Introduction

- **Function**
  - A self-contained program segment that carries out some specific, well-defined task.

- **Some properties:**
  - Every C program consists of one or more functions.
    - One of these functions must be called "`main`".
    - Execution of the program always begins by carrying out the instructions in "`main`".
  - A function will carry out its intended action whenever it is *called* or *invoked*.

- In general, a function will process information that is passed to it from the calling portion of the program, and returns a single value.
  - Information is passed to the function via special identifiers called *arguments* or *parameters*.
  - The value is returned by the "`return`" statement.

- Some functions may not return anything.
  - Return data type specified as "`void`".

```c
#include  <stdio.h>

int  factorial (int m)
{
    int i, temp=1;
    for (i=1; i<=m; i++)
    temp = temp * i;
    return (temp);
}
```

```c
main()
{
    int  n;
    for  (n=1; n<=10; n++)
          printf ("%d! = %d \n",
              n, factorial (n) );
}
```

Output:

1! = 1

2! = 2

3! = 6 …….. upto 10!

Why Functions?

- **Functions**
  - **Allows one to develop a program in a modular fashion.**
    - **Divide-and-conquer approach.**
  - **All variables declared inside functions are local variables.**
    - **Known only in function defined.**
    - **There are exceptions (to be discussed later).**
  - **Parameters**
    - **Communicate information between functions.**
    - **They also become local variables.**

- **Benefits**
  - **Divide and conquer**
    - Manageable program development.
    - Construct a program from small pieces or components.
  - **Software reusability**
    - Use existing functions as building blocks for new programs.
    - Abstraction: hide internal details (library functions).

Defining a Function

- **A function definition has two parts:**
  - **The first line.**
  - **The body of the function.**

```
return-value-type  function-name  ( parameter-list )
{
   declarations and statements
}
```

- The first line contains the return-value-type, the function name, and optionally a set of comma-separated arguments enclosed in parentheses.
  - Each argument has an associated type declaration.
  - The arguments are called *formal arguments* or *formal parameters*.

- Example:

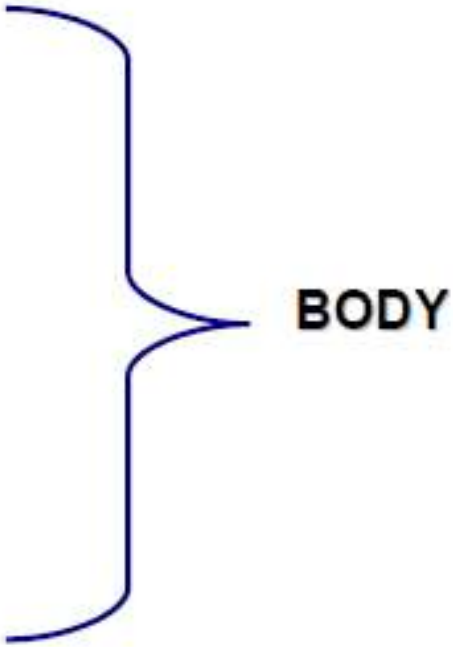
```
int gcd (int A, int B)
```

- The argument data types can also be declared on the next line:

```
int gcd (A, B)
{ int A, B; ----- }
```

- **The body of the function is actually a compound statement that defines the action to be taken by the function.**

```
int  gcd  (int A, int B)
{
   int  temp;
   while ((B % A) != 0)  {
     temp = B % A;
     B = A;
     A = temp;
   }
   return (A);
}
```



- When a function is called from some other function, the corresponding arguments in the function call are called *actual arguments* or *actual parameters*.
  - The formal and actual arguments must match in their data types.
  - The notion of positional parameters is important

- Point to note:
  - The identifiers used as formal arguments are "local".
    - Not recognized outside the function.
    - Names of formal and actual arguments may differ.

```
#include  <stdio.h>
/* Compute the GCD of four numbers */

main()
{
   int  n1, n2, n3, n4, result;
   scanf ("%d %d %d %d", &n1, &n2, &n3, &n4);
   result  =  gcd ( gcd (n1, n2), gcd (n3, n4) );
   printf ("The GCD of %d, %d, %d and %d is %d \n",
            n1, n2, n3, n4, result);
}
```

