Perils of Pointers: Expired Scopes

What will this program output when it runs?

```c
int * weird(int a) {
    return &a;
}
int main(void) {
    int p_content = 0;
    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

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

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

---

End of Stack Trace
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.

Problem: Vector Product

Implement the function

```c
void vector_product(const struct vector *p1,
                   const struct vector *p2,
                   int *dot,
                   struct vector *cross);
```

// vector_product(p1, p2, dot, cross):
// modifies *dot and *cross so that they represent the
dot and cross products of the vectors p1 and p2.
// Effects:
// * Modifies *dot and *cross

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)
Declarations vs. Definitions

// Declaration:
bool is_impossible(int power_level);

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

Why use definitions?

• 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

/* header.h */

// This is a description of the file.
#include "cs136.h"

bool predicate_fn(int n);
int side_effects_fn(int *p);

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

Example Program

Interface: impossible.h
#include "cs136.h"

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

Implementation: impossible.c
#include "impossible.h"

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

Stack

Define the following C function from A2 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.