CS11001/CS11002 Programming and Data Structures (PDS) (Theory: 3-0-0)

Teacher: Sourangshu Bhattacharya
sourangshu@gmail.com
http://cse.iitkgp.ac.in/~sourangshu/

Department of Computer Science and Engineering
Indian Institute of Technology Kharagpur

Functions

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 computation / task 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 function may not return anything.
    - Return data type specified as "void".

Function Example

```c
#include  <stdio.h>

int  factorial (int m)
{
   int i, temp=1;
   for (i=1; i<=m; i++)
    temp = temp * i;
   return (temp);
}
int main()
{
   int  n,fact;
   for  (n=1; n<=10; n++) {
     fact=factorial (n);
      printf (“%d! = %d
               \n”,n,fact);
   }
   return 0;
}
```

```c
#include  <stdio.h>

int  factorial (int m)
{
   int i, temp=1;
   for (i=1; i<=m; i++)
    temp = temp * i;
   return (temp);
}
int main()
{
   int  n;
   for  (n=1; n<=10; n++)
    printf (“%d! = %d
   \n”,n,factorial (n));
   return 0;
}
```

Functions: Why?

- Functions
  - Modularize a program
  - All variables declared inside functions are local variables
    - Known only in function defined
  - Parameters
    - Communicate information between functions
    - They also become local variables.
- Benefits
  - Divide and conquer
    - Manageable program development
  - Software reusability
    - Use existing functions as building blocks for new programs
    - Abstraction - hide internal details (library functions)
  - Avoids code repetition

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
  }
```

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)
```

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

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

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

Declarations and statements: function body (block)

- Variables can be declared inside blocks (can be nested)
- Function can not be defined inside another function

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

Function Not Returning Any Value

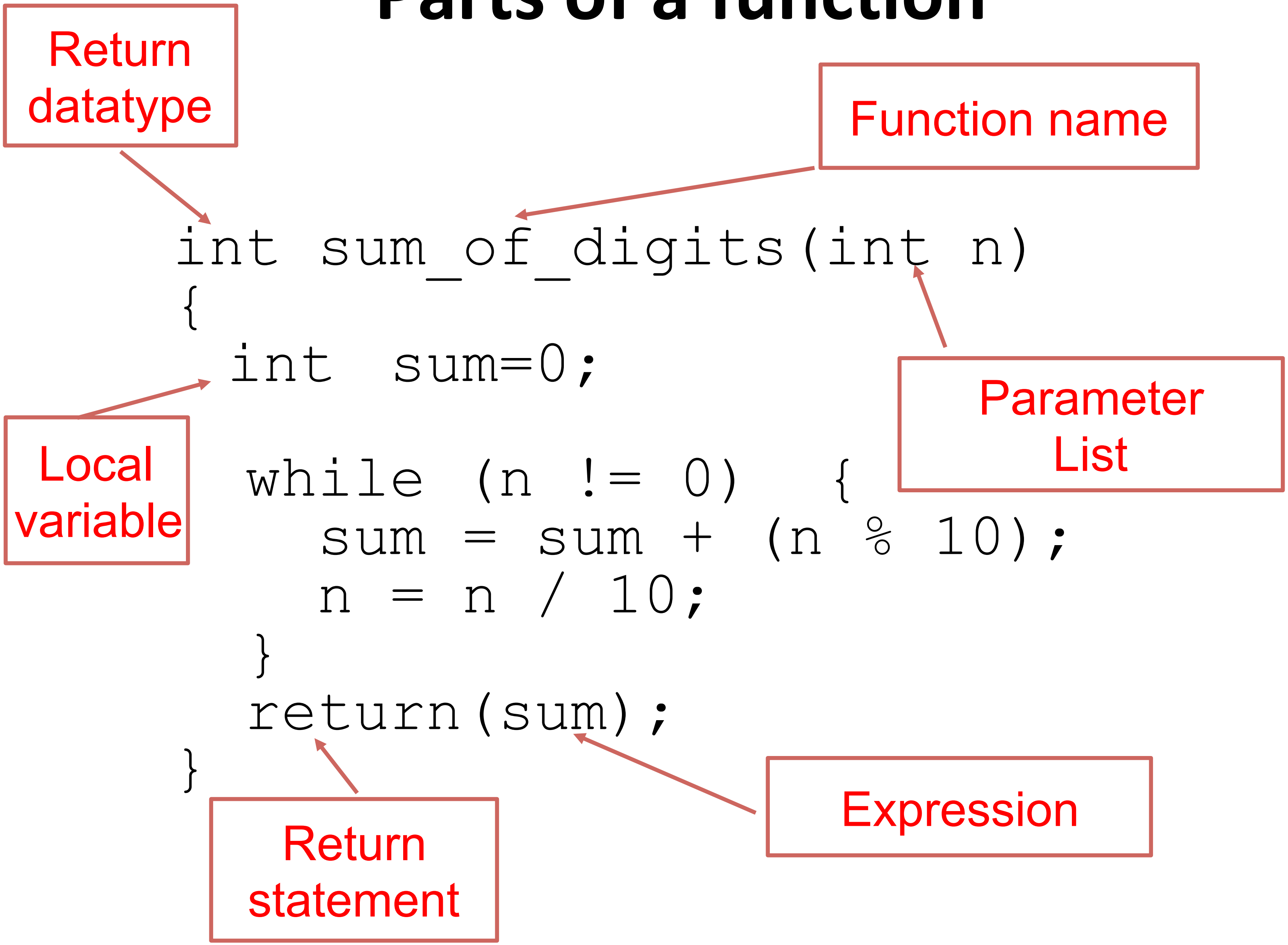
- Example: A function which only prints if a number if divisible by 7 or not.

```
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
}
```