Function Not Returning Any Value

- **Example: A function which prints if a number is divisible by 7 or not.**

```
void  div7 (int n)
{
   if  ((n % 7) == 0)
        printf ("%d is divisible by 7", n);
   else
        printf ("%d is not divisible by 7", n);

   return;
}
```

OPTIONAL

- **Returning control**
  - **If nothing returned**
    - `return;`
    - or, until reaches right brace
  - **If something returned**
    - `return` *expression*`;`

Some Points

- A function cannot be defined within another function.
  - All function definitions must be disjoint.
- Nested function calls are allowed.
  - A calls B, B calls C, C calls D, etc.
  - The function called last will be the first to return.
- A function can also call itself, either directly or in a cycle.
  - A calls B, B calls C, C calls back A.
  - Called *recursive call* or *recursion*.

Example:: `main` calls `ncr`, `ncr` calls `fact`

```
#include  <stdio.h>

int ncr (int n, int r);
int fact (int n);

main()
{
    int i, m, n, sum=0;
    scanf ("%d %d", &m, &n);

    for (i=1; i<=m; i+=2)
       sum = sum + ncr (n, i);

    printf ("Result: %d \n", sum);
}
```

```
int  ncr (int n, int r)
{
    return (fact(n) / fact(r) /
    fact(n-r));
}

int  fact (int n)
{
    int  i, temp=1;
    for (i=1; i<=n; i++)
       temp *= i;
    return (temp);
}
```

Variable Scope

```
#include  <stdio.h>
int A;
void main()
{  A = 1;
   myProc();
   printf ( "A = %d\n", A);
}

void myProc()
{    int A = 2;
     while( A==2 )
   {
      int A = 3;
      printf ( "A = %d\n", A);
      break;
   }
   printf ( "A = %d\n", A);
}
```

Output:

A = 3

A = 2

A = 1

Math Library Functions

- **Math library functions**
  - perform common mathematical calculations

```
#include <math.h>
```

- **Format for calling functions**

```
FunctionName (argument);
```

  - If multiple arguments, use comma-separated list

```
printf ("%f", sqrt(900.0));
```

  - Calls function *sqrt*, which returns the square root of its argument.
  - All math functions return data type *double*.
  - Arguments may be constants, variables, or expressions.

Math Library Functions

double ceil(double x) - Get smallest integral value that exceeds x.
double floor(double x) – Get largest integral value less than x.
double exp(double x) - Compute exponential of x.
double fabs (double x) – Compute absolute value of x.
double log(double x) - Compute log to the base e of x.
double log10 (double x) – Compute log to the base 10 of x.
double pow (double x, double y) - Compute x raised to the power y.
double sqrt(double x) – Compute the square root of x.

Function Prototypes

- Usually, a function is defined before it is called.
  - `main()` is the last function in the program.
  - Easy for the compiler to identify function definitions in a single scan through the file.

- However, many programmers prefer a top-down approach, where the functions follow `main()`.
  - Must be some way to tell the compiler.
  - Function prototypes are used for this purpose.
    - Only needed if function definition comes after use.

- **Function prototypes are usually written at the beginning of a program, ahead of any functions (including *main()*).**

- **Examples:**

```
int  gcd (int A, int B);
void div7 (int number);
```

  - **Note the semicolon at the end of the line.**
  - **The argument names can be different; but it is a good practice to use the same names as in the function definition.**

Call by Value (Random No. Generation)

- **rand function**
  - Prototype defined in `<stdlib.h>`
  - Returns "random" number between `0` and `RAND_MAX`

```
i = rand();
```

  - Pseudorandom
  - Preset sequence of "random" numbers
    - Same sequence for every function call
- **Scaling**
  - To get a random number between `1` and `n`

```
1 + (rand() % n )
```

  - To simulate the roll of a dice:

```
1 + (rand() % 6)
```

Random Number Generation: Contd.

