# CMSC 313
# COMPUTER ORGANIZATION
# &
# ASSEMBLY LANGUAGE
# PROGRAMMING

LECTURE 12, FALL 2012

# TOPICS TODAY

- **Assembling & Linking Assembly Language**
- **Separate Compilation in C**
- **Scope and Lifetime**

# LINKING IN ASSEMBLY

# FUNCTIONS & SEPARATE COMPILATION IN C

# C Parameter Passing Notes

- We'll say *formal parameter* vs *actual parameter*.
  - Formal parameters are place holders in function definition.
  - Actual parameters (aka arguments) actually have a value.
- In C, all parameters are passed by value.
- Parameter passing by reference is simulated by passing the *address* of the variable.

```
scanf("%d", &n) ;
```

- Array names represent the address of the array. In effect, arrays are passed by reference.

```
int UpdateArray (int A[], int n) {
   A[0] += 5 ;
   ...
```

# A Simple C Program

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

double circleArea( Radius radius ) {
   return PI * radius * radius ;
}

double calcCircumference( Radius radius ) {
   return 2 * PI * radius ;
}

int main() {
   Radius radius = 4.5;
   double area = circleArea( radius );
   double circumference = calcCircumference( radius );

   printf ("Area = %10.2f, Circumference = %10.2f\n",
      area, circumference);

   return 0;
}
```

# Separate Compilation: Why?

- Keeps files small.
- Different people can work on different parts of the program.
- Easier to find functions.
- Keeps large program logically organized.
- Do not have to re-compile entire program when changes are made to a small portion.
- Parts of the program (e.g., code for a data structure) may be reusable in other programs.

**Problem: need a mechanism for external references.**

# circleUtils.h

```
/* circleUtils.h*/

/* #includes required by the prototypes, if any */


/* typedefs and #defines */

typedef double Radius;


/* function prototypes */

double circleArea( Radius radius );

double calcCircumference( Radius radius );
```

# circleUtils.c

```
/* circleUtils.c */

#include "circleUtils.h"
#define PI 3.1415

/* Function implementations */

double circleArea( Radius radius ) {

   return ( PI * radius * radius );
}

double calcCircumference( Radius radius ) {

   return (2 * PI * radius );
}
```

# main program

```
/* sample.c */
#include <stdio.h>
#include "circleUtils.h"

int main( ) {
   Radius radius = 4.5;
   double area, circumference ;

   area = circleArea( radius );
   circumference = calcCircumference( radius );

   printf ("Area = %lf, Circumference = %lf\n",
        area, circumference);

   return 0;
}
```

# Header Files

- Header files should contain
  - function prototypes
  - type definitions
  - #define constants
  - extern declarations for global variables
  - other #includes
- Header files should end with .h
- System header files #included with < >

```
#include <stdio.h>
```

- Your own header files #included with " "

```
#include "circleUtils.h"
```

- Header files are expected to include all other header files needed to work with implemented functions.

# Guarding Header Files

- Header files should not be included multiple times.
- multiple declaration of function prototypes: OK
- multiple type definition: BAD
- multiple `#include` can lead to loops where a `.h` file includes itself.

- Solution:

```
#ifndef _UNIQUE_VAR_NAME_
#define _UNIQUE_VAR_NAME_

...
#endif
```

# Guarded circleUtils.h

```
#ifndef CIRCLEUTIL_H
#define CIRCLEUTIL_H

/* circleUtils.h*/

/* #includes required by the prototypes, if any */

/* typedefs and #defines */

typedef double Radius;

/* function prototypes */

double circleArea( Radius radius );

double calcCircumference( Radius radius );

#endif
```

# Compiling and linking

- **How to compile:**

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

- **Or**

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

# Compiler vs linker

- Compiler: translates one .c file into a .o file
  - Verifies that all functions are being called correctly
  - Verifies that all variables exist
  - Verifies language syntax

- Linker: combines .o files and C libraries into executable file
  - "Finds" functions called by one .c/.o file, but defined in another E.g. printf( ), scanf( ).
  - "Finds" global variables used by one .c/.o file, but defined in another (more on this soon)

- `gcc` uses `ld` to link & load
  - Easier to invoke `ld` through `gcc`

# Linking with C libraries

- By default, the standard C library which includes printf, scanf and char and string functions is always linked with your program.

- Other libraries must be explicitly linked with your code.

- Typical C libraries have the form `libxxx.a`.
  - Standard C library: `libc.a`.
  - Math library:`libm.a`.

- Use the `-l` flag and the `xxx` part of the library name to link.

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

# Project Organization

- main( ) is generally defined in its own .c file and generally just calls helper functions
  - E.g. project1.c
- Project-specific helper functions may be in the same .c file as main( )
  - main( ) comes first
  - Helper function order that makes sense to you
- 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

# SCOPE & LIFETIME

# Variable Scope and Lifetime

- **The scope of a variable refers to that part of a program that may refer to the variable.**
- **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.**

# static and extern

- In C/C++, the keyword static is overloaded.
  - A static local variable has lifetime = duration of program.
  - A static global variable has file scope
  - A static function has file scope
- extern is means that the variable is defined in another file.

```
extern int other_variable ;
```

- an extern declaration is an example of a declaration that is not a definition. (Another example is a function prototype.)

# NEXT TIME

- **Pointers**