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

Parts of a function



```
int sum_of_digits(int n)
{
 int  sum=0;

 while (n != 0)  {
    sum = sum + (n % 10);
    n = n / 10;
 }
 return(sum);
}
```

Return datatype

Function name

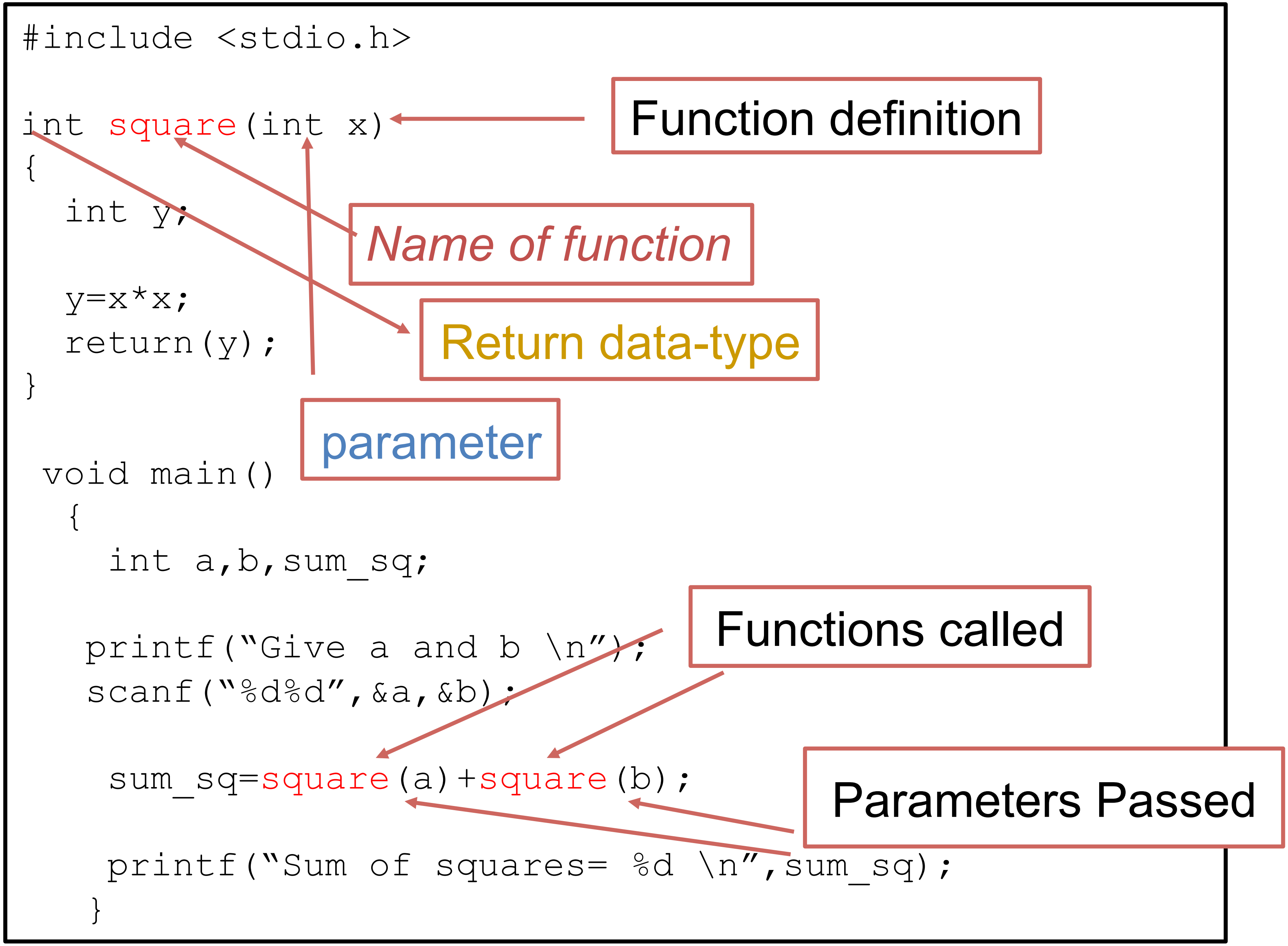
Parameter List

Local variable

Return statement

Expression

Function: An Example



```
#include <stdio.h>

int square(int x)
{
  int y;

  y=x*x;
  return(y);
}

 void main()
  {
    int a,b,sum_sq;

   printf(“Give a and b \n”);
   scanf(“%d%d”,&a,&b);

    sum_sq=square(a)+square(b);

    printf(“Sum of squares= %d \n”,sum_sq);
   }
```

Function definition

Name of function

Return data-type

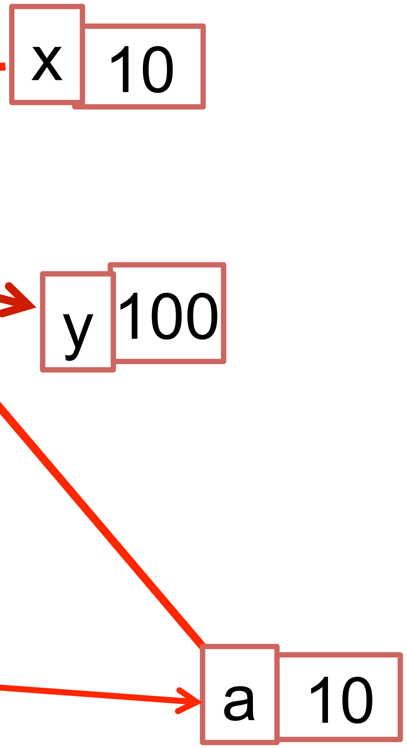
parameter

Functions called

Parameters Passed

Invoking a function call : An Example

```
#include <stdio.h>

int square(int x)
{
  int y;

  y=x*x;
  return(y);
}

void main()
 {
    int a,b,sum_sq;

   printf(“Give a and b \n”);
   scanf(“%d %d”,&a,&b);


    sum_sq=square(a)+square(b);

    printf(“Sum of squares= %d \n”,sum_sq);
 }
```



Function Prototypes

- Usually, a function is defined before it is called.
  - 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 Prototype (Contd.)

- 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 (optional too); but it is a good practice to use the same names as in the function definition.

Function Prototype: Examples

```
#include  <stdio.h>

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

int main()
{
   int i, m, n, sum=0;
   printf(“Input m and n \n”);
      scanf (“%d %d”, &m, &n);

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

   printf (“Result: %d \n”,
   sum);
   return 0;
}
```

Prototype declaration

Function prototype is optional if it is defined before use (call).

```
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);
}
```

Function definition

Function: Summary

```
