60-141-02 LECTURE 2: FUNCTIONS

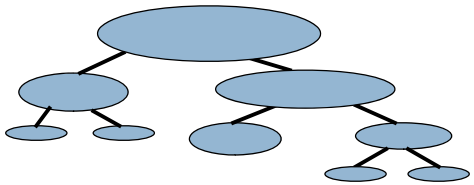
Dr. Mina Maleki

Outline

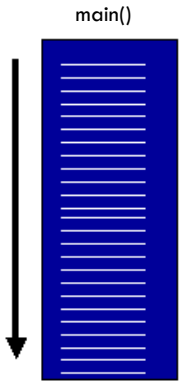
- The Function Concept
- Predefined Library Functions
- User-defined Functions
- Local and Global Variables
- Variable Scope and Storage
- Recursive Functions

Modularity

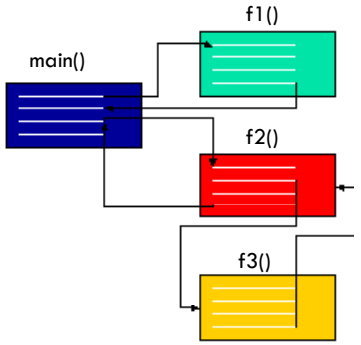
- How do you solve a big/complex problem?
  - Divide it into small tasks and solve each task. Then combine these solutions.
- Divide and conquer
  - Divide program into small pieces/components
  - Construct a program from smaller pieces/components
  - These smaller pieces are called modules



Modular Style Programming



One Program



Many Modules

Advantages of using modules

- Modules can be written and tested separately (more manageable)
- Modules can be reused → avoid repeating (reusability)
- Large projects can be developed in parallel
- Reduces length of program, making it more readable
- Promotes the concept of abstraction
  - A module hides details of a task
  - We just need to know what this module does
  - We don't need to know how it does it

C Function

- Modules in C → Function:
  - Has its own declarations and statements
  - Performs a single, well-defined task
- As a general rule, any piece of logic that is likely to be repeated is worth placing it in a separate function.
- Every C program has at least one function, which is **main()**
- Functions could be
  - **Pre-defined library functions** (e.g., printf, pow)
  - **User-defined functions**

C Library Functions

Predefined Functions

C Standard Libraries

- Standard means that the solution or function is supported by a national and international committee (ANSI / ISO) and is known to work the same way across different compilers.
- A Library is a term used to refer to the packaged functions and files shipped together.
- Avoid reinventing the wheel. Use C Standard Library functions instead of writing new functions:
  - Save on development time
  - Increase the reliability of our code.
    - standard functions are often rigorously tested and well documented.

C Standard Libraries …

- To use existing C library functions, we have to include the header file of the corresponding library.
  - Contain the function prototypes for all the functions in that library
  - Load with `#include <filename.h>`, #include <math.h>
- Programs that refer to any functions in a library must include all the functions in that library.
  - Because most programming work that requires one function, most likely will also use the other functions
  - Since all function codes must be included in the fully compiled program, avoid including unnecessary libraries

C Libraries …

- C library functions provide most commonly used functions.
- The system is subdivided into specific libraries, each devoted to a particular topic (set of functions)
  - <stdio.h> input / output
  - <math.h> mathematic functions
  - <stdlib.h> random numbers, conversions of numbers to text and text to numbers, memory allocations
  - <string.h> string processing
  - <time.h> time and date

Some of the standard library headers

Header	Explanation
`<assert.h>`	Contains information for adding diagnostics that aid program debugging.
`<ctype.h>`	Contains function prototypes for functions that test characters for certain properties, and function prototypes for functions that can be used to convert lowercase letters to uppercase letters and vice versa.
`<errno.h>`	Defines macros that are useful for reporting error conditions.
`<float.h>`	Contains the floating-point size limits of the system.
`<limits.h>`	Contains the integral size limits of the system.
`<locale.h>`	Contains function prototypes and other information that enables a program to be modified for the current locale on which it's running. The notion of locale enables the computer system to handle different conventions for expressing data such as dates, times, currency amounts and large numbers throughout the world.
`<math.h>`	Contains function prototypes for math library functions.
`<setjmp.h>`	Contains function prototypes for functions that allow bypassing of the usual function call and return sequence.
`<signal.h>`	Contains function prototypes and macros to handle various conditions that may arise during program execution.
`<stdarg.h>`	Defines macros for dealing with a list of arguments to a function whose number and types are unknown.
`<stddef.h>`	Contains common type definitions used by C for performing calculations.
`<stdio.h>`	Contains function prototypes for the standard input/output library functions, and information used by them.
`<stdlib.h>`	Contains function prototypes for conversions of numbers to text and text to numbers, memory allocation, random numbers, and other utility functions.
`<string.h>`	Contains function prototypes for string-processing functions.
`<time.h>`	Contains function prototypes and types for manipulating the time and date.

Standard Input/Output Library <stdio.h>

- Predefined functions to perform input and output of characters and constants such as EOF
- **printf**: `printf("Value = %f", 0.5 * (X+Y) ) ;`
- **scanf**: `scanf("%d%f", &intVal,&floatVal ) ;`
- I/O involves a single character
  - **putchar**: `putchar( Ch ) ;`
  - **getchar**: `char Ch = getchar( ) ;`
- I/O involves a string of characters
  - **puts**: `puts( "This is an output string 1 2 3" ) ;`
  - **gets**: `char name[10]; gets(name);`

printf/scanf conversion specification

13

Data type	printf conversion specification	scanf conversion specification
long double	%Lf	%Lf
double	%f	%lf
float	%f	%f
unsigned long int	%lu	%lu
long int	%ld	%ld
unsigned int	%u	%u
int	%d	%d
unsigned short	%hu	%hu
short	%hd	%hd
char	%c	%c

Standard I/O Library <stdio.h> …

14

- Definition of a special constant, called the end-of-file marker, **EOF**.
  - The actual value of **EOF** is typically the value **-1**.
  - The `getchar()` and `scanf()` functions return a value which can be assigned to a variable (int or char)
    - This variable can then be compared with EOF to "detect" the end-of-file condition

```
while ((Ch = getchar())!= EOF )
{
 .....
}
```

```
Ch = getchar() ;
while ( Ch != EOF )
{
    .....
    Ch = getchar() ;
 }
```

