Functions

A concept from Math function, e.g., cosine, sine, logarithm, square, etc.
Reusable code – help you to “divide and conquer” a problem

Outline

1. Basic Concepts
   - Understand what a function is
   - Call a function and trace a function call
   - Characteristics of the variables declared inside a function
   - Why we define function
2. Parameters (IN)
   - Define a function with parameters
   - Pass data into a function
3. Return value (OUT)
   - Define a function with a return value
   - The "`return`" keyword
4. Function prototypes

You will need to create many functions in your “Project”

1.1. What is a Function?

- A *function* consists of a sequence of statements for performing "a specific task"!

- C programs are usually made up of one or more functions.

  e.g., `printf(), scanf(), sqrt()  // built-in func.`

  `main()` is the starting point of a program. `// user-defined`

Note:

- In the lecture notes, we use the notation `foobar()` to mean "a function named **`foobar`**".
- It is also known as *procedure* or *subroutine* in some other programming languages.
- Each function (e.g., printf()) should have a clear (specific) task to do.

1.2. A Simple Function

```
1   #include <stdio.h>
2
3   void greet()
4   {
5       printf( "Hi! How are you?\n" );
6   }
7
8   int main()
9   {
10      greet() ;
11      return 0 ;
12  }
```

1.3. Function Name

See lec note 1b

A function has a name (identifier)

```
1   #include <stdio.h>
2
3   void greet()
4   {
5       printf( "Hi! How are you?\n" );
6   }
7
8   int main()
9   {
10      greet() ;
11      return 0 ;
12  }
```

Another user-defined function named "greet"

A function named "main"

main() is also the starting point of a C program.

1.4. Call/Invoke A Function

```
1   #include <stdio.h>
2
3   void greet()
4   {
5       printf( "Hi! How are you?\n" );
6   }
7
8   int main()
9   {
10      greet() ;
11      return 0 ;
12  }
```

We *call* (or *invoke*) a function by the function's name, followed by a pair of parentheses.

Call a function
➔ execute the code in that function

1.5. Terminology: *Caller* and *Callee*

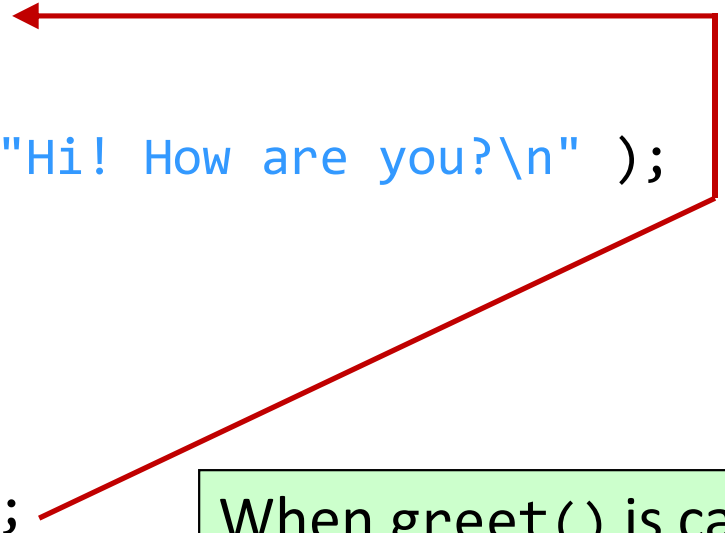
```
1   #include <stdio.h>
2
3   void greet()
4   {
5       printf( "Hi! How are you?\n" );
6   }
7
8   int main()
9   {
10      greet() ;
11      return 0 ;
12  }
```

At line 10, `main()` initiates the function call "`greet()`". In this situation, we say

- `main()` is the *caller*, and
- `greet()` is the *callee*.

