Lecture Notes for Lecture 3 of CS 5001 (Foundations of CS) for the Fall, 2017 session at the Northeastern University Silicon Valley Campus.

Functions and scopes

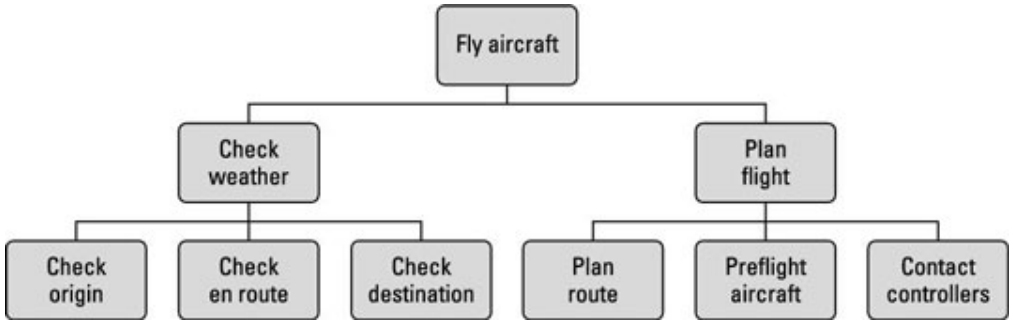
Philip Gust,
Clinical Instructor
Department of Computer Science

Lecture 2 Review

- C programs consists of data and instructions that compute a result.
- Identifiers specify name of storage that describes its purpose, and the data type for selecting the correct operations.
- Data types include Boolean, Character, Integer, and Real Number. Variations for different size integers (2, 4, 8, or 10 bytes) and floating point precision (7, 16, or 19 decimal digits).
- Operators available for assignment, arithmetic, comparison, logical, and conditional operations.
- Expressions provide a way to combine operators, variables, and constants.
- Programs consist of variable declaration, and statements within functions that are executed one after another to produce a result.
- A program has main function where execution begins.

What is a Function?

- A function is a way to create a new operation that can be combined with built-in operators and other library functions.
  - Saves coding by creating a frequently used computation once, and reuse it wherever needed
  - Helps make program more understandable by replacing sequence of statements by the use of a function that *encapsulates* a computation and whose name describes its purpose
  - Provides a way to design a complex computation by decomposing it into a number of functions that are simpler to design, use, test, and maintain (*functional decomposition*).



C Library Functions

- The C runtime includes many function libraries for a range of operations that can be used in a program.
  - Libraries are collections of functions that perform related functions
  - Use of libraries usually requires include statement for definitions
- Examples of C standard libraries
  - math: mathematical functions:
    - transcendentals, min/max, square root, ceiling/floor, etc.
  - stdarg: variable number of function parameters
  - stdio: input/output operations
    - printing/writing values, reading/inputting values, etc.
  - stdlib: miscellaneous C functions
    - sorting, searching, random numbers, memory management
  - time: functions for getting and processing time values

Anatomy of a C Function

- Here is the general outline of a C function

```
/*
 * Description of function.
 * @param param1 description of param1
 * …
 * @param paramN description of paramN
 * @return description of return value
 */
return-type function-name (type1 param1, … , typeN paramN) {
    statement-1;
    …
    statement-M;
    return return-value;
}
```

Anatomy of a C Function

Function Documentation

- A function begins with a documentation block that describes what it does, its parameters, and its return value
  - Note: A function need not have parameters or a return value
- Style of documentation facilitates automatically-generated external documentation using Doxygen.

```
/*
 * Description of function.
 * @param param1 description of param1
 * …
 * @param paramN description of paramN
 * @return description of return value
 */
```

Anatomy of a C Function

Function Declaration

- A function declaration includes the type returned by function, and identifiers for its *formal parameters*
- Formal parameters are assigned actual values when the function is *called*. This is known as *call-by-value*.
  - specify that the function takes no parameters using *void*.
- Parameters act as local variables and constants within the function. Changing them has no effect outside the function.
- *Return-type* is the type of value returned by the function.
  - specify that the function returns no value using *void.*

return-type function-name (type1 *param1*, … , typeN *paramN*) {
    …
}

Anatomy of a C Function

Function Body

- Function body is a sequence of statements that perform a computation using function parameters, and global and local variables and constants.
- Return statement returns return-value to caller.
  - May be multiple return statements within function, or none if return type is *void*.

```
statement-1;

…

statement-N;

return return-value;     // not specified if return type is void
```

