Call Stack and Stack Frames

Suppose the function `main` calls `f`, then `f` calls `g`, and `g` calls `h`.

As the program jumps from function to function, we need to remember the history of the return addresses, as well as all the parameters and local variables. This history is known as the **call stack**.

The entries that are pushed onto the call stack are known as **stack frames**.

Each function call creates a new stack frame that contains the following:

```c
// ===========================
// <function name>:
// <parameter one>: <value>
// <parameter two>: <value>
// ...
// <local variable one>: ( <value> || ??? )
// <local variable two>: ( <value> || ??? )
// ...
// return address: <caller function name>:<line>
// ===========================
```

If a question asks you to draw the call stack, then for each stack frame pushed onto the call stack you must provide the above.
For example:

```c
int g(void){
    int x = 2;
    return x;
}

int f(int x){
    int a = g();
    return x * a;
}

int main(void){
    int x = 4;
    int a = 1;
    int y = f(x) + a;
}
```

**example: draw the call stack**

1. int g(void){
2.    int x = 2;
3.    return x;
4. }
5. int f(int x){
6.    int a = g();
7.    return x * a;
8. }
9. int main(void){
10.   int x = 4;
11.   int a = 1;
12.    int y = f(x) + a;
13. }

```
1. int g(void){
2.    int x = 2;
3.    return x;
4. }
5. int f(int x){
6.    int a = g();
7.    return x * a;
8. }
9. int main(void){
10.   int x = 4;
11.   int a = 1;
12.    int y = f(x) + a;
13. }
```

```c

```
```c
int g(void){
    int x = 2;
    return x;
}

int f(int x){
    int a = g();
    return x * a;
}

int main(void){
    int x = 4;
    int a = 1;
    int y = f(x) + a;
}
```

```
g:
    x: 2
    return: f: 7
    ===============
f:
    x: 4
    a: ?
    return: main: 14
    ===============
main:
    x: 4
    a: 1
    y: ?
    return address: OS
    ===============
```
Memory Addresses and Pointers

A pointer can be declared by placing an **indirection operator** (*) before the identifier.

C requires you to specify the type of the value stored at an address.

```c
int i = 42;
int *p = &i;
```

Note that in this example we are declaring the variable named `p` of the type `int *`.

You can get the starting address of where the value of an identifier is stored in memory by using the **address operator** (`&`).

```c
// To print the value p is pointing at
printf("The value of i is: \%d", *p);
```

---

Content vs Address of Pointers

Recap on the difference between content and address of pointers

```c
int main(void) {
    int i = 42;
    int *p = &i;

    printf("\%d\n", i);  // 42
    printf("\%p\n", &i);  // address of i

    // You can only dereference pointers!
    // printf("\%d\n", *i);  // This is not valid

    printf("\%p\n", p);  // address of i;
    printf("\%p\n", &p);  // address of p
    printf("\%d\n", *p);  // 42
}
```
A Quick Recap of Pointers

- When defining a pointer use * between the identifier and the type
- The value of a pointer is a memory address.
- The address operator, &, gives us the address in memory where the value of a variable is stored.
- The type of the pointer specifies how the value stored at that memory address should be interpreted.

Pointers as Function Parameters

Pointers may be used as parameters to allow the function to mutate its inputs or save copying a large structure.

By passing the address of x, we can change the value of x. It is also common to say “pass a pointer to x”.

To pass the address of x, we use the address operator (&x). If x is declared to be an int, then the corresponding parameter type is an int pointer (int *).

Note: by using the address operator we get the address of x, not the value of x.

Example 1: Mutating Values in Functions

Compare these two segments of code

```c
void inc(int p) {
    p += 1;
}

int main(void) {
    int x = 42;
    inc(x);
    printf("%d
", x);
}
```

Output: 42

```c
void inc(int *p) {
    *p += 1;
}

int main(void) {
    int x = 42;
    inc(&x);
    printf("%d
", x);
}
```

Output: 43
Example 2: Passing Pointers by Value

// assign_if_greater(a, b) Assigns the value
// of b to a if b is greater than a
// effects: The value of b is assigned to a
// if b is greater than a
void assign_if_greater(int *a, int *b) {
    if (*a < *b) {
        *a = *b;
    }
}

