Programming and Data Structures

Debasis Samanta

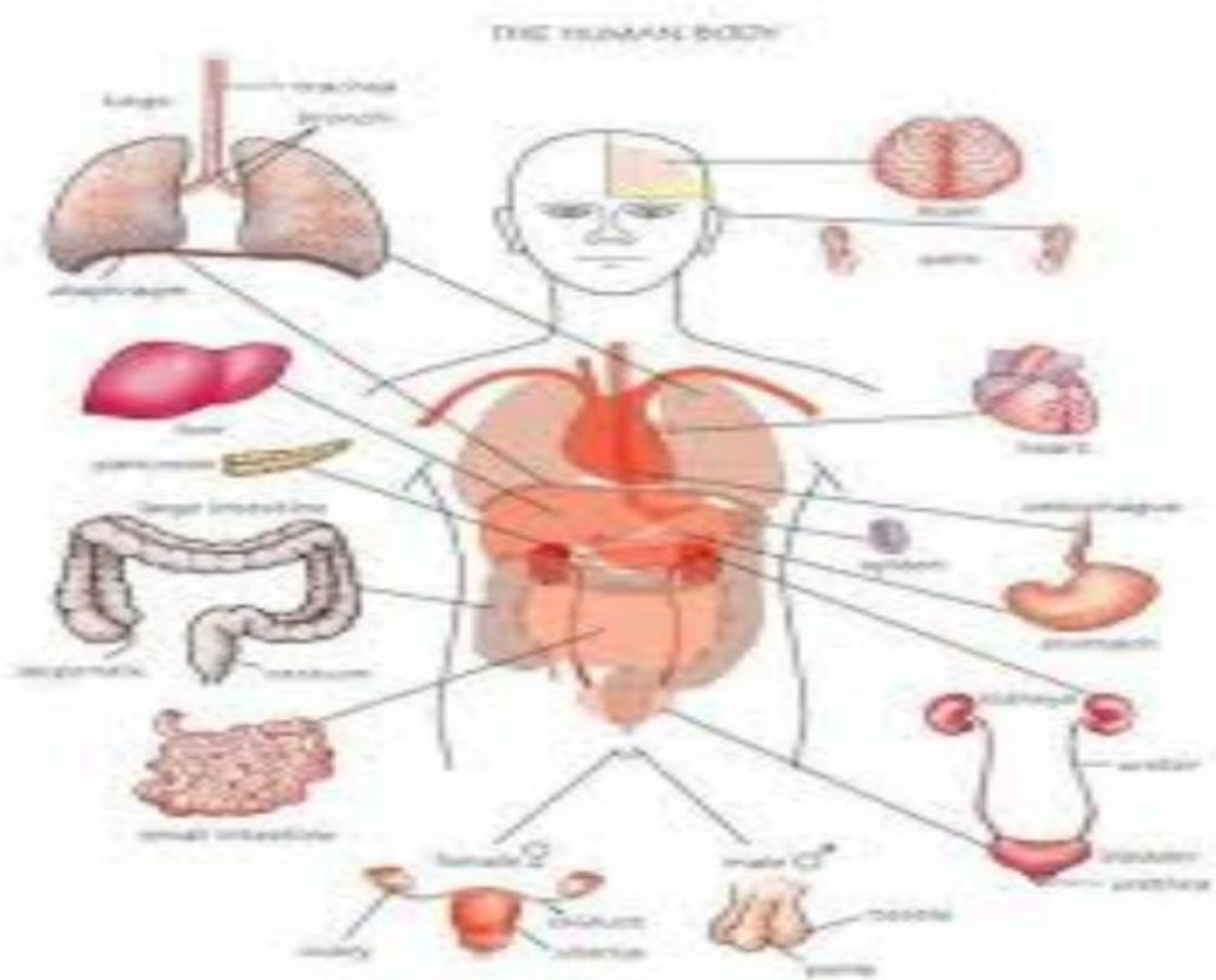
Computer Science & Engineering

Indian Institute of Technology Kharagpur

Spring-2017

Lecture #5

Functions in C


THE HUMAN BODY

Today's discussion…

* Concepts of functions in C
  * Definition
  * Properties
  * Benefits
* Different types of functions
  * Built-in functions
  * User-defined functions
* Scope of variables
* Parameter passing techniques
* Macro definition in C
* Storage class of variables

