Welcome to Week 5!

This week you should ...

Required Activities

- Go through the various **videos, movies, and lecture notes** in the **Week 5 folder**.
- Read **Chapter 7** in the latest edition of the textbook (chapter 8 in earlier editions).
- Begin **Quiz 5**. It is due Sunday at midnight.
- Begin **Assignment 4**. It is due Sunday at midnight.

Recommended (optional) activities

- Attend **chat** this Thursday night, from 8:00 pm - 9:00 pm Eastern Time. Although chat participation is optional, it is highly recommended.
- Post any questions you might have on this week's topic in the **Week 5 Discussion Forum** located in the course **Discussion Board**. Please ask as many questions as needed, and don't hesitate to answer one-another's questions.
- Try out the various **code examples** in the lecture notes. Feel free to modify them and conduct your own **"What ifs"**.

"That's easy Larry, we have broken up our ship's software into a series of simple to complex functions that run everything from life support to weapon systems."

--- Montgomery Scoot
Chief Engineer, USS Enterprise
(Appearing on Larry Cling Live)

Functions

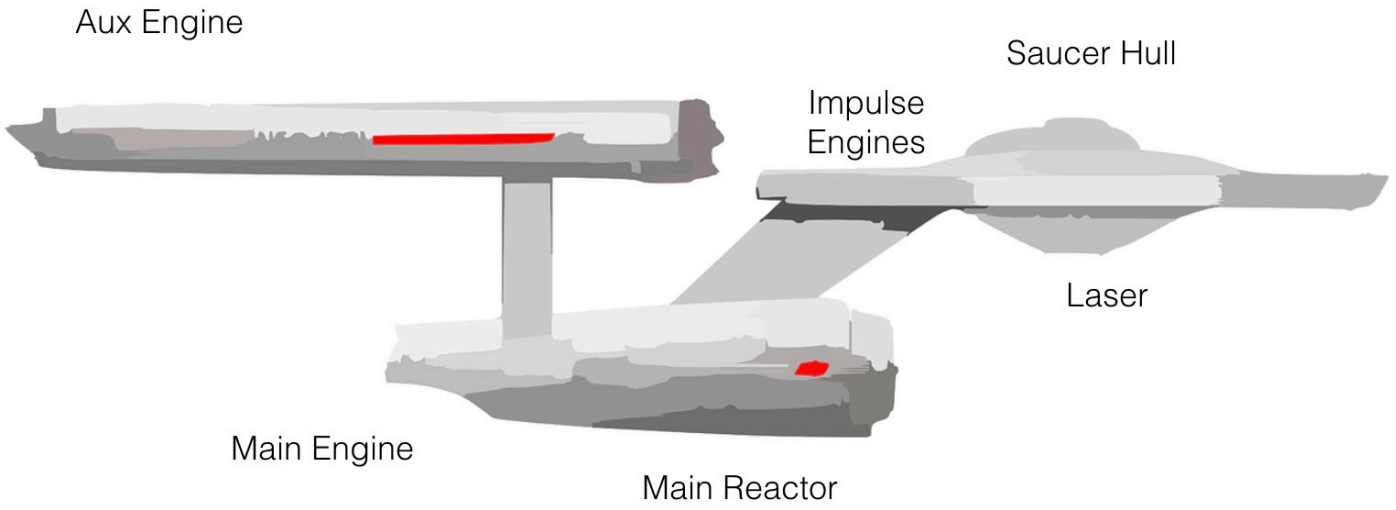
We have now reached the most important week in the class. It will be critical to understand that a C program is simply a bunch of functions calling other functions that starts with a **function called main**. If you are familiar with other languages, you might be used to other names, such as *subroutine*, *procedure*, and *module* ... they are all basically the same as a *function* in C.

There is a right way to design and implement functions in C, and there are many wrong ways. Don't be overwhelmed by vast information on functions in this week's chapter, there is a lot of information. You'll notice that there are many **videos** that are available this week. It can be difficult to learn functions just by reading a book and my lecture notes (as good as they are). The videos provide you with both carefully crafted animation and my words of wisdom that can really go a very long way in helping you navigate through the concept of functions. Read the book and watch the videos first, and then read carefully through the lecture notes to master your understanding of functions in C.

DO try out various examples (and there are many) presented in both the book and lecture notes on your compiler, and feel free to create your own functions. There is no substitute for "hands on" experience, and making mistakes and correctly them is a best way to learn!

Functions - Star Trekking Style

Let's review an example of a functional based concept by looking at a public domain image via *Pixabay.com* depicting the fictional USS Enterprise from the original Star Trek series. I've annotated the image to show some of the various functional components that comprise the ship.



Source: *Pixabay.com (public domain image)*

I learned of various functional components of such a ship through a review of a web site called "*Build the*

Enterprise", that was authored by an actual systems engineer who has proposed real life plans for building such as ship in the next twenty years. Complete details can be found at:
http://www.buildtheenterprise.org if you are interested, it is quite a web site!

Star Trekking - How the Ship Functions

To introduce you to concept of **functions**, let's look in on an interview with talk-master **Larry Cling**, as he hosts his show "Larry Cling Live" on the USS Enterprise. In this segment, he interviews Chief Engineer Mr. Scoot, a.k.a. "**Scootty**", on how a series of functions are used to run all the systems on the ship. Click the image below or go back to the link on this week's class page to watch this informative two minute **movie**.

Video - Functions in a Nutshell

Take a look at the concept of a function at the 1000 foot level before we get into the programming details with the C Programming language. This video will show you real world examples of how functions are used in many applications that we use today. Learn why functions are needed and why they are an important way to design and implement programs.

Introduction

The word **function** comes from the **Latin** word *functio*, which means activity or performance. In C, it is best described as:

- **An independent set of statements to accomplish one nameable task**

There are functions that C provides in its library, such as **printf** and **scanf**, as well as functions that you and other team members will need to develop to integrate into your programs.

A **program** is simply one or more functions where execution begins with a function called **main**.

You will find that the C language itself has very limited operators and functionality *built into the language*. Remember, its built for speed. Instead, it relies on many available functions that are stored in the **C runtime library** to perform various needed activities. These libraries are automatically linked into program after a successful compile. Using them correctly requires the right header file (we'll learn more about these in Chapter 13) to be included. We've already used the standard input output library functions like printf and scanf and have included the header file stdio.h. Some common ones include:

- math.h - math functions (trig, log, square root)
- time.h - converting various time and date formats
- string.h - string handling functions (copy strings, compare, etc)

While we will cover many of the libraries over the semester, we just don't have time to cover them all. There are many books out there that provide details and examples of each C library function, but for a complete listing of all available libraries in the C language, the Wikipedia web site is a great start:

- http://en.wikipedia.org/wiki/C_standard_library

The books I've personally used to learn initially about each library function were "Topics in C Programming" by Stephen Kochan and the "C Pocket Reference". As I said before, there are many reference books and web sites that go into detail about each C library function. We'll cover a good majority of them in this class over the semester.

Benefits of using Functions

There are many benefits to using functions. Often a new programmer will just write one large chunk of code and be satisfied that it works correctly, no realizing how inefficient and difficult their program will be to understand and maintain into the future. Here are just a few things to keep in mind about why you should use functions as you read deeper into these lecture notes.

1. ***Improves Programming Productivity*** - Work can be divided among project

members to allow for design and coding to be completed in parallel.

2. ***Simplifies the Testing Process*** - Easy to locate and isolate a faulty function for further investigation.
3. ***Facilitates SW Reuse*** - A function may be used later by many other programs; this means that a C programmer can use a function written by others, instead of starting over from scratch.
4. ***Eliminates Duplication of Effort*** - A function can be used to eliminate the rewriting of the same block of code from going into multiple locations within a program.
5. ***Functions Support Abstraction*** - Once a function is written, it serves as a black box. All that any programmer would have to know to invoke a function would be to know its name, and the input (called parameters in C) that it expects.

Design Concepts

The following **design concepts** was taken from *STRUCTURAL DESIGN,* IBM Systems Journal, No. 2, 1974, by Stevens, Meyers, and Constantine.

COHESIVENESS: Why code statements are together:

bad

coincidental

better

logical
temporal
communicational
sequential

best

functional

- **COINCIDENTAL** means no apparent relationship
- **LOGICAL** means some logical relationship between element of module
- **TEMPORAL** means time related, as in things done at start of module
- **COMMUNICATIONAL** means reference to same Input or Output
- **SEQUENTIAL** means the actions are not related but happen in sequence

The **best reason** for code statements to be together in a **function** is for the purpose of accomplishing one nameable activity (such as "calculate a square root"). This would be referred to in this class as **functional**.

Some Key Terms:

COUPLING: How two modules or functions interact with one another.

COMPLEX INTERFACE: Bad! Either function alters internal code or data in another function.

SIMPLE INTERFACE: Better - what happens in one function cannot damage another function.

TIP:

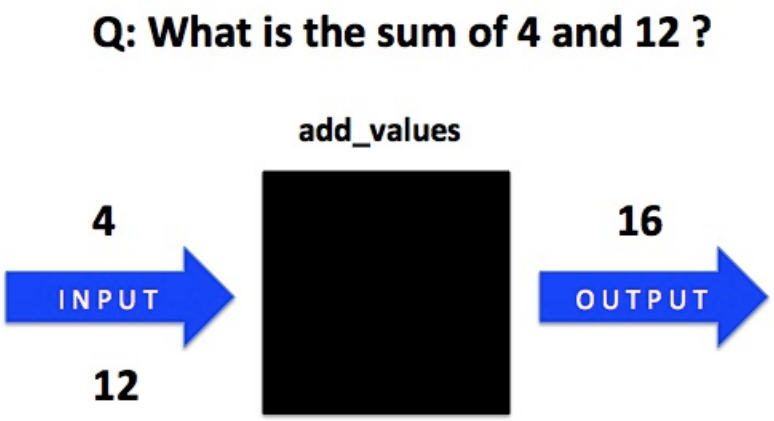
A called function should be designed so that it is **self contained**, able to resist errors made by the calling module.

Video - Function Overview

In C, a program is really just a bunch of functions calling other functions that starts execution with a function called "main". This video introduces you to how functions are designed, how they are managed within memory, and more importantly, how to pass values back and forth between functions.

General Function Syntax

What if you were asked to design a function that would return the sum of two integer values. An initial design might be something that looks like this **Black Box** type approach where there are **two inputs** fed into it and it **outputs** the sum of those two inputs:



The basic syntax and usage of a function called **add_values** is shown below along with how it would be called from another function (in this case, the main function). It is **passed two integer values** and it **returns** the **sum** of those two integer values back to the **calling function** (in this, main). The benefit of this function is you can now call it with **ANY two integer values** (not just 4 and 12) in order to figure out their sum. Try it out at: http://ideone.com/So2BTF

```
#include <stdio.h>

// ************************************************************************
// Function:  add_values
//
// Description:  Adds two integer numbers together and returns
//                  the sum
//
// Parameters:  number1 - an integer value to be summed
//                   number2 - another integer value to be summed
//
// Returns:        result - the sumer of number1 and number2
//
// ************************************************************************
int add_values (int number1, int number2)    /* two parameters */
{    /* start of function body */

    int result;  /* local variable to hold add result */

    result = number1 + number2;
```

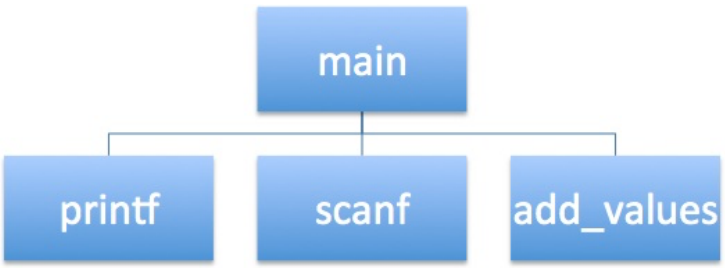
```
    return (result);   /* to calling function ... main  */

}   /* end of function body */

main ( )
{

   int value1;     /* local variable */
   int value2;     /* another local variable */
   int answer;     /* and yet another local variable */

   printf ("\n Enter an integer value: ");
   scanf ("%i", &value1);

   printf ("\n Enter another integer value: ");
   scanf ("%i", &value2);

   answer = add_values (value1, value2);   /* pass two arguments */

   printf ("\n Adding %i to %i equals %i", value1, value2, answer);

   return (0);

}
```

It would print the following **output**:

```
Enter an integer value: 4
Enter another integer value: 12
Adding 4 to 12 equals 16
```

In general, there are two types of functions in terms of how they are called. There is a **calling function** that calls the function to be executed, which is known as the **called function**.