<math.h>

15

Function	Description	Example
sqrt(x)	square root of *x*	sqrt(900.0) is 30.0 sqrt(9.0) is 3.0
cbrt(x)	cube root of *x* (C99 and C11 only)	cbrt(27.0) is 3.0 cbrt(-8.0) is -2.0
exp(x)	exponential function e^x	exp(1.0) is 2.718282 exp(2.0) is 7.389056
log(x)	natural logarithm of *x* (base *e*)	log(2.718282) is 1.0 log(7.389056) is 2.0
log10(x)	logarithm of *x* (base 10)	log10(1.0) is 0.0 log10(10.0) is 1.0 log10(100.0) is 2.0
fabs(x)	absolute value of *x* as a floating-point number	fabs(13.5) is 13.5 fabs(0.0) is 0.0 fabs(-13.5) is 13.5
ceil(x)	rounds *x* to the smallest integer not less than *x*	ceil(9.2) is 10.0 ceil(-9.8) is -9.0
floor(x)	rounds *x* to the largest integer not greater than *x*	floor(9.2) is 9.0 floor(-9.8) is -10.0
pow(x, y)	*x* raised to power *y* (x^y)	pow(2, 7) is 128.0 pow(9, .5) is 3.0
fmod(x, y)	remainder of *x*/*y* as a floating-point number	fmod(13.657, 2.333) is 1.992
sin(x)	trigonometric sine of *x* (*x* in radians)	sin(0.0) is 0.0
cos(x)	trigonometric cosine of *x* (*x* in radians)	cos(0.0) is 1.0
tan(x)	trigonometric tangent of *x* (*x* in radians)	tan(0.0) is 0.0

Example

16

- If a = 4.99, b = 13.0, c = 3.0 and d = 4.0.

Function	Description	Example
`fabs(x)`	Return the absolute value of X	`fabs(-13.5) = 13.5`
`fmod(x,y)`	Return the remainder of X/y	`fmod(13.5,12) = 1.5`
`pow(x,y)`	Return the value of X^y	`pow(d,3) = 64.0`
`sqrt(x)`	Return the square root of X	`sqrt(b+c*d) = 5.00`
`floor(x)`	rounds X down	`floor(a) = 4`
`ceil(x)`	rounds X up	`ceil(4.01) = 5`

Standard Utilities Library <stdlib.h>

17

- Definitions for a number of **general purpose** functions
- Convert data from one data type to another
  - atoi, atof, atol: Converts a string to int, float, long int
- Request memory allocation
  - malloc , calloc
- Generate random numbers
  - rand, srand

<stdlib.h>- Random Number Generation

18

- Random numbers are any sets of numbers whose pattern cannot be predicted
- function is: `int rand (void)`
  - rand( ) generates an int between 0 and 32767
- `int n = rand() % L ;`
  - L is called the ***scaling factor***
  - Generate values within a range [0..L) by ***scaling*** with L
- Generate a number between a min and a max number

```
min + (rand() % (max + 1 - min))
```

Example 1

19

- Generates an integer between 0 and 9
  - `rand() % 10`
- Generates an integer between 1 and 100
  - `1 + (rand() % 100 ))`
- Generate a random number between 3 and 20
  - `3 + (rand() % (20+1-3))`
  - `3 + (rand() % 18)`

Example 2

20

- Rolling the dice

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

// function main begins program execution
int main( void )
{
   unsigned int i; // counter

   // loop 20 times
   for ( i = 1; i <= 20; ++i ) {

      // pick random number from 1 to 6 and output it
      printf( "%10d", 1 + ( rand() % 6 ) );

      // if counter is divisible by 5, begin new line of output
      if ( i % 5 == 0 ) {
         puts( "" );
      } // end if
   } // end for
} // end main
```

First Run

6	6	5	5	6
5	1	1	5	3
6	6	2	4	2
6	2	3	4	1

Second Run

6	6	5	5	6
5	1	1	5	3
6	6	2	4	2
6	2	3	4	1

rand() always produces the same sequence of values, starting at the same value. Eventually the sequence repeats itself.

<stdlib.h>- Random Number Generation …

- Solution to overcome the limitations of rand() → **srand**
- srand(time(NULL));
  - This function seeds the random number generator used by the function **rand**
  - It uses `time(NULL)` in `<time.h>` library to generate a unique seed from the system clock

Example

- Print 5 random numbers from 1 to 50

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

int main()
{
   /* Intializes random number generator */
   srand(time(NULL));

   /* Print 5 random numbers from 1 to 50 */
   for( int i = 0 ; i < 5 ; i++ )
   {
      printf("%d\n", rand() % 50 + 1);
   }
   return(0);
}
```

User-defined Functions

Function Definition


```
return_type function_name (parameters)
{
      declarations;
      statements;
}
```

```
double average(double a,double b)
{
  double average;
  average = (a + b) / 2;
  return average;
}
```

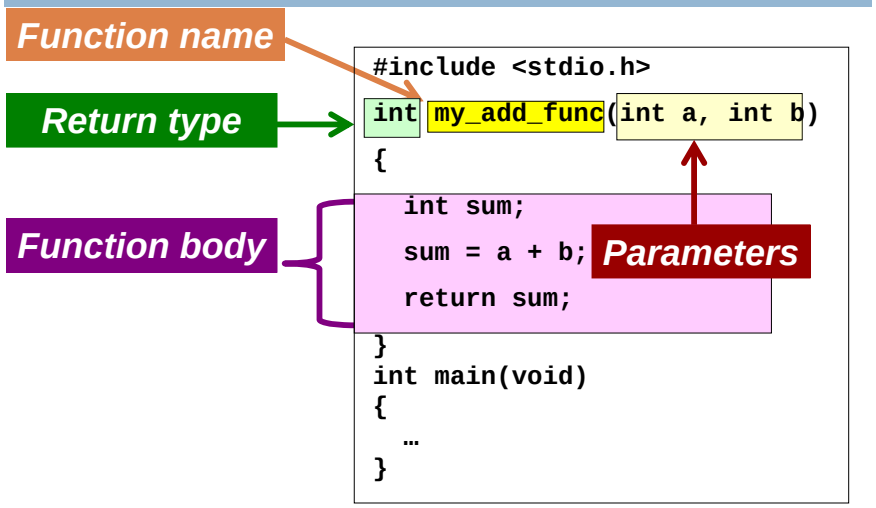
Example

Function definition

-occurs ONE time for all
-outside other functions

```
#include <stdio.h>
int my_add_func(int a, int b)
{
  int sum;
  sum = a + b;
  return sum;
}
int main(void)
{
  …
}
```

Example 2



Function Name

- Function name should effectively express the task of the function.
- No two functions have the same name in your C program.
- The naming rule is the same as variable names.