Concepts of Functions in C

Introduction

- **The C language is termed as function-oriented programming**
- Every C program consists of one or more functions.
  - The concept is based on the "divide-and conquer" policy
    - A large program can be decomposed into a number of relatively smaller segments
      - Easy to code, debug, maintain, etc.
  - In C, there is no limit on the number of such functions in a program.
    - In C programs, any function can call any other function, as many time as it may be.
  - One of these functions, there shall be one must be called "main".
    - Execution of a program always begins by carrying out the instructions in "main".

Properties of C Functions

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

Example

$y = n!$

```
#include  <stdio.h>

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

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!

Whether this `factorial(n)` works, if it is called with parameter value `n = 0`?

Why 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.

Illustration …

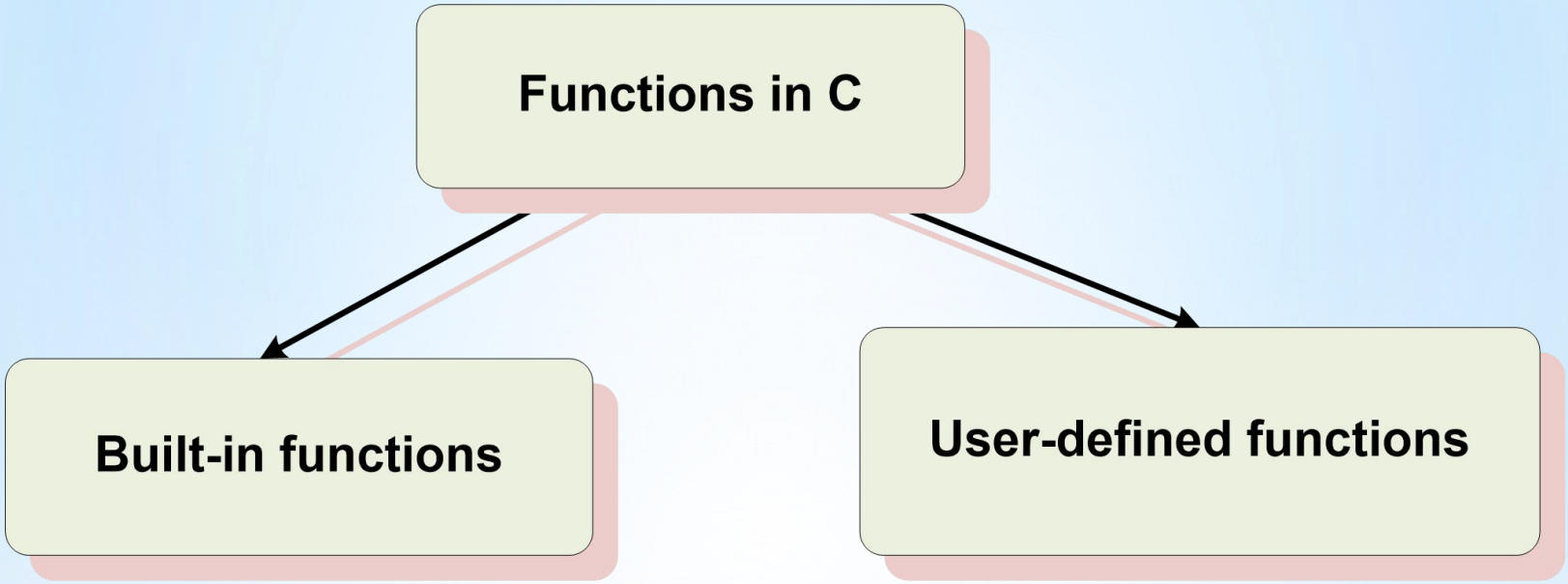
```
#include  <stdio.h>

int count;

