CS113: Lecture 4

Topics:

- Functions
- Function Activation Records

Why functions?

- Functions add no expressive power to the C language in a formal sense.

- Why have them?
  - Breaking tasks into smaller ones make them easier to think and reason about
  - Facilitates code re-use (not just within one program, but in others)
  - Makes it possible to hide away details of one task from the rest of program, which does not care

- The ideal function performs a single, well-defined task: testing if a given number is prime, computing the number of days in a given month, etc.
  - *The less specific a function is to the program in which it is initially used, the more reusable it tends to be*
  - On the other hand, don't waste lots of time writing a general function if it's not needed for your task. Code reuse is nice, but atypical in practice.

Example #1: A simple function

```
int power( int base, int exp ) {
    int i, p = 1;
    for( i = 1; i <= exp; i++ )
        p *= base;
    return p;
}
```

Anatomy of a function:

- Function has the general form

  ```
  return-type function-name( parameters ) {
      declarations
      statements
  }
  ```

- Function definitions can appear in any order

- Names used by `power` (`base`, `exp`, `p`) are "local" to `power`, and are not visible to any other functions. DO NOT USE GLOBAL VARIABLES.

- If the `return-type` is non-`void`, every path of execution must end in a `return` statement

- If the `return-type` is `void`, then `return;` can be used to terminate the function at any point

Example #1 in context

```
int power( int base, int exp );

void main() {
   int i = 3, j = 4;
   printf( "%d raised to the power %d is %d.\n",
           i, j, power( i, j ));
}

int power( int base, int exp ) {
    int i, p = 1;
    for( i = 1; i <= exp; i++ )
        p *= base;
    return p;
}
```

Notes:

- `int power( int base, int exp );` at the top is a *function prototype*; tells compiler what kind of function `power` is, so that it can check function calls

- Distinction between
  - *parameters* of a function – variables in parenthesized list (for `power`, parameters are `base`, `exp`), and
  - *arguments* of a function call – the actual values passed to the function (here, `3`, `4`)

Yet another example

```
void print_square( int a );
int square( int a );

void main() {
   int i;
   for( i = 1; i <= 10; i++ )
      print_square( i );
}

void print_square( int a ) {
   printf( "The square of %d is: %d\n", a, square( a ));
}

int square( int a ) {
   return( a * a );
}
```

Call by value

- In C, all arguments to functions are passed *by value*: the function is given *copies* of the arguments, and *not* the originals

- A called function is given the values of its arguments in temporary variables, not the originals

- A called function cannot directly alter a variable in the calling function – it can only alter its private, temporary copy

- Example.

```
void increment( int a ) {
   a++;
}

void main() {
   int b = 3;
   increment( b );
   printf( "%d\n", b );
}
```

Call by value: an asset?

K & R says, "Call by value is an asset . . . not a liability."

Here's one example:

```
int power( int base, int n ) {
   int i, p;
   p = 1;
   for( i = 1; i <= n; i++ )
      p = p * base;
   return( p );
}
```

which is equivalent to a program where we "use up" the function argument `n`:

```
int power( int base, int n ) {
   int p;
   for( p = 1; n > 0; n-- )
      p = p * base;
   return( p );
}
```