1.6. Control Flow during a Function Call – Part 1

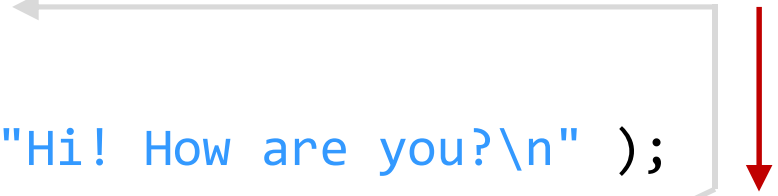
```
1   #include <stdio.h>
2
3   void greet()
4   {
5       printf( "Hi! How are you?\n" );
6   }
7
8   int main()
9   {
10      greet() ;
11      return 0 ;
12  }
```



When `greet()` is called at line 10, control is "transferred" to the beginning of `greet()`.

1.6. Control Flow during a Function Call – Part 2

```
1   #include <stdio.h>
2
3   void greet()
4   {
5       printf( "Hi! How are you?\n" );
6   }
7
8   int main()
9   {
10      greet() ;
11      return 0 ;
12  }
```

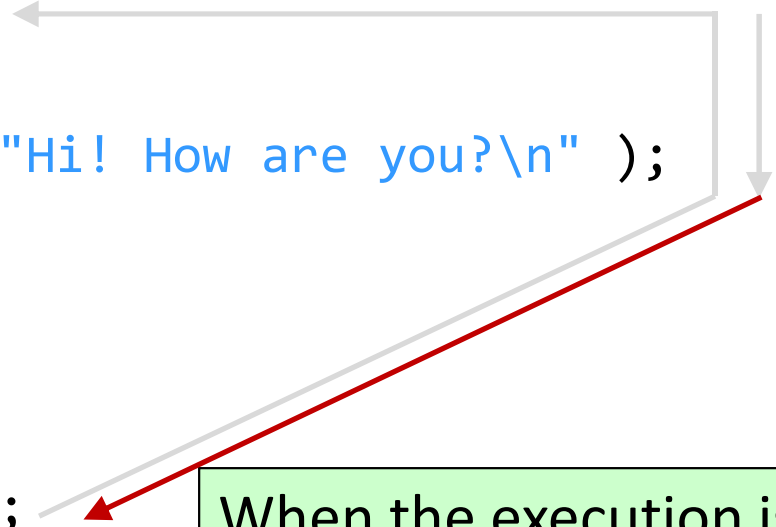


Statement(s) in `greet()` will then be executed sequentially from the beginning.

```
Hi! How are you?
```

1.6. Control Flow during a Function Call – Part 2

```
1   #include <stdio.h>
2
3   void greet()
4   {
5       printf( "Hi! How are you?\n" );
6   }
7
8   int main()
9   {
10      greet() ;
11      return 0 ;
12  }
```



When the execution is completed in greet( ), control is returned to the location where greet( ) is called.

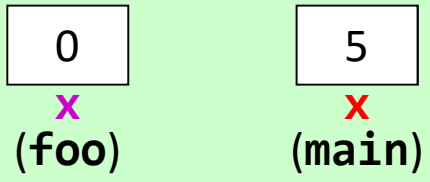
```
Hi! How are you?
```

1.7. Variables declared in a function are **local** to that function

```
1   void foo()
2   {
3       int x = 0 ;
4       printf( "In foo(): x = %d\n" , x );
5   }
6
7   int main()
8   {
9       int x = 5 ;
10
11      printf( "Before: In main(): x = %d\n" , x );
12      foo() ;
13      printf( "After: In main(): x = %d\n" , x );
14
15      return 0 ;
16  }
```

A function may have its own variable(s).

x in foo() and x in main() are two different variables.




```
Before: In main(): x = 5
In foo(): x = 0
After: In main(): x = 5
```

Note: “different” means they are stored at different locations in the computer’s main memory

1.7. Variables declared in a function are **not** accessible outside the function

```
1   void bar()
2   {
3       int y = 0 ;
4       printf( "In bar(): y = %d\n" , y );
5   }
6
7   int main()
8   {
9       bar() ;
10      printf( "%d\n" , y );
11      return 0 ;
12  }
```

compile-time error

Variables declared in a function are *local variables* and are only accessible inside that function.

y, being declared in `bar()`, is not accessible in `main()`.