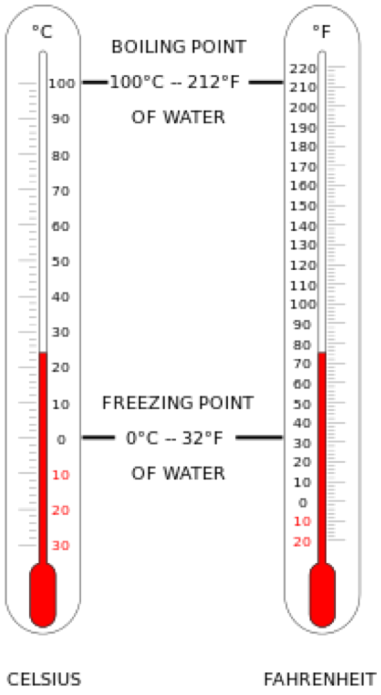
Anatomy of a C Function

Function Body

- The int returned by the main function indicates the exit status of the program.
- Returning EXIT_SUCCESS (0) indicates successful execution, and EXIT_FAILURE (1) indicates that a problem occurred. These values are defined in the stdlib standard C library.
- The C standard has a special rule that makes a return statement from the main function optional. This is a convenience since most programs terminate normally.
- If execution reaches the *}* that terminates the main function, the function implicitly returns the value EXIT_SUCCESS.

Anatomy of a C Function

- Example: Converting between Fahrenheit and Celsius
  - The Fahrenheit and Celsius temperature scales are both tied to the temperature at which water freezes and boils at sea level.
  - In the Fahrenheit scale, water freezes at 32° F and boils at 212° F.
  - In the Celsius scale, water freezes at 0° C and boils at 100° C.
  - There are simple formulas for converting between the two scales:
    - T(°C) = (T(°F) - 32) / 1.8
    - T(°F) = 1.8 T(°C) + 32
  - We can encode these as functions that take one argument and return a value.



Anatomy of a C Function

- Example: Converting between Fahrenheit and Centigrade

```
/**
 * @file: fahrenheit_celsius.c
 * This program demonstrates functions that convert between
 * the fahrenheit and celsius temperature scales
 */
#include <stdio.h>                                  // for input/output operations

/**
 * Convert temperature from fahrenheit to celsius
 * @param fahrenheit temperature in fahrenheit
 * @return temperature in celsius
 */
float fahrenheitToCelsius(float fahrenheit) {
        return (fahrenheit - 32.0f) / 1.8f;
}
```

Anatomy of a C Function

- Example: Converting between Fahrenheit and Centigrade

```
/**
 * Convert temperature from celsius to fahrenheit
 * @param centigrade temperature in centigrade
 * @return temperature in centigrade
 */
float centigradeToFahrenheit(float celsius) {
        return 1.8f * celsius + 32.0f;
}
```

Anatomy of a C Function

- Example: Converting between Fahrenheit and Celsius

```
/** Tests fahrenheit-celsius conversion.
*/
int main(void) {
        float freezingF = celsiusToFahrenheit(0.0f);            // test freezing temperature
        float freezingC = fahrenheitToCelsius(32.0f);
        printf("%f °F: %f °C\n", 32.0f, freezingC);
        printf("%f °C: %f °F\n", 0.0f, freezingF);

        float boilingF = celsiusToFahrenheit(100.0f);           // test boiling temperature
        float boilingC = fahrenheitToCelsius(212.0f);
        printf("%f °F: %f °C\n", 212.0f, boilingC);
        printf("%f °C: %f °F\n", 100.0f, boilingF);

        float roomF = celsiusToFahrenheit(22.0f);               // test room temperature
        float roomC = fahrenheitToCelsius(72.0f);
        printf("%f °F: %f °C\n", 72.0f, roomC);
        printf("%f °C: %f °F\n", 22.0f, roomF);
}
```

Anatomy of a C Function

- Example: Present Value
  - Present Value (PV) is a formula used in finance that calculates the present day value of an amount received at a future date. The premise is that there is "time value of money".
  - Formula may be applied to various areas of finance including corporate finance, banking finance, and investment finance.
  - Apart from the various areas of finance, the formula is also used as a component of other financial formulas.
  - $pv = fv / (1+r)^n$
    - fv: value received in future
    - r: rate of return
    - n: number of periods