Example 3: Call Stacks with Pointers

Trace the content of the Call Stack for the following program

```c
void swap_int(int *i, int *j) {
    int temp = *i;
    *i = *j;
    *j = temp;
}

void swap_posn(struct posn *p1, struct posn *p2) {
    swap_int(&p1->x), &p2->x));
    swap_int(&p1->y), &p2->y));
}

int main(void) {
    struct posn p1 = {3, 4};
    struct posn p2 = {5, 6};
    swap_posn(&p1, &p2);
}
```
Example 3: Call Stacks with Pointers

```c
void swap_int(int *i, int *j) {
    int temp = *i;
    *i = *j;
    *j = temp;
}

void swap_posn(struct posn *p1, struct posn *p2) {
    swap_int(&(p1->x), &(p2->x));
    swap_int(&(p1->y), &(p2->y));
}

int main(void) {
    struct posn p1 = {3, 4};
    struct posn p2 = {5, 6};
    swap_posn(&p1, &p2);
}
```

```c
void swap_int(int *i, int *j) {
    int temp = *i;
    *i = *j;
    *j = temp;
}

void swap_posn(struct posn *p1, struct posn *p2) {
    swap_int(&(p1->x), &(p2->x));
    swap_int(&(p1->y), &(p2->y));
}

int main(void) {
    struct posn p1 = {3, 4};
    struct posn p2 = {5, 6};
    swap_posn(&p1, &p2);
}
```

```c
void swap_int(int *i, int *j) {
    int temp = *i;
    *i = *j;
    *j = temp;
}

void swap_posn(struct posn *p1, struct posn *p2) {
    swap_int(&(p1->x), &(p2->x));
    swap_int(&(p1->y), &(p2->y));
}

int main(void) {
    struct posn p1 = {3, 4};
    struct posn p2 = {5, 6};
    swap_posn(&p1, &p2);
}
```
Example 3: Call Stacks with Pointers

```c
void swap_int(int *i, int *j) {
    int temp = *i;
    *i = *j;
    *j = temp;
}

void swap_posn(struct posn *p1, struct posn *p2) {
    swap_int(&(p1->x), &(p2->x));
    swap_int(&(p1->y), &(p2->y));
}

int main(void) {
    struct posn p1 = {3, 4};
    struct posn p2 = {5, 6};
    swap_posn(&p1, &p2);
}
```

---

Example 3: Call Stacks with Pointers

```c
void swap_int(int *i, int *j) {
    int temp = *i;
    *i = *j;
    *j = temp;
}

void swap_posn(struct posn *p1, struct posn *p2) {
    swap_int(&(p1->x), &(p2->x));
    swap_int(&(p1->y), &(p2->y));
}

int main(void) {
    struct posn p1 = {3, 4};
    struct posn p2 = {5, 6};
    swap_posn(&p1, &p2);
}
```

---

Example 3: Call Stacks with Pointers

```c
void swap_int(int *i, int *j) {
    int temp = *i;
    *i = *j;
    *j = temp;
}

void swap_posn(struct posn *p1, struct posn *p2) {
    swap_int(&(p1->x), &(p2->x));
    swap_int(&(p1->y), &(p2->y));
}

int main(void) {
    struct posn p1 = {3, 4};
    struct posn p2 = {5, 6};
    swap_posn(&p1, &p2);
}
```
Example 3: Call Stacks with Pointers

```c
template
1 void swap_int(int *i, int *j) {
2     int temp = *i;
3     *i = *j;
4     *j = temp;
5 }
```

```c
template
1 void swap_posn(struct posn *p1, struct posn *p2) {
2     swap_int(&(p1->x), &(p2->x));
3     swap_int(&(p1->y), &(p2->y));
4 }
```

```c
template
1 int main(void) {
2     struct posn p1 = {3, 4};
3     struct posn p2 = {5, 6};
4     swap_posn(&p1, &p2);
5 }
```
Example 3: Call Stacks with Pointers

```c
void swap_int(int *i, int *j) {
    int temp = *i;
    *i = *j;
    *j = temp;
}

void swap_posn(struct posn *p1, struct posn *p2) {
    swap_int(&(p1->x), &(p2->x));
    swap_int(&(p1->y), &(p2->y));
}

int main(void) {
    struct posn p1 = {3, 4};
    struct posn p2 = {5, 6};
    swap_posn(&p1, &p2);
}
```

---

Example 3: Call Stacks with Pointers

```c
void swap_int(int *i, int *j) {
    int temp = *i;
    *i = *j;
    *j = temp;
}

void swap_posn(struct posn *p1, struct posn *p2) {
    swap_int(&(p1->x), &(p2->x));
    swap_int(&(p1->y), &(p2->y));
}

int main(void) {
    struct posn p1 = {3, 4};
    struct posn p2 = {5, 6};
    swap_posn(&p1, &p2);
}
```

---

Example 3: Call Stacks with Pointers

```c
void swap_int(int *i, int *j) {
    int temp = *i;
    *i = *j;
    *j = temp;
}

void swap_posn(struct posn *p1, struct posn *p2) {
    swap_int(&(p1->x), &(p2->x));
    swap_int(&(p1->y), &(p2->y));
}

int main(void) {
    struct posn p1 = {3, 4};
    struct posn p2 = {5, 6};
    swap_posn(&p1, &p2);
}
```

---

Example 3: Call Stacks with Pointers

```c
void swap_int(int *i, int *j) {
    int temp = *i;
    *i = *j;
    *j = temp;
}

void swap_posn(struct posn *p1, struct posn *p2) {
    swap_int(&(p1->x), &(p2->x));
    swap_int(&(p1->y), &(p2->y));
}

int main(void) {
    struct posn p1 = {3, 4};
    struct posn p2 = {5, 6};
    swap_posn(&p1, &p2);
}
```

