Functions, Separate Compilation, Macros

Review

- Introduction to C
  - C History
  - Compiling C
  - Identifiers
  - Variables
    - Declaration, Definition, Initialization
    - Variable Types
  - Logical Operators
  - Control Structures
    - i.e. loops
  - Functions

C Functions

- Have a:
  - Name
  - Return Type
  - Parameters
- Uniquely identified by name
  - No overloading
  - E.g. - this is invalid
    - int MyAddition(int a, intb);
    - char MyAddition(char a, charb);

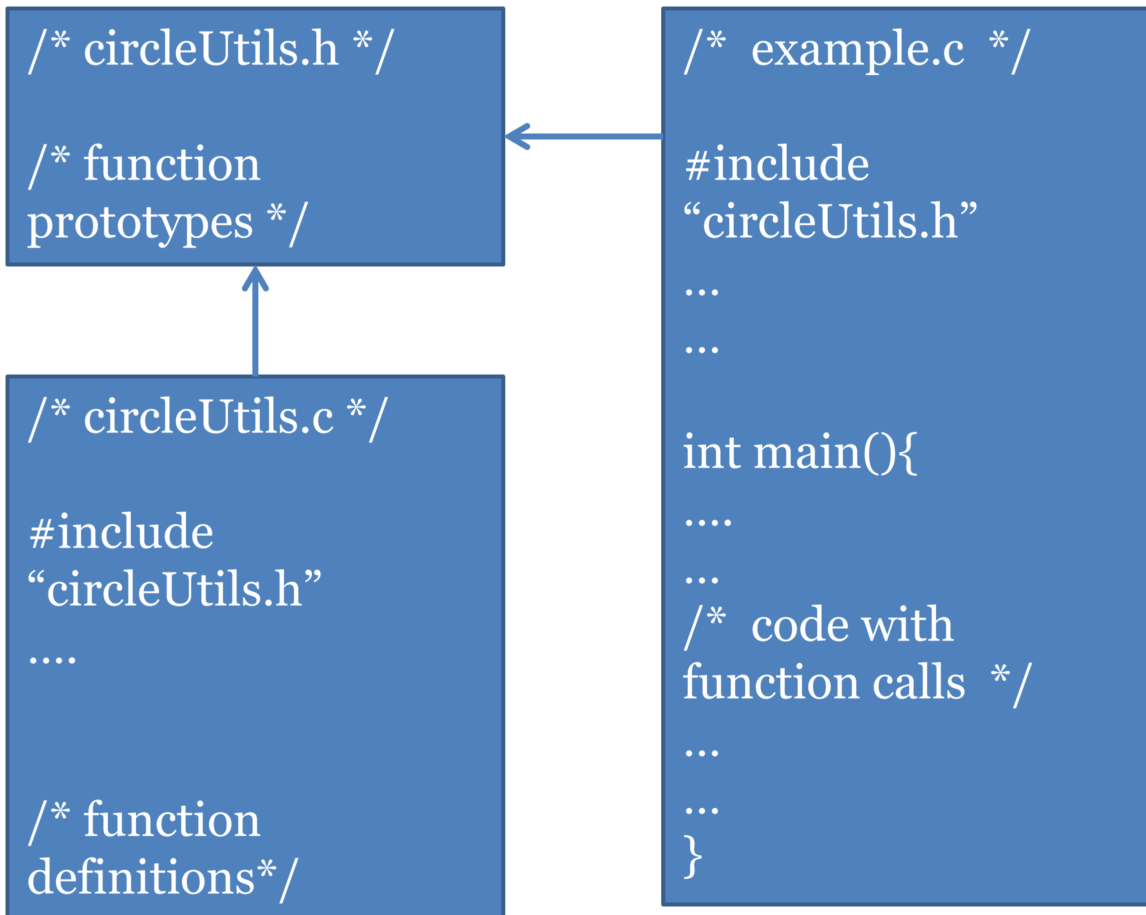
Function Arguments

```
int ArraySum(int array[], int size){
    int k, sum=0;
    for (k=0; k<size; k++)
        sum+= array[k];
    return sum;
}
int main(){
    int ages[6] = {19,18,17,22,44,66};
    int sumAge = ArraySum(ages,6);
    printf("The sum of ages is %d\n",sumAge);
    return 0;
}
```

Function Reuse

- Some functions are general functions that may be used by multiple applications
  - E.g. CircleArea, Circumference
- To make them available to multiple applications, must be placed in separate .c file
- Compiler requires that function prototype is provided
  - Place prototypes and supporting declarations in .h file
  - .h file included in .c files that wish to call the functions

Function Reuse



In example.c, the circle functions " CircleArea" and "Circumference" are to be used.