In the above program, the calling function is **main**, it calls two **library functions**, **printf** and **scanf**, and a function that has been created by a programmer called **add_values**. A **hierarchy** of the functions would look like this:



In each function there are two variable types: **parameters** and **local variables**. Both have a scope that is only applicable to the function that they belong, meaning, if you

reference number1, number2, or result inside the main function code, it won't know what you are referring to. Likewise, the add_values function knows nothing about value1, value2, or answer.

Each function can have a set of parameters that can accept arguments passed from a function. If a function has **three parameters**, it will expect that the calling function is passing **three corresponding arguments**. The add_values function has **two parameters**, number1 and number2.

```
int add_values (int number1, int number2)
```

When it is called in the main function, it is passed two **arguments**, value1 and value2, as shown below:

```
add_values (value1, value2);
```

After it is called, the function will **return a value** which is the **sum of the parameters value1 and value2** into another **local variable** called **answer**. For example, if value1 is 5 and value2 is 10, then the function will return a value of 15 (5 + 10) into the local variable answer.

The table below shows all the local variables and parameters associated with each of the functions in the above program (note that in this case, main does not have any parameters):

Function	Variable Type	Name	Data Type
add_values	parameter	number1	integer
add_values	parameter	number2	integer
add_values	local variable	result	integer
main	local variable	value1	integer
main	local variable	value2	integer
main	local variable	answer	integer

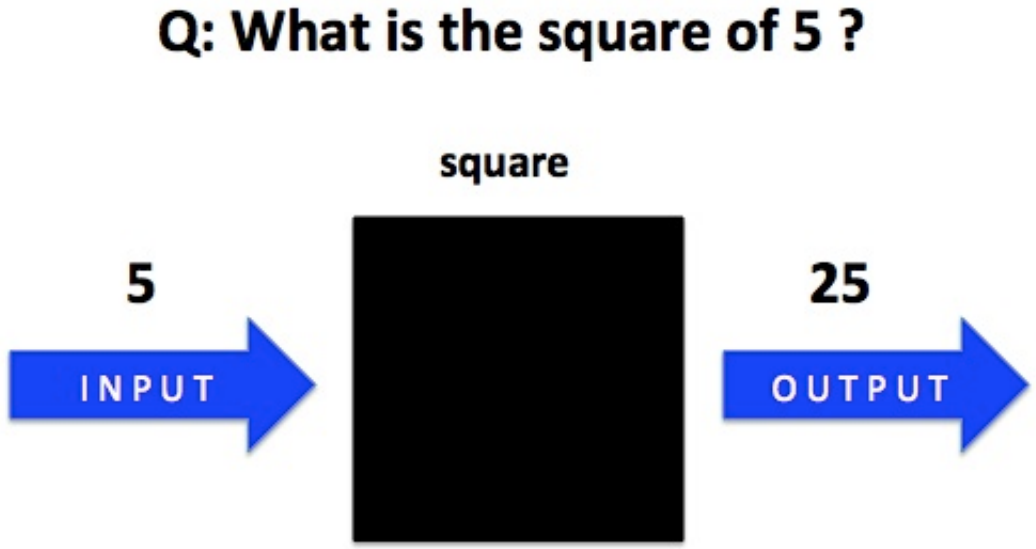
On the next set of lecture sides, let's look into more on how functions are passed values and how a value is returned back to the calling function.

Function Return Type

In C, functions normally return a value back to the function that called it. What C needs to know for each function is what TYPE of value is it returning ... an integer, a float, an address, or maybe nothing. Consider the syntax shown below. The **default return type** for a function in C is **int**. Even if no return type is in the code, C will assume that it is an "int".

One important fact about functions in C is that a function can ***only return at most ONE value***. Saying that, you are probably asking yourself, but Tim, what if I want to return more than one value? While it is true that a function can only return at most one value, we'll see that when we cover **pointers,** that you'll be able to use them to reference arrays and other containers that will allow you to get access to multiple data items. As far as this week, I'll show you how to work and manipulate multiple values inside an array that is passed to a function.

A **square** is simply a number multiplied by itself. For example, the square of 5 is ***5 x 5*** or 25. You can envision a **design** for such a function working like this:



Below is a sample function in C that is passed a number and it returns its square.

Watch it work at: http://ideone.com/XKzQ5H

```
#include <stdio.h>

// ********************************************************
// Function: square
//
// Description: Takes a floating point number and returns
//              its square
//
// Parameters: number - a floating point number to square
//
// Returns:    square_value - the square of the number
//
// ********************************************************
float square (float number)
{
    float square_value; /* number squared */

    /* square the number and store it in square_value */
    square_value = number * number;

    return (square_value); /* return to calling function */
}

int main ()
{

    float value1; /* the number to be squared */

    float answer; /* holds the square value */

    /* Prompt the user for a number to be squared */
    printf ("\n Enter a number: ");
    scanf ("%f", &value1);

    /* Pass value1 to the square function, process it and return the */
```

```
        /* the squared value into the answer local variable */
        answer = square ( value1 );

        printf ("\n The square of %5.2f is %8.2f \n", value1, answer);

        return (0);
}
```

Output:

```
Enter a number:  5

The square of 5.00 is 25.00
```

The default type returned is **int**, but it is better programming to be explicit.

Let's take a more in depth look at this statement used in the program above:

```
answer = square ( value1 );
```

When you call the *square* function and pass it a value, as in:

```
square (value1);
```

It will call the **square function** and pass it the contents of **value1**. Let's say if *value1* contains 10.0, then 10.0 is passed to the function *square*.

The function square now becomes active, and that 10.0 value is passed into the parameter named **number** in the *square* function:

```
float square (float number)
```

Inside the *square* function, the number is multiplied by itself and stored in **square_value**, which is a local variable known only to the function *square*:

```
square_value = number * number;
```

Both parameters and local variables are stored in memory locations that are available to each function. In the *square*

function, at this point, the memory locations would look like the table below if the *square* function was passed a value of 10.0

Variable Type	Name	Contents
Parameter	number	10.0
Local Variable	square_value	100.0

The **return** statement is now key here, as it provides a way for the function to return the value stored in square_value to the function is was called from, in this case, **main**. Once the *return* statement is executed, control returns back to the calling function. A couple things about the return statement:

1) It can only return one value, you can't say for example: **return** (value1, value2);
2) You can have multiple returns in a function, but its best if you can design a function with one exit point, using one return statement.

In our example where 10.0 was passed to the square function, a value of 100.0 would now be in the local variable *square_value*, and 100.0 would be the value that would be passed back to the *main* function.

```
return (square_value);
```

We now come full circle and come back to where this all started, with the call in the *main* function to the called function named *square*. With execution returning back to the calling function, in this case *main*, the function *square* returns a value of 100 and that value is assigned into the variable "answer" below:

```
answer = square (value1);
```

This type of function call is known as **Call by Value**, as only a value is passed back, no change is made by the called function to argument, *value1*, which is a local variable in the *main* function.

While our example showed how you could return a value from a function and assign it into a variable, you can use the return value in many different ways. The table shows a few ways you might call the square function and utilize its returned value. Please review this and make sure you take the time to understand this concept, it is probably the most important lesson you'll learn this week!

Statement	Description	If square (value1) returns 10.0, its like saying:
value1 = **square** (value1);	Passes the contents of value1 to the **square** function, AND then value1 will be assigned the squared value upon the return.	value1 = 10.0;
answer = **square** (value1);	Passes the contents of value1 to the **square** function, AND then answer will be assigned the squared value upon the return.	answer = 10.0
if (**answer = square (value1)**)	Function will return result into the local variable answer, and will evaluate the contents of answer to see if it is TRUE (not 0) or FALSE (0).	answer = 10.0; if (10.0)
if (**square (value1**))	Function will return a result which will either be a True (non-Zero) or False (zero) value, no variables updated.	if (10.0)
printf ("%f", **square (value1)**);	Function will return a result which will be used as an argument in the printf function, no variables updated.	printf ("%f", 10.0);
x = square (value1) + 12;	Function will return a result which will be used as part of an expression that will be assigned to the variable x, no other variables updated.	x = 10.0 + 12;
square (value1);	Passes the contents of value1 to the function **square**, but when **square** is finished executing, the value being returned will have no use in the statement above, it will just be ignored, and no variables will be updated.	nothing

Default Return Type and type VOID

C also has a return type of **void**. This return is used when a function does not return any value. A good example are functions who sole job is to print information that is passed to it. Using our square function from above, we could have

a function that passes a number, and the squared value of that number to a function that prints both in a nice table format:

```
void printSquare (float value, float squareVal)
{
    printf ("A Number \t The Number Squared \n");

    printf ("%5.2f \t %8.2f \n", value, squareVal);

    return; /* return statement is optional, but no value returned */
}
```

Output

```
A Number    The Number Squared
10.0            100.0
```

If the calling program tried to call the printSquare function and expected a return value such as:

```
value = printSquare (value1, answer);
```

... the compiler would flag this as an error, because the printSquare function was declared as type **void.**

Returning Function Results

In Summary, the return is necessary because

If you alter a parameter in a CALLED FUNCTION, it has no effect on the variable in the CALLING FUNCTION.

The CALLING FUNCTION passes a copy of the argument, NOT the real argument.

This is referred to as "CALL BY VALUE"

Video - Function Design Guidelines

This video will show you the right way to start developing your C Program using functions. Proper design guidelines will be discussed, most notably how to design using a top down approach. Also discussed is how to develop and implement function prototypes.

Top-Down Programming

INSIDE EVERY LARGE PROGRAM THERE ARE MANY SMALL PROGRAMS

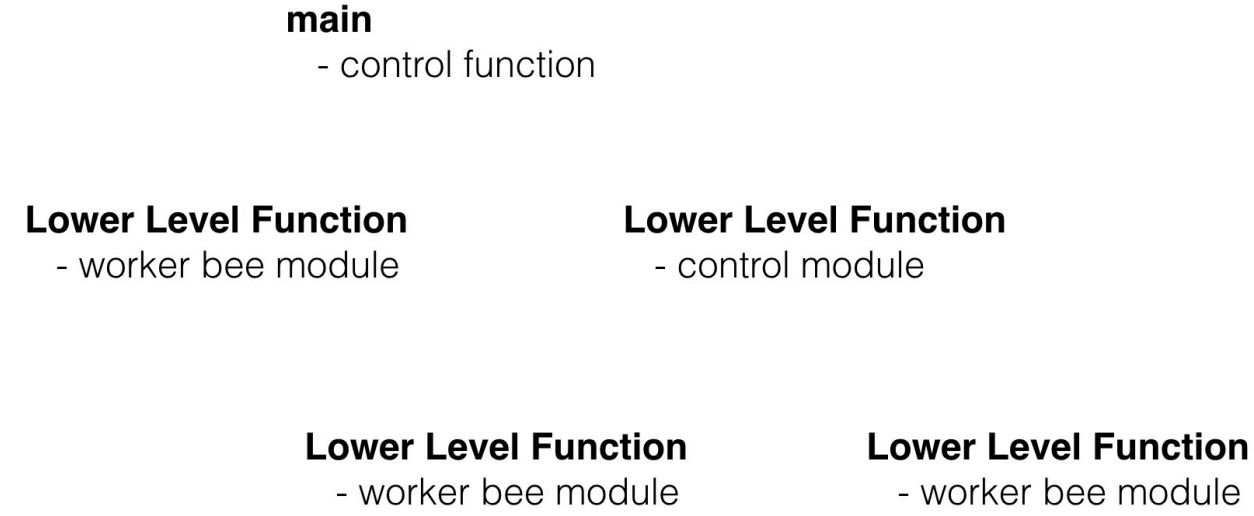
- Divide and Conquer

USE FUNCTIONAL BUILDING BLOCKS

- Abstraction and control functions are at the top
- Worker bee functions are at the bottom

WRITE SMALL INDEPENDENT FUNCTIONS

- that perform one well defined task
- that may be re-used



Inside every large problem are several small problems. Divide and Conquer, solving one problem at a time.