Return Value

- Function definition declares the type of value to be returned
- Rules governing the return type:
  - Functions may not return arrays.
  - If the function doesn't return a value → the return type is `void` ; `void` is a type with no values.
- How to return a value in a function
  - return; //for void return type
  - return variable_Name;
  - return expression;

Input Parameters

- Arguments
- Information passed to a function
- A comma-separated list of parameters
- A type must be listed explicitly for each parameter
- Example: `(int num1, double num2, char ch)`
- Function with no parameters → The word `void` is placed in parentheses after the function's name

```
void sayHello ( void )
{
  printf("Hello World!\n");
}
```

Function Body

- block of statements which is enclosed in braces to be executed when the function is called
- The body of a function may include both declarations and statements
- Variables declared in the body of a function can't be examined or modified by other functions.
- In C89, variable declarations must come first, before all statements in the body of a function.
- In C99, variable declarations and statements can be mixed, as long as each variable is declared prior to the first statement that uses the variable.

Function Call

Function Call

- A function call consists of a **function name** followed by a **list of arguments**, enclosed in parentheses:

```
average(x,y)
my_add_function(a,b)
sayHello()
```

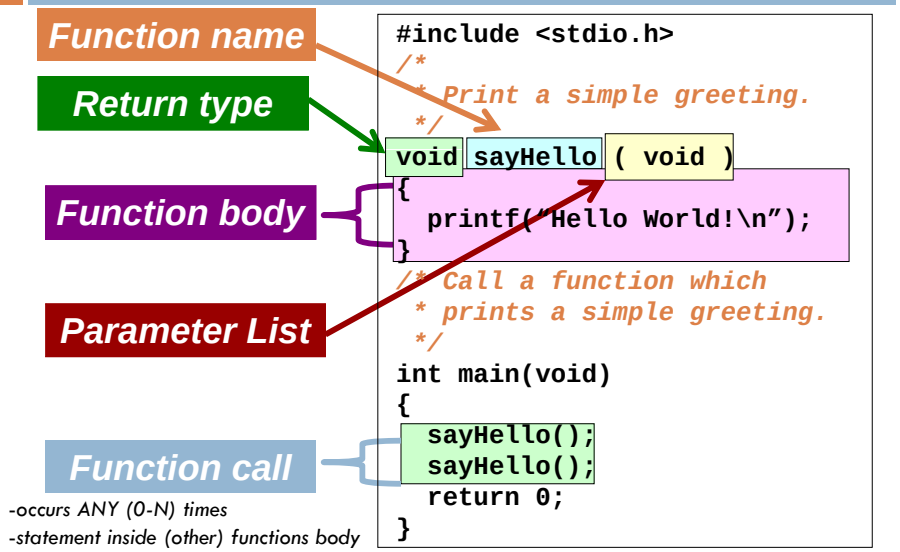
- If the parentheses are missing, the function won't be called:

```
sayHello;    /*** WRONG ***/
```

This statement is legal but has no effect.

Example

33



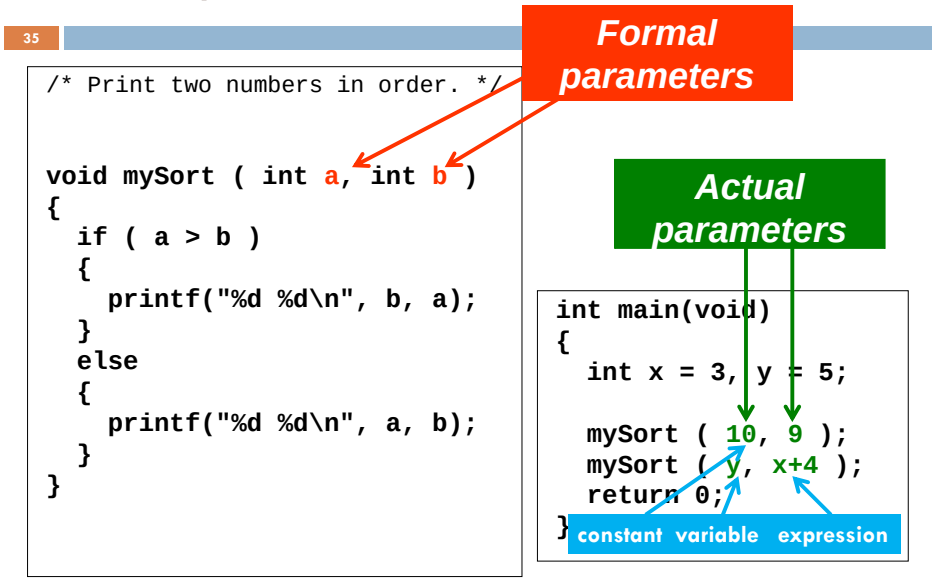
Function Call-Parameters

34

- "Formal" parameters are variables declared in the function declaration.
- "Actual" parameters are values passed to the function when it is called.
  - May be constants, variables, or expressions
- Formal parameters must match with actual parameters in order, number and data type.
- Parameters are passed by copying the value of the actual parameters to the formal parameters.

Example

35



Function Call-Return Value

36

- A call of a `void` function with no return value is always followed by a semicolon to turn it into a statement:

```
print_count(i);
sayHello();
```

- The value returned by a non-`void` function can be assigned to a variable, printed, or used in an expression.
- Example
  - `avg = average(x, y);`
  - `printf("The average is %g\n", average(x, y));`
  - `if (average(x, y) > 0)`
    `printf("Average is positive\n");`

Example 1

37

```
/* Returns the larger of two numbers. */
int max (int a, int b)
{
  int result;

  if (a > b)
  {
    result = a;
  }
  else
  {
    result = b;
  }
  return result;
}
```

Return type

Function call:

```
int val = max(7,5);
```

Note: Type of `val` and return value of the function `max` should be the same.

Example 2

38

```
double CalcArea(double hight, double width)
{
   return height * width;
}
int main()
{
   double h, w;

   printf("Please input height and width\n");
   scanf("%f, %f", &h, &w);
   double area = CalcArea(h, w);
   printf("The area is %f", area);

   return 0;
}
```

Function Declarations

Function prototype

Example 1

40

```
#include <stdio.h>

double average(double a, double b)
{
  return (a + b) / 2;
}

int main(void)
{
  double x, y, z;

  printf("Enter three numbers: ");
  scanf("%f%f%f", &x, &y, &z);
  printf("Average of %g and %g: %g\n", x, y, average(x, y));
  printf("Average of %g and %g: %g\n", y, z, average(y, z));
  printf("Average of %g and %g: %g\n", x, z, average(x, z));

  return 0;
}
```

Function definition

YES Function definition before function call

Function calls

Example 2

41

```
#include <stdio.h>

int main(void)
{
  double x, y, z;

  printf("Enter three numbers: ");
  scanf("%f%f%f", &x, &y, &z);
  printf("Average of %g and %g: %g\n", x, y, average(x, y));
  printf("Average of %g and %g: %g\n", y, z, average(y, z));
  printf("Average of %g and %g: %g\n", x, z, average(x, z));

  return 0;
}

double average(double a, double b)
{
  return (a + b) / 2;
}
```

NO!