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

main()
{
        int  n;
        for  (n=1; n<=10; n++) {
            count = 0;
           printf ("%d! = %d \n",n, factorial (n) );
            printf("Number of multiplications is %d", count);
         }
}
```

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).

Functions in C



- First let us discuss on built-in functions

Built-in Functions in C

Built-in Functions

- Many functions are already there in C language
  - These are defined by the C developer
  - These are called standard library

- Standard library

<assert.h>	<float.h>	**<math.h>**	<stdarg.h>	<stdlib.h>
<ctype.h>	<limits.h>	<setjmp.h>	<stddef.h>	**<string.h>**
<errno.h>	<locale.h>	<signal.h>	**<stdio.h>**	<time.h>

- In all these **header files**, functions, type and macros are declared and defined

Header Files

- **Standard header files**
  - A header file can be accessed by

    ```
    #include <headerFileName.h>
    ```

  - A header must be included outside of any external declaration.
  - A programmer can include as many as headers they need in his program.
  - Headers may be included in any order.

- **Custom header files**
  - Create file(s) with function definitions.
  - Save as `filename.h` (say).
  - Load in other files with `#include "filename.h"`
  - Reuse functions.

Input and Output Functions

- Many functions dealing with input and output to-and-from programs

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

- For file operation
  - To write and read from an external files (stored in secondary memory)

    (We shall study about it in details in a later class.)
- For formatted output
  - `int printf(char *format, ….);`
  - `int sprintf(char *s, char *format, ….);`
- For formatted input
  - `int scanf(char *format, ….);`
  - `int sscanf(char *s, char *format, ….);`

Formatted Output Functions

`printf (…)` function

- The `printf(…)` function is unusual in that it can take a variable number of parameters.
- The function take at least one parameter : a string literal
- This argument contains text and may contain many placeholders
- Number of arguments is decided by the number of placeholders

Examples

```
printf("Hello World");

printf("Hi %s darling! %d years\n", name, age);
```

Formatted Output Functions

`printf (...)` conversion character

Conversion format	Meaning
`c`	Printed as a single character
`d`	Printed as a signed decimal integer
`e`	Printed as a floating-point value with an exponent
`f`	Printed as a floating-point value without an exponent
`g`	Printed as a floating-point value with e- or f-type; trailing zero or trailing decimal point will not be printed
`i`	Printed as a decimal, hexadecimal or octal integer
`o`	Printed as an octal integer, without a leading zero
`s`	Printed as a string
`u`	Printed as an unsigned decimal integer
`x`	Printed as a hexadecimal integer, without leading 0x

Formatted Output Functions

`sprintf (...)` function

- The `sprintf(...)` is the same as the `printf(...)` except that the output is written into the string `s` terminated with `'\0'`.
- `s` must be big enough to hold the result
- The return value is the number of characters written into the string `s`
  - The return count does not include the '\0'.

Examples

```
sprintf(date, "%d-%d-%d", dd, mm, yy);
```

Formatted Input Functions

`scanf (…)` function

- The `scanf(…)` function is the input analog of `printf(…)`.
- This argument contains text and may contain many placeholders
- Number of arguments is decided by the number of placeholders
- The argument for each placeholder must be a pointer, indicating where the corresponding converted input should be stored

Examples

```
scanf("Enter name = %s and age = %d", name, &age);

scanf("%d%d%d", &x, &y, &z);
```

Formatted Input Functions

`scanf (…)` conversion character

Conversion format	Meaning
`c`	Read as a single character
`d`	Read as a signed decimal integer
`e`	Read as a floating-point value with an exponent
`f`	Read as a floating-point value without an exponent
`g`	Read as a floating-point value with e- or f-type; trailing zero or trailing decimal point will not be read
`h`	Read as a short integer
`i`	Read as a decimal, hexadecimal, or octal integer
`o`	Read as an octal integer
`s`	Read as a string followed by a whitespace character, the null character '\0' will automatically be added at the end
`u`	Read as an unsigned decimal integer
`x`	Read as a hexadecimal integer
`[…]`	Read as a string, which may include whitespace characters

Formatted Output Functions

`sscanf (…)` function

- The `sscanf(…)` reads from a string instead of the standard input.
- `It sscanf(char *s, char *format, ..);`
- It scans the string `s` according to the format in `format` (i.e., the conversion specifications) and stores the resulting values through arg1, arg2, …
  - These arguments must be pointers.

Examples