---

Example 3: Call Stacks with Pointers

```c
void swap_int(int *i, int *j) {
    int temp = *i;
    *i = *j;
    *j = temp;
}

void swap_posn(struct posn *p1, struct posn *p2) {
    swap_int(&(p1->x), &(p2->x));
    swap_int(&(p1->y), &(p2->y));
}

int main(void) {
    struct posn p1 = {3, 4};
    struct posn p2 = {5, 6};
    swap_posn(&p1, &p2);
}
```
Problem: Safe
Suppose we have a safe structure. The structure supports locking and unlocking as shown below.

```c
struct safe {
    bool is_locked;
    int key;
};
```

---

Problem: Safe

// make_safe(&s) initializes a safe s
// effects: Creates a new unlocked safe
void make_safe(struct safe *s);

// is_locked_safe(&s) returns true if the safe is locked
// otherwise, false is returned
bool is_locked_safe(const struct safe *s);

// recover_key(&s, &k) recovers the key of a safe s
// effects: Mutates the content of k given a valid safe
void recover_key(struct safe *s, int *k);

---

Problem: Safe

// lock_safe(&s, k) locks the safe if s was unlocked
// effects: s was unlocked, it is now locked with key k
void lock_safe(struct safe *s, int k);

// unlock_safe(&s, k) attempts to unlock the safe s
// with the key k.
// effects: Unlock s if k is the correct key
void unlock_safe(struct safe *s, int k);

Refer to Seashell for Coding Exercise
**scanf() function**

Reads data and stores it based on the parameter format.

Returns the number of values that have been successfully read. It returns EOF if it hits the end of the input.

For example:

```c
char char1;
char char2;
printf("Enter a character:");
scanf("%c", &char1); // not ignore whitespace
printf("The input is %c\n", char1);
scanf(" %c", &char2); // ignore whitespace
printf("The input is %c\n", char2);
```

(See Seashell)

---

**Perils of Pointers: Expired Scopes**

What will this program output with `foo();` commented?

How about if `foo();` is uncommented?

```c
#include <stdio.h>

// Returns an address to invalid memory
int * weird(int a, int b) {
    return &b;
}

// Doesn't do anything?
void foo(void) {
    int x = 1;
}

int main(void) {
    int *p = weird(5, 6);
    // foo();
    int star_p = *p;
    printf("%d\n", star_p);
}
```

---

**Example 4: Perils of Pointers: Stack Trace**

Trace the content of the Call Stack for the following program:

```c
// larger(a, b) returns the address of the larger integer
int *larger(int a, int b) {
    if (a > b) {
        return &a;
    } else {
        return &b;
    }
}

int main(void) {
    int x = 42;
    int y = 24;
    int *p = larger(x, y);
    printf("%d\n", *p);
}
```
Example 4: Perils of Pointers: Stack Trace

// larger(a, b) returns the address
// of the larger integer
int *larger(int a, int b) {
    if (a > b) {
        return &a;
    } else {
        return &b;
    }
}

int main(void) {
    int x = 42;
    int y = 24;
    int *p = larger(x, y);
    printf("%d
", *p);
}

Example 4: Perils of Pointers: Stack Trace

// larger(a, b) returns the address
// of the larger integer
int *larger(int a, int b) {
    if (a > b) {
        return &a;
    } else {
        return &b;
    }
}

int main(void) {
    int x = 42;
    int y = 24;
    int *p = larger(x, y);
    printf("%d
", *p);
}

Example 4: Perils of Pointers: Stack Trace

// larger(a, b) returns the address
// of the larger integer
int *larger(int a, int b) {
    if (a > b) {
        return &a;
    } else {
        return &b;
    }
}

int main(void) {
    int x = 42;
    int y = 24;
    int *p = larger(x, y);
    printf("%d
", *p);
}
Example 4: Perils of Pointers: Stack Trace

1 // larger(a, b) returns the address
2 // of the larger integer
3 int *larger(int a, int b) {
4     if (a > b) {
5         return &a;  <!-- Original comment
6     } else {
7         return &b;
8     }
9 }
10
11 int main(void) {
12     int x = 42;
13     int y = 24;
14     int *p = larger(x, y);
15     printf("%d\n", *p);
16 }

End of Stack Trace

Notice that the stack frame for main has a pointer to addr_1. However, the stack frame which contains the content of addr_1 has been popped. This means the memory at addr_1 no longer belongs to the program.

Accessing the content of memory that no longer belongs to the program will result undefined behaviour.

Function Pointers

- In C, functions are not first-class values, but function pointers are.

- A function pointer stores the (starting) address of a function, which is an address in the code section of memory.

The syntax to declare a function pointer with name fpname is:

```
return_type (*fpname)(param1_type, param2_type, ...)
```

A function pointer can only point to a function that already exists.
Problem: Find_Max_Min

Implement the function \texttt{find\_max\_min(fp, maxp, minp, upper, lower)}. 

(On Seashell)

The function does the following:

\begin{verbatim}
// find_max_min(fp, maxp, minp, upper, lower) finds
// and mutates maxp and minp with maximum and minimum
// values of fp(x), in range of the given lower
// and upper values (i.e. [lower, upper])
// requires: upper > lower

Note: you need to check every \texttt{integer} in the range [lower, upper]
\end{verbatim}