Calling a function for which the compiler has not yet seen a declaration or definition is an error.

Function calls

Function definition

Function Declaration

42

- C doesn't require that the definition of a function precede its calls.
  - It may make the program harder to understand by putting its function definitions before its function calls
- Functions must declare before calling (like variables)
  - When the called function is defined after the calling function
- Function declaration is called a "prototype"
- Function prototype provides the compiler with a brief glimpse at a function whose full definition will appear later.

Function Declaration …

43

- General form of a function prototype:

  return-type ***function-name*** **(** ***parameters*** **)** **;**

```
double average(double x, double y);
int max (int a, int b);
```

- A function prototype doesn't have to specify the names of the function's parameters, as long as their types are present:

```
double average (double, double);
int max(int, int);
```

- It's usually best not to omit parameter names.

Note

44

- The function prototype must be consistent with the function definition and function call
- They should all agree in the number, type, and order of parameters and in the type of return value.

- Function Prototype
  - `int my_add_func(int a, int b);`
  - OR `int my_add_func(int, int);`
- Function Call
  - `int result = my_add_func(5, X);`
- Function Header (definition)
  - `int my_add_func(int a, int b)  {…}`

Example 1

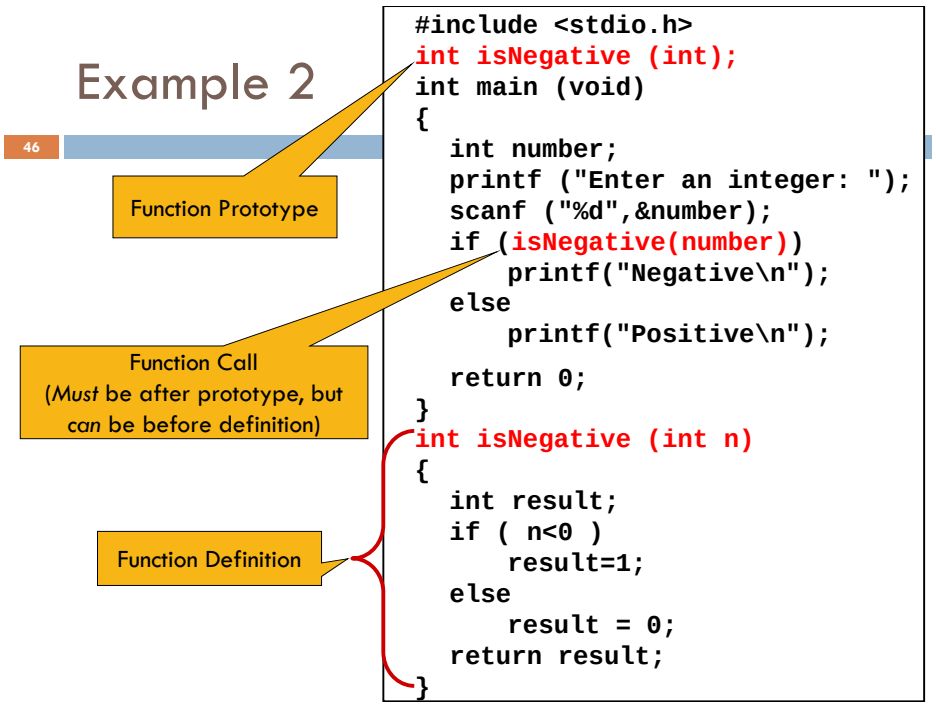

```
#include <stdio.h>

double average(double a, double b);

int main(void)
{
  double x, y, z;

  printf("Enter three numbers: ");
  scanf("%f%f%f", &x, &y, &z);
  printf("Average of %g and %g: %g\n", x, y, average(x, y));
  printf("Average of %g and %g: %g\n", y, z, average(y, z));
  printf("Average of %g and %g: %g\n", x, z, average(x, z));

  return 0;
}

double average(double a, double b)
{
  return (a + b) / 2;
}
```

Function prototype

Function calls

Function definition

Example 2



Local and Global Variables

Symbolic Referencing Issues

- We use symbolic names as a vital part of C programming in order to declare **names** of variables, functions and other items.
- We have already encountered one important rule concerning referencing of symbol names – we ***must declare all names before*** they may be referenced.
- It is necessary to understand all the rules concerning how we may reference symbol names
  - And whether we may even be prohibited from referencing names under certain conditions

Variables

49

- Each variable has name, type, size and value
  - int x = 4; → name= 'x', type= int, size= 4 bytes, value=4
- Local Variables
  - Defined variables in a function body
  - Can only be used inside the function they were declared in.
- Global Variables
  - Created by placing variable declarations outside any function definition
  - Can be referenced by any function that follows their declarations or definitions in the file.

Local Variables

50

- Defined variables in a function body
- Created when the function is active (called)
- Why need local variables?
  - To store temporary information
  - Values are private to the function they were declared in, and not elsewhere.
  - Can't be accessed by other functions
- Arguments (function parameters) are also local variables
  - Each parameter is initialized automatically when a function is called

Example 1

51

```
double CalcMax(double a[10])
{
  int i;
  double maxValue;
  …;
}
int main()
{
  int i;
  double a[10];
  double maxValue;

    maxValue = CalcMax(a) ;
}
```

Local variables

Local variables

Example 2

52

```
#include <stdio.h>
// Function to calculate the nth triangular number
void calculateTriangularNumber ( int n )
{
   int i, triangularNum = 0;
   for ( i = 1; i <= n; ++i )
      triangularNum += i;
   printf ("Triangular num %i is %i\n", n,triangularNum);
}
int main (void)
{
   calculateTriangularNumber (3);
   calculateTriangularNumber (5);
   return 0;
}
```

Local variables

Triangular num 3 is 6
Triangular num 5 is 15

Global Variables

53

- Passing arguments is one way to transmit information to a function.
- Functions can also communicate through **global variables**
- How to define a global variable
  - Same as other variable declaration, except it is outside any function
- Global variables are sometimes known as **external variables.**

Global Variables …

54

- Global variable:
  - A variable that is declared outside the body of any function.
  - A variable does not belong to any function
  - A variable can be accessed by any function in a program
  - A variable can retain its value throughout the execution of the program.
- Why need a global variable
  - avoid passing frequently-used variables continuously throughout several functions
- Make sure that external variables have meaningful names.