- `srand` function
  - Prototype defined in `<stdlib.h>`.
  - Takes an integer seed, and randomizes the random number generator.

```
srand (seed);
```

```c
/* A programming example
   Randomizing die-rolling program */
#include <stdlib.h>
#include <stdio.h>

int main()
{
   int i;
   unsigned seed;

   printf( "Enter seed: " );
   scanf( "%u", &seed );
   srand( seed );

   for ( i = 1; i <= 10; i++ ) {
      printf( "%10d ", 1 + ( rand() % 6 ) );

      if ( i % 5 == 0 )
         printf( "\n" );
   }

   return 0;
}
```

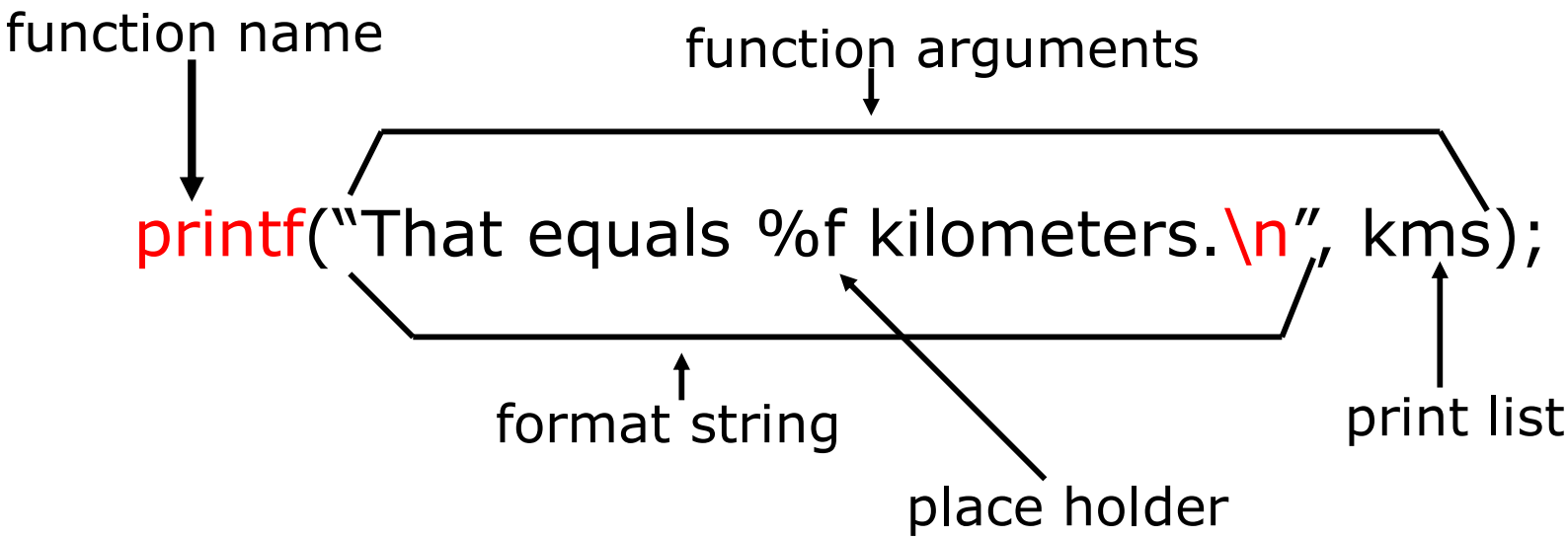
Program Output

```
Enter seed: 67
          6          1          4          6          2
          1          6          1          6          4
```

```
Enter seed: 867
          2          4          6          1          6
          1          1          3          6          2
```

```
Enter seed: 67
          6          1          4          6          2
          1          6          1          6          4
```