#include  <stdio.h>
int  factorial
   (int m)
{
   int i, temp=1;

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

Returned data-type

Function name

parameter

Local vars

Return statement

Self contained programme

```
int main()
{
   int  n;
   for  (n=1; n<=10; n++)
      printf (“%d! = %d \n”,
   n, factorial (n) );
   return 0;
}
```

main()
is a function

Calling a function

Functions: Some Facts

- 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.

Functions: Some Facts

- A function can be declared within a function.

- The function declaration, call and definition must match with each other.

  -
    ```
    int gcd(int a, int b);     // function declaration
    ```

  -
    ```
    gcd(a,b);     //function call, a and b is int
    ```

  -
    ```
    int gcd(int a, int b)   // function definition
     {
         …..
     }
    ```

Header Files

- Header files
  - contain function prototypes for library functions
  - <stdio.h>, <stdlib.h> , <math.h>, etc
  - Load with

    #include <filename>
  - #include <math.h>

- Custom header files
  - Create file with functions
  - Save as *filename.h*
  - Load in other files with #include "*filename.h*"
  - Reuse functions

Math Library Functions

- Math library functions
  - perform common mathematical calculations
  - `#include <math.h>`
  - `cc <prog.c>` **`-lm`**

- Format for calling functions

  `FunctionName(argument);`
    - If multiple arguments, use comma-separated list
  - `printf("%.2f",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 ceil(double x)	-- Get smallest integral value that exceeds x.
double floor(double x)	-- Get largest integral value less than 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.
double exp(double x)	-- Compute exponential of x
double fabs(double x)	-- Compute absolute value of x.
double log(double x)	-- Compute log(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.

