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 (&).

Memory Addresses and Pointers

To get the value of what a pointer "points at", we use the indirection operator (*). This is often known as the dereference operator and it is the inverse of the address operator (&).

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

- Allows functions to mutate variables that live outside the function.
- Allows functions to avoid copying large structures.

Example: 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: 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: 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: 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: 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: 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: 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);
}
```

CS 136 Fall 2018 Tutorial 6
Example: 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: 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);
    End of Stack Trace
}
```
**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); // can read whitespace
printf("The input is %c\n", char1);
scanf(" %c", &char2); // ignores whitespace
printf("The input is %c\n", char2);
```

(See Seashell)

---

**Perils of Pointers: Expired Scopes**

What will this program output when foo runs? When it doesn’t run?

```c
// Get a bad address
int *weird(int a) {
    return &a;
}

// #JUSTFOOTINGS
void foo(void) {
    int x = 1;
}
```

```c
int main(void) {
    int p_content;
    int *p = weird(5);
    p_content = *p;
    printf("%d\n", p_content);
}
```

---

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

Trace the content of the Call Stack for the following program

```c
int *foo(int a) {
    return &a;
}

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

```c
int *foo(int a) {
    return &a;
}

int main(void) {
    int x = 42;
    int *p = foo(x);
    printf("%d\n", *p);
}
```

---

```
=====================================
main:
  x: 42
  p: ?
  return address: OS

```

---

Example: Perils of Pointers: Stack Trace

```c
int *foo(int a) {
    return &a;
}

int main(void) {
    int x = 42;
    int *p = foo(x);
    printf("%d\n", *p);
}
```

---

```
=====================================
foo:
  a: 42 [addr]
  return address: main:7

```

---

```
=====================================
main:
  x: 42
  p: ?
  return address: OS

```

---

Example: Perils of Pointers: Stack Trace

```c
int *foo(int a) {
    return &a;
}

int main(void) {
    int x = 42;
    int *p = foo(x);
    printf("%d\n", *p);
}
```

---

```
End of Stack Trace

```

---

```
=====================================
main:
  x: 42
  p: addr
  return address: OS

```

---
Example: Perils of Pointers: Stack Trace

===================================== main:
  x: 42
  p: addr
  return address: OS

Notice that the stack frame for main has a pointer to addr. However, the stack frame which contains the contents of addr has been popped. Accessing the contents of this memory will result in undefined behaviour.

Function Pointers

- Functions are not first-class values in C.
- Function pointers, on the other hand, 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:

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

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

Problem: find_max

Implement the function `find_max(fp, lower, upper)` on Seashell.

The function does the following:

```c
// find_max(fp, lower, upper) returns the maximum value of fp(x) in the range [lower, upper]
// requires: upper > lower
```
Declarations vs. Definitions

// Declaration:
bool is_impossible(int power_level);

// Definition:
bool is_impossible(int power_level) {
    return (power_level > 9000);
}

Why?

- Reusability.
- Maintainability.
- Abstraction.