```
sscanf(date, "%d-%d-%d", &dd, &mm, &yy);
scanf(biodata,"%s%s%d", &fName, &lName, &age);
```

Other Input and Output Functions

Function	Meaning
`int fgetc();`	Returns the character (converted to an `int`), which is typed on the keyboard
`int getchar();`	Returns the character (converted to an `int`), which is typed on the keyboard
`int putc(int c);`	Writes the character `c` (converted to an unsigned `char`); it returns the character return or EOF for an error.
`Int putchar(int c);`	Writes the character `c` (converted to an unsigned `char`); it returns the character return or EOF for an error.
`char *gets(char *s, int n);`	Reads an input line into the array `s`; it replaces the terminating newline with '\0'. It returns s or NULL, if an error occurs
`int puts(char *s);`	Writes the string `s` and a newline to stdout; It returns EOF, if an error occurs

Strings Functions

- The C library functions `<string.h>` for manipulating strings
  - It is required to add the line.

```
#include  <string.h>
```

```
char *strcpy (s1, s2);
char *strncpy(s1, s2, n);
char *strcat(s1, s2);
char *strncat(s1, s2, n);
char *strcmp (s1, s2);
char *strncmp(s1, s2, n);
char *strchr(s1, c);
char *strrchr(s1, c);
```

Strings Functions

```
char *strstr (s1, s2);
char *strpbrk(s1, s2);
char *strerror(n);
char *strtok(s1, s2);
int strspn (s1, s2);
int strcspn(s1, s2, n);
int strlen(s1);
```

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 acos(double x) – Compute arc cosine of x.
- double asin(double x) – Compute arc sine of x.
- double atan(double x) – Compute arc tangent of x.
- double atan2(double y, double x) – Compute arc tangent of y/x.
- double cos(double x) – Compute cosine of angle in radians.
- double cosh(double x) – Compute the hyperbolic cosine of x.
- double sin(double x) – Compute sine of angle in radians.
- double sinh(double x) – Compute the hyperbolic sine of x.
- double tan(double x) – Compute tangent of angle in radians.
- double tanh(double x) – Compute the hyperbolic tangent of x.

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.

User-Defined Functions in C

Function Prototypes

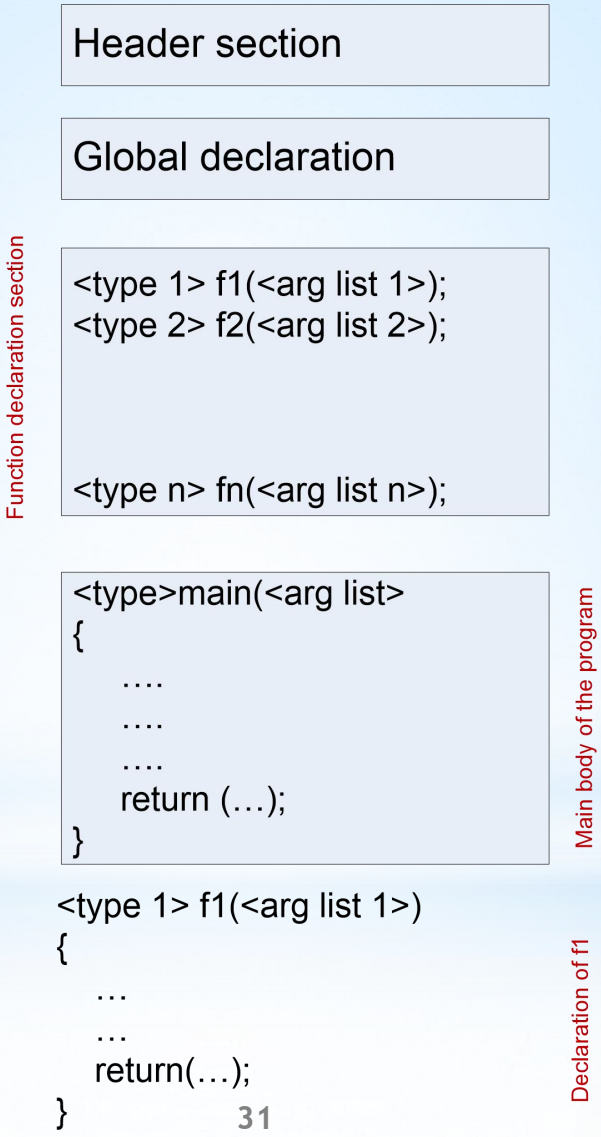
Way 1

- 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.

Way 2

- 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



Function Prototypes

- 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.

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
                …
         statements
                …
        return (…);
}
```

