all the children will be "indirectly lost".)*

4. Example

```c
#include <stdlib.h>

int main()
{
    char *x = (char*)malloc(20); // line 5

    return 0;
}
```

```
==42== HEAP SUMMARY:
==42==     in use at exit: 20 bytes in 1 blocks
==42==   total heap usage: 1 allocs, 0 frees, 20 bytes allocated
==42==
==42== 20 bytes in 1 blocks are definitely lost in loss record 1 of 1
==42==    at 0x483E899: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==42==    by 0x10914A: main (memory_leak.c:5)
==42==
==42== LEAK SUMMARY:
==42==    definitely lost: 20 bytes in 1 blocks
==42==    indirectly lost: 0 bytes in 0 blocks
==42==      possibly lost: 0 bytes in 0 blocks
==42==    still reachable: 0 bytes in 0 blocks
==42==         suppressed: 0 bytes in 0 blocks
==42==
==42== For lists of detected and suppressed errors, rerun with: -s
==42== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```