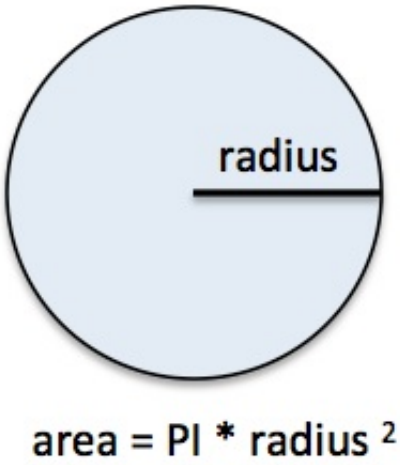
Consider your algorithm carefully before you write code. Try to identify low level functions that can be reused. Write the high level control functions with the low level functions in mind. Then write and test the low level functions, one by one.

Example: if you wish to compare two times, each consisting of hours, minutes, and seconds - convert each to seconds, compare, then convert the difference back.

Functions Calling Functions

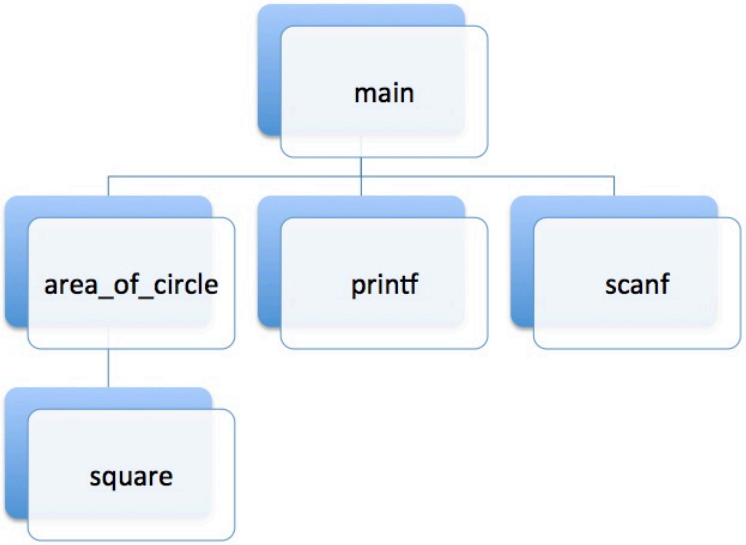
In C, it is all about functions calling functions that starts with a function called **main**. Every C program, at a minimum, must have at least one function, the **main** function. It will be the first function called and it will be active through the life of the entire program, and it will the last function executing when the program completes. It all starts and ends with the main function.

Let's look at a simple mathematical problem such as determining the **area of a circle**. To solve it, you must know the ***radius*** of a circle, which you can then square and multiply by ***PI*** (which is approximately 3.1415927) to figure out the circle's area. If you want to better understand how and why this works, optionally check out this link: http://www.mathsisfun.com/geometry/circle-area.html.



Shown below is an example of three functions in a program, along with two library functions, **printf** and **scanf**. The **square** function returns the square of a given number. The **area_of_circle** function returns the area of a circle given a radius. Note that it needs to call the **square** function in its calculation of an area. The **main** function calls the **area_of_circle** function and passes it the radius value that was prompted for in the main function. Note how all the functions work together to accomplish a set of tasks. The main function completes the cycle by printing a nice output of the circle's area.

An illustration of the function hierarchy, along with the actual code and output (given a sample input value of 5.6 for a radius) is shown below. Feel free to try it out in IDEOne at: `http://ideone.com/HxKgB`




```
#include <stdio.h>
#define PI 3.14      /* define the value of PI as a constant */

// ********************************************************
//
// Function: square
//
// Description:  Squares a given number and returns it
//
// Parameter:  number - number to be squared
//
// Returns:    square_value - the number squared
//
// ********************************************************
float square   (float number)
{
    float square_value;  /* value to square */

    square_value = number * number;  /* square the number */
```

```
    return (square_value);

}


// ********************************************************
//
// Function: area_of_circle
//
// Description:  Calculates the Area of a circle given the
//               its Radius
//
// Parameter:  radius - The radius of the circle
//
// Returns:    area - Calculated area of the circle
//
// ********************************************************

float area_of_circle (float radius)
{

    float area;   /* area of a circle */

    /* compute area of a circle: PI * radius squared */
    area  = PI * square (radius);

    return (area);
}

int main ()
{

    float area;     /* the area of the circle */
    float radius;   /* radius of a circle to be entered */

    printf ("Enter the circle radius: ");
    scanf ("%f", &radius);
```

```
    /* Pass value1 to the square function, process it and return the */
    /* the squared value into the answer local variable */
    area = area_of_circle (radius);

    printf ("\nThe Area of a Circle with a radius of %5.2f is %8.2f \n",
            radius, area);

    return (0);

}
```

OUTPUT:

```
Enter a circle radius: 5.6
The Area of a Circle with a radius of 5.60 is 98.47
```

Function Prototypes

A C Program can be characterized as functions calling other functions that starts with a function called main. In traditional C, the compiler needs to know the function name and return type if it encounters a function call before the function is specified. The compiler will start at the top of the file and work its way down. ANSI C adds a more complete **function prototype** with the type of each parameter.

```
int add_two (int, int);
int double_it (int);
```

You could even put the **parameter names** in the prototypes to be more descriptive, but at a minimum, it will look for the function name, its return type, and the types of each parameter. The rest is ignored. Also, they don't even have to be the same names as the parameter in the actual function code ... it is really just ignored by the compiler anyway. The benefit of the approach below is that you can use **descriptive parameter names** that will aid in **readability**.

```
int add_two (int firstNum, int secondNum);
int double_it (int theNumber);
```

If a function appears in a program before it is called, the compiler will need to know what type it returns. For example, **main** may be the first function in a program. This is **Method 1**.

You can explicitly declare the function outside all functions (**Method 2a**) or inside a function before it is called (**Method 2b**). Some function design guidelines:

- Each function should be written so that it is invulnerable to errors made by the calling function.
- A **return** statement can appear more than once in a function, but it is not the best design - try to use just **one return statement** in a function.

- A function should never depend of the intelligence of the programmer who coded the calling function.

```
/* https://www.ideone.com/Fm2wDi */

#include <stdio.h>
```

```c
/* global definition of a function prototype */
/* known to all functions                    */
int add_two (int, int); /* Method 2a */

int main ()
{

  int doubled;  /* the result of a number doubled */
  int sum;      /* the sum of two numbers         */
  int val;      /* just a number to work with     */

  val = 100;

  /* local declaration of a function prototype      */
  /* known only to calls to it in the main function */
  int double_it (int); /* Method 2b */

  sum = add_two (val, val*5);
  doubled = double_it (val);

  printf("sum = %i and doubled = %i\n", sum, doubled);

  return(0);

}

// ************************************************
// Function: add_two
//
// Description:  Adds to numbers together and returns
//               their sum.
//
// Parameters: num1 - first integer to sum
//             num2 - second integer to sum
//
// Returns:    result - sum of num1 and num2
//
// ************************************************
```

```
int add_two (int num1, int num2)
{
   int result;  /* sum of the two number */

   result = num1 + num2;
   return (result);
}

// ************************************************
// Function: double_it
//
// Description:  Adds to numbers together and returns
//               their sum.
//
// Parameters: num - number to double
//
// Returns:    answer - the value of num doubled
//
// ************************************************
int double_it (int num)
{
   int answer; /* num being doubled */

   answer = num + num;

   return (answer);
}
```

With **Method 1,** all functions are placed after main in a file assuming it is main that is calling each of these functions. While this works, it best to have a function prototype for each function (other than main) and then either have the *main* function *first or last* so you can easily find it.

The other reason is that it is very possible that you might have one non-main function that calls another function. Putting all your function prototypes up front allows you to order your functions anywhere in a file, or later on, when you might have each function in its own separate file (see Chapter 15). I find that this is the **best method**.

Try it out at: https://ideone.com/HTCFmz

```
#include <stdio.h>

/* global definition of a function prototype */
/* known to all functions                    */
int add_two (int, int);
int double_it (int);

int main ()
{

  int doubled;  /* the result of a number doubled */
  int sum;      /* the sum of two numbers         */
  int val;      /* just a number to work with     */

  val = 100;

  /* call the functions */
  /* The function prototypes before help the compiler to             */
  /* determine the types each function is passed and what it returns */
  sum = add_two (val, val*5);
  doubled = double_it (val);

  printf("sum = %i and doubled = %i\n", sum, doubled);

  return(0);

}



// ************************************************
// Function: add_two
//
// Description:  Adds to numbers together and returns
//               their sum.
//
// Parameters: num1 - first integer to sum
```

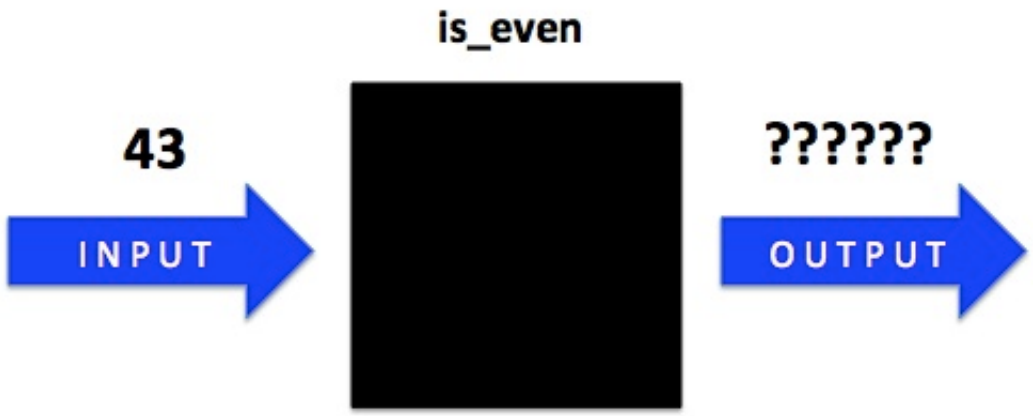
```
//              num2 - second integer to sum
//
// Returns:     result - sum of num1 and num2
//
// ************************************************

int add_two (int num1, int num2)
{

   int result;  /* sum of the two number */

   result = num1 + num2;

   return (result);

}

// ************************************************
// Function: double_it
//
// Description:  Adds to numbers together and returns
//               their sum.
//
// Parameters: num - number to double
//
// Returns:     answer - the value of num doubled
//
// ************************************************
int double_it (int num)
{

   int answer; /* num being doubled */

   answer = num + num;

   return (answer);
}
```

Good/Bad Functions

Let's look at an example of two ways to write a function to check to see if a given **integer** is an **even number**. A good name for such a function is **is_even**. A number is **even** if you divide two integer numbers and get **no remainder**. The ***modulus (mod) function*** in C (the % operator) will do this for you. Take one integer number and divide it with another ... if you have ***no remainder*** (i.e., 0), then the number is **even**, otherwise it is **odd**. An integer number can only be even or odd, it has to be one of the two. Some important rules to keep in mind when designing functions:

- Always **pass** the **function result** back to the **calling function** whenever possible
- Let the **calling function *decide*** what to do with the **result** (does it want to print, assign to a variable, pass it as an argument to another function, ...)
- Make the function as **usable** as possible (let it work in many ways, don't limit it)

Let's look at the contrasting code of two functions to determine if a number is odd or even. The first one, Function 1, is a **bad** design in that it **does not return a result** back to the calling function, instead, it decides on its own to just print the values to the screen regardless if that is the intention of the **calling function**. You could envision it with a black box approach as:




```
/* Function 1 - BAD */
int is_even ( int number )
{
    /* this will ALWAYS print */
    if ( number % 2 == 0 )
        printf ("number is even \n");
    else
        printf ("number is odd \n");
}
```

Contrast that with the Function 2 below, which is a much better design in that it returns a TRUE (1) or FALSE (0) value back to the calling function to let it decide what to do with the result. If the function returns a TRUE value, then the number is **Even** (after all, look at the function name: **is_even**), otherwise it returns a FALSE value to indicate

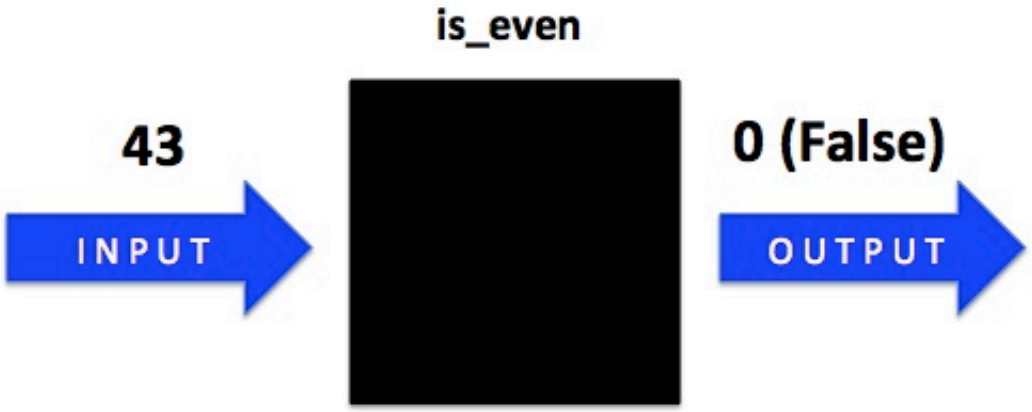
that the number is **Odd**.