#define: Macro definition

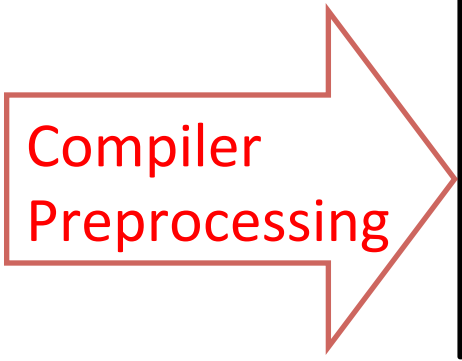
- Preprocessor directive in the following form

  #define string1 string2

- Replaces the *string1* by *string2* wherever it occurs before compilation, e.g.

  #define PI 3.14

```
#include <stdio.h>
#define PI 3.14
main()
{
  float r=4.0,area;
  area=PI*r*r;
  return 0;
}
```



Compiler Preprocessing

```
#include <stdio.h>
int main()
{
  float r=4.0,area;
  area=3.14*r*r;
  return 0;
}
```

#define with argument

- It may be used with argument e.g.

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

Which one is faster to execute?

```
#include <stdio.h>
int sqr(int x)
{
   return (x*x);
}
int main()
{
  int y=5;
  printf(“value=%d \n”,
sqr(y)+3);
  return 0;
}
```

```
#include <stdio.h>
#define sqr(x) ((x)*(x))

int main()
{
  int y=5;
  printf(“value=%d \n”,
sqr(y)+3);
  return 0;
}
```

```
#include <stdio.h>

int main()
{
  int y=5;
  printf(“value=%d \n”,
((y)*(y))+3);
  return 0;
}
```

#define with arguments: A Caution

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

– How macro substitution will be carried out?

r = sqr(a) + sqr(30); ➔ r = a*a + 30*30;

r = sqr(a+b); ➔ r = a+b*a+b;

WRONG?

– The macro definition should have been written as:

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

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

An Example: Random Number Generation

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

void 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");
  }
}
```

Algorithm

1. Initialize seed
2. Input value for seed

2.1 Use srand to change random sequence

2.2 Define Loop

3. Generate and output random numbers

```
Enter seed: 293
       2         4         1         5         3
       3         1         1         2         6
```

```
Enter seed: 67
       6         4         4         6         4
       3         6         1         4         2
```

```
Enter seed: 867
       5         5         2         3         5
       4         2         2         3         4
```

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 1: Minimum of a set of numbers

```
#include  <stdio.h>

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.

```
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);
}
```

Parameter Passing mechanism

- When an array is passed to a function, the values of the array elements are *not passed* to the function.
  - The array name is interpreted as the *address* of the first array element.
  - The formal argument therefore becomes a *pointer* to the first array element.
  - When an array element is accessed inside the function, the address is calculated using the formula stated before.
  - Changes made inside the function are thus also reflected in the calling program.

Parameter Passing mechanism

- Passing parameters in this way is called

  call-by-reference.

- Normally parameters are passed in C using

  call-by-value.

- Basically what it means?
  - If a function changes the values of array elements, then these changes will be made to the original array that is passed to the function.
  - This does not apply when an individual element is passed on as argument.

Example: Average of numbers

```
#include <stdio.h>

float avg(float [], int );

void main()
{
  float a[]={4.0, 5.0, 6.0,
7.0};

  printf("%f \n", avg(a,4) );
}
```

prototype

Array name passed

Array as parameter

```
float  avg (float x[], int
n)
{
  float sum=0;
  int i;

  for(i=0; i<n; i++)
     sum+=x[i];
  return(sum/(float) n);
}
```

Number of Elements used

Call by Value and Call by Reference

- Call by value
  - Copy of argument passed to function
  - Changes in function do not effect original
  - Use when function does not need to modify argument
    - Avoids accidental changes
- Call by reference
  - Passes original argument
  - Changes in function effect original
  - Only used with trusted functions

Example: Max Min function

```
/* Find maximum and minimum from a
list of 10 integers */
#include <stdio.h>

void getmaxmin(int array[],int
size,int maxmin[]);

void main()
{
        int a[20],i,maxmin[2];

        printf("Enter 10 integer
values: ");
        for(i=0;i<10;i++)
                scanf("%d",&a[i]);
        getmaxmin(a,10,maxmin);
        printf("Maximum=%d,
Minimum=%d\n",
    maxmin[0],maxmin[1]);
}
```

Return type of any function may be void

Still it can return value(s).

```
void getmaxmin(int array[],int
size,int maxmin[])
{
        int
i,max=-99999,min=99999;

        for(i=0;i<size;i++) {
                if(max<array[i])

max=array[i];
                if(min>array[i])

min=array[i];
        }
        maxmin[0]=max;
        maxmin[1]=min;
}
```

Returning multiple values from a function.

Scope of a variable

```
#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
```

Storage Class of Variables

What is Storage Class?

- 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.

auto: Example

```
#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.

static: Example

```
#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.

- 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.
  - Declare them outside the function, at the beginning.

global: Example

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

```
#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>
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.

```
#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);
}
```

#include: Revisited

- Preprocessor statement in the following form