Example 1

55

```
#include <stdio.h>

int x;

void f1 (void) {
      x++;
}

void f2 (void) {
      x++;
}

int main(void) {
      x=7;
      f1();
      f2();
      printf("x=%i \n",x);
}
```

Global Variable

Example 1 …

56

```
#include <stdio.h>

int x;

void f1 (void) {
      x++;   // x = 8
}

void f2 (void) {
      x++;   // x = 9
}

int main(void) {
      x=7;
      f1();
      f2();
      printf("x = %i \n",x);
}
```

Global Variable

x = 9

Example 2

57

```
int x;       /* Global variable */
int y = 10;  /* Initialized global variable */

int foo(int z)
{
  int w;   /* local variable */
  x = 42;  /* assign to a global variable */
  w = 10;  /* assign to a local variable */
  return (x % y + z / w);
}
int main(void) {
  printf("foo = %i \n", foo (20));
  return 0;
}
```

foo = 4

Example 3

58

```
#include<stdio.h>
int A; int B;

int Add() {
    int x = 8;
    return A + x;
}
int main() {
  int answer;
  A = 5;
  B = 7;
  answer = Add();
  printf("Answer is %d\n", answer);
  return 0;
 }
```

Global Variables

Local Variables

Local Variables

Answer is 13

Pros and Cons of Global Variables

59

- Global variables are convenient when:
  - Many functions must share a variable
  - OR a few functions share a large number of variables.
- In most cases, it's better for functions to communicate through parameters rather than by sharing variables:
  - If we change a global variable during program maintenance (by altering its type, say), we'll need to check every function in the same file to see how the change affects it.
  - If a global variable is assigned an incorrect value, it may be difficult to identify the guilty function.
  - Functions that rely on global variables are hard to reuse in other programs.

Variable Scopes

Scope

61

- In a C program, the same variable may have several different meanings.
- A Scope defines:
  - when the variable is VISIBLE
  - what the meaning of the variable is
- A variable that is out of scope means it is not accessible (may be it no longer exists!).
- C's scope rules enable the programmer (and the compiler) to determine which meaning is relevant at a given point in the program.
- Scope is sometimes referred to as a name space.

Scope Rules

62

- The **Scope** of a variable refers to the part(s) of a program where
  - It is valid to reference the variable
  - The variable is visible or accessible
- There are four aspects of symbolic name scope
  - Function scope
  - File scope
  - Block scope
  - Function-Prototype scope

Function Scope

63

- Variable names that are declared within functions have function scope
  - Such variables may be referenced only within the function in which they are declared
  - References from outside the function generate compiler errors, as if the variable name had never been declared

```
int  FuncName (  ......  )
{
  int A, B ;  /* function scope */
  float  X ;  /* function scope */

  printf ( "&d &d &f",  A, B, X ) ;
}
```

All references to A, B or X must stay "within the box" of the function definition.

Example 1

64

```
int F1 (void) {
     int X = 5 ;   //Local variable
     return X ;
}
int main ( ) {
     printf ( "%d", X ) ;
     return 0 ;
}
```

Does NOT compile!

X is a local variable with a function scope→ All references to X must stay "within the box" of the function definition.

Example 2

65

```
int F1 (void) {
     int X = 5 ;    //Local variable
     return X ;
}
int main ( ) {
    int X = 5;
    printf ( "%d", X ) ; // Output:   5
    return 0 ;
}
```

Two different versions of X exist :
--One variable being the "X that exists only inside of F1"
--One variable being the "X that exists only inside of main".

File Scope

66

- Global (external) variables can be declared **outside** of any functions
- Having ***file scope*** means that a variable is visible from its point of declaration to the end of the enclosing file
- Global variable names that are declared before function definitions may be referenced directly from within those functions
- **Global** (or external) variables have **File scope**

Example

```
#include <stdio.h>
int A = 4;
float X = 6.5;
char C;
int Func1 ( ) {
   return  A +
}
float Func2 ( int Z )
{
   X = X - Z ;
   return  3.0 * X ;
}
char Func3 ( ) {
  return (C = 'w');
}
```

File scope

Function scope

```
int main ( ) {
   int B ;
   float Y;
   B = Func1( );
   printf("A= %d,B=%d\n",A,B);
   Func3( );
   printf( "C = %c\n", C );
   Y = Func2( X - 0.5 );
   printf("X = %.1f,Y=%.1f\n",
                          X,Y);
   retrun 0;
}
```

OUTPUT: A= 4 , B= 5
C= w
X= 0.5 , Y= 1.5

Block

68

- Compound statement is more than one statement grouped together with in a braces
  { *statements* }
- C allows compound statements to contain declarations as well as statements:
  { *declarations statements* }
- This kind of compound statement is called a ***block.***
- The body of a function is a block.
- Blocks are also useful inside a function body when we need variables for temporary use.

Block Scope

69

- Block scope provides a specific programming mechanism for strongly enforcing the principle of ***localization*** of variable referencing and algorithmic logic
- Variables defined inside a block have block scope.
- Block scope ends at the terminating right brace (}) of the block.
- The most important scope rule
  - When blocks are nested, and a variable in an outer block has the same name as a variable in an inner block, the variable in the outer block is temporarily "hidden" until the inner block terminates.
  - At the end of the block, the variable regains its old meaning.

Block Scope …

70

- Local variables have **Block Scope.**
  - A local variable is visible from its point of declaration to the end of the enclosing function body.
- Any block may contain variable definitions.
- C99 allows variables to be declared anywhere within a block.
- It's possible for a lc a very small scope:

```
void f(void)
{
  …
  int i;
  …
}
```

scope of i

Example – output?

71

```
int main ( ) {
    int X = 3, K = 10 ;
    printf( "%d %d\n", K, X );
    {
       int K ;
       for( K = 0 ; K < 4 ; K++ )
       {
            X = X + K ;
            printf( "%d %d\n", K, X ) ;
       }
     }
    printf( "%d %d\n",K,X );
    return 0 ;
}
```

Function scope

Block scope

OUTPUT:

```
10 3
0 3
1 4
2 6
3 9
10 9
```

When blocks are nested, and a variable in an outer block has the same name as a variable in an inner block, the variable in the outer block is temporarily "hidden" until the inner block terminates.