By including "circleUtils.h" the prototypes are referenced. The actual definition, then, is in the .c file

Header File

- A header file (.h) contains everything necessary to compile a .c file that includes it

```
/* circleUtils.h */
/* #includes required by the prototypes, if any */
/* supporting typedefs and #defines */
typedef double Radius;

/* function prototypes */
// given the radius, returns the area of a circle
double CircleArea( Radius radius );
// given the radius, calculates the circumference of a circle
double Circumference( Radius radius );
```

Header File

- Each .h file should be "stand alone"
  - It should declare, #define, and typedef anything needed by prototypes and include any .h files it needs to avoid compiler errors
- In our example prototypes for CircleArea() and Circumference are placed in circleUtils.h
  - circleUtils.h included in circleUtils.c
  - circleUtils.h included in any other .c file that uses CircleArea() or Circumference()

Guarding Header Files

- A .h file may include other .h files
  - Possibility that one or more .h files may included by a single .c file more than once
    - Compiler error – “multiple name definitions”
- To avoid errors .h files should be guarded
  - “#ifndef” and “#endif”
    - If not defined
- Other compiler directives
  - “#ifdef” – if defined
  - “#else”
  - “#elif” – else if

Guarding Example

```
#ifndef CIRCLEUTIL_H
#define CIRCLEUTIL_H
/* circleUtils.h */
/* include .h files as necessary */
/* supporting typedefs and #defines */
typedef double Radius;
/* function prototypes */
// given the radius, returns the area of a circle
double Area( Radius radius );
// given the radius, calcs the circumference of a circle
double Circumference( Radius radius );
#endif
```

Separate Compilation

- If code is separated into multiple .c files
  - Must compile each .c file
  - Combine resulting .o files to create executable
- Files may be compiled separately and then linked together
  - -c flag tells gcc to “compile only”
  - Creates .o files

Separate Compilation Example

```
gcc -c -Wall circleUtils.c   //creates .o file
gcc -c -Wall sample.c        //creates .o file
gcc -Wall -o sample sample.o circleutils.o
```

- OR if only a few files, compiling and linking can be done in one step
  - gcc -Wall -o sample sample.c circleUtils.c

Program Organization

- main is generally defined in own .c file
  - Calls helper functions
- Program-specific helper functions in another .c file
  - E.g. example1Utils.c
  - If there are very few helpers, they can be in the same file as main
- Reusable functions in own .c file
  - Group related functions in same file
- Prototypes, typedefs, #defines, for reusable function in .h file

Scope/Lifetime

- Variable “scope” refers to part of the program that may access the variable
  - Local, global, etc…
- Variable “lifetime” refers to time in which a variable occupies memory

- Both determined by how and where variable is defined

Global Variables

- Global (external) variables are defined outside of any function, typically near the top of .c file
  - May be used anywhere in the .c file in which they are defined
  - Exist for the duration of your program
  - May be used by any other .c file in your application that declares them as "extern" unless also defined as static
- Static global variables may only be used in .c file that declares them
  - "extern" declarations for global variables should be placed into a header file

Local Variables

- Defined within opening and closing braces of function, control-structure, etc…
  - Are usable only within the block in which they are defined
  - Exist only during the execution of the block unless also defined as static
  - Initialized variables are reinitialized each time the block is executed if not defined as static
- Static local variables retain their values for the duration of the program
  - Usually used in functions to retain values between calls to function
- Function parameters are local to the function

Static Variables

- Static variables are initialized to zero upon memory allocation
  - Good style to explicitly code it to make clear-zero initialization was intended
  - May initialize to other constants
  - Exception – pointers variables initialize to NULL

Static Example

```
int trackTillTen(){
    static int i = 1;
    if (i>10) return(1);
    i++;
    return(0);
}
int main(){
    int i = 0;
    while(i==0) i=trackTillTen();
}
```

Function Scope

- All functions are external
  - C does not allow nesting of function definitions
  - No “extern” declaration is needed
  - All functions may be called from any .c file in your program UNLESS they are also declared as static
- Static functions may only be used within .c file in which they are defined

- Exception: GNU C will allow nested helper functions inside other functions only usable inside that function. Not part of C standard – not portable

Recursion

- C functions may be called recursively
  - Typically called by itself
- A properly written recursive function has the following properties
  - A “base case” – a condition which does NOT make a recursive call because a simple solution exists
  - A recursive call with a condition (usually a parameter value) that is CLOSER to the base case than the current condition
- Each invocation of the function gets its own set of arguments and local variables

Recursion Example

```
/* print an integer in decimal
** K & R page 87 (may fail on largest negative int) */
#include <stdio.h>
void printd( int n ){
   if ( n < 0 ){
      printf( “-” );
      n = -n;
   }
   if ( n / 10 ) /* (n / 10 != 0) --more than 1 digit */
      printd( n / 10 ); /* recursive call: n has 1 less digit */
   printf( “%c”, n % 10 + ‘0’); /* base case ---1 digit */
}
```

Inline Functions

- C99 Only
- Short functions may be defined as “inline”
  - Suggestion to compiler that calls to the function should be replaced by body of the function
    - Suggestion, not a requirement
- Inline functions provide code the structure and readability advantages of using functions without overhead of actual function calls
  - i.e. inline bool isEven(int n);
- Generally, inline is more important in embedded environments than in other environments

Macros

- C provides macros as an alternative to small functions
  - More common prior to C99 (inline functions)
- Handled by preprocessor
- Inline functions are usually better
  - Some situations macros don't handle well
- Macro format
  - #define NAME(params (if any)) code here
    - Note: no space b/w name and left paren

Macro Example

- #define SQUARE(x) (x*x)
- Like all #defines, the preprocessor performs text substitution. Each occurrence of the parameter is replaced by argument text
  - int y=5; int z=SQUARE(y);
- NEVER FORGET THE ()
  - #define DOUBLE_IT(x) x+x
  - Will inevitably be called with
  - X =DOUBLE_IT(x)*3
    - Becomes x = x+x*3