C Functions

- C functions:
  - Have a **name** - the actual name of the function. The function name and the parameter list together constitute the function signature.
  - Have a **return type** - the data type of the value the function returns. If no value is to be returned, the return type is the keyword **void**.
  - May have **parameters** – value(s) that may be sent to the function. Parameters are also optional (and may also be set as such with the keyword **void**.)

- A function **declaration** tells the compiler about a function's name, return type, and parameters. It is at the top of program and tells the compiler about a function name and how to call the function. A function declaration is required when a function is defined in one source file and called in a different file. The actual body of the function can be defined separately from the function declaration.
  - Function Declaration and it's parts:

```
return_type function_name( parameter list );
```

  - Parameter names are not important in function declaration only their type is required

```
int max(int, int);
```

- A function **definition** provides the actual body of the function:

```
return_type function_name( parameter list )
{
   body of the function
}
```

C Functions

- Unlike Java methods, a function in C is uniquely identified by its name. Therefore, there is no concept of method overloading in C as there is in Java. There can be only one **main( )** function in a C application.

```
int MyAddition(int a, int b){
return (a+b);
}

int MyAddition(char a, char b){
return (a+b);
}

char MyAddition(int a, int b){
return ((char)(a+b));
}
```

You can only have one of these…

- UMBC coding standards dictate that function names begin with an UPPERCASE letter

Passing Arguments

- To call a function, you simply need to pass the required parameters along with function name, and if function returns a value, then you can store returned value. For example:

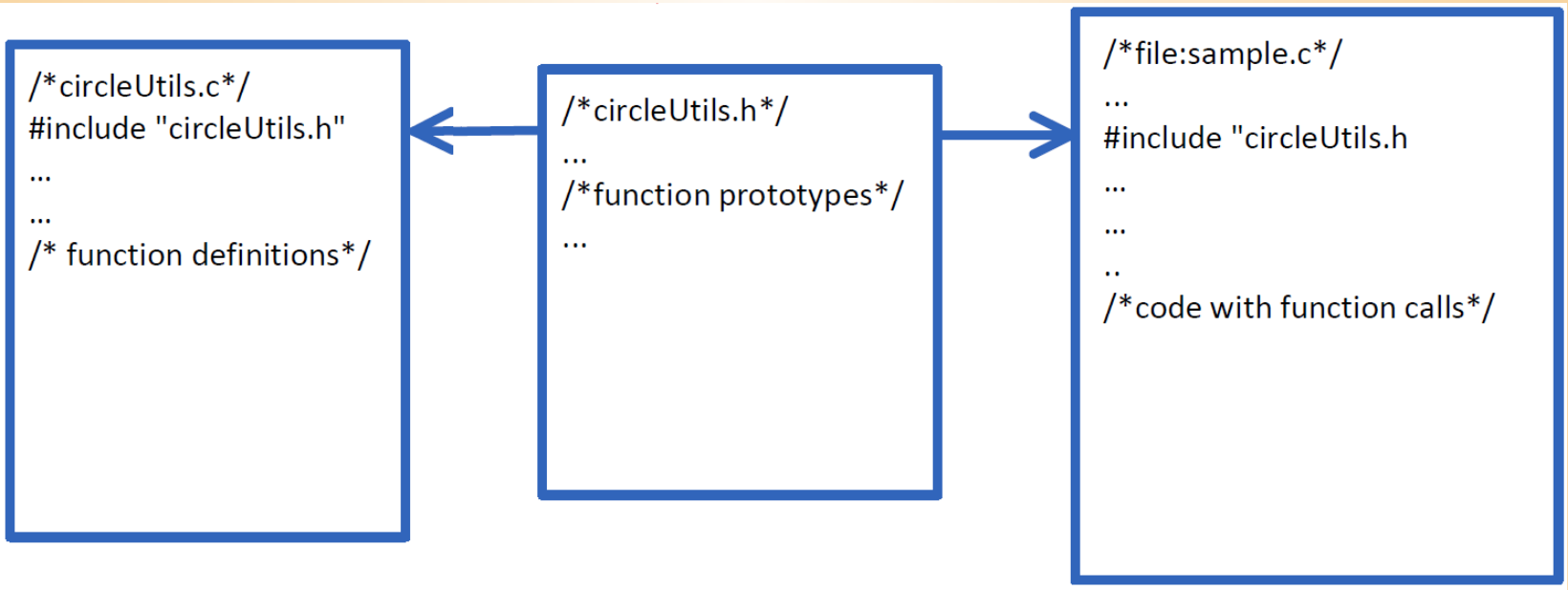
```
/* ages.c */

#include <stdio.h>

int ArraySum( int array[ ], int size)
{
  int k, sum = 0;
  for (k = 0; k < size; k++)
    sum += array[ k ];
  return sum;
}

int ArrayAvg( int array[ ], int size)
{
  double sum = ArraySum( array, size );
  return sum / size;
}

int main( )
{
int ages[ 6 ] = {19, 18, 17, 22, 44, 55};
  int avgAge = ArrayAvg( ages, 6 );
  printf("The average age is %d\n", ageSum);
  return 0;
}
```

