CS113: Lecture 4

Topics:

- Functions
Why functions?

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

- Why then?
  - 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
  - For functions that return “true/false” values, representing true by 1 and false by 0 is very conventional
**Example #1: A simple function**

```c
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

  ```c
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

```c
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

```c
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 );
}
```
Converting lengths

float cm_to_inches( float cm );
float inches_to_feet( float inches );
float cm_to_feet( float cm );

void main()
{
    float input;
    printf( "Enter a length in centimeters:" );
    scanf( "%f", &input );
    printf( "%f cm is equal to %f feet.\n",
            input, cm_to_feet( input ));
}

float cm_to_inches( float cm )
{
    return( cm / 2.54 );
}

float inches_to_feet( float inches )
{
    float feet;
    feet = inches / 12;
    return( feet );
}

float cm_to_feet( float cm )
{
    return( inches_to_feet( cm_to_inches( cm )));
}
Call by value

- In C, all arguments to functions are passed \textit{by value}: the function is given \textit{copies} of the arguments, and \textit{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.

```c
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.”

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

...is equivalent to...

```c
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 age )
{
    printf( "Please enter your age.\n" );
    scanf( "%d", &age );
}
#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.\n", 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( 2002 - year );
}
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 );
}

void do_one_score( int total )
{
    int score;
    printf( "Enter a score: " );
    scanf( "%d", &score );
    total += score;
}
An example of a recursive function

```c
int fact( int number );

void main()
{
    printf( "6 factorial is %d\n", fact( 6 ) );
}

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