Anatomy of a C Function

- Example: Present Value (present_value.c)

```
/** @file present_value.c
 * This program demonstrates computing the present value of an investment.
 */
#include <stdio.h>          // for input/output operations
#include <math.h>           // for pre-defined pow() math library function

/**
 * Calculates present day value of amount received at a future date.
 * @param futureValue future value of investment
 * @param returnRate rate of return per period (e.g. 0.05 for 5%)
 * @param nPeriods the number of periods
 * @return present value discounted over nPeriods by return rate
 * @see <a href="http://financeformulas.net/Present_Value.html">Present Value</a>
 */
double presentValue(double futureValue, double returnRate, unsigned int nPeriods) {
        return futureValue/ pow(1 + returnRate, nPeriods);  // library function
}
```

Anatomy of a C Function

- Example: Present Value (present_value.c)

```
/**
 * Tests presentValue function with different inputs.
*/
int main(void) {
	double pv;                                  // present value
	double fv = 100.0, rate= 0.05;              // future value, rate of return
	pv = presentValue(fv, rate, 1);             // test 1 year period
	printf("presentValue(%.2f,  %.2f, %u) : %.2f\n", fv, rate, 1, pv);

	pv = presentValue(fv, rate, 4);             // test 4 year period
	printf("presentValue(%.2f,  %.2f, %u) : %.2f\n", fv, rate, 4, pv);
}
```

Anatomy of a C Function

- Example: Present Value (present_value.c)
  - Actual values copied to formal parameter variables
  - Function called with formal parameters as local variables
  - Result copied to result variable

```
pv = presentValue(100.00, 0.05, 1);              // test 1 year period
                  |       |       |       '--> nPeriods
                  |       |       '--> returnRate
                  |       '-->  future value (cashFlow)
     95.24  <--'
```

Anatomy of a C Function

- Example: Computing Square Root Using Transcendentals
  - Pocket calculators typically implement good routines to compute the exponential function and the natural logarithm, and then compute the square root of *x* using the identity found using the properties of logarithms ($\ln x^n == n \ln x$) and exponentials ($e^{\ln x} == x$):
  - $\text{squareRoot}(x) = e^{\frac{1}{2} \ln x}$
  - The denominator in the fraction corresponds to the *n*th root. In the case above the denominator is 2, hence the equation specifies that the square root is to be found.

Anatomy of a C Function

- Example: Computing Square Root Using Transcendentals

```
/**
 * @file square_root.c
 * This program demonstrates computing the square root function
 * using transcendental functions.
 */
#include <stdio.h>                      // standard input/output functions
#include <math.h>                       // math library functions

/**
 * Compute square root of number using transcendental function
 * exp(x) – e raised to the power x, and log(x) – the log base e of x.
 * Note: named squareRoot(x) to avoid replacing math library sqrt(x).
 * @param x the number
 * @return the square root of the number
 */
double  squareRoot(double x) {
        return exp(log(x)/2);           // library functions
}
```