A Simple C Program

```
/* sample.c */

#include <stdio.h>
typedef double Radius;
#define PI 3.1415

/* given the radius, calculates the area of a circle */
double CircleArea( Radius radius ){
  return ( PI * radius * radius );
}

// given the radius, calcs the circumference of a circle
double Circumference( Radius radius ){
  return (2 * PI * radius );
}

int main( )
{
  Radius radius = 4.5;
  double area = CircleArea( radius );
  double circumference = Circumference( radius );
  printf ("Area = %10.2f, Circumference = %10.2f\n", area,
    circumference);
  return 0;
}
```

Function Reuse – Header File introduction

- The functions CircleArea and Circumference are general functions that may be used by multiple applications.
- To make them available to multiple applications, we must place them into a separate .c file
- However, recall that the compiler requires that we must provide the function prototypes to the calling code. We do this by placing the prototypes and supporting declarations into a **header (.h) file** which is then included in .c files that wish to call the functions.



Header Files

- A header file is the **.h** file associated with a **.c** that typically defines a library of functions
- When a file contains functions to be reused in several programs, their prototypes and important **#defines** and **typedefs** are placed into a header (**.h**) that is then included where needed. If certain **#defines** and **typedefs** are only needed internally by the associated **.c** file and are not be used by other files, they can and arguably should be placed in the associated **.c** file.
- Each **.h** file should be "stand alone". That is, it should declare any **#define** and **typedef** needed by the prototypes and **#include** any **.h** files it needs to avoid compiler errors. The **.h** file should contain everything needed to successfully compile any **.c** file that includes it.
- In the following example, the prototypes for **CircleArea()** and **Circumference( )** are placed into the file **circleUtils.h** which would then be included in **circleUtils.c** and any other **.c** file that uses **CircleArea( )** and / or **Circumference( )**.

Function Reuse

- **circleUtils.c**

```
/* circleUtils.c
** Utilites for circle calculations
*/
#include "circleUtils.h"
#define PI 3.1415    // why not in the .h file??

/* given the radius, calculates the area of a circle */
double CircleArea( Radius radius )
{
   return ( PI * radius * radius );
}

// given the radius, calcs the circumference of a circle
double Circumference( Radius radius )
{
   return (2 * PI * radius );
}
```

- **circleUtils.h - A header (.h) file 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 Area( Radius radius );
// given the radius, calcs the circumference of a circle
double Circumference( Radius radius );
```

A Simple C Program - revisited

