Perils of Pointers: Expired Scopes

Suppose we write a function that outputs the memory address of the function parameter.

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

What will this program output when it runs?

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

Trace the content of the Call Stack for the following program

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

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

```
=====================================  main:
x: 42
p: ?
return address: OS
=====================================  weird:
a: 42 [addr]
return address: main:7
```

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

Example: Perils of Pointers: Stack Trace

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

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

```
=====================================  main:
x: 42
p: ?
return address: OS
=====================================  weird:
a: 42 [addr]
return address: main:7
```

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

Example: Perils of Pointers: Stack Trace

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

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

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

Example: Perils of Pointers: Stack Trace

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

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

```
=====================================  main:
```
```c
int * weird(int a) {
    return &a;
}

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

End of Stack Trace

---

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.

---

**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.
scanf() function

What does this program do?

```c
#include <stdio.h>

int main(void) {
    char c;
    while (scanf(" %c", &c) == 1) {
        printf("%c", c);
    }
    printf("\n");
}
```

Declarations vs. Definitions

```c
// Declaration (impossible.h):
bool is_impossible(int power_level);

// Definition (impossible.c):
bool is_impossible(int power_level) {
    return (power_level > 9000);
}
```

Why use interface?

- Reusability
- Maintainability
- Abstraction
Interface vs. Implementation

We can provide the interface to the client. We can also hide the implementation.

Why?

• Keep code safe
• Ease of Use
• Hide sensitive information
• Flexibility to change implementation

Typical Interface/Header File

```c
#include <stdbool.h>
// Stack: provides a Integer Stack ADT
struct stack;
// stack_create() creates a new empty stack
// effects: allocates memory
struct stack *stack_create(void);

// stack_destroy(s) frees all memory for s
// effects: s is no longer valid
void stack_destroy(struct stack *s);

/* modules don't contain the main function! */
```

Example Program

Interface: impossible.h

```c
#include "cs136.h"

// is_impossible(power_level) determines whether the given
// power level is plausible.
bool is_impossible(int power_level);
```

Implementation: impossible.c

```c
#include "impossible.h"

bool is_impossible(int power_level) {
    return (power_level > 9000);
}
```
Stack

Define the following C function using stack.h:

Write a program that reads in ints, and determines the count of how many numbers were read in. It prints the numbers in reverse order, adding the count to each number.

Reverse

Define the following C functions using stack.h:

// read_nums(s) reads int from input and pushes them into s, returns the total number of int successfully read
// effects: reads from input
// modifies *s
// requires: s is not NULL

// print_reverse(count, s) prints (num + count) for each num in the stack from top to bottom
// effects: produces output
// requires: s is not NULL