Note: concepts “life-time” and “scope” of variables will be discussed in more detail in Lecture 8

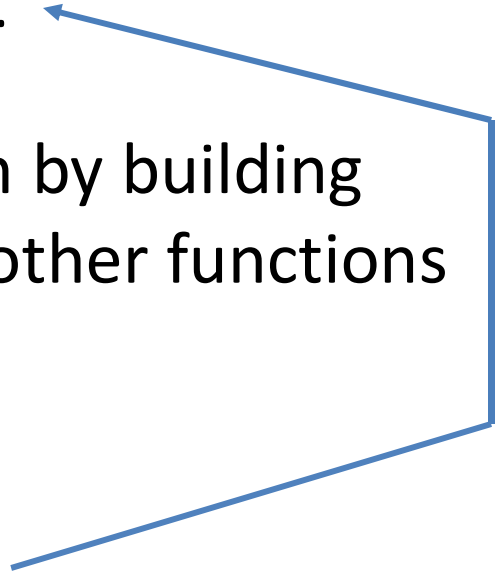
1.1. Why Function?

- **Rule of thumb**: a *function* should perform a specific task.
  - **Reusable**
  - **Readable** – "users" easily knows how to use it
  - Each function should be **complete** (for a **task**) and **not too long**

    e.g., `printf`, `getchar`, `isascii`, `toupper`, etc.

- We can "**divide and conquer**" a problem by building small functions that can be "reused" in other functions

- **Good programming practice**:
  - Use meaningful names for your functions!



Outline

1. ~~Basic Concepts~~
   - ~~Understand what a function is~~
   - ~~Call a function and trace a function call~~
   - ~~Characteristics of the variables declared inside a function~~
   - ~~Why we define function~~
2. Parameters (IN)
   - Define a function with parameters
   - Pass data into a function
3. Return value (OUT)
   - Define a function with a return value
   - The "`return`" keyword
4. Function prototypes

2. Parameters

- Parameters are for passing data from a caller to a callee.

- Proper use of parameters allows programmers to reuse code for processing different data.
  - e.g., `printf()` can be used to print many kinds of data in different formats.

2.1. Formal and Actual Parameters

```
1   #include <stdio.h>
2
3   void foo( int n )
4   {
5       printf( "%d\n" , n );
6   }
7
8   int main()
9   {
10      int x = 10 ;
11
12      foo( 3 );
13      foo( x );
14      foo( x + 3 );
15
16      return 0 ;
17  }
```

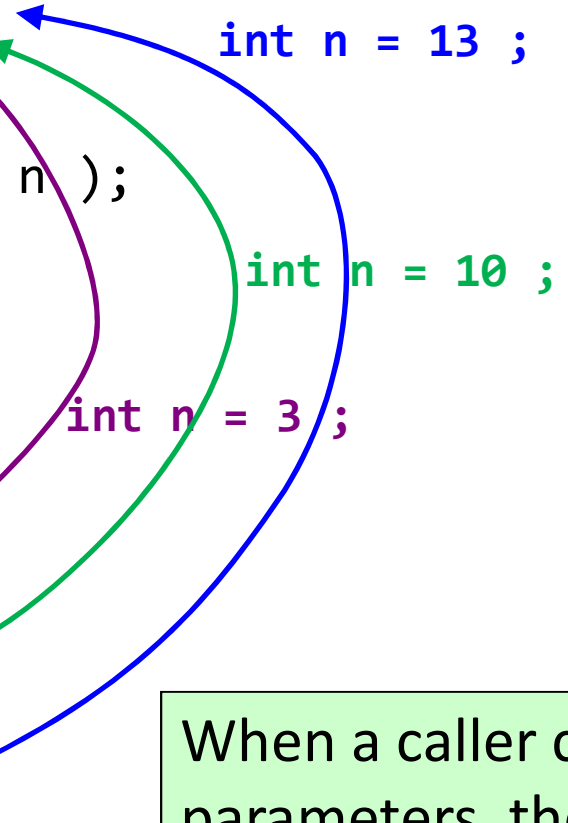
The variables for holding the values passed to a function are called the *formal parameters*.

They have local scope in the function (like local variables).

The expressions specified in the function calls are called the *actual parameters* or *arguments*.