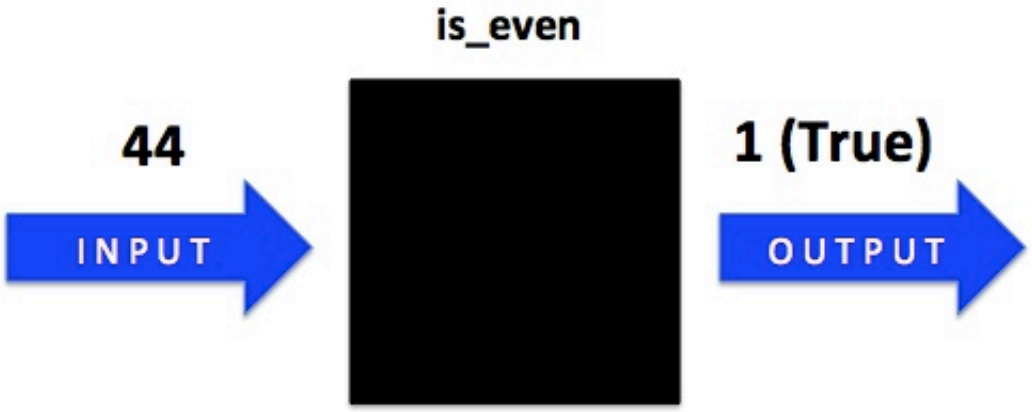
```
/* Function 2 - BETTER */
int is_even ( int number )
{
    /* return TRUE if even, otherwise FALSE for odd */
    if ( number % 2 == 0 )
        return ( 1 ); /* its even */
    else
        return ( 0 ); /* its odd */
}
```

The only improvement I might make to Function 2 above is to **return only once**. It makes your function SO MUCH easier to test and validate (using my best Software Engineer voice). In Function 3 below, I added a **flag** that can be set to True or False (i.e., 1 or 0) to indicate if the number is odd or even. I simply return the value of that flag at the end with my only return statement in my function. You can watch my function work at: http://ideone.com/TdATV3

```
/* Function 3 - BEST */
int is_even ( int number )
{
    int evenFlag;   /* flag to indicate odd or even */

    if ( number % 2 == 0 )
        evenFlag = 1; /* its even */
    else
        evenFlag = 0; /* its odd */

    return (evenFlag);  /* 0 or 1, or True/False */
}
```

Below are two black box examples of what happens in either my Function 2 or Function 3 example when you pass both an **odd** number (43) and an **even** number (44):





When you ***pass back the result of a function*** to the calling function, you now have MANY options! Let's look at just a few ways we can now work with the value **returned** from Functions 2 or 3:

```
/* Conditionally print in the calling function based on the result */

if ( is_even ( number ) )   /* the result is TRUE, not zero */
     printf ("number is even \n");
else
     printf ("number is odd \n");
```

You could also just print the result, in this case, a TRUE (1) or FALSE (0) value

```
printf ( "%i \n", is_even (number) ); /* will print either T (1) or F (0) */
```

Finally, you could use the result in an if statement, the ! means NOT, so not TRUE ... means FALSE

```
if (  ! is_even ( number ) )    /* its Odd */
{
     x = 12;
     b++;
}
```

which could also be written to check for a FALSE (0) value

```
if ( is_even ( number ) = = 0 )  /* Odd */
 {
      x = 12;
      b++;
 }
```

... and so on ... note that none of the options above with Functions 2 or 3 are possible with our design of Function 1 ... it is the bad design!.

Video - The Good, the Bad, and the Ugly Function Designs

This video will provide some design rules and tips to help you create functions that are readable, maintainable, and most importantly ... usable! Check out this video that discusses the various design rules and then walks you through a poorly designed function.

Function Example: Triangular Number

Remember the triangular number problem we did a few lectures ago? A sample program was posted there where all the code needed to figure out a triangular number resided in the main function. Shown below is how you would create a function to calculate a triangular number for any given integer. This function could be called from the main function, or any other function in your program.

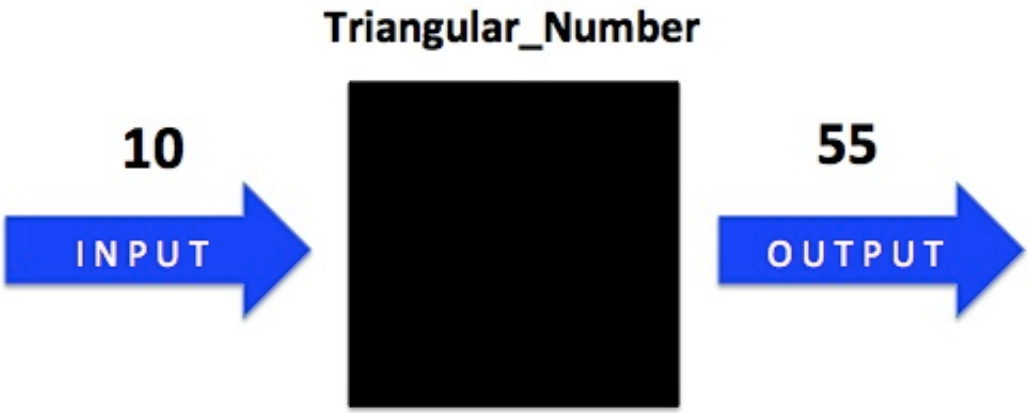
Recall that a triangular number can be generated by the formula:

- **Triangular number = n (n + 1) / 2**

For example, if we wanted to find the triangular number of 10, we just substitute in 10 for the value of n in the equation to get 55.

= (10 * (10 + 1)) / 2
= (10 * (11)) / 2
= (110) / 2
= 55

You could envision its black box function design as:



So, how would you write a function to calculate the triangular number for any integer value n? Furthermore, how would you incorporate that function into a program that you could call to generate every fifth triangular number between 5 and

50 (5, 10, 15 ... 50)?

The code and output is show below. The beauty of this function is that we can now reuse it in any future program that needs to calculate a triangular number. Try it out at: http://ideone.com/F9IwAK

```
#include <stdio.h>

/* no prototype needed, but add them here if desired */
int Triangular_Number (int);

//*************************************************************
//
// Function: Triangular_Number
//
// Description: A triangular number can be generated by the
//              the following formula:
//
//         Triangular number = n (n + 1) / 2
//
//    This function will return the triangular number
//    calculated based on it being passed an integer value
//
// Parameters: theNumber - An integer value for input
//
// Returns: The Triangular Number that is calculated based on theNumber
//
//*************************************************************

int Triangular_Number (int theNumber)
{

    int tri_num; /* triangular number to be calculated */

    /* Calculate the Triangular Number */
    tri_num = (theNumber * (theNumber + 1))/2;

    /* return to the calling function */
```

```
    return(tri_num);

} /* end Triangular_Number */


int main (void)
{

    int n;       /* a number to calculate a triangular number */
    int tri_num; /* The triangular number calculated from n */

    printf ("TABLE OF TRIANGULAR NUMBERS from 5 to 50 by 5\n\n");
    printf (" n  Triangular Number\n");
    printf ("--- ----------------\n");

    tri_num = 0;

    /* Calculate and print triangular numbers from 5 to 50 in steps of 5 */
    for (n = 5; n <= 50; n += 5)
    {
        tri_num = Triangular_Number(n);
        printf (" %-5i %-5i\n", n, tri_num);
    }

    return (0); /* indicate successful completion */

} /* end main */
```

OUTPUT:

```
 TABLE OF TRIANGULAR NUMBERS from 5 to 50 by 5

     n     Triangular Number
    ---    ----------------
```

5	15
10	55
15	120
20	210
25	325
30	465
35	630
40	820
45	1035
50	1275

Video - Passing Arrays to Functions

This video will show you how to pass Arrays to Functions. There are really two ways to do this. One way is to pass an Array Element, while the other method passes an address of an Array Element.

Functions and Arrays

When you **pass** just the array name to a function, you are passing the **address** of the array. Remember that in C, the name of an array by itself is seen as the address of the first element of the array (i.e., **array = &array[0]**). Consider the function *double_array* below that is passed an array along with its size. It will actually access each element in the array that is passed to it from the **calling function** and it will modify the contents by doubling the value stored at each array element. When you pass an item by Reference, you have the ability of actually changing its contents of the **argument** that was passed to it in the function from which it was called.

```
void double_array (int somearray[], int size)
{
    int i; /* index */

    for (i=0; i < size; ++i)
    {

        /* changes array in calling function */
        somearray[i] *= 2;
    }

    /* no return needed */
}
```

Note that you do not need to give the array parameter a size, as in:

```
void double_array (int somearray [3] , int size)
```

It will work, but it will simply be ignored. All you need with arrays in terms of setting it up as a parameter for a function, is to provide three basic things:

1) An array type,
2) An array name

3) Indicate it is an array with the two brackets [ ]

Consider the example below. When called, the function *double_array* is passed the starting address of the array.

```
double_array( myArray, 3 )
```

It could also be called the exact same way by passing the **starting address** of the first element, which is exactly the same as just saying the array name, *myArray*.

```
double_array( &myArray [0], 3 );
```

Once this is known, it can figure out how to get to each element of the array by calculating array offsets (see last week's notes) since each element is the same size and type. Again, when accessing the parameter *somearray* in the *double_array* function, it is in reality accessing the local variable *myArray* that was passed to it as an argument from the calling function, in this case, main.

```
/* https://ideone.com/b03ed9 */

#include <stdio.h>

// ********************************************
// Function: double_it
//
// Descriptions: Double the value of any given
//               integer value
//
// Parameters:  number - integer value to double
//
// Returns:     result - the number doubled
// ********************************************
int double_it (int number)
{
  int result; /* holds value of number doubled */

  result = number + number; /* double it */

  return (result);
```

```
} /* double it */

// ***********************************************
// Function: double_array
//
// Descriptions: Given an array, doubles the value
//                in each array element
//
// Parameters:  somearray - array of integer
//              size      - size of the array
//
// Returns:     void
// ***********************************************
void double_array (int somearray[], int size)
{
    int i; /* index */

    for (i=0; i < size; ++i)
    {

        /* contents changed in calling function */
        somearray [i] *= 2;

    }

    /* no return needed */

} /* double_array */

int main ()
{
  int i;                              /* index */
  int myArray [3] = {10, 20, 30}; /* local Array */

  /* pass one element of array by value           */
  /* this is really just passing an integer value */
  /* The contents of MyArray[2] are unchanged     */
```

```
    printf ("%i doubled is %i \n", myArray[2], double_it (myArray[2]) );
    printf("After call to double_it \n\n");
    printf("myArray[2] = %i \n", myArray[2] );

    /* Simply pass double_it any integer value */
    printf("\n\n 5 doubled is %i \n\n", double_it (5) );

    /* If you pass an array name, you are passing the */
    /* the array address:  &myArray[0]                */

    /* Each array element in MyArray will get doubled */
    double_array(myArray, 3);

    printf ("After call to double_array: \n\n");


   /* Verify each array element has been doubled */
   for (i=0; i < 3; ++i)
   {
       printf("myArray[%i] = %i \n", i, myArray[i] );
   }

   return(0);

  } /* main */
```

Output:

```
30 doubled is 60

After call to double_it

  myArray[2] = 30
```

```
5 doubled is 10

After call to double_array:

  myArray[0] = 20
  myArray[1] = 40
  myArray[2] = 60
```

Multidimensional Arrays

Passing multidimensional arrays needs to be done carefully, consider the code below. The first dimension does not need to be specified in the two dimensional array parameter. However, the size of the second dimension must be specified.

```
void scalar_multiply (int measured_values [ ] [20], int constant)
{
   /* body */
}


int main ()
{
   int measured_values[10][20], constant;
   ...
   scalar_multiply(measured_values, constant);

   return (0);
}
```

To send one row of multidimensional array to a called function:

```
scalar_multiply (measured_values [i][4])
```

This sends the ith row of the two dimensional array measured_values to the function, and tells the function how many columns there are. The called function thinks it is getting a 1 dimensional array.

