CS 33

Introduction to C
Part 4
Even though \( a \) is given a value the first time \( \text{func} \) is called, on \( \text{func} \)'s second invocation \( a \) is not given a value and thus the result that's printed is “undefined”. This is because the lifetime of \( a \) is just for the length of time its scope is active, which is from when the execution of \( \text{func} \) starts to when \( \text{func} \) returns. The \( a \) in the next invocation of \( \text{func} \) is different from the previous \( a \).
In this case, \( a \) is global and thus the value set for it in one invocation of \( \text{func} \) is still there for the next invocation – the lifetime of \( a \) is that of the program itself.
Here $a$ is local again. `func` is called (recursively) from within itself: the recursive invocation of `func` modifies a different $a$ than is used in the first invocation. Thus the value printed is 2.
When a function returns, its local variables become out of scope and no longer active – the lifetime of local variables is from the instant the function is called to when it returns. Thus a pointer to a local variable refers to an undefined value if the variable is of a function invocation that is no longer active.
Similarly, a the lifetime of function arguments is the same as the lifetime of the function.
Rules

- Global variables exist for the duration of program’s lifetime
- Local variables and arguments exist for the duration of the execution of the procedure
  - from call to return
  - each execution of a procedure results in a new instance of its arguments and local variables
Function calling in C (and in most other languages) is implemented on stacks. Associated with an invocation of a function is a stack frame, which contains, among other things, its arguments and local variables. When a function is called, a stack frame for it is pushed onto the stack. When it returns, its stack frame is popped off the stack.
int main() {
    int a;
    func1(0);
    ...
}

int func1(int x) {
    int a,b;
    if (x==0) func2(a,2);
    ...
}

int func2(int x, int y) {
    int a,b,c;
    func1(1);
    ...
}
void proc(int a) {
    int b=1;
    if (a == 1) {
        proc(2);
        printf("%d\n", b);
    } else {
        b = a*(b++)*b;
    }
}

int main() {
    proc(1);
    return 0;
}
Static local variables have the same scope as other local variables, but their values are retained across calls to the procedures they are declared in. Like global variables, uninitialized static local variables are implicitly initialized to zero.

```c
int *sub1() {
    int var = 1;
    ...
    return &var;
    /* amazingly illegal */
}

int *sub2() {
    static int var = 1;
    ...
    return &var;
    /* (amazingly) legal */
}
```

- **Scope**
  - like local variables
- **Lifetime**
  - like global variables
int sub() {
    static int svar = 1;
    int lvar = 1;
    svar += lvar;
    lvar++;
    return svar;
}

int main() {
    sub();
    printf("%d\n", sub());
    return 0;
}
The function `scanf` is called to read input, doing essentially the reverse of what `printf` does. Its first argument is a format string, like that of `printf`. Its subsequent arguments are pointers to locations where the input should be copied (after format conversion as specified in the format string). Note that we must have pointers for these arguments, not simple values, since arguments are passed by value. (Make sure you understand why this is important!)

The format conversion done is the reverse of what `printf` does. For example, `printf`, given the `%d` format code, converts the machine representation of an integer into its string representation in decimal notation. `scanf` with the same format code takes the string representation of a number in decimal notation and converts it to the machine representation of an integer.
#define (again)

```
#define CtoF(cel) (9.0*cel)/5.0 + 32.0
```

Simple textual substitution:

```c
float tempc = 20.0;
float tempf = CtoF(tempc);
// same as tempf = (9.0*tempc)/5.0 + 32.0;
```
Be careful with how arguments are used! Note the use of parentheses in the second version.
Structures

```c
struct ComplexNumber {
    float real;
    float imag;
};
```

```c
cstruct ComplexNumber x;
x.real = 1.4;
x.imag = 3.65e-10;
```
Note that when we refer to members of a structure via a pointer, we use the “->” notation rather than the “.” notation.
structs and Functions

struct ComplexNumber ComplexAdd(
    struct ComplexNumber a1,
    struct ComplexNumber a2) {
    struct ComplexNumber result;
    result.real = a1.real + a2.real;
    result.imag = a1.imag + a2.imag;
    return result;
}
This doesn’t work, since it returns a pointer to result that would not be in scope once the procedure has returned. Thus the returned pointer would point to an area of memory with undefined contents.
This works fine: the caller provides the location to hold the result.

```c
void ComplexAdd(
    struct ComplexNumber *a1,
    struct ComplexNumber *a2,
    struct ComplexNumber *result) {
    result->real = a1->real + a2->real;
    result->imag = a1->imag + a2->imag;
    return;
}
```
Using It ...

```c
struct ComplexNumber j1 = {3.6, 2.125};
struct ComplexNumber j2 = {4.32, 3.1416};
struct ComplexNumber sum;

ComplexAdd(&j1, &j2, &sum);
```
Arrays of structs

struct ComplexNumber j[10];
j[0].real = 8.127649;
j[0].imag = 1.76e18;
Arrays, Pointers, and structs

/* What's this? */
struct ComplexNumber *jp[10];

struct ComplexNumber j0;
jp[0] = &j0;
jp[0]->real = 13.6;

Subscripting (i.e., the “[]” operator) has a higher precedence than the “*” operator. Thus `jp` is an array of pointers to `struct ComplexNumbers`. 
Quiz 3

```c
struct list_elem {
    int val;
    struct list_elem *next;
} a, b;

int main() {
    a->val = 1;
    a->next = &b;
    b->val = 2;
    printf("%d\n", a->next->val);
    return 0;
}
```

- What happens?
  a) syntax error
  b) seg fault
  c) prints something and terminates
Quiz 4

```c
struct list_elem {
    int val;
    struct list_elem *next;
} a, b;

int main() {
    a.val = 1;
    a.next = &b;
    b.val = 2;
    printf("%d\n", a.next.val);
    return 0;
}
```

- What happens?
  a) syntax error
  b) seg fault
  c) prints something and terminates
Quiz 5

```
struct list_elem {
    int val;
    struct list_elem *next;
} a, b;

int main() {
    a.val = 1;
    b.val = 2;
    printf("%d\n", a.next->val);
    return 0;
}
```

- What happens?
  a) syntax error
  b) seg fault
  c) prints something and terminates
Quiz 6

```c
struct list_elem {
    int val;
    struct list_elem *next;
} a, b;

int main() {
    a.val = 1;
    a.next = &b;
    b.val = 2;
    printf("%d\n", a.next->val);
    return 0;
}
```

- **What happens?**
  
  a) syntax error  
  b) seg fault  
  c) prints something and terminates
for (;;)
    printf("C does not have objects!\n");
This seems pretty weird at first glance. But keep in mind that the name of an array refers to the address its first element, and does not represent the entire array. But the name of a structure refers to the entire structure.

```c
struct Array {  
    int A[6];  
} S1, S2;

int A1[6], A2[6];

A1 = A2;  // not legal: arrays don’t know how big they are

S1 = S2;  // legal: structures do
```
A Bit More Syntax ...

- Constants
  ```
  const double pi =
  3.141592653589793238;
  
  area = pi*r*r;    /* legal */
  pi = 3.0;         /* illegal */
  ```
Note that constant_ptr_to_constant’s value may not be changed, and the value of what it points to may not be changed.
And Still More …

- **Array initialization**

```c
int SomeMorePrimes[] = {17, 19, 23, 29};
int MoreWithRoomForGrowth[10] = {31, 37};
int MagicSquare[][] = {{2, 7, 6},
                      {9, 5, 1},
                      {4, 3, 8}};
```