2.1. Formal and Actual Parameters

```
1   #include <stdio.h>
2
3   void foo( int n )
4   {
5       printf( "%d\n" , n );
6   }
7
8   int main()
9   {
10      int x = 10 ;
11
12      foo( 3       );
13      foo( x       );
14      foo( x + 3 );
15
16      return 0 ;
17  }
```



Console output:

```
3
10
13
```

When a caller calls a function with parameters, the values of the arguments are copied to initialize (or assigned to) the formal parameters.

2.3. Define a function with parameters (syntax)

```
void function_name( parameter_list )
{
    declarations and statements
}
```

- `parameter_list`
  - Zero or more parameters separated by commas in the form

    $\texttt{type}_1\ \texttt{param}_1\ ,\ \texttt{type}_2\ \texttt{param}_2\ ,\ \ldots\ ,\ \texttt{type}_N\ \texttt{param}_N$

2.4. Example

- Design and implement a function that can be used to print an input character N times.

2.4. Example

```
1  /* A function that prints character "ch" n times */
2  void printChars( char ch , int n )
3  {
4      for ( int i = 0 ; i < n ; i++ )
5          printf( "%c" , ch );
6      printf("\n");
7  }
8
9  int main()
10 {
11     printChars( '#' , 17 );
12     printf( "  Hello World!\n" );
13     printChars( '*' , 17 );
14     return 0 ;
15 }
```

```
#################
  Hello World!
*****************
```

2.4. Example

```
1  /* A function that prints character "ch" n times */
2  void printChars( char ch , int n )
3  {
4      for ( int i = 0 ; i < n ; i++ )
5          printf( "%c" , ch );
6      printf("\n");
7  }
8
9  int main()
10 {
11     printChars( '#' , 17 );
12     printf( "  Hello World!\n" );
13     printChars( '*' , 17 );
14     return 0 ;
15 }
```

`char ch` is the character to be printed.

%c is the format specifier for printing a character

Character constant is enclosed by a pair of single quotes, e.g., '#'

This code shows the power of "**reusable**" function!

2.5. Parameters are matched by position

```
1   void foo( int x , int y )
2   {
3       printf( "%d %d\n" , x , y );
4   }
5
6   int main()
7   {
8       int x = 3 , y = 2 ;
9       foo( x , y );
10      foo( y , x );
11      return 0 ;
12  }
```

What is the output?

- Arguments and formal parameters are **matched by position** (NOT BY names and NOT BY types).

2.6. Common Mistake

```
1  void foo( int x , int y )   /* Correct */
2  {
3    …
4  }
```

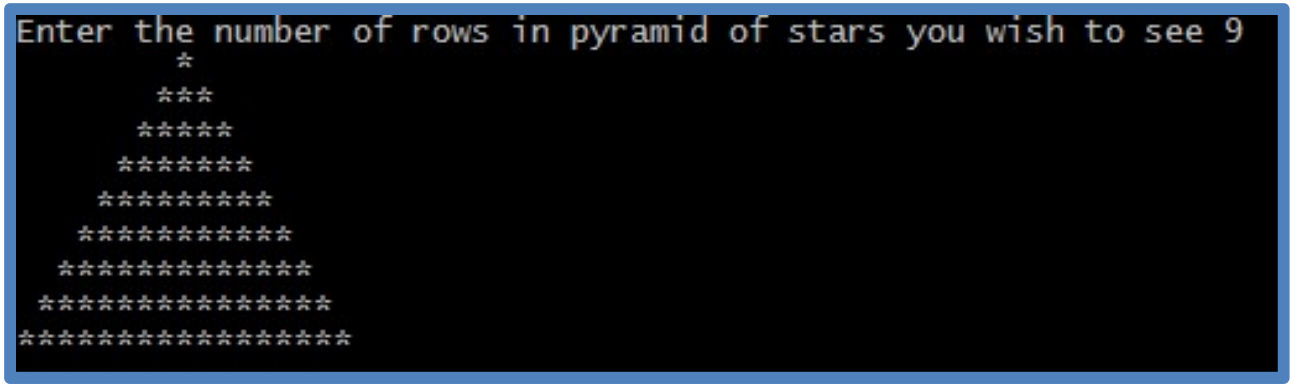
```
1  void foo( int x , y )   /* Incorrect */
2  {
3    …
4  }
```

Note:

Need to individually specify a data type for each parameter even though multiple parameters are of the same type.