Function-prototype Scope

72

- Any symbol names declared within the prototype of a function may only be referenced within the prototype
- This is **not** the same as a function reference where a symbolic variable name identified in the function interface of the definition is referenced within the function body
- If a name is used in the parameter list of a function prototype, the compiler ignores the name.
- Identifiers used in a function prototype can be reused elsewhere in the program without ambiguity.
- Example:

```
int Func ( float X ) ; /* X is not required! */
int main ( ) {....}
int Func ( float Y ) {....} /* Y is required */
```

Scope resolution rules

73

- Where is variable x?

1. Search in the current block {…} for a declaration of x
2. if you find it, then use it!
3. if you don't find it in the current block then search in the next outer block…
4. repeat steps 2 and 3 until you find it!
5. if you reach the outer most block (file level) and you don't find a declaration… then it must be an undeclared variable! ERROR!

Example 1

74

```
int i ;              /* Declaration 1 */
void f(int i )       /* Declaration 2 */
{
  i = 1;
}

void g(void)
{
  int i = 2;         /* Declaration 3 */
  if (i > 0) {
    int i ;          /* Declaration 4 */
    i = 3;
  }
  i = 4;
}

void h(void)
{
  i = 5;
}
```

- Declaration 1, `i` is a variable with file scope.
- Declaration 2, `i` is a parameter with block scope.
- Declaration 3, `i` is a variable with block scope.
- Declaration 4, `i` is also a variable with block scope.
- C's scope rules allow us to determine the meaning of `i` each time it's used (indicated by arrows).

Example 2

75

```
#include <stdio.h>
int f1(int x)
{
  x = 2;
  printf("Out1 = %d\n",x);
  return(x+1);
}
int main()
{
  int x = 4, y;
  y = f1(x);
  printf("Out2 = %d\n",x);
  printf("Out3 = %d\n",y);
  return 0;
}
```

Example 2 …

76

```
#include <stdio.h>
int f1(int x1)   //x1 is a local variable for f1
{
  x1 = 2;
  printf("Out1 = %d\n",x1);
  return(x1+1);
}
int main()
{
  int x = 4, y; // local variables of main
  y = f1(x);
  printf("Out2 = %d\n",x);
  printf("Out3 = %d\n",y);
  return 0;
}
```

Out1 = 2
Out2 = 4
Out3 = 3

Storage Classes

Automatic

Static

Storage Classes

- The period during which the identifier exists in *memory*.
- In C programs, variables fall into two categories of storage class depending on their **duration** (lifetime)
- Automatic storage duration (short lifetime)
  - Variables exist in memory for only a brief period
  - `auto` and `register`
- Static storage duration (long lifetime)
  - Variables exist for the entire time the program itself exists in RAM
  - `static` and `extern`

Automatic Storage Class - `auto`

- Automatic storage duration is used for identifiers that exist only briefly.
- All local variables that are defined inside of any function (unless explicitly defined otherwise) have ***automatic storage duration***
- Auto storage is the **default** storage class.
  - `auto int N ;` *is the same as* `int N ;`
- Storage, space in RAM, is "automatically" allocated when the enclosing function is called.
- Storage space is "automatically" deallocated when the function returns (leaving the function).

Automatic Storage Class - `register`

- Register storage is a special class and refers to data that is stored in CPU **hardware registers**
  - `register int N ;`
- In this example, N does not refer to a RAM location, rather a CPU register location. Data is moved to the register and subsequently manipulated at that location
  - This avoids the need to move data back and forth between RAM and CPU
- The register should only be used for variables that require quick access such as counters
- Both auto and register storages are sometimes called **local** identifiers.

Static Storage Class

81

- Static storage duration refers to variables whose lifetime is as long as the program's lifetime.
- Static variables are allocated ***permanent*** storage in RAM when programs are first loaded into memory.
- **Global variables** have **static** storage class
  - They are allocated RAM storage locations when the program is loaded and the data stored is **persistent** for the lifetime of the program itself.

Static Storage Class - `static`

82

- A local variable that belongs to a block can be declared `static` to give it **static storage duration**.
- Define static local variable: `static int N = 5 ;`
  - Once a value is assigned to a static storage (N), the value stays even if the function is exited and then re-entered.
  - Any changes in N are also kept until the next reference or modification.
  - Static local variables are **destroyed** (deallocated) only when the program **terminates**.
- A **static local variable** still has **block scope**, so it's not visible to other functions.

Example

83

```
#include <stdio.h>
int add2 ( void ) ;
int main ( )
{
   int N ;
   N = add2() ;
   printf( "%d\n", N );      /* outputs value of N as 2 */
   N = add2() ;
   printf( "%d\n", N );      /* outputs value of N as 4 ! */
   return 0 ;
}
int add2 ( void )
{
   static int A = 0;
   A += 2 ;
   return A ;
}
```

A is initialized to 0 only the first time that the function add2 is called. Thereafter, 2 is added each time.

Static Storage Class - `extern`

84

- Used for global variables
- Extern storage refers to a symbol which, like `static`, is permanent for the execution lifetime of the program.
- Such identifiers permit compilers and linkers to reference identifiers that may be declared inside of other compiled program modules
  - `extern int N ;`
- Both static and extern identifiers are sometimes called global identifiers

Important Note

- Default properties of local variables:
  - Automatic storage duration.
  - Block scope.
- Properties of global variables:
  - Static storage duration
  - File scope

Example


```
int i ;              /* Declaration 1 */
void f(int i )       /* Declaration 2 */
{
  i = 1;
}

void g(void)
{
  int i = 2;         /* Declaration 3 */
  if (i > 0) {
    int i ;          /* Declaration 4 */
    i = 3;
  }
  i = 4;
}

void h(void)
{
  i = 5;
}
```

- Declaration 1, `i` is a global variable with **static** storage duration and file scope.
- Declaration 2, `i` is a parameter with an **automatic** storage and block scope.
- Declaration 3, `i` is an **automatic** local variable with block scope.
- Declaration 4, `i` is also **automatic** local variable and has block scope.

Passing Parameters

Call by Value Parameters

Call by Reference Parameters

Passing Parameters to a Function

- Call by Value (default)
  - When the argument is the name of a variable
  - **A copy of the argument's value** is sent to the function to view/edit it locally.
  - The outcome of editing the copy does not affect the original.
    - Any changes made to the parameter during the execution of the function don't affect the argument.
- Call by Reference

Example

89

```
/* Swap the values of two
   variables. */
void mySwap ( int a, int b )
{
  int temp;

  temp = a;
  a = b;
  b = temp;

  printf("%d %d\n", a, b);
}
```

```
int main(void)
{
  int a = 3, b = 5;

  printf("%d %d\n",a,b);
  mySwap ( a, b );
  printf("%d %d\n",a,b);

  return 0;
}
```

Example ...