Function: First Line

- 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)
```

- Theoretically, there is no limit on the number of formal parameters in C functions
  - Function name should adhered to the declaration of identifiers in C.

Function: Body

- 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);
}
```

BODY

Function: Call

- 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

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

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

  - If nothing is returned
    - `return;`
    - or, until reaches right parenthesis
  - If something returned
    - `return` *`expression;`* For example, `return(0);`

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: Nested Calls

```
#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);
}
```

Example: Random Number Generation

- **Function generating random number**
  - Prototype defined in `<stdlib.h>`
  - Follows PRNG (Pseudo-Random Number Generator) algorithm
  - `rand( )` function
    - Returns "random" number between `0` and `RAND_MAX`

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

  - To get a random number between `x` and `y`

```
x + (rand() % y )
```

  - `srand( )` function
    - Seeds the random number generator used by the function `rand()`
    - Takes an unsigned integer value as a seed.

```
srand (seed);
```

Example: Randomizing die-rolling program

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

void main()
{
      int i, t;
      time_t t;
      unsigned seed;

    /* Initialize random number generator */
      srand((unsigned)time(&t));

    /* Print 10 random numbers between 1 and 6 */
      for(i=1;i<=10;i++)
      {
         printf("%d",1+rand()%6);
         if (i%5 == 0)printf("\n");
      }
  return 0;
}
```

Output

6 1 4 6 2
1 6 1 6 4

Scope of Variables in C

Scope of Variables

```
#include  <stdio.h>

int A;   1

void myProc()
{     int A = 2;     2
      while( A==2 )
      {
          int A = 3;     4
          printf ( "A = %d\n", A);
          break;
      }
      printf ( "A = %d\n", A);
}

void main()
{
       int A = 1;     3
       myProc();
       printf ( "A = %d\n", A);
}
```

Output!
A = 3
A = 2
A = 1

Scope of Variables

```
#include  <stdio.h>

int A;          1

void myProc()
{     int A = 2;        2
      while( A==2 )
      {
          int A = 3;          4
          printf ( "A = %d\n", A);
          break;
      }
      printf ( "A = %d\n", A);
}

void main()
{
       myProc();
       printf ( "A = %d\n", A);
}
```

Output!
A = 3
A = 2
A = 1

Scope of a Variable : Illustration