```
/* sample.c */

#include <stdio.h>
#include "circleUtils.h"  //now it's all defined in
                          //  the .h file
int main( )
{
  Radius radius = 4.5;
  double area = CircleArea( radius );
  double circumference = Circumference( radius );
  printf ("Area = %10.2f, Circumference = %10.2f\n",
    area, circumference);
  return 0;
}
```

Guarding Header Files

- Because a **.h** file may include other **.h** files, there is the possibility that one or more **.h** files may unintentionally be included in a single **.c** file more than once, leading to compiler errors (multiple name definitions).
- To avoid these errors, **.h** files should be "guarded" using the compiler directives **#ifndef** (read as "if not defined") and **#endif**
- Other compiler directives for conditional compilation include:
  - **#ifdef** - read as "if defined"
  - **#else**
  - **#elif** - read as "else if"
- Example:

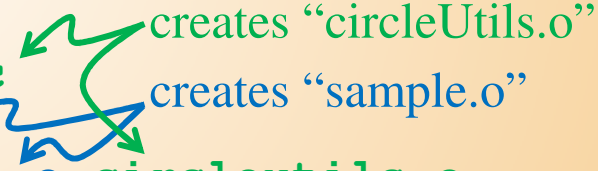
```
#ifndef CIRCLEUTIL_H
#define CIRCLEUTIL_H

/* 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
```

Compiling and Linking

- When a program's code is separated into multiple .c files, we must compile each .c file and then combine the resulting .o files to create an executable program.

- The files may be compiled separately and then linked together. The –c flag in the first two command tells gcc to "compile only" which results in the creation of .o(object) files. In the 3rd command, the presence of the .o extension tells gcc to link the files into an executable

```
gcc -c -Wall circleUtils.c
gcc -c -Wall sample.c
gcc -Wall -o sample sample.o circleutils.o
```



- Or if there only a few files, compiling and linking can be done all in one step

```
gcc -Wall -o sample sample.c circleUtils.c
```

Program Organization

- main( ) is generally defined in its own .c file and generally just calls helper functions
  - e.g. project1.c
- Program-specific helper functions in another .c file
  - e.g. proj1Utils.c
  - If there are very few helpers, they can be in the same file as main( )
- Reusable functions in their own .c file
  - Group related functions in the same file
  - e.g. circleUtils.c
- Prototypes, typedefs, #defines, etc. for reusable function in a .h file
  - Same file root name as the .c file. e.g. circleUtils.h

Variable Scope and Lifetime

- The **scope** of a variable refers to that part of a program that may refer to the variable. local, global, etc...
- The **lifetime** of a variable refers to the time in which a variable occupies a place in memory
- The scope and lifetime of a variable are determined by how and where the variable is defined
- There are three places where variables can be declared in C programming language:
  - Inside a function or a block which is called **local** variables,
  - Outside of all functions which is called **global** variables.
  - In the definition of function parameters which is called **formal** parameters.

Local Variables

- **Local** variables are defined within the opening and closing braces of a function, loop, if-statement, etc. (a code "block") Function parameters are local to the function.
  - 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 your program. Usually used in functions, they retain their values between calls to the function.

Global Variables

- **Global** (**external**) variables are defined outside of any function, typically near the top of a .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(see bullet immediately below…)
  - **Static** global variables may only be used in the .c file that declares them
  - "**extern**" declarations for global variables should be placed into a header file

Initialization of Static Variables

- Static variables are initialized to zero upon memory allocation.
- Still good idea / good style to explicitly code it to make it clear zero-initialization was intended
- May initialize to other constants too. (Pointers variables, which we'll learn about later, initialize to NULL)

```
int count (){
  static int i=0;
  i++;
  return(i);
}

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

Initialization of Static Variables

- All functions are external because C does not allow nesting of function definitions.
  - So 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 the .c file in which they are defined

- NOTE: some languages do allow defining "helper" functions that are only defined inside another function and can be used only in the parent function it is embedded in. While standard C doesn't support such helper functions, the GNU extensions of gcc will allow it, but its use is not wide-spread and code using these isn't portable across all C compilers.