The printf Function



Conversion Specification	utput
%a	Floating-point number, hexadecimal digits and p-notation (C99).
%A	Floating-point number, hexadecimal digits and P-notation (C99).
%c	Single character.
%d	Signed decimal integer.
%e	Floating-point number, e-notation.
%E	Floating-point number, e-notation.
%f	Floating-point number, decimal notation.
%g	Use %f or %e, depending on the value. The %e style is used if the exponent is less than –4 or greater than or equal to the precision.
%G	Use %f or %E, depending on the value. The %E style is used if the exponent is less than –4 or greater than or equal to the precision.
%i	Signed decimal integer (same as %d).
%o	Unsigned octal integer.
%p	A pointer.
%s	Character string.
%u	Unsigned decimal integer.
%x	Unsigned hexadecimal integer, using hex digits 0f.
%X	Unsigned hexadecimal integer, using hex digits 0F.
%%	Prints a percent sign.

- /* width.c -- field widths */

```
#include <stdio.h>
#define PAGES 931
int main(void) {
printf("*%d*\n", PAGES);
printf("*%2d*\n", PAGES);
printf("*%10d*\n", PAGES);
printf("*%-10d*\n", PAGES);
return 0; }
```

```
*931*

*931*

*       931*

*931       *
```

•%d with no modifiers. It produces a field with the same width as the integer being printed. This is the default option.

•The second conversion specification is %2d. This should produce a field width of 2, but because the integer is three digits long, the field is expanded automatically to fit the number.

•The next conversion specification is %10d. This produces a field 10 spaces wide, and, indeed, there are seven blanks and three digits between the asterisks, with the number tucked into the right end of the field.

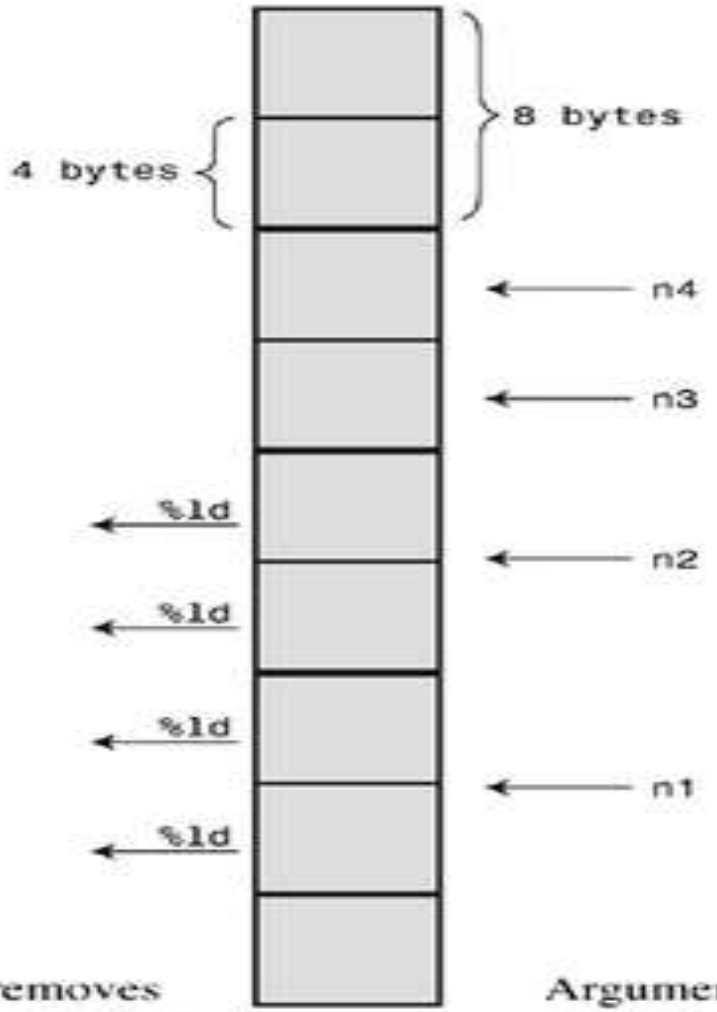
• The final specification is %-10d. It also produces a field 10 spaces wide, and the - puts the number at the left end

```
#include <stdio.h>
int main(void) {
        const double  RENT = 3852.99;
        printf("*%f*\n", RENT);
        printf("*%e*\n", RENT);
        printf("*%4.2f*\n", RENT);
        printf("*%3.1f*\n", RENT);
        printf("*%10.3f*\n", RENT);
        printf("*%10.3e*\n", RENT);
        printf("*%+4.2f*\n", RENT);
        return 0;
}
```

```
*3852.990000*
*3.852990e+03*
*3852.99*
*3853.0*
*   3852.990*
* 3.853e+03*
*+3852.99*
*0003852.99*
```