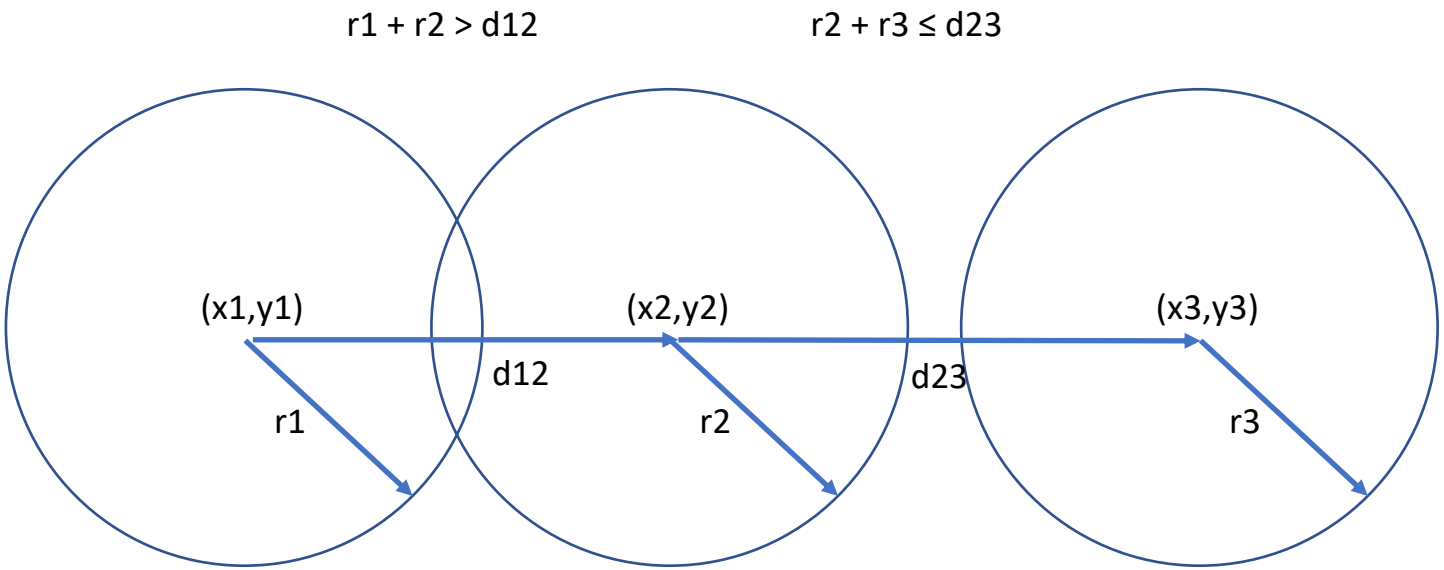
Anatomy of a C Function

- Example: Computing Square Root Using Transendentals

```
/**
 * Verify the squareRoot function with various test values.
*/
int main (void) {
        printf (“squareRoot(%f): %f\n”, 0.0, sqareRoot(0.0));
        printf (“squareRoot(%f): %f\n”, 1.0, sqareRoot(1.0));
        printf (“squareRoot(%f): %f\n”, 2.0, sqareRoot(2.0));
        printf (“squareRoot(%f): %f\n”, 100.0, sqareRoot(100.0));
        printf (“squareRoot(%f): %f\n”, -1.0, sqareRoot(-1.0));  // error!
}
```

Anatomy of a C Function

- Example: Determining whether two circles intersect
  - Two circles intersect if the distance from the center of one circle to its perimeter plus the distance from center of the other circle to its perimeter is less than the distance from the center of one circle to the center of the other circle.



Anatomy of a C Function

- Example: Determining whether two circles intersect

```
/** @file circles_intersect.c
 * This program demonstrates determining whether
 * two circles intersect, give their center and radius.
 */
#include <math.h>
#include <stdbool.h>
#include <stdio.h>
/**
 * Compute distance between two points.
 * @param p1x  x value of first point
 * @param p1y  y value of first point
 * @param p2x  x value of second point
 * @param p2y  y value of second point
 */
double  distance(double p1x, double p1y, double p2x, double p2y) {
        double deltax = p1x – p2x;
        double deltay = p1y – p2y;
        return sqrt(deltax*deltax + deltay*deltay);
}
```

Anatomy of a C Function

- Example: Determining whether two circles intersect

```
/**
 * Determines whether two circles intersect.
 * @param c1x circle1 x
 * @param c1y circle1 y
 * @param c1r circle1 radius
 * @param c2x circle2 x
 * @param c2y circle2 y
 * @param c2r circle1 radius
 * @return true if intersects, false otherwise
*/
bool intersects(double c1x, double c1y, double c1r,
                double c2x, double c2y, double c2r) {
    // intersects if sum of radii greater than distance between centers
    return (c1r + c2r) > distance(c1x, c1y, c2x, c2y);
}
```

Anatomy of a C Function

- Example: Determining whether two circles intersect

```
/**
 * Test function that determines whether two circles intersect.
*/
int main (void) {
        bool test;
        // circle at 0,0 radius 10, and circle at 21,0, radius 10: do not intersect
        test = intersects(0.0, 0.0, 10.0, 21.0, 0.0, 10.0);
        printf ("intersects(%f, %f, %f, %f, %f, %f): %f\n",  0.0, 0.0, 10.0, 21.0, 0.0, 10.0,  test);

        // circle at 0,0 radius 10, and circle at 20, 0, radius 10: do not intersect
        test = intersects(0.0, 0.0, 10.0, 20.0, 0.0, 10.0);
        printf ("intersects(%f, %f, %f, %f, %f, %f): %f\n", 0.0, 0.0, 10.0, 20.0, 0.0, 10.0, test);

        // circle at 0,0 radius 10, and circle at 19,0, radius 10:  intersect
        test = intersects(0.0, 0.0, 10.0, 19.0, 0.0, 10.0);
        printf ("intersects(%f, %f, %f, %f, %f, %f): %f\n", 0.0, 0.0, 10.0, 19.0, 0.0, 10.0, test);
}
```