Self-Exercise – The power of reusability!

```
1  /* A function that prints character "ch" n times */
2  void printChars( char ch , int n , int any_newline )
3  {
4      for ( int i = 0 ; i < n ; i++ )
5          printf( "%c" , ch );
6      if ( any_newline ) printf("\n");
7  }
8
9  int main()
10 {
11     // your code
12
13
14     return 0 ;
15 }
```



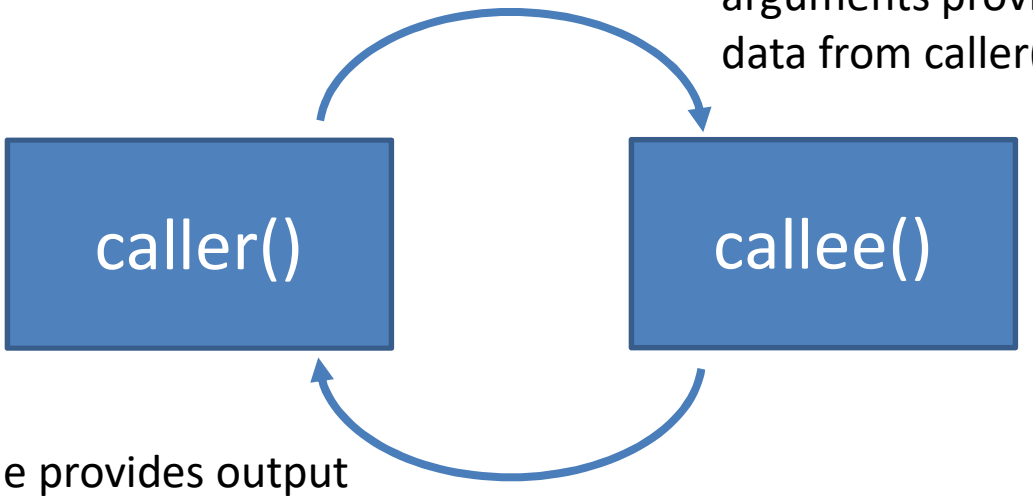
Outline

1. ~~Basic Concepts~~
   - ~~Understand what a function is~~
   - ~~Call a function and trace a function call~~
   - ~~Characteristics of the variables declared inside a function~~
   - ~~Why we define function~~
2. ~~Parameters (IN)~~
   - ~~Define a function with parameters~~
   - ~~Pass data into a function~~
3. Return value (OUT)
   - Define a function with a return value
   - The "`return`" keyword
4. Function prototypes

3. Return Value

- How to return a value from a function to its caller?

- How does the `return` keyword affect the execution flow inside a function?



Note: this is like math functions,
e.g., value = cos(45°)

3.1. Return a Value from a Function

```
1  #include <stdio.h>
2
3  int cube( int x )
4  {
5      int y ;
6      y = x * x * x ;
7      return y ;
8  }
9
10 int main()
11 {
12     int result ;
13     result = cube( 3 ) ;
14     printf( "Cube of 3 is %d\n" , result );
15     return 0 ;
16 }
```

A function can return a value to its caller.

```
Cube of 3 is 27
```

3.1. Return a Value from a Function

```
1  #include <stdio.h>
2
3  int cube( int x )
4  {
5      int y ;
6      y = x * x * x ;
7      return y ;
8  }
9
10 int main()
11 {
12     int result ;
13     result = cube( 3 ) ;
14     printf( "Cube of 3 is %d\n" , result );
15     return 0 ;
16 }
```

`int` indicates that `cube()` must return a value of type `int` when the function finishes its execution

`return` is the keyword for specifying the value to be returned from callee to caller. In this example, the value of y is returned.

```
Cube of 3 is 27
```

3.2. Define a function that returns a value (Syntax)

```
return_type function_name( parameter_list )
{
    …
    return expression ;
}
```

- `return_type`
  - Data type of the data to be returned by callee function
  - Use `void` for functions that do not need to return any value
- `return expression ;`
  - `return` is a keyword in C
  - When this `return` statement is executed, `expression` will first be evaluated and the resulting value will be passed to the caller (similar to break, the control flow will go back to the caller)

3.3. Evaluating an expression containing function calls

Functions are called first if they are part of an expression.

```
x = cube(1) + cube(2) * cube(3) ;
```

```
x = 1 + cube(2) * cube(3) ;
```

```
x = 1 + 8 * cube(3) ;
```

```
x = 1 + 8 * 27 ;
```

```
x = 1 + 216 ;
```

```
x = 217 ;
```

Note: some compilers call the functions from right to left.