90

```
/* Swap the values of two
   variables. */
void mySwap ( int a, int b )
{
  int temp;

  temp = a;
  a = b;
  b = temp;

  printf("%d %d\n", a, b);
}
```

```
int main(void)
{
  int a = 3, b = 5;

  printf("%d %d\n",a,b);
  mySwap ( a, b );
  printf("%d %d\n",a,b);

  return 0;
}
```

Output:

```
3  5
```

Example ...

91

```
/* Swap the values of two
   variables. */
void mySwap ( int a, int b )
{
  int temp;

  temp = a;
  a = b;
  b = temp;

  printf("%d %d\n", a, b);
}
```

```
int main(void)
{
  int a = 3, b = 5;

  printf("%d %d\n",a,b);
  mySwap ( a, b );
  printf("%d %d\n",a,b);

  return 0;
}
```

Output:

```
3 5
5 3
```

Example ...

92

```
/* Swap the values of two
   variables. */
void mySwap ( int a, int b )
{
  int temp;

  temp = a;
  a = b;
  b = temp;

  printf("%d %d\n", a, b);
}
```

```
int main(void)
{
  int a = 3, b = 5;

  printf("%d %d\n",a,b);
  mySwap ( a, b );
  printf("%d %d\n",a,b);

  return 0;
}
```

Output:

```
3 5
5 3
3 5
```

Example …


```
/* Swap the values of two
   variables. */
void mySwap ( int a, int b )
{
  int temp;

  temp = a;
  a = b;
  b = temp;

  printf("%d %d\n", a, b);
}
```

```
int main(void)
{
  int a = 3, b = 5;

  printf("%d %d\n",a,b);
  mySwap ( a, b );
  printf("%d %d\n",a,b);

  return 0;
}
```

Called function's environment: `a: 5  b: 3`

Calling function's environment: `a: 3  b: 5`

Changes made in function swap are lost when the function execution is over

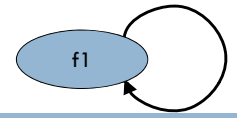
Passing Parameters to a Function

- Call by Value (default)
  - When the argument is the name of a variable
  - **A copy of the argument's value** is sent to the function to view/edit it locally.
  - The outcome of editing the copy does not affect the original.
    - Any changes made to the parameter during the execution of the function don't affect the argument.
- Call by Reference
  - When the argument is a pointer variable such as an array
  - **A copy of the 'memory address'** or pointer that tells the function where to find the original value is sent to the function
  - The outcome of editing the copy affects the original.
  - we will discuss this topic further when we talk about pointers.

Recursion

Recursive Functions

Recursion



- A function is ***recursive*** if it calls itself.
- A recursive function consists of two types of cases
  - A base case(s) and
  - A recursive case
- The base case is a small problem
  - The solution to this problem should not be recursive, so that the function is guaranteed to terminate
  - There can be more than one base case
- The recursive case defines the problem in terms of a smaller problem of the same type
  - The recursive case includes a recursive function call
  - There can be more than one recursive case

Recursive function

- Steps Leading to Recursive Solutions
  - How can the problem be defined in terms of smaller problems of the same type?
  - What is the base case that can be solved without recursion?
  - Will the base case be reached as the problem size is reduced?

```
If this is a simple case
    solve it
else
    redefine the problem using recursion
```

Example 1 - Factorial

- 1! = 1
- 2! = 2 x 1
- 3! = 3 x 2 x 1
- 4! = 4 x 3 x 2 x 1
- 5! = 5 x 4 x 3 x 2 x 1

Example 1 - Factorial

- 1! = 1 = 1 *Base case*
- 2! = 2 x 1 = 2 x 1!
- 3! = 3 x 2 x 1 = 3 x 2!
- 4! = 4 x 3 x 2 x 1 = 4 x 3!
- 5! = 5 x 4 x 3 x 2 x 1 = 5 x 4!

$n! = n*(n-1)!$

Example 1- Factorial …

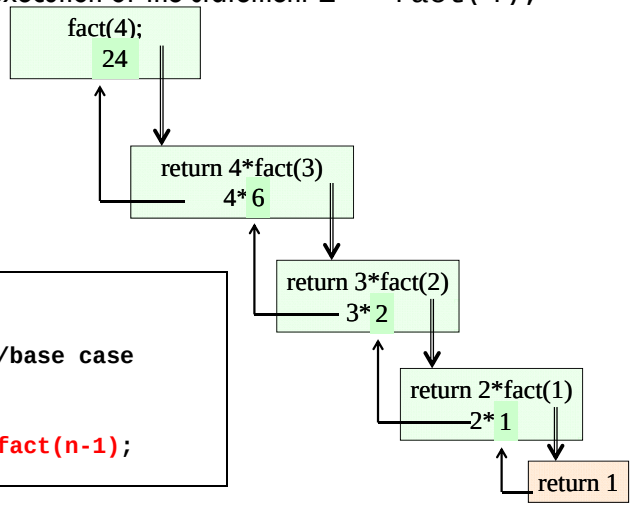
$fact(n) = n*(n-1)*(n-2)* \ldots *1$

$$fact(n) = \begin{cases} 1 & \text{if } n=1 \\ n * fact(n-1) & \text{if } n>1 \end{cases}$$

```
int fact(int n)
{
  if (n == 1) //base case
    return 1;
  else
    return n * fact(n-1); //recursive case
}
```

Example 1- Factorial …

- Trace the execution of the statement `i = fact(4);`




```
int fact(int n)
{
  if (n == 1) //base case
    return 1;
  else
    return n * fact(n-1);
}
```

Example 1- Factorial …

- The factorial can be calculated iteratively and recursively:

Iterative function n! =n×(n-1)×…×2× 1

```
int fact(int n)
{
   int factorial = 1;
   for (int i=n; i>1; i--)
      factorial *= i;
   return factorial;
}
```

Recursive function *n*! = *n* × (*n* – 1)!

```
int fact(int n)
{
  if (n == 1) //base case
    return 1;
  else
    return n * fact(n-1);
}
```

Example 2: Sum

- Can we write loops recursively?
- Try this function using recursion:

```
/* calculate the sum of n + (n-1) + … + 1 */
int Sum(int n)
{
  int total = 0;
  for (int c = n; c >= 1; c--)
  {
     total = total + c;
  }
  return total;
}
```