You can not pass one column of a two dimensional array

Video - Stick Figures: The Problem with Global Variables

Here is a quick stick figure animated movie that stars me and another student ... I'll call her "Debbie Downer". It seems like with every C Class I teach, no matter how much I tell the students NOT to use global variables, a small few still don't get the message.

Global Variables

A **global variable** is a variable that is defined outside all functions, including main. The main function is no different than any other function in this case. Some say that global variables are good because any function can access them without having to pass them between functions. I would argue that they are bad in that ANY function can access and possible corrupt their values. Furthermore, it can make debugging and testing a nightmare in that you can't be certain when a variable value is changing, since it could be done in any function. Saying all that, there are times where they might be useful, but use caution. Don't be the lazy programmer that has all their functions in the form where there are no parameters or return values:

- void foo ( )

Advantages

- makes values available to all modules, without passing
- global arrays can be initialized without static
- initializes variables to 0

Disadvantages

- reduces function generality
- reduces function reliability
- is harder to debug
- They always take up space in memory during your entire program execution, makes your executable size bigger too.

Again, things to remember about global variables within a program:

- Global variables can be **accessed** by ANY function
- Global variables can be **changed** by ANY function

Important Points:

1**)** If you have a **conflict where a local variable is the same name as a global variable**, and it is referenced in a given function, then the local variable will always be used instead of the global variable.

2) In general, you should ***never use global variables*** unless it is absolutely necessary. Don't use them on your exams or homework problems ... it just makes you a lazy programmer

Run the code example below, what do you expect the final values of localValue and globalValue to be when they are printed below?

```
/* https://ideone.com/m5O0Xu */

#include <stdio.h>

/* - A global variable, does not belong to any function */
/* - It is initialized to zero automatically            */
/* - It can be any valid C variable name.               */
int globalValue;

// ************************************************
// Function:  somefunc
//
// Description: Updates a global variable
//
// Parameter: number - an integer number
//
// Returns:  void (global variable is updated)
// ************************************************

void somefunc (int number)
{
    int answer;  /* squared value */

    globalValue *= 10;  /* update global var */

    answer = globalValue * globalValue; /* use it */
}

int main ( )
{
    int localValue;  /* a local variable value */

    globalValue += 10;  /* add 10 to globalValue */

    localValue = globalValue;  /* set values to it */

    somefunc (localValue);

    printf("localValue = %i, globalValue = %i\n", localValue,
globalValue);

    return (0);
}
```

Output

```
localValue = 10, globalValue = 100
```

AUTO and STATIC VARIABLES

Local (a.k.a., *Automatic*) and **static variables** are a bit different than global variables. Let's make sure we understand the differences.

```
void somefunction()
{

   /* auto (local) variable - could also */
   /* be:  auto int value1 = 5           */
   int value1 = 5;

   static int value2 = 10;
}
```

Auto Variable

- Initialized **each time**
- Creates **no default** value
- Disappears, recreates, etc.
- **Auto** is short for "Automatic" ... think of them as "Local Variables"

Static Variable

- Initialized **only once**
- Maintains its value throughout the life of the program
    - Is actually stored in memory like a global variable
- **Defaults** to 0, unless otherwise stated

We'll discuss this more in the next class once we go through the C Run Time Environment. It will be important to first understand the program stack, heap, data, and text areas. Once we go over how these areas work, it will then be clear how static, local, and global variables work.

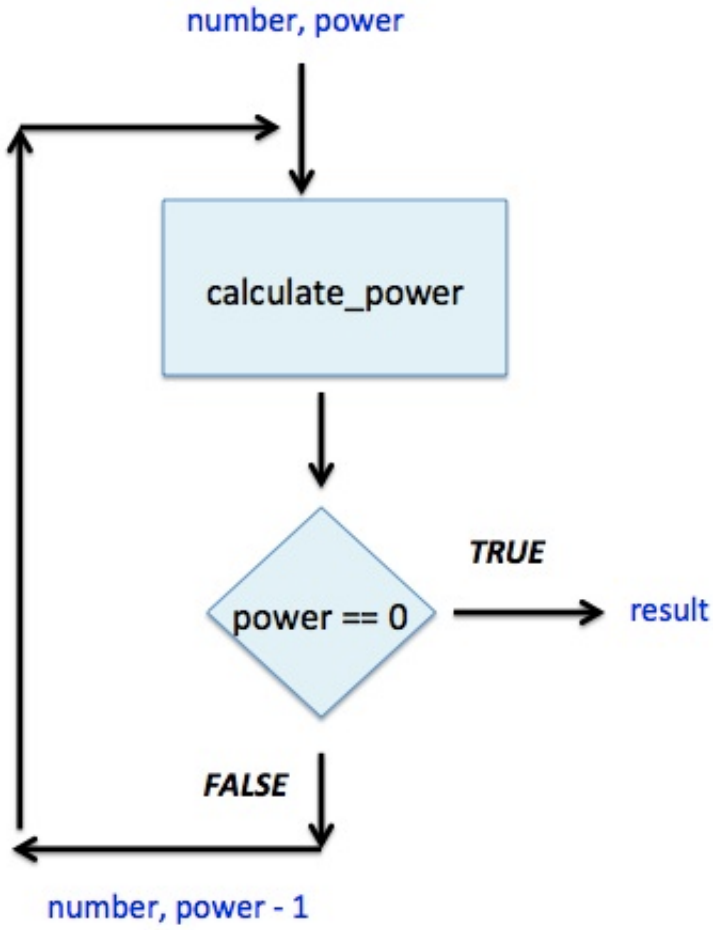
Recursive Functions

A **recursive function** is a function that calls itself during its execution. This enables the function to repeat itself several times, outputting the result at the end of each iteration. Recursive functions are common in programs because they allow developers to write efficient programs using a minimal amount of code.

You must include logic in your function to stop the recursion and return to the calling function. If proper cases are not included in the function to stop the execution, the recursion will repeat forever, causing the program to crash or tie up valuable resources on your computer.

- For example: if (power = = 0) return (result);

Review the code for the recursive function, **calculate_power**, which is shown below. You can envision it working like this:



Don't worry about recursive functions. They can be **complicated** to explain, design, and test. For these reasons, many companies and organizations recommend that they not be used in their coding standards. We won't be using them anymore in this class, but I wanted to cover them so you know how they work and you'll understand them if you see them in the future. Also, if you do use them, remember not to use static variables, that can cause infinite looping as their values are maintained each time the function calls itself.

Look to the book for other examples, but here is one that I wrote a few years ago.

```
/* http://ideone.com/ZVBsP */

#include <stdio.h>

/* Function prototypes */
long int calculate_power (int number, int power);

/*****************************************************/
/* Function: calculate_power                         */
/*                                                   */
/* Overview: Raises an integer to a positive integer  */
/*           power.                                  */
/*                                                   */
/* Parameters:  number - a integer                   */
/*              power  - a positive power to raise   */
/*                                                   */
/* Written by:  Timothy Niesen                       */
/*                                                   */
/* Date Written: 10/22/91                            */
/*****************************************************/

long int calculate_power (int number, int power)
{
    long int result = 1; /* final result */

    /* anything to the zero power is 1 */
    if (power == 0)
        return(result);

    else
    {
        /* Recursive call: a function calling itself */
        result =  number * calculate_power(number, power - 1);
    } /* else */

    return (result); /* to calling function */

} /* calculate_power */

int main (void)
{

    /* Call this function with 4 different arguments */

    printf("Passed with 5,0 is %li\n", calculate_power(5,0));
    printf("Passed with 5,1 is %li\n", calculate_power(5,1));
    printf("Passed with 5,2 is %li\n", calculate_power(5,2));
    printf("Passed with 5,3 is %li\n", calculate_power(5,3));
```

```
        return(0);
    } /* main */
```

Output:

```
Passed with 5,0 is 1
Passed with 5,1 is 5
Passed with 5,2 is 25
Passed with 5,3 is 125
```

Compiling and Building Programs in C with gcc

The are many compilers out there that will work with C and C++. One of the most popular ones out there is the **GNU Compiler Collection (GCC)**, which is a collection of programming compilers. It is commonly available on UNIX and LINUX systems, and you can even find versions to run on most systems. Let's look at **gcc** to help us understand how to compile and work with multiple files.

If you have just one file to compile, **gcc** would do all four steps of the compilation process (preprocessor, compiler, assembler, and linker) to create an executable. The key point here is that you must have at a minimum a function called **main**, as any C program is essentially a bunch of functions calling other functions that starts with a function called "**main**". Let's take a look at each of the four C compilation steps below:

- **Step 1 (Preprocessing)** - The first step which is somewhat unique to C will pre-process the source code, that begins with the # directive, initially by expanding include files (such as stdio.h), removing comments, and expanding macros, the latter of which we will cover in a future week. It will also replace your symbolic constants with their value, for example, given:

  ```
  #define STD_HOURS 40.0
  ```

  ... the statement: `if (hours > STD_HOURS)`

  ... is expanded to the following statement that will be compiled: `if (hours > 40.0)`

- **Step 2 (Compiling)** - The second step will take the C source code that has been preprocessed and generate assembly language, which is a human readable language specific to the target machine. A statement in C will generally translate to many equivalent assembly instructions. Below is a simplified translation to assembly language of our initial conditional statement along with comments to the right. There are various flavors of assembly language out there.

  ```
  CMP REG5, 40.0    ;  compare REG5 to the value 40.0
  JE   LH8          ;  if equal, then jump to label LH8
  ```

- **Step 3 (Assembly)** - The third step will take all the assembly code and covert it to machine code, a set of zeros and ones which is better known as object code. If you are a computer or robot, you'll have no problem figuring out the code that is actually being processed ... as now the code is in a form a machine can quickly understand and process. I am greatly simplifying things here, but you don't want to be reading machine code as it is not viewed very well on your screen as text.

10010111000110011001100111 0

- **Step 4 (Linking)** - The fourth and final step will link all our object code together, incorporating various functions and libraries into an executable file that you can invoke to run your program. There are two types of linking methods that can be integrated into your executable. In ***static linking***, library object code as needed is copied into and lives within your executable. With ***dynamic linking***, only information about what is being linked is placed in your executable, whereby, at run time, the executable will retrieve what it needs. Don't be too concerned at this point about static vs dynamic linking. In our last lecture week, I'll show you how to create, link, and utilize both static and dynamic libraries.

Four Step C Compilation Process - Hello World Program

Remember our first program? The **Hello World** program? It includes a call to **printf**, which is part of the standard C library. The function prototype for **printf** can be found in the stdio.h file. On most UNIX systems, header files can be found in the directory: ***/usr/include***

Most of the library files to compile your program on a UNIX machine are found in ***/usr/lib*** or ***/usr/local/lib*** ... but it varies depending on the administrator who is setting up the system. The file that contains most of the C standard library functions is called" **libc.a**" (if static linking is used) or **libc.so** (for dynamic linking). It contains pre-compiled functions stored in corresponding **object files** (a file that is in a compiled state) and stored them all together in a container we call a **binary archive file** (hence, the *.a or *.so) extension.