3.4. Interrupting Control Flow with `return`

A **`return`** statement also forces the execution to leave a function (like “`break`”) and return to its caller immediately.

```
int min( int x , int y )
{
   if ( x > y )
      return y ;

   return x ;
}
```

When "`return y`" is executed, execution immediately stops in `min()` and resumes at its caller.

In this example, if "`x > y`" is true, "`return x`" will not be executed.

3.5. Example

- Implement a function that accepts a month and a year as parameters, and returns the number of days in the given month and year.

3.5. Example (with multiple **`return`**'s)

```
1  /* Returns # of days in a particular month */
2  int days_per_month( int m , int y )
3  {
4     if ( m == 1 || m == 3 || m == 5  ||
5          m == 7 || m == 8 || m == 10 || m == 12 )
6        return 31 ;
7
8     if ( m == 4 || m == 6 || m == 9 || m == 11 )
9        return 30 ;
10
11    /* if y is a leap year */
12    if ( … )
13       return 29 ;
14
15    return 28 ;
16 }
```

Only one of the "`return`" statements will be executed.

Similar to "`break`", once "`return`", get out already!

3.5. Example (with only one **return**)

```
1   /* Returns # of days in a particular month */
2   int days_per_month( int m , int y )
3   {
4      int days ;
5      if ( m == 1 || m == 3 || m == 5  ||
6           m == 7 || m == 8 || m == 10 || m == 12 )
7         days = 31 ;
8      else
9      if ( m == 4 || m == 6 || m == 9 || m == 11 )
10        days = 30 ;
11     else
12     if ( … )  /* if y is a leap year */
13        days = 29 ;
14     else
15        days = 28 ;
16     return days ;
17  }
```

A “long” function is easier to debug if there is only one **return**, since we know exactly where the execution leaves the function.

3.6. Using `return` without a returning value

- When the function return type is `void`, we can use `return` without a return value.

```
void ask_something( int code )
{
   if ( code != 7 )
   {
      printf( "Who are you?\n" );
      return ;   /* Leave the function immediately */
   }
   printf( "How are you today, James?\n" );
   return ;   /* This return statement is optional */
}
```

If the return type is **`void`**, placing a return as the last statement is optional (it is implied).

3.7. Additional info about returning a value

- A function can only return **one value** of a specific data type

- If a function's return type is not **`void`**, then **all paths** leaving the function **must return a value** that matches the specified return type.

```
double reciprocal( double x )
{
   if ( x != 0.0 )
      return 1.0 / x ;
}
```

If x is 0.0, then what value to be returned?

Compiler may warn

Outline

1. ~~Basic Concepts~~
   - ~~Understand what a function is~~
   - ~~Call a function and trace a function call~~
   - ~~Characteristics of the variables declared inside a function~~
   - ~~Why we define function~~
2. ~~Parameters (IN)~~
   - ~~Define a function with parameters~~
   - ~~Pass data into a function~~
3. ~~Return value (OUT)~~
   - ~~Define a function with a return value~~
   - ~~The "`return`" keyword~~
4. Function prototypes

You may see this in “Project”

4. Function Prototypes

- Also known as *function declarations*
- Why do we need function prototypes?
- How to define function prototypes?

```
1   #include <stdio.h>
2
3   int main()
4   {
5       printf( "%d\n" , square( 4 ) );
6       return 0 ;
7   }
8
9   /* Function definition */
10  int square( int y )
11  {
12      return y * y ;
13  }
```

Compile-time warning: undefined '**square**'; assume returning **int**. Why?