Variable Scope – part 1

```
#include <stdio.h>

// extern definition of randomInt and prototype for getRandomInt

#include "randomInt.h"

/* a global variable that can only be used
by functions in this .c file */

static int inputValue;

/* a function that can only be called by other functions
in this .c file */

static void inputPositiveInt( char *prompt )
{
  /* init to invalid value to enter while loop */

  inputValue = -1;

  while (inputValue <= 0)
  {
    printf( "%s", prompt);
    scanf( "%d", &inputValue);
  }

}
```

Variable Scope – part 2

```
/* main is the entry point for all programs */

int main( )
{
  /* local/automatic variables that can only be used in
     this function and that are destroyed when the
     function ends */

  int i, maxValue, nrValues;

  inputPositiveInt("Input max random value to generate: ");

  maxValue = inputValue;
  inputPositiveInt("Input number of random ints to generate: ");

  nrValues = inputValue;

  for (i = 0; i < nrValues; i++)
  {
    getRandomInt( maxValue );
    printf( "%d: %d\n", i + 1, randomInt );
    return 0;
  }
}
```

randomint.c – global variable example:

```
/* a global variable to be used by code in other .c files.
** This variable exists until the program ends
** Other .c files must declare this variable as "extern"
** holds the random number that was generated */

int randomInt;

/* a function that can be called from any other function
** returns a random integer from 1 to max, inclusive */

void getRandomInt( int max )
{

  /* max is a local variable that may used within this function */
  /* lastRandom is a local variable that can only be used inside
     this function, but persists between calls to this function */

  static long lastRandom = 100001;

  lastRandom = (lastRandom * 125) % 2796203;
  randomInt = (lastRandom % max) + 1;

}
```

randomint.h (to go with randomint.c):

```
#ifndef RANDOMINT_H
#define RANDOMINT_H

// global variable in randomint.c
// set by calling getRandomInt( )

extern int randomInt;

// prototypes for function in randomInt.c

void getRandomInt(int max );

#endif
```

Recursion

- C functions may be called recursively.
  - Typically a function calls 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 condition (parameter value) of the current function call
- Each invocation of the function gets its own set of arguments and local variables

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**". This is a suggestion to the compiler that calls to the function should be replaced by the body of the function.
  - SUGGESTION TO COMPILER NOT A REQUIREMENT
- **inline** functions provide code the structure and readability advantages of using functions, but can avoid the overhead of actual function calls

```
inline bool isEven( int n )
  { return n % 2 == 0; }
inline max( int a, int b )
  { return a > b ? a : b; }
```

- Generally, inline is more important in embedded code than in other environments.

Macros

- C provides macros as an alternative to small functions.
- More common prior to C99 (which now provides inline functions)
- Handled by the preprocessor
- Several "gotcha"s
- Inline functions are usually better
- General macro format.
  - #define NAME(params if any) code here
    - Note: there is NO space between NAME and the left paren, **this is important!!!**

- A simple macro to square a variable

```
#define SQUARE( x ) (x * x)
```

- Like all #defines, the preprocessor performs text substitution. Each occurrence of the parameter is replaced by the argument text.

```
int y = 5;
int z = SQUARE( y );
```

- NEVER FORGET THE ()

```
#define DOUBLE_IT( x ) x + x
```

…somewhere in the code:

--> expands to `x=x+x*3` instead of `(x+x)*3`

- Consider this statement

```
int w = SQUARE( y + 1 );
```

--> `(y + 1 * y + 1)` which is not the same as $(y + 1)^2$

A better Macro for SQUARE():

- This version is better

```
#define SQUARE( x ) ( (x) * (x) )

int y = 5;
int z = SQUARE( y );
int w = SQUARE( y + 1 );
```

- But still doesn't work in this case:

```
int k = SQUARE( ++y );
```

--> expands to `((++y)*(++y))` which has two pre-increments

- So, be aware of what macros are, that they are just text replacement scripts. Don't abuse their use, misuse them, or carelessly think of them as functions.