Thus, when you compile your program, a C compiler like **gcc** will link in that **libc** file automatically by default so it is available with the code you have developed to link them all together to create an executable if everything is successful. Given our very simple C program below:

```
#include <stdio.h>
int main ( )
{
    printf ("Hello World\n");
}
```

You could use **gcc** to compile the program in the following manner:

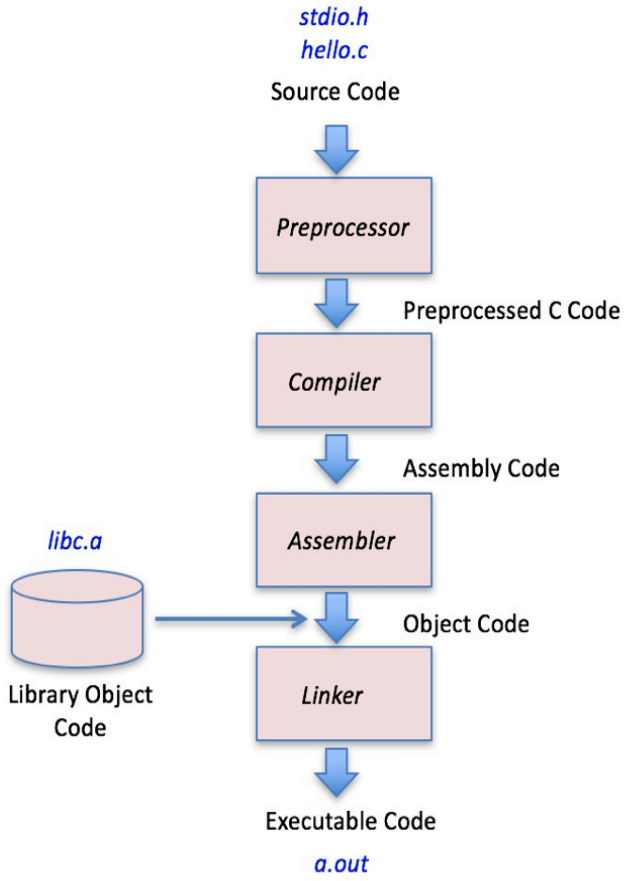
```
gcc hello_world.c
```

Alternatively, it would be equivalent to:

```
gcc hello_world.c -lc
```

Where **-lc** would be shortcut for linking in the **libc** library. When you use the -l option with **gcc** to indicate a library to include, the "**lib**" in **libc.a** (or **libc.so**) is not required, as well as the file extension part: (***.a** or ***.so**). Yet both **gcc** statements above are equivalent as the object files in the C Library are automatically included by default, whether you name the **libc** library file or not.

This will compile the source code in the file **hello_world.c** first, and if successful, will link in the standard C library (**libc**) by default, which is where the compiled **printf** function is found. It will successfully compile, build, and create a default executable called "**a.out**" that can be used to run your program. Below is an illustrated view of this "compile and build" process in C:



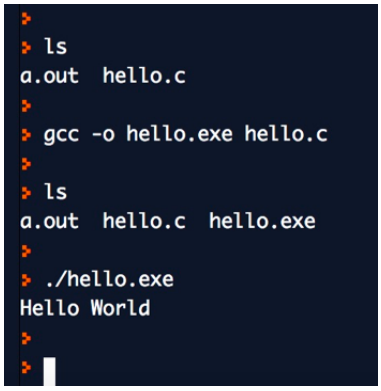
Below is a sample session on a UNIX system, from the command line, compiling and building the Hello World program. We have a single file called **hello.c** in my directory, and you can use the "**ls**" UNIX command to show all files in a directory. When I compile the file **hello.c** using **gcc**, a default executable file called "**a.out**" will be created (assuming everything compiled and built correctly) that I can use to run the program at the command line.

gcc hello.c

```
> ls
hello.c
> gcc hello.c
> ls
a.out  hello.c
> ./a.out
Hello World
>
```

You can also redirect an executable file using the -o option with gcc.

```
gcc -o hello.exe hello.c
```



Other Useful Tips

There are many available options that you can run with **gcc**, for a full list, just do an Internet search on:

gcc C Compiler Options

Should you want to see the translated assembly language of your C code, you can use the **-S** option with **gcc**.

```
gcc -o hello.exe -S hello.c
```

It will create a file called **hello.s** that you can open up and see the assembly code.

Even better, if you want to see the C code and associated translated assembly code together to help you follow along easier, try:

```
gcc -Wa,-adhln -g hello.c > hello.s
```

The **-Wa** option for **gcc** will pass multiple options to the assembler (**-adhln**), add useful debugging information (**-g**) and then redirect (using ">") to a file called **hello.s** which you can then view using your favorite editor (as it is just a text file).

If you wanted to convert your C source code to corresponding compiled code format (known as an **object file**), you can do:

```
gcc -c hello.c
```

It will generate an object file, which has a *.o file extension:

```
hello.o
```

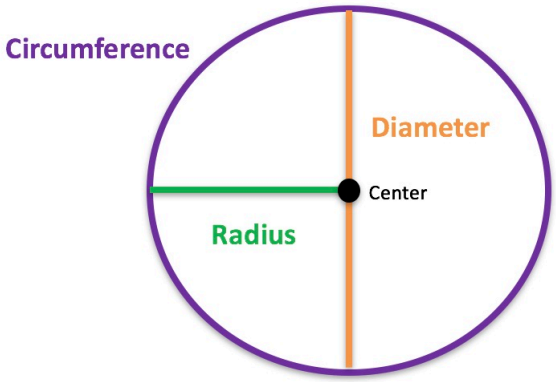
... we can link multiple object files together which will be useful if your program is made up of multiple files.

One final tip, you can open and look at any **C source file** (*.c, *.h, or *.cpp extension),

you can look at any **assembly file** (usually a *.s extension), but I would not recommend tying to view or edit an **object file** (*.o) or an **executable file** (a.out, *.exe, etc.). Something will display, but you won't be able to make any use of it as a human, and your computer screen might not be too happy trying to render it.

A Circle C Program (Single File)

A **C Program** is essentially a bunch of functions calling other functions that starts with a function called **main**. Let us take a look at an example program the calculates various aspects of a circle. The three defining aspects of a circle are the Circumference, Diameter, and Radius. The **Circumference** is the distance around the outside of the circle, much like a perimeter for a square or rectangle. The **Diameter** is the distance between one side of the circle to the other at its widest point. It will pass through the **Center** of the circle. The **Radius** is half the distance of the diameter. The diagram below illustrates each of the items presented above:



The interesting thing is knowing the size of any one of these three items: **Circumference**, **Diameter**, or **Radius** ... can help you figure out any of the others with the help of remarkable value called **Pi**, which is the ratio of the **Circumference** of a circle to the **Diameter**. Given any circle, just divide the **Circumference** by the **Diameter** and you always get exactly the same number, regardless of the size of the circle. For example, while there are many ways to figure out the three defining aspects of a circle, here are a group of three equations that all use the value of **Pi** (which in my notes, I will round to 3.1416).

- *Circumference = Pi * Diameter*
- *Radius = Circumference / (2 * Pi)*
- *Diameter = Circumference / Pi*

A program is shown below that incorporates functions to calculate each of the three defining aspects of a circle. Note that all functions within this program exist in one single file. You can call the file whatever name you like, but a program must at least contain a "main" function. You can also try it out at: https://ideone.com/4ZIIS4

```
/* A program to figure out the three defining aspects of a Circle */

#include <stdio.h>

/* Constants */
#define PI 3.1416
```

```
/* Function Prototypes */
float circleCircumference (float Diameter);
float circleRadius (float Circumference);
float circleDiameter (float Circumference);

int main ( )
{

   float circumference;  /* circumference of a circle */
   float diameter;       /* diameter of a circle */
   float radius;         /* radius of a circle */

   printf ("Enter the Circumference in Centimeters: ");
   scanf ("%f", &circumference);

   /* calculate the diameter and radius based on the circumference */
   diameter = circleDiameter (circumference);
   radius = circleRadius (circumference);

   printf ("\nThe Diameter of our Circle is: %5.1f \n", diameter);
   printf ("\nThe Radius of our Circle is %5.1f \n", radius);

   printf ("The Circumference given the Diameter %5.1f is %5.1f \n",
            diameter, circleCircumference (diameter));

   return (0);

} /* main */

// ************************************************************************
// Function:  circleCircumference
//
// Description:  Calculates the Circumference of a Circle
//               given the Diameter of a Circle
//
// Parameters:  Diameter - The Diameter of the Circle
//
// Returns:  Circumference - The Circumference of a Circle
// ************************************************************************
float circleCircumference (float Diameter)
{
   return (PI * Diameter);
}

// ************************************************************************
// Function:  circleRadius
//
// Description:  Calculates the Radius of a Circle
//               given the Circumference of a Circle
//
```

```
// Parameters:  Circumference - The Circumference of the Circle
//
// Returns:  Radius - The Radius of a Circle
// ************************************************************************
float circleRadius (float Circumference)
{
   return (Circumference / (2 * PI));
}

// ************************************************************************
// Function:  circleDiameter
//
// Description:  Calculates the Diameter of a Circle
//                  given the Circumference of a Circle
//
// Parameters:  Circumference - The Circumference of the Circle
//
// Returns:  Diameter - The Diameter of a Circle
// ************************************************************************
float circleDiameter (float Circumference)
{
   return (Circumference / PI);
}
```

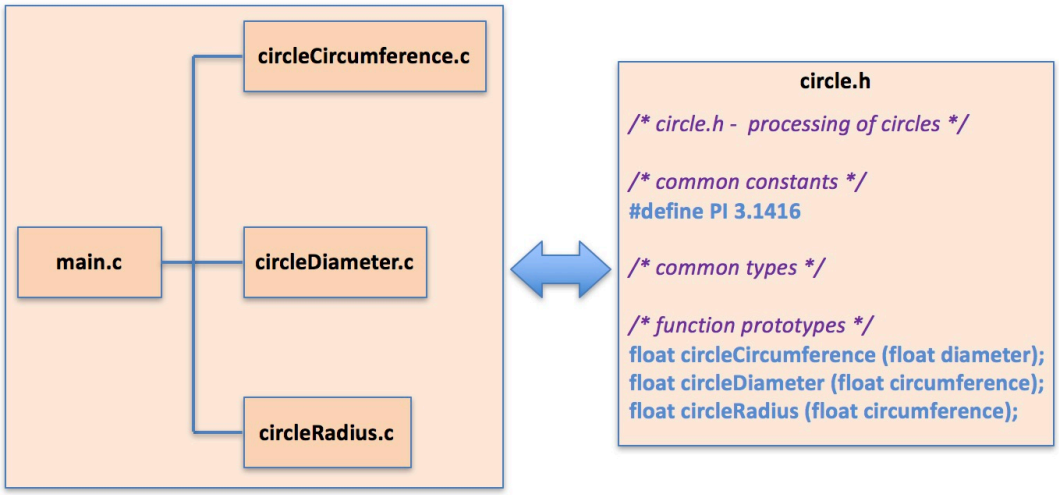
Sample Output

```
Enter the Circumference in Centimeters: 32
The Diameter of our Circle is: 10.2
The Radius of our Circle is 5.1
The Circumference given the Diameter 10.2 is 32.0
```

A Circle C Program (Multiple Files)

Up to now, every program we have developed has been contained within a single C source file. For this class, you can certainly continue to use this method for all your programs. However, in the real world, when you start dealing with larger and more complex programs, this approach is no longer practical. As the number of lines of code increases, so does the time needed to edit and compile your program. Making any change (however large or small) to your source file requires that everything be recompiled to keep your executable that you run up to date. In the real world, most program development requires the resources of more than one developer. Having everyone work on the same file, even their own copy, would be totally unworkable. Let's look at the Circle Program we just did as a single file and envision how it could be done by separating each function into its own file. For library functions such as **printf** and **scanf** contained in our C Standard Library (**libc**), we'll continue to link those into our program during the linking compilation stage.

The key design issue now is that if a function is all by itself within its own file, how would it get access to any constants, types, and function prototypes it would need to allow it to compile successfully. The trick is you can encapsulate those items into one or more header files that you can create and store either in your current directory or somewhere on your computer to share as needed with multiple programs. For our example Circle Program, let's put it into a file called **circle.h** and include it as needed into our separate C source code files.