Example 2: Sum …


```
/* calculate Sum(n)= n + (n-1) + … + 1 */

int Sum(int n)
{
  int total = 0;
  for (int c = n; c >= 1; c--)
  {
    total = total + c;
  }
  return total;
}
```

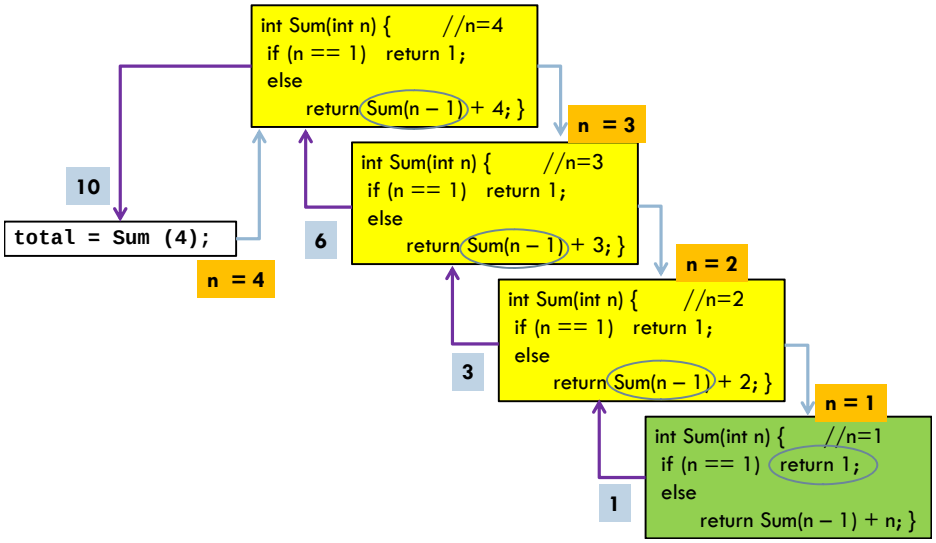
Iterative

Recursive

```
/* calculate Sum(n) = n + sum (n-1)*/

int Sum(int n)
{
   if (n == 1)   // base case
      return 1;

   else   // recursive case
     return Sum(n - 1) + n;
}
```

Example2: Sum …

105



Example 3 - x^n

106

- The following recursive function computes x^n, using the formula $x^n = x \times x^{n-1}$.

```
int power(int x, int n)
{
  if (n == 0) // base case
    return 1;
  else
    return x * power(x, n - 1); // recursive case
}
```

Example 4 - Fibonacci

107

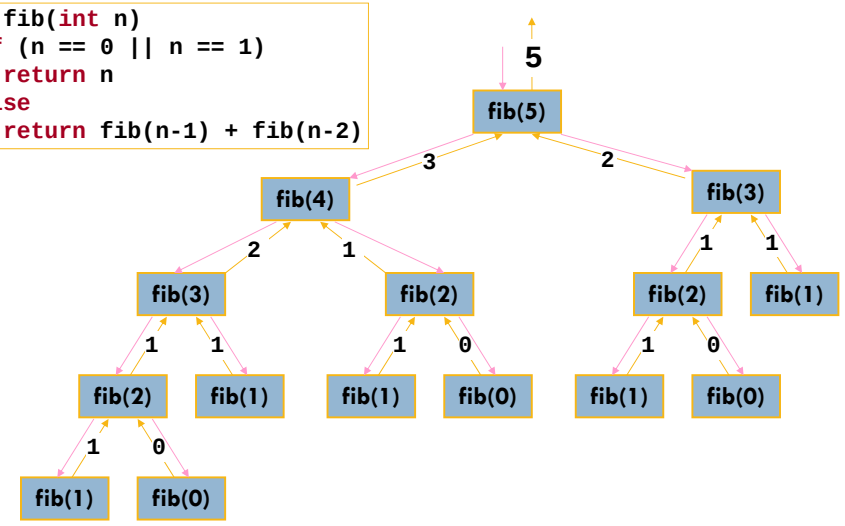
- Sequence {f0,f1,f2,…}. First two values (f0,f1) are 1, each succeeding number is the sum of previous two numbers.
- 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, …
- Base case: `fib(0)=0, fib(1) = 1`
- Recursive step: `fib(n) = fib(n-1)+fib(n-2)`

```
int fib(int n)
  if (n == 0 || n == 1)
    return n
  else
    return fib(n-1) + fib(n-2)
```

Example 4 - Fibonacci ….

108

```
int fib(int n)
  if (n == 0 || n == 1)
    return n
  else
    return fib(n-1) + fib(n-2)
```



Question 1

- f(1) = 1,
- f(2) = 2,
- f(n) = 2f(n-1) +f(n-2) , for n>=3
- What is f(5)?

n	f(n)
1	1
2	2
3	2 x f(2) + f(1) = 2 x 2 + 1 = 5
4	2 x f(3) + f(2) = 2 x 5 + 2 = 12
5	2 x f(2) + f(1) = 2 x 12 + 5 = **29**

Question 2 - What does the program do?


```
1  #include <stdio.h>
2
3  unsigned int mystery( unsigned int a, unsigned int b ); // function prototype
4
5  // function main begins program execution
6  int main( void )
7  {
8     unsigned int x; // first integer
9     unsigned int y; // second integer
10
11    printf( "%s", "Enter two positive integers: " );
12    scanf( "%u%u", &x, &y );
13
14    printf( "The result is %u\n", mystery( x, y ) );
15 } // end main
16
17 // Parameter b must be a positive integer
18 // to prevent infinite recursion
19 unsigned int mystery( unsigned int a, unsigned int b )
20 {
21    // base case
22    if ( 1 == b ) {
23       return a;
24    } // end if
25    else { // recursive step
26       return a + mystery( a, b - 1 );
27    } // end else
28 } // end function mystery
```

Recursion vs. Iteration …

Similarities	Iteration	Recursion
based on a control structure	repetition structure	selection structure.
Involving repetition	explicitly uses a repetition structure	Achieves through repeated function calls
involving a termination test	when the loop-continuation condition fails	when a base case is recognized.
occurring infinitely	if the loop-continuation test never becomes false	if the recursion step never converges on the base case.

Lecture 2: Summary

- C Library Functions
- User Defined Functions
- Storage Classes and Scope Rules
- Recursion
- Assigned Reading:
  - Chapter 5 – C Functions
  - Chapter 6 – C Arrays
- Assignment
  - Deadline of first assignment is Jan. 29.
  - Labs begin this week.