- Next is the default for %e. It prints one digit to the left of the decimal point and six places to the right.
- Notice how the fourth and the sixth examples cause the output to be rounded off.
- Finally, the + flag causes the result to be printed with its algebraic sign, which is a plus sign in this case, and the 0 flag produces leading zeros to pad the result to the full field width.

```
float n1;  /* passed as type double */
double n2;
long n3, n4;
...
printf("%ld %ld %ld %ld\n", n1, n2, n3, n4);
```



printf() removes values from stack as type long

Arguments n1 and n2 placed on stack as type double values, n3 and n4 as type long

- The printf() function also has a return value;
- it returns the number of characters it printed.
- If there is an output error, printf() returns a negative value.
- (Some ancient versions of printf() have different return values.)

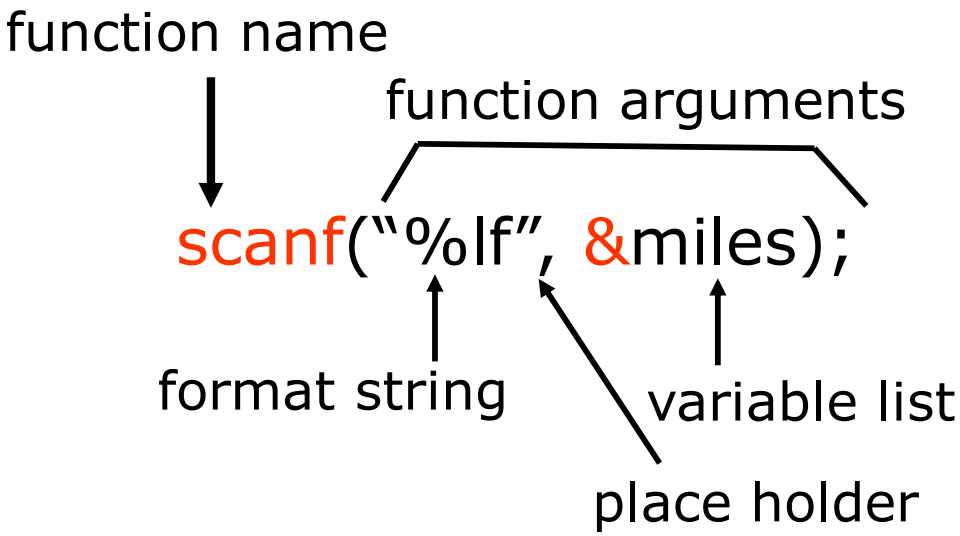
- #include <stdio.h>

```
int main(void)
{
        int  bph2o = 212; int  rv;
        rv = printf("%d F is
        water's boiling point.\n",
        bph2o);
        printf("The printf()
        function printed %d
        characters.\n", rv);
        return 0;
 }
```

The output is as follows:

- O/P:
  212 F is water's boiling point.
  The printf() function printed
  32 characters.

The scanf Function



- When user inputs a value, it is stored in variable `miles`.
- The placeholder type tells the function what kind of data to store into variable miles.

- The & is the C address of operator. The & operator in front of variable `miles` tells the `scanf` function the location of variable miles in memory.

•The scanf() function returns the number of items that it successfully reads.

• If it reads no items, which happens if you type a nonnumeric string when it expects a number, scanf() returns the value 0.

• It returns EOF when it detects the condition known as "end of file." (EOF is a special value defined in the stdio.h file. Typically, a #define directive gives EOF the value –1.)