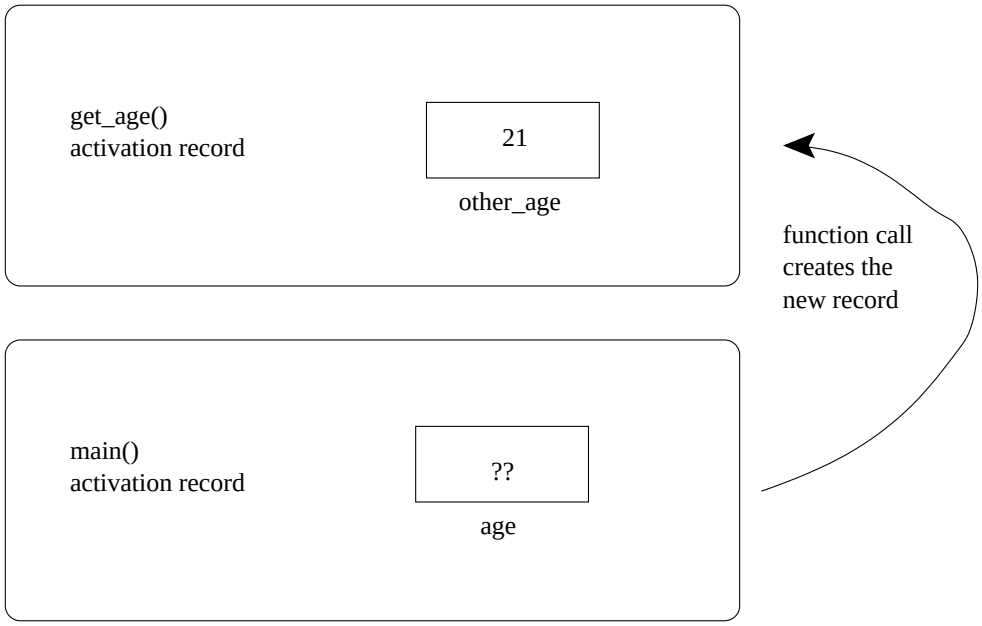
What's the problem here?

```
#include <stdio.h>

void get_age( int age );

void main() {
   int age;
   get_age( age );
   printf( "Your age is: %d\n", age );
}

void get_age( int other_age ) {
   printf( "Please enter your age.\n" );
   scanf( "%d", &other_age );
}
```

The value of `other_age` gets destroyed when the function `get_age` returns!

Activation Records in C: A First Look

A high-level view of memory:



- C is call-by-value, so a new `other_age` memory location gets created, and when the function gets called, the argument to the function gets copied into it.

- When `get_age()` finishes, its activation record is destroyed. so the value stored in `other_age` is lost.

- Meanwhile, the `age` variable in function main is still uninitialized...

Activation Records

More formally,

- An activation record is a chunk of memory that is allocated by the program every time a function is invoked.

- It includes space (in memory) for the *parameters* of the function, as well as all the variables declared by the function.

- The parameters are initialized by the *arguments* of the function.

- The activation record also includes a "pointer" to the calling function, so the program knows where to go when the function finishes.

- The activation records form an *execution stack*, with main() at the bottom and the current function at the top.

Other languages, like Java, also include activation records. It isn't strictly necessary to understand how they work to use the language effectively, but with a low-level language like C it definitely pays off.