```
#include<stdio.h>
void print(int a)
{
   printf("3.1 in function value of a: %d\n",a);
   a+=23;
   printf("3.2 in function value of a: %d\n",a);
}
void main()
{
   int a=10,i=0;
   printf("1. value of a: %d\n",a);
   while(i<1)
   {
      int a;
      a=20;
      printf("2. value of a: %d\n",a);
      i++;
   }
   printf("3. value of a: %d\n",a);
   print(a);
   printf("4. VALUE of a: %d\n",a);
}
```

3.1 in function value of a: 10

3.2 in function value of a: 33

1. value of a: 10

2. value of a: 20

3. value of a: 10

4. value of a: 10

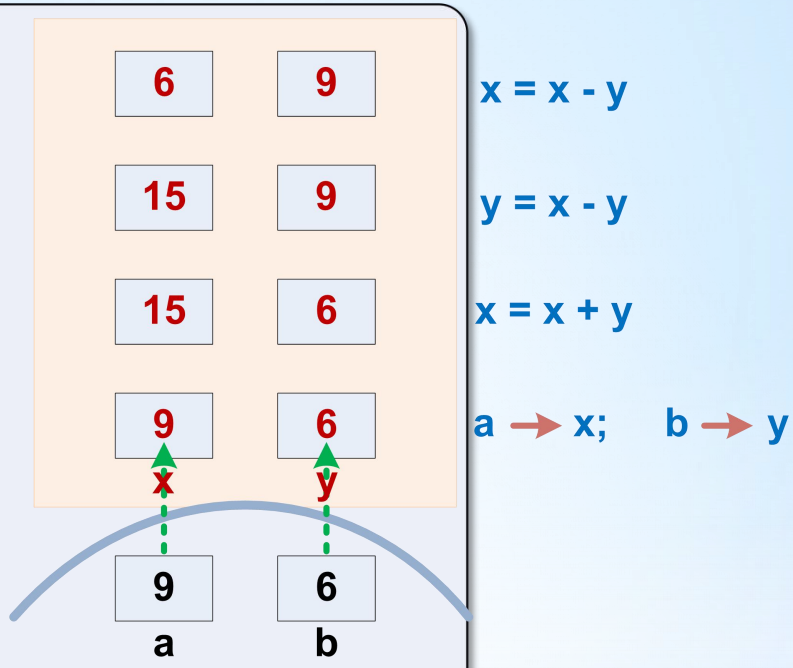
Parameter Passing Techniques in C

Parameter Passing Techniques

- Used when invoking functions.
- **Technique 1: Call by value**
  - Passes the value of the argument to the function.
  - Execution of the function does not affect the original.
  - Used when function does not need to modify argument.
    - Avoids accidental changes.
- **Technique 2: Call by reference**
  - Passes the reference to the original argument.
  - Execution of the function may affect the original.
  - Not directly supported in C – can be effected by using pointers

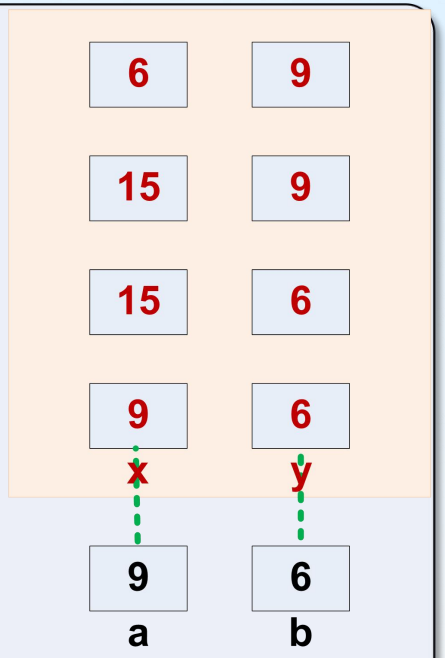
Example: Call by Value

```
#include <stdio.h>

void swap (int x, int y)
{
      x = x + y;
      y = x - y;
      x = x - y;

      return;
}

void main()
{
      int a, b;

      scanf ("a = %d, b = %d", &a, &b);
      swap(a, b);

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



Example: Call by Reference

```
#include <stdio.h>

void swap (int *x, int *y)
{
      *x = *x + *y;
      *y = *x - *y;
      *x = *x - *y;

      return;
}

void main()
{
      int a, b;

      scanf ("a = %d, b = %d", &a, &b);
      swap(&a, &b);

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



x = x - y

y = x - y

x = x + y

&a → &x; &b → &y

Passing Arrays to a Function

- An array name can be used as an argument to a function.
  - Permits the entire array to be passed to the function.
  - Array name is passed as the parameter, which is effectively the address of the first element.

- Rules:
  - The array name must appear by itself as argument, without brackets or subscripts.
  - The corresponding formal argument is written in the same manner.
    - Declared by writing the array name with a pair of empty brackets.
    - Dimension or required number of elements to be passed as a separate parameter.

Example: Minimum of a Set of Numbers

```
#include <stdio.h>

int minimum (int x[], int size)
{
      int i, min = 99999;
      for (i=0; i<size; i++)
         if (min > x[i])
            min = x[i];
return (min);
}

void main()
{
      int a[100], i, n;

      scanf ("%d", &n);
      for (i=0; i<n; i++)
         scanf ("%d", &a[i]);

      printf ("\n Minimum is %d",minimum(a,n);
}
```

We can also write int x[100];
But the way the function is written makes it general; it works with arrays of any size.

Macro Definition in C

#define: Macro Definition

- Pre-processor directive in the following form:

```
#define  string1  string2
```

- Replaces string1 by string2 wherever it occurs before compilation.
  - For example:-

```
#define  PI  3.1415926
```

#define: Macro Definition

- Use `#define` at the header section
  - It treats as global declaration

