Lecture 15: Program organization
Function Declaration & Definition!

```
return-type function-name (parameters) ;
```

```
return-type function-name (parameters)
{
    declarations
    statements
}
```
```c
#include <stdio.h>

double average(double a, double b);//Declaration

main(void)
{
    double x=8, y=10, z;

    z = average(x, y);               //Call
}

double average(double a, double b) //Definition
{
    return (a + b) / 2;
}
```
Program Organization
Local Variables

• A variable declared in the body of a scope is said to be *local* to the scope:

```c
int multiplication(int a, int b)
{
    int result = 0;   /* local variable */
    result = a * b;
    return result;
}
```
Local Variables

• Default properties of local variables:
  
  – **Automatic storage duration.** Storage is "automatically" allocated when the enclosing function is called and deallocated when the function returns.

  – **Block scope.** A local variable is visible from its point of declaration to the end of the enclosing scope body.

```c
void f(void)
{
    ...
    int i;
    ...
}  // scope of i
```
Local Variables

```c
void function1() {
    int x=0;
}

void function2() {
    int x=0;
}

main() {
    int x=0;
}
```
External Variables

• Passing arguments is one way to transmit information to a function.
• Functions can also communicate through *external variables*—variables that are declared outside the body of any function.
• External variables are sometimes known as *global variables*. 
Global Variables

```c
int x;
void function1() {
    x=0;
}
void function2() {
    x=0;
}
main() {
    x=0;
}
```
Pros and Cons of External Variables

• External variables are convenient when many functions must *share a variable* or when a *few functions share a large number of variables*.

• In most cases, it’s better for functions to communicate through parameters rather than by sharing variables:
  – If an external variable is assigned an *incorrect value*, it may be difficult to identify the guilty function.
  – Functions that rely on external variables are hard to *reuse* in other programs.
• Suppose that several functions need a variable named \( i \) to control a `for` statement.

• Instead of declaring \( i \) in each function that uses it, some programmers declare it just once at the top of the program (This practice is misleading).
• Make sure that external variables have meaningful names.

• Local variables don’t always need meaningful names: it’s often hard to think of a better name than \( i \) for the control variable in a \texttt{for} loop.
• Making variables external when they should be local can lead to some rather frustrating bugs.
• Code that is supposed to display a $10 \times 10$ arrangement of asterisks:

```c
int i;

void print_one_row(void)
{
    for (i = 1; i <= 10; i++)
        printf("*");
}

void print_all_rows(void)
{
    for (i = 1; i <= 10; i++) {
        print_one_row();
        printf("\n");
    }
}

Instead of printing 10 rows, `print_all_rows` prints only one.
#include<stdio.h>

// Global variables
int A;
int B;

int Add() {
    return A + B;
}

int main() {
    int answer; // Local variable
    A = 5;
    B = 7;
    answer = Add();
    printf("%d\n", answer);
    return 0;
}
• Develop a program that generates a random number between 1 and 100, which the user attempts to guess in as few tries as possible:

Guess the secret number between 1 and 100.

A new number has been chosen.
Enter guess: 55
Too low; try again.
Enter guess: 65
Too high; try again.
Enter guess: 60
Too high; try again.
Enter guess: 58
You won in 4 guesses!

Play again? (Y/N) y
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define MAX_NUMBER 100

void initialize_number_generator(void);
int new_secret_number(void);
void read_guesses(int secret_number);

int main(void)
{
    char command;
    int secret_number;

    printf("Guess the number between 1 and %d.\n\n", MAX_NUMBER);
    initialize_number_generator();
    do {
        secret_number = new_secret_number();
        printf("A new number has been chosen.\n");
        read_guesses(secret_number);
        printf("Play again? (Y/N) \n");
        scanf(" %c", &command);
    } while (command == 'y' || command == 'Y');

    return 0;
}
void initialize_number_generator() { srand((unsigned) time(NULL)); }

int new_secret_number() { return rand() % MAX_NUMBER + 1; }

void read_guesses(int secret_number) {
    int guess, num_guesses = 0;
    while (1) {
        num_guesses++;
        printf("Enter guess: ");
        scanf("%d", &guess);
        if (guess == secret_number) {
            printf("You won in %d guesses!\n\n", num_guesses);
            return;
        }
        else if (guess < secret_number)
            printf("Too low; try again.\n");
        else
            printf("Too high; try again.\n");
    }
}
Blocks

• C allows compound statements to contain declarations as well as statements:

\[
\{ \text{declarations statements} \}
\]

• This kind of compound statement is called a **block**.

```c
#include <stdio.h>

int main()
{
    {  
        int x=0;
    }
    {  
        x = 100;               //Undeclared identifier
    }
    return 0;
}
```
• Example of a **block**:

```c
int i, j; //global to block
if (i > j) {
    int temp = i; //local to block
    i = j;
    j = temp;
}
```
• By default, the storage duration of a variable declared in a block is automatic: storage for the variable is *allocated* when the block is entered and *deallocated* when the block is exited.

• The variable has block scope; it *can’t be referenced outside the block*. 
Scope

• In a C program, the same identifier may have several different meanings.

• C’s scope rules enable the programmer (and the compiler) to determine which meaning is relevant at a given point in the program.

• The most important scope rule: When a declaration inside a block names an identifier that’s already visible, the new declaration temporarily “hides” the old one, and the identifier takes on a new meaning.