Thus, we include the **circle.h** file in every source file that needs it, in this case, all files (except **main.c**) use **PI** as a

symbolic constant, and the **main** function needs to use the function prototypes to allow the compiler to check, verify, and compile everything correctly. To include the header file, **circle.h**, into each of the files, just put it with any other includes as needed near the top of the file (just after any other standard C library include files such as **stdio.h**).

```
#include "circle.h"   /* would look for it in the current directory where it is being compiled */
```

or

```
#include "/usr/local/programs/include/circle.h"   /* some common area on the computer to share */
```

For example, the file **circleDiameter.c** would only need to include the **circle.h** file. It is needed so that it knows the value of the PI symbolic constant:

```
#define PI 3.1416
```

The include file **stdio.h** would not be needed here since there are no library I/O function calls (such as printf or scanf) in this function. If you do include **stdio.h**, all will work just fine as the function code will only consume what is needs to compile and link correctly. Note that **stdio.h** would be needed in the **main.c** file as the main function does call functions like printf and scanf.

```
#include "circle.h"

// ************************************************************************
// Function:  circleDiameter
//
// Description:  Calculates the Diameter of a Circle
//                  given the Circumference of a Circle
//
// Parameters:  Circumference - The Circumference of the Circle
//
// Returns:  Diameter - The Diameter of a Circle
// ************************************************************************
float circleDiameter (float Circumference)
{
   return (Circumference / PI);
}
```

Given all these files within a given directory, you can now use **gcc** to compile and build your program with the following

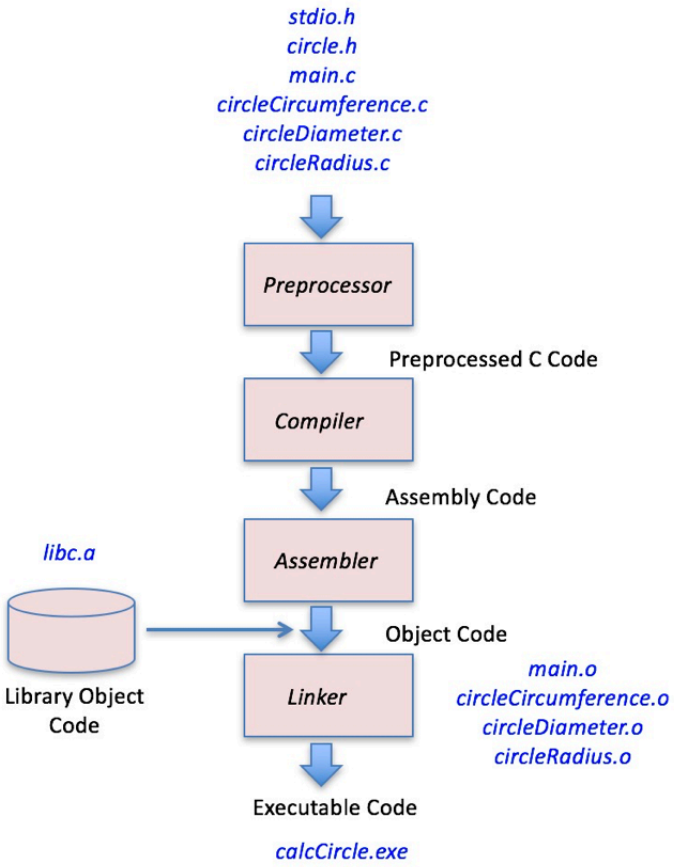
single command:

```
gcc -o calcCircle.exe main.c circleCircumference.c circleDiameter.c circleRadius.c
```

When you are using a full blown native compiler and IDE, steps like this are often done behind the screens for you. The executable file will only be created if the entire compilation and build process is successful with all four files. The same result can be achieved using the following five commands:

```
gcc -c main.c

gcc -c circleCircumference.c

gcc -c circleDiameter.c

gcc -c circleRadius.c

gcc -o calcCircle.exe main.o circleCircumference.o circleDiameter.o circleRadius.o
```

Using **gcc -c** will create corresponding object files (*.o extensions) that represent the successful compilation of a C file. Together, the object files can be linked together along with any library objects to create a program executable. Below is a illustration of what is happening at each of the five compilation steps above:



The disadvantage of the methods already discussed is that you have to compile each source file every time. A better method would be to only compile source files if they had changed after the creation of the executable file. Only files that have been modified after the successful creation of the executable file will need to go through the compilation process again since they would be "out of date". For example, if only the file **circleRadius.c** were subsequently modified, then the following commands would correctly update the **calcCircle.exe** executable.

gcc -c **circleRadius.c**

gcc -o calcInterest.exe main.o circleCircumference.o circleDiameter.o **circleRadius.o**

Makefiles

It is important to note in our example above that only **circleRadius.c** needed to be recompiled, so the time required to

rebuild **calcCircle.exe** is shorter than if the first method for compiling and building **calcCircle.exe** were used that consisted of all five steps (four compiles and one link/load). When there are numerous source files that make up a program, and a change is only made to just a few of them, the time savings can be significant.

This automated compile and build process is generally handled consistently through the use of a **Makefile**. If you are interested, there are various tutorials on the Internet on Makefiles. I'll provide a brief and simple tutorial and example here.

Below are the contents of a Makefile I created (**filename: makefile**) to automate the compile and build process for our Circle program:

```
all: calcCircle.exe

calcCircle.exe: main.o circleCircumference.o circleDiameter.o circleRadius.o
     gcc -o calcCircle.exe main.o circleCircumference.o circleDiameter.o circleRadius.o

main.o: main.c circle.h
     gcc -c main.c

circleCircumference.o: circleCircumference.c circle.h
     gcc -c circleCircumference.c

circleDiameter.o: circleDiameter.c circle.h
     gcc -c circleDiameter.c

circleRadius.o: circleRadius.c circle.h
     gcc -c circleRadius.c

clean:
     rm -rf *.o calcCircle.exe
```

In the statements above, note that one indents with a **TAB** instead of consecutive spaces within a **Makefile** (this is a small but important point to adhere to in getting the make process to work correctly). Three **Makefile** commands come to mind to use it ... assuming the name of my file is "**makefile**":

```
make clean
make all
make
```

You can always just start with "**make**" to start the process. The ***"clean"*** directive within the Makefile will remove any object and executable files. Running the ***"all"*** directive right after that will force a recompilation of all files to create (assuming no errors) new object files and an executable. If you then change the contents of the ***calculateRadius.c***

file, and rerun ***"make all"***, it would then force a recompilation of ***calculateRadius.c*** to create an new object file (***calculateRedius.o***) and then proceed with the build process to create a new version of the executable (**calc*Circle.exe***).

How does the Makefile know if an existing object file (file.o type) is out of date and needs to be recompiled? It is actually very easy; every file has an associated modification date that gets changed anytime it is saved. Two things come to mind:

1. ***Source File*** - If a source file (**foo.c**) is *newer* than its corresponding object file (**foo.o**), then a recompilation is needed
2. ***Target Executable*** - If any *source file has changed*, then the executable is not the latest, a recompilation and build is needed

In summary, a **Makefile** provides both an automated and orderly process to ensure your program and all its components are kept up to date with each set of changes.

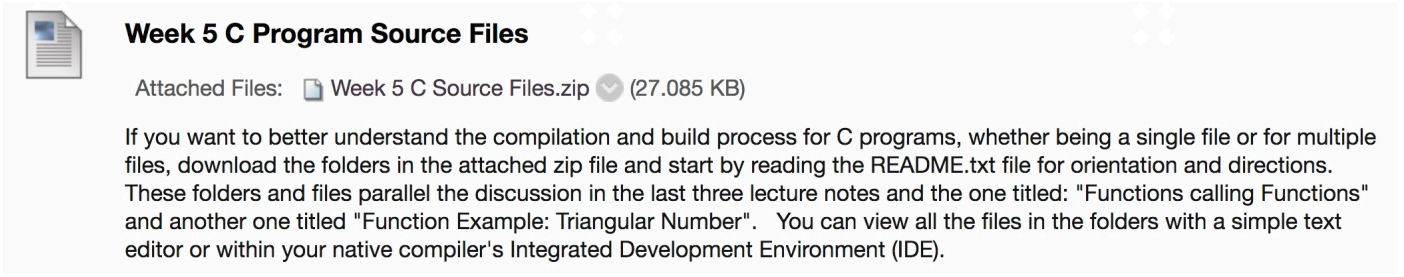
Below is an example of running the actual **Makefile** on a LINUX system. The "**ls"** UNIX command is used to list the files in the current directory. It shows the base files we started with, the makefile, the four C source files, and a header file. Once I run the **Makefile** (i.e., make), note how corresponding **object files** (*.o) are created for each **C source file** (*.c) along with an **executable** called **calcCircle.exe**. One can then invoke the executable file to run the program, whereby I typed in **sample data** as prompted. Finally, it shows how I can **clean up** the files (objects and executable) using the "**make clean**" command to get back to my original state of a Makefile and four C source files. If I edit the **circle.h** file in any way, it will require that every associated C file will have to be recompiled and the executable file rebuilt.

```
> 
> 
> ls
circleCircumference.c  circleDiameter.c  circle.h  circleRadius.c  main.c  makefile
> 
> 
> make
gcc -c main.c
gcc -c circleCircumference.c
gcc -c circleDiameter.c
gcc -c circleRadius.c
gcc -o calcCircle.exe main.o circleCircumference.o circleDiameter.o circleRadius.o
> 
> 
> ls
calcCircle.exe         circleDiameter.c  circleRadius.c  main.o
circleCircumference.c  circleDiameter.o  circleRadius.o  makefile
circleCircumference.o  circle.h      main.c
> 
> 
> ./calcCircle.exe
Enter the Circumference in Centimeters: 32

The Diameter of our Circle is:  10.2

The Radius of our Circle is   5.1
The Circumference given the Diameter  10.2 is  32.0
> 
> 
> make clean
rm -rf *.o calcCircle.exe
> 
> 
> ls
circleCircumference.c  circleDiameter.c  circle.h  circleRadius.c  main.c  makefile
```

Week 5 C Program Source Files

We covered many different programs with functions this week. Some of our programs contained just a main function, while others implemented a few other functions as well. In the real world, most C programs are implemented using multiple files where generally each function is stored in its own C source file. This helps to better isolate issues for debugging as well as speed up the compilation process, as you only need to to recompile functions/files that have changed. In programming classes you take going forward, you may very well be creating programs with multiple source files.

I've converted some of the programs we reviewed in the lecture notes this week to work with multiple source files. To access the files, download the folders stored in the zip file available back in our current week's notes. You will find the item right near the end of the list of lecture notes near the Quiz area, and it looks like this:



Most folders have the following file which you should read first:

- **README.txt** - general information on how to compile, build, and use the template files

Use the **Makefile** provided if you wish to compile and build your program from the command line, such as on a UNIX or LINUX system. **Alternatively**, you can load files into your **native compiler's** Integrated Development Environment (IDE) and have it build and run from there. In every case, the files will compile and build correctly out of the box. You are welcome to expand upon them by updating files as needed and adding new C source files for any additional functions you plan to design into it.

At the very least, feel free to just download the zip file and associated folders and take a look at all the files within your favorite text editor.

Summary

There is quite a bit of information to digest with functions. I have provided many detailed lecture notes this week with many examples that you can try out ... so take the time to do just that. The book is another great resource. Although you have plenty to do this week with your readings in the book/notes, as well as the homework assignment, try out some of the exercises at the end of the chapter this week in the book.

Start working on Homework 4 as soon as you can, don't wait until the last minute (**hints** and **templates** are provided). You want to create as many functions as possible. **Do not use global variables** (except if you want to specify a *file pointer*). I want you to learn how to pass arguments and work with parameters. It will be significantly bigger than your previous homework, yet in the end, accomplishes then same solution.

Take your time and do it right. I expect there will be questions when doing this, so don't hesitate to send them over the class discussion board or email me directly if you are shy. There are NO dumb questions. I've posted two templates in the assignment area if you need something to get started. Both templates do the job in the end ... you don't need to use both, just pick one to get started. Of course, you are welcome to improvise and design your own unique program from scratch, or modify the design from my templates. The templates are just to get you started in the right direction.

For Next Week

Going forward, we'll continue to use functions, as well as a new data structure called a **structure**. As a bonus, I will also go into detail about the **C Run Time Environment**, a topic not covered in your book. I'll show you exactly how a program is loaded into memory, how functions are executed, and where exactly global, static, local, and parameter variables are stored within memory.

Good Luck with your homework! Don't forget your **Quiz** this week as well.

HOMEWORK 4
FUNCTIONS

Write a C program using multiple functions that will calculate the gross pay for a set of employees.

See the last lecture note this week for a template you can use if you feel you need something to start with. However, feel free to implement this assignment that way you see fit, the template is just there if you need it. There are multiple ways to effectively implement this assignment.