```
#include <stdio.h>
#define PI 3.1415926

main()
{
   float r = 4.0, area;
   area = PI*r*r;
}
```

```
#include <stdio.h>

main()
{
   float r = 4.0, area;
   area = 3.1415926*r*r;
}
```

#define Macro with Arguments

- `#define` statement may be used with arguments.

Example:

```
#define   sqr(x)   x*x

r = sqr(a) + sqr(30);        r = a*a + 30*30;
r = sqr(2+5);                r = 2+5*2+5;
```

- The macro definition should have been written as

```
#define  sqr(x)  (x)*(x)

r = sqr(a+b);      r = (a+b)*(a+b);
```

C Storage Classes

Storage Class of Variables

- It refers to the permanence of a variable, and its scope within a program.
- Four storage class specifications in C
  - Automatic: **auto**
  - External: **extern**
  - Static: **static**
  - Register: **register**

Automatic Variables

- These are always declared within a function and are local to the function in which they are declared.
  - Scope is confined to that function.
- This is the default storage class specification.
  - All variables are considered as auto unless explicitly specified otherwise.
  - The keyword `auto` is optional.
  - An automatic variable does not retain its value once control is transferred out of its defining function.

Example: Auto

```
#include <stdio.h>

int factorial(int m)
{
   auto int i;
   auto int temp = 1;

   for (i=1; i<=m; i++)
      temp = temp * i;
   return (temp);
}

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

Static Variables

- Static variables are defined within individual functions and have the same scope as automatic variables.
- Unlike automatic variables, static variables retain their values throughout the life of the program.
  - If a function is exited and re-entered at a later time, the static variables defined within that function will retain their previous values.
  - Initial values can be included in the static variable declaration.
    - Will be initialized only once.
- An example of using static variable:
  - Count number of times a function is called.

Example: Static

```
#include <stdio.h>
void print()
{
   int count = 0;
   printf("Hello World!! ");
   count++;
   printf("is printing %d times.\n",count);
}

int main()
{
   int i = 0;
   while(i<10)
   {
      print();
      i++;
   }
return 0;
}
```

Output

```
Hello World!! is printing 1 times.
Hello World!! is printing 1 times.
Hello World!! is printing 1 times.
Hello World!! is printing 1 times.
Hello World!! is printing 1 times.
Hello World!! is printing 1 times.
Hello World!! is printing 1 times.
Hello World!! is printing 1 times.
Hello World!! is printing 1 times.
Hello World!! is printing 1 times.
```

Example: Static

```
#include <stdio.h>
void print()
{
    static int count = 0;
    printf("Hello World!! ");
    count++;
    printf("is printing %d times.\n",count);
}

int main()
{
    int i=0;
    while(i<10)
    {
        print();
        i++;
    }
return 0;
}
```

Output!

Hello World!! is printing 1 times.
Hello World!! is printing 2 times.
Hello World!! is printing 3 times.
Hello World!! is printing 4 times.
Hello World!! is printing 5 times.
Hello World!! is printing 6 times.
Hello World!! is printing 7 times.
Hello World!! is printing 8 times.
Hello World!! is printing 9 times.
Hello World!! is printing 10 times.

External Variables

- They are not confined to single functions.
- Declare them outside the function, at the beginning.
- Their scope extends from the point of definition through the remainder of the program.
  - They may span more than one functions.
  - Also called global variables.
    - Alternate way of declaring global variables.

Example: Global

```
#include <stdio.h>
extern int count = 0;

void print()
{
   printf("Hello World!! ");
   count++;
}

int main()
{
   int i=0;
   while(i<10)
   {
      print();
      i++;
      printf("is printing %d times.\n",count);
   }
return 0;
}
```

Output!

Hello World!! is printing 1 times.
Hello World!! is printing 2 times.
Hello World!! is printing 3 times.
Hello World!! is printing 4 times.
Hello World!! is printing 5 times.
Hello World!! is printing 6 times.
Hello World!! is printing 7 times.
Hello World!! is printing 8 times.
Hello World!! is printing 9 times.
Hello World!! is printing 10 times.