Functions and Scopes

- In addition to a type and a name, an identifier also has a *scope*. The scope of an identifier determines where the identifier can be referred to and the location of its storage.
  - A function is always defined in the *global scope*, and can be referenced and called from anywhere within the program.
  - Variables and constants that are defined outside of a function are also defined in the global scope, and can be referenced, accessed, and updated anywhere within the program.
  - Variables and constants that are defined within a function are defined within its local scope and can only be referenced, accessed, and updated within that function.

Functions and Scopes

Functions in the global scope

- A function declared in a C file can be accessed and called within the same file or from any other C file that is part of the same program.
- The visibility of a function can be limited to the C file where it is defined by preceding it with the *static* qualifier.

```
/** @file cfile1.c
 */
/** This function is visible only within this file*/
static void function1(void) {
    …
}
/** This function is visible everywhere in the global scope */
void function2(void) {
    function1();
}
```

Functions and Scopes

Variables and constants in the global scope

- Variables and constants declared outside of any function are in the global scope. They can be accessed within same file or from any other C file that is part of the same program.
- The lifetime of global variables and constants is the lifetime of the program.
- The visibility of a variable or constant can be limited to the C file where it is defined by preceding it with the *static* qualifier.

```
/** @file cfile2.c
*/

/** This variable is visible only within this file*/
static int variable1;

/** This variable is visible everywhere in the global scope */
int variable2;
```

Functions and Scopes

Variables and constants in the global scope

```
/** @file: initialize_global.c
 */
#include <stdio.h>                                  // for input/output operations

/** global transaction count */
unsigned int tranCount = 0;                         // available anywhere in program

/** Increment global counter */
void processTransaction(void) {
        tranCount++;                                // increment global transaction count
}

/** Demonstrates global variables.
*/
int main(void) {                                    // main function
        processTransaction();                       // process transaction
        printf("tranCount: %d\n", tranCount);       // print global transaction count
}
```

Functions and Scopes

- **Variables and constants in the global scope**
- Global variables can be initialized when declared, or within a function. Global constants must be initialized when declared.
- If a global is not initialized when declared, it is has an initial value of 0.
- If a global variable or constant is initialized when declared, a constant expression must be used.

```
/** initialize maxSize based to size of a double */
const unsigned int minSize = sizeof(double);          // 8 bytes

/** initialize curSize  based on minSize */
unsigned int curSize = minSize + 1;                   // 9 bytes

/** initialize maxSize based on curSize */
unsigned int nextSize = 2*curSize;                    // error! – not constant
```

Functions and Scopes

- **Variables and constants in a local scope**
- Variables and constants declared within a function are within the function's *local scope* and is only accessible within the function.
- The storage for a local variable or constant is within the function, and its lifetime is only during a call to the function.
- The storage for a local variable or constant can remain available across function calls by adding the keyword static to the variable or constant.
- A local variable or constant must be initialized within the function. Otherwise, its value is undefined.

Functions and Scopes

- Declaring and initializing local variables

```
/** @file: initialize_local.c
 */
#include <stdio.h>                                  // for input/output operations

 /** This function returns the next value on each call
  *  @return the next transaction count
  */
 unsigned int processTransaction(void) {
     static unsigned int transCount = 0;   // static storage within function
     return transCount++;                  // next transaction count on each call
 }
```

Functions and Scopes

- Declaring and initializing local variables

```
/**
 * Demonstrates local variables.
*/
int main(void) {                        // main function
        int tranCount;                                  // local to main
        tranCount = processTransaction();       // process first transaction
        printf("tranCount: %d\n", tranCount);   // print local transaction count

        tranCount = processTransaction();       // process second transaction
        printf("tranCount: %d\n", tranCount);   // print local transaction count

         = processTransaction();        // process third transaction
        printf("tranCount: %d\n", tranCount);   // print local transaction count
}
```