The program determines the overtime hours (anything over 40 hours), the gross pay and then outputs a table in the following format shown below. Column alignment, leading zeros in Clock number, and zero suppression in float fields is important. Use 1.5 as the overtime pay factor.

```
-------------------------------------------------------------------------
   Clock#   Wage  Hours    OT    Gross
-------------------------------------------------------------------------
   098401  10.60   51.0   11.0  598.90
   526488   9.75   42.5    2.5  426.56
   765349  10.50   37.0    0.0  388.50
   034645  12.25   45.0    5.0  581.88
   127615   8.35    0.0    0.0    0.00
```

You should implement this program using one array for clock number, one array for wage rate, etc.

- Continue to use constants and build upon what you learned in previous assignments
- Read in the hours worked for each employee.
- Limit your use of global variables - Learn how to pass parameters!
- You should have at least 3-4 functions designed and implemented in addition to the main function.
- Try to limit how much code you have in your main function, call other functions to do the work that is needed.
- Remember to use constants and all the other things we have covered up to this assignment.
- Re-read the homework coding standards ... make sure that each local variable is commented in EACH function, and EACH function has a descriptive function comment header. Note that a function comment header is not needed for your main function.
- Feel free to initialize the clock and wage values into your arrays with the test data above.
- DO NOT pre-fill the hours, overtime, and gross pay arrays with test data values. It is OK however to initialize them to zero if you wish, but

it is not needed since these values get overwritten via input prompts or calculations.

I would recommend that your clock, wage, hours, overtime, and gross values be stored in individual arrays. As a challenge, you can have your program prompt the user to enter the number of hours each employee worked. When prompted, key in the hours shown below.

Do define your array size up front in your variable declaration. Don't define the array size at run time with a variable. This strategy does not always work on every C compiler.

Create a separate function whenever possible to break up your program. For instance, you might have a function for obtaining the hours from the user, another function for calculating overtime hours, another for calculating gross pay and another function for producing the output. At a minimum, you should have a main function and three or more other functions.

Finally, as always, don't use concepts we have not yet covered.

Optional Challenge

For those of you more experienced programmers, continue with the challenges below that were offered in last week's assignment. The additional challenge this week would be encapsulating the logic into functions that can be called to do this work.

- Calculate and print the total sum of Wage, Hours, Overtime, and Gross values
- Calculate and print the average of the Wage, Hours, Overtime, and Gross Values

Clock#	Wage#	Hours	OT	Gross
098401	10.60	51.0	11.0	598.90
526488	9.75	42.5	2.5	426.56
765349	10.50	37.0	0.0	388.50
034645	12.25	45.0	5.0	581.88
127615	8.35	0.0	0.0	0.00
Total	51.45	175.5	18.5	1995.84
Average	10.29	35.1	3.7	399.17

Additional Optional Challenge

If you want an additional challenge, a code template has also been provided that starts you in the right direction if you want to implement the homework using separate C source files for each function as well as including a header file as needed. You can load the template files provided into your native compiler and Integrated Development Environment (IDE) to get started.

Sample Homework 4 (Functions) Templates

For those that need a hint to get started, I've attached some code below that will give you a good starting point for the assignment. There are two general ways to solve this problem. The **first template** shows you how to pass values to functions to read in and calculate values (overtime and gross), and return them back to the calling function. It also includes a function that will print all your employees information at the end by pass the array names (i.e., addresses) needed using a ***Call by Reference*** design. The **second template** is a pure ***Call by Reference*** design, all functions are passed array names whereby any changes to any of the elements in those functions is reflected back in the arrays from the calling function.

These templates implement the basic design (a main function that calls 4-5 functions) laid out in the beginning of the video this week titled ***"Function Design Guidelines"***. Note there was a more elegant design that followed in the video using ***control functions*** calling ***worker bee functions*** that you might want to alternatively utilize to challenge yourself.

As always, the *templates are only a suggestion*. Feel free to implement your own unique designs or pick and optionally choose code segments from below.

Template #1 - Combination of Call by Value and Call by Reference

This sample code template has a loop inside the main function that has functions that pass array elements ***by value*** to read in the hours worked for each employee, and calculates their corresponding overtime and gross pay. After the loop, I have a **stub** (no body of code, just the basic set up so it can be called) for a function, **printData**, that is passed the array names (**call by reference**) containing all the employee information so that you can print all the employee information at once on demand. The code below will compile, I've tested it in IDEOne already. Your task if you decide to use this template is to fill in the missing code, add other functions as needed, add comments (including the file header and comment block headers for EACH function other than main), and test it out with input. You can retrieve the template in IDEOne at: https://ideone.com/16EEjj

```
/* Add file comment block here per coding standards with your name */

#include <stdio.h>

/* constants */
```

```
#define NUM_EMPL 5
#define OVERTIME_RATE 1.5f
#define STD_WORK_WEEK 40.0f

/* function prototypes                    */
/* I added one below to get you started */
float getHours (long clockNumber);
void printEmp  (long int clockNumber[], float wageRate[], float hours[],
                float overtime[], float gross[], int size);

/* IMPORTANT:  Add other function prototypes here as needed */

int main()
{

    /* Variable Declarations */

    long  int clockNumber[NUM_EMPL] = {98401,526488,765349,34645,127615}; /* ID */
    float gross[NUM_EMPL];     /* gross pay */
    float hours[NUM_EMPL];     /* hours worked in a given week */
    int   i;                   /* loop and array index */
    float overtime[NUM_EMPL];  /* overtime hours */
    float wageRate[NUM_EMPL] = {10.60,9.75,10.50,12.25,8.35}; /* hourly wage rate */

    for (i = 0; i < NUM_EMPL; ++i)
    {

        /* Function call to get input from user. */
        hours[i] = getHours (clockNumber[i]);

        /* Function call to calculate overtime */

        /* Function call to calculate gross pay */

    }

    /* Print the header info */

    /* Print all the employees - call by reference */
```

```
    printEmp (clockNumber, wageRate, hours,
              overtime, gross, NUM_EMPL);

    return (0);

}

//*************************************************************/
// Function: getHours
//
// Purpose: Obtains input from user, the number of hours worked
// per employee and stores the result in a local variable
// that is passed back to the calling function.
//
// Parameters: clockNumber - The clock number of the employee
//
// Returns: hoursWorked - hours worked by the employee in a given week
//
//*************************************************************/

float getHours (long int clockNumber)
{

    float hoursWorked; /* hours worked in a given week */

    /* Get Hours for each employee */
    printf("\nEnter hours worked by emp # %06li: ", clockNumber);
    scanf ("%f", &hoursWorked);

    return (hoursWorked);
}

//*************************************************************/
// Function: printEmp
//
// Purpose: Prints out all the employee information in a
// nice and orderly table format.
//
```

```
// Parameters:
//
//     clockNumber - Array of employee clock numbers
//     wageRate - Array of employee wages per hour
//     hrs - Array of number of hours worked by an employee
//     overtime - Array of overtime hours for each employee
//     gross - Array of gross pay calculations for each employee
//     size - Number of employees to process
//
// Returns: Nothing (call by reference)
//
//*************************************************************/

void printEmp (long int clockNumber[], float wageRate[], float hours[],
                float overtime[], float gross[], int size)
{

    /* This is an example of a "Stub" ... you need to add code ...     */
    /* but you can set up and call it and fill in the details later    */

    /* Add a loop here and print information on each employee          */
    /* in a nice orderly table, just like you have done before         */
    /* HINT:  The loop and code setup is similar to how the getHours   */
    /* function is implemented in the alternate template (by reference)*/
}


/* Add other functions here as needed                              */
/* ... remember your comment block headers for each function */
```

Template #2 - Call by Reference only

Below is an alternative homework template to consider. Here you exclusively pass entire arrays to functions as needed. Because you passed the array name, which is an address, meaning **call by reference**, any updates to array elements that you make in those functions will be reflected in the arrays from the calling function, which in this case is the main function. The benefit of this approach is that the main function acts more like a driver and calls functions as needed to get the job done. You do want to strive not to overload the main function with code, rather, call functions as

needed to get the job done. You can retrieve this alternative template in IDEOne at: http://ideone.com/FV3LOj

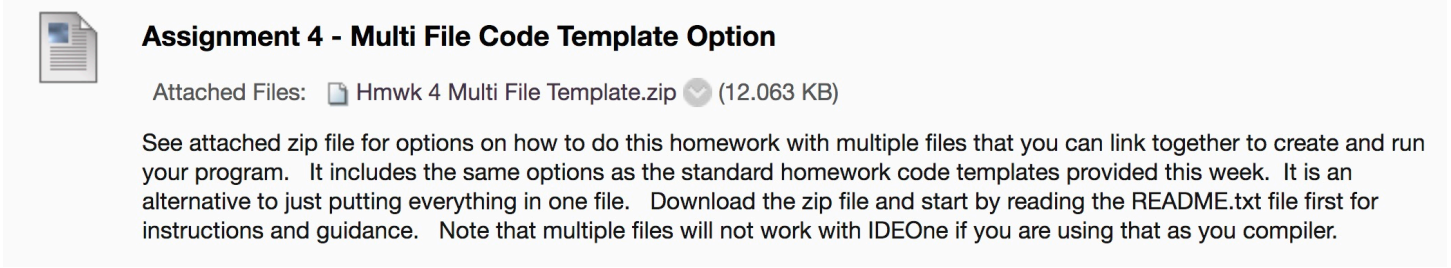
```
/* Add file comment block here per coding standards with your name */

#include <stdio.h>

/* constants */
#define NUM_EMPL 5
#define OVERTIME_RATE 1.5f
#define STD_WORK_WEEK 40.0f

/* function prototypes */
/* I added one below to get you started */
void getHours (long int clockNumber[], float hours[], int size);

/* IMPORTANT: Add other function prototypes here as needed */

int main()
{

    /* Variable Declarations */

    long  int clockNumber[NUM_EMPL] = {98401,526488,765349,34645,127615}; /* ID */
    float gross[NUM_EMPL];      /* gross pay */
    float hours[NUM_EMPL];      /* hours worked in a given week */
    float overtime[NUM_EMPL];   /* overtime hours */
    float wageRate[NUM_EMPL] = {10.60,9.75,10.50,12.25,8.35}; /* hourly wage rate */

    /* Function call to get hours worked for each employee */
    getHours (clockNumber, hours, NUM_EMPL);

    /* Function call to calculate overtime */

    /* Function call to calculate gross pay */

    /* Function call to output results to the screen */

    return (0);
```

```
}

//*************************************************************/
// Function: getHours
//
// Purpose: Obtains input from user, the number of hours worked
// per employee and stores the results in an array that is
// passed back to the calling function by reference.
//
// Parameters:
//
// emp_num - Array of employee clock numbers for each employee
// hrs - Array of hours worked by each employee
// size - Number of employees to process
//
// Returns: Nothing (call by reference)
//
//*************************************************************/

void getHours (long emp_num[], float emp_hrs[], int size)
{

    int count; /* loop counter */

    /* Get Hours for each employee */
    for (count = 0; count < size; ++count)
    {
        printf("\nEnter hours worked by emp # %06li: ", emp_num[count]);
        scanf ("%f", &emp_hrs[count]);
    }

    printf("\n\n");
}

/* Add other functions here as needed */
/* ... remember your comment block headers for each function */
```

Homework 4 - Multi File Template Option

Take the optional challenge this week and implement Homework Assignment 4 using multiple files.

To access the files, download the folder stored in the zip file available back our current week's notes. You will find the item right near the end of the list of lecture notes, and it looks like this:



It contains two folders, one for a design using *Call by Value*, and the other with a design using *Call by Reference*. The same set of files are tailored and implemented into each folder based on the design:

- **employees.h** - header file with common constants, types, and prototypes
- **main.c** - the main function to start the program
- **getHours.c** - a function that will read in the number of hours an employee worked
- **makefile** - a file you can use if you want to compile and build from the command line
- **printEmp.c** - a function that will print out the current values in our arrays
- **README.txt** - read this first! ... general information on how to compile, build, and use the template files

Use the **makefile** provided if you wish to compile and build your program from the command line, such as on a UNIX or LINUX system. **Alternatively**, you can load the *employee.h* file and the two source files (*main.c* and *print_list.c*) into your **native compiler** Integrated Development Environment (IDE) and have it build the template from there. In either case, the files will compile and build correctly out of the box. Your job will be to expand upon them by updating files as needed and adding new C source files for any additional functions you plan to design into it.

Of course, even if you decide to just implement your homework 4 assignment within a single file using the other homework code template(s) provided, feel free to just download the Multi File Code Template Option folder and take a look at all the files within your favorite text editor.