Static versus Extern

```
#include <stdio.h>
void print()
{
   static int count=0;
   printf("Hello World!! ");
   count++;
   printf("is printing %d
    times.\n",count);
}

int main()
{
   int i=0;
   while(i<10)
   {
      print();
      i++;
   }
return 0;
}
```

```
#include <stdio.h>
extern int count=0;
void print()
{
   printf("Hello World!! ");
   count++;
}

int main()
{
   int i=0;
   while(i<10)
   {
       print();
       i++;
       printf("is printing %d
        times.\n",count);
   }
return 0;
}
```

Register Variables

- These variables are stored in high-speed registers within the CPU.
  - Commonly used variables like loop variables/counters may be declared as register variables.
  - Results in increase in execution speed.
  - User can suggest, but the allocation is done by the compiler.

Example Register Variables

```
#include<stdio.h>
int main()
{
    int sum;
    register int count;

    for(count=0;count<20;count++)
        sum=sum+count;

     printf("\nSum of Numbers:%d", sum);
     return(0);
}
```

Any question?

You may post your question(s) at the "Discussion Forum" maintained in the course Web page.

Problems for Ponder…

1. Explain what is likely to happen when the following situations are ther in any C program
   a) Actual arguments are less than the formal arguments in a function.
   b) Data type of one of the actual arguments does not match with the type of the corresponding formal argument.
   c) The type of the expression in return(expr); does no match with the type of the function.
   d) The same variable name is declared in two different functions
   e) Reference to a variable name, which is not declared in anywhere in the program

2. State whether the following statements are true or false.
   a) Functions should be arranged in the order in which they are called.
   b) A function can return only one value.
   c) We can pass nay number of arguments to a function.
   d) A function in C should have at least one argument.
   e) A function always returns an integer values.
   f) A function ca call itself.
   g) Every function should have a return statement.

Problems for Ponder…

3. Which of the following function definitions are invalid? Why?

   a) average(x, y, z);

   b) sqrt(int n);

   c) Rand();

   d) power (int x, int n)

   e) double minimum(float x; float y);

4. The following is the function prototype to retune the value of x/y.

```
double divide(x, y)
 float x; float y;
 {
     return (x/y);
 }
```

What will be the value of the following function calls

a) `divide (10, 2);`
b) `divide (4.5, 1.0);`
c) `divide (1, 0);`

Problems for Ponder…

5. Determine the output of the following program.

```
#include <stdio.h>
int p, q;

int producr(int i, int j);

void main () {
     int x = 10, y = 20;
     p = product(x, y);
     q = product(p, product(x,2));
     printf("%d %d \n", p, q);
}
int producr(int a, int b){
    return(a*b);
}
```

6. What will be the output of the following programs given that s = "d%samanta";

a) `printf(s);`

b) `printf("%s", s);`

Problems for Ponder…

7. Is `printf(…)` function returns anything? If so, what it returns?
8. What is the format for printing a number in hexadecimal format?
9. Is `scanf(…)` function returns anything? If so, what it returns?
10. How using `scanf(…)` function one can read the string "Debasis Samanta" typed on the keyboard?
11. What the following `printf` statements will print?

a) `printf( );`

b) `printf("\n \n");`

c) `printf("%d", 6789);`

d) `printf("%6d", 6789);`

e) `printf("%2d", 6789);`

f) `printf("%-6d", 6789);`

g) `printf("%.6d", 6789);`

h) `printf("%7.4d", 6789);`

i) `printf("%2.2f", 456.789);`

Problems for Ponder…

12. What the following program segments will print?

a)
```
int d = 65;
printf("%c \n",d);
```

b)
```
char c = '\t';
printf("%d \n",c);
```

13. In response to the input statement

```
scanf("%4d %c %f", &x, &y, &z);
```

The following data is keyed in

```
05011964
```

What values does the computer assigns to the variable x, y and z?

Problems for Practice…

* **You can check the Moodle course management system for a set of problems for your own practice.**
  - Login to the Moodle system at http://cse.iitkgp.ac.in/
  - Select "PDS Spring-2017 (Theory) in the link "My Courses"
  - Go to Topic 5: Practice Sheet #05 : Functions in C

* Solutions to the problems in Practice Sheet #05 will be uploaded in due time.

If you try to solve problems yourself, then you will learn many things automatically.

Spend few minutes and then enjoy the study.