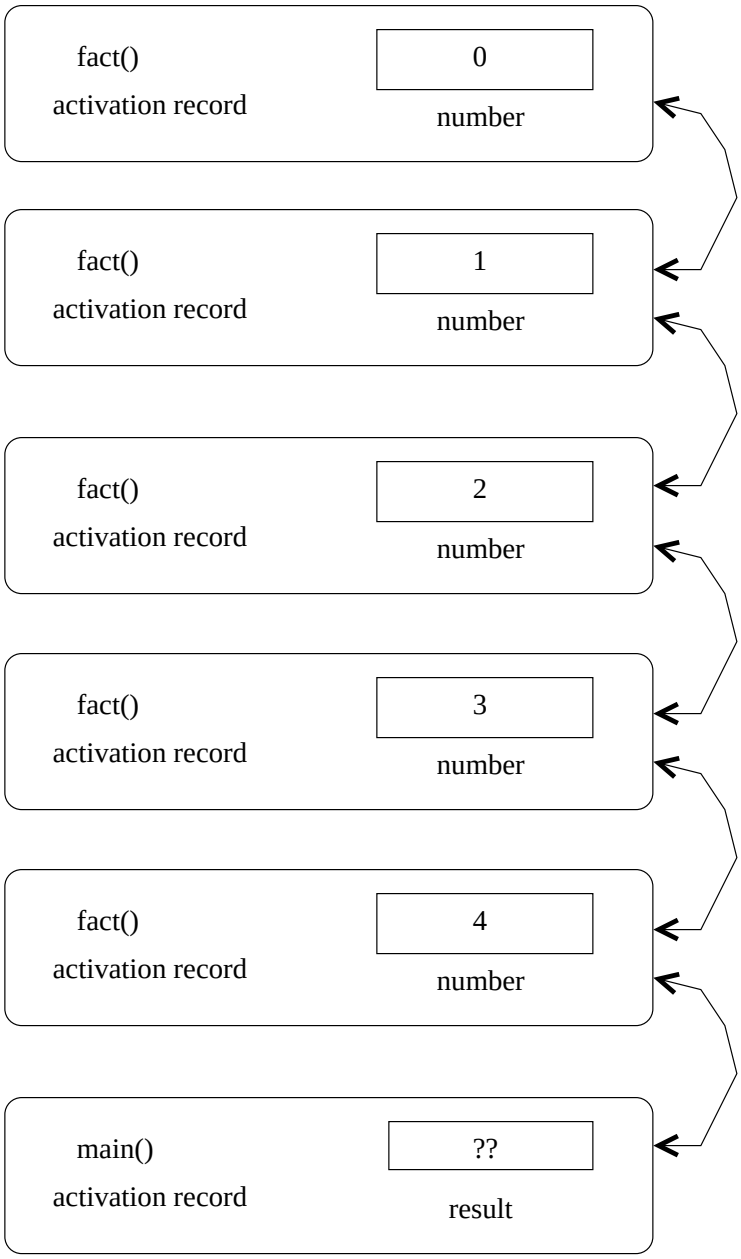
An example of a recursive function

Because a fresh activation record is created each time a function is called, C supports *recursion*, where a function calls itself:

```
int fact( int number );

void main() {
   int result;
   result = fact(4);
   printf( "4 factorial is %d\n", result );
}

int fact( int number ) {
   if( number == 0 )
      return 1;
   /* else */
   return( number * fact( number - 1 ));
}
```

Activation records with recursion

Here's what the execution stack looks like after the last call to `fact()` in the previous program, but before it returns:



Apples and Oranges

```
#include <stdio.h>

void print_fruits( int oranges, int apples );

void main() {
   int apples, oranges;
   printf( "How many apples do you have?" );
   scanf( "%d", &apples );
   printf( "How many oranges do you have?" );
   scanf( "%d", &oranges );
   print_fruits( apples, oranges );
}

void print_fruits( int oranges, int apples ) {
   printf( "You have %d oranges and %d apples.",
           oranges, apples );
}
```

How old are you?

```
#include <stdio.h>

int get_age( int year );

void main() {
   int age, year;
   year = 0;
   age = get_age( year );
   printf( "Your age is: %d\n", age );
}

int get_age( int year ) {
   printf( "What year were you born?" );
   scanf( "%d", &year );
   return( 2005 - year );
}
```

Play again?

```
int play_again() {
   char response;
   printf( "Would you like to play again (Y/N)?" );
   scanf( "%c", &response );
   if( response == 'Y' )
      return( 1 );
   else if( response == 'N' )
      return( 0 );
}
```

Tallying scores

```
#include <stdio.h>

void do_one_score( int total );

void main() {
   int total, i;
   total = 0;
   for( i = 1; i <= 10; i++ )
      do_one_score( total );
   printf("Total: %d\n", total);

}

void do_one_score( int total ) {
   int score;
   printf( "Enter a score: " );
   scanf( "%d", &score );
   total += score;
}
```

Tallying scores, again

Global variables are visible to every function in the source file.

```
#include <stdio.h>
int total = 0;
void do_one_score( void );

void main() {
   int i;
   for( i = 1; i <= 10; i++ )
      do_one_score( );
   printf("Total: %d\n", total);

}

void do_one_score( void ) {
   int score;
   printf( "Enter a score: " );
   scanf( "%d", &score );
   total += score;
}
```

Tallying scores, yet again

`static` variables persist between function calls.

```
#include <stdio.h>
int running_total( void );

void main() {
   int i, total;
   for( i = 1; i <= 10; i++ )
      total = do_one_score( );
   printf("Total: %d\n", total);

}

int running_total( void ) {
   static int total = 0;
   int score;
   printf( "Enter a score: " );
   scanf( "%d", &score );
   total += score;
   return total;
}
```