- For efficiency, a C compiler performs a 1-pass sequential scan of the source code during the compilation.
- When it sees the identifier "**square**" at line 5, it does not know that "**square**" is a function defined at line 10.

```c
#include <stdio.h>

/* Function definition */
int square( int y )
{
    return y * y ;
}

int main()
{
    printf( "%d\n" , square( 4 ) );
    return 0 ;
}
```

- We could rearrange the functions, so that all callees are defined before their callers, but such approach is not always possible (and actually troublesome). Why?
  - E.g., we have multiple functions, and the functions call one another…
- A better solution is to declare function prototypes.

```
1   #include <stdio.h>
2
3   /* Function prototype */
4   int square( int );
5
6   int main()
7   {
8      printf( "%d\n" , square( 4 ) );
9      return 0 ;
10  }
11
12  /* Function definition */
13  int square( int y )
14  {
15      return y * y ;
16  }
```

Tells the compiler that:

- **square** is a function name.
- It takes one argument of type **int**.
- It returns a value of type **int**.

A function prototype provides compiler info (I/O) about a function to be defined later. Using function prototype, you can use a function before you define it.

However, a function prototype is a "promise" that the function will be defined *somewhere in the program*; if you miss it, eventually…

4.1. When and why do we need function prototypes?

```
void foo( void );
void bar( void );

int main()
{
   foo() ;
}

void foo()
{
   if (…) bar() ;
}

void bar()
{
   if (…) foo();
}
```

- When a callee is defined after its caller in the same file.
- When a callee and its caller are defined in separate source files
  - Common in large software project

  E.g., your project!

foobar.h

```
void foo( void );
void bar( void );
```

main.c

```
#include "foobar.h"
int main()
{
   foo() ;
}
```

foobar.c

```
void foo()
{
   if (…) bar() ;
}

void bar()
{
   if (…) foo();
}
```

4.2. Specifying Function Prototypes (Syntax)

- Function prototype is like a function definition but without the body.

Function definition

```
double foo( int x, double y, char z ) {
    …
}
```

Function prototype

```
double foo( int x, double y, char z );
```

or

```
double foo( int , double , char );
```

  - Parameter names are optional in function prototypes. Why?
  - Function name, return type, and parameter types must match between a function definition and its function prototype.

5. Calling Pre-defined Functions

- C language provides many built-in functions, e.g., `printf`. To use them, you have to know the following info (which can be found in manuals or API):
  - name, functionality, parameters & return value

- You also need to know which *header file(s)* to include, e.g.:
  - To use `printf(…)`, you have to include "stdio.h" as
    ```
    #include <stdio.h>
    ```
  - To use math functions, you have to include "math.h" as
    ```
    #include <math.h>
    ```

Summary

- Understand what "functions", "**parameters/arguments**", "**return value/type**" are
- Understand what is happening during a function call
- **Know how to define and call a function**
- Understand why we need **function prototypes** and how to declare them

Make sure you understand function prototype, you will need them in the *course project* a lot.

Final note: Why we need Function?

- Without functions, what will the code look like?
- With functions,
  - "Divide and conquer" a problem -> larger software
  - Code becomes more reusable -> efficiency in code development
  - Code becomes more readable
    - Same function can be called by many callers, as well as in different programs

Do your lab. exercises with functions!!

Will save your time!!

Let's try this in lab 7 ex 1?
What function that you write ?

Self-exercise: any substring is a palindrome?

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>

// What is a palindrome?  e.g., madam and noon
// What is a palindrome substring?  e.g., CSCI, tool

int is_palindrome( const char str[ 103 ] , int i , int j )
{
    ...
}

int check_sub_palindrome( const char str[ 103 ] , int i , int j , int n )
{
    ... // We may call is_palindrome somewhere here
}

int main( void )
{
    char str[ 103 ] ;

    while ( fgets( str , 102 , stdin ) != NULL )
    {
        ... // We may call is_palindrome and check_sub_palindrome
    }
    return 0 ;
}
```

When you use fgets(), beware of \n (which could be 1 or 2 char) at the end

```
int lastPos = strlen(str)-1 ;
while ( str[lastPos] == 10 || str[lastPos] == 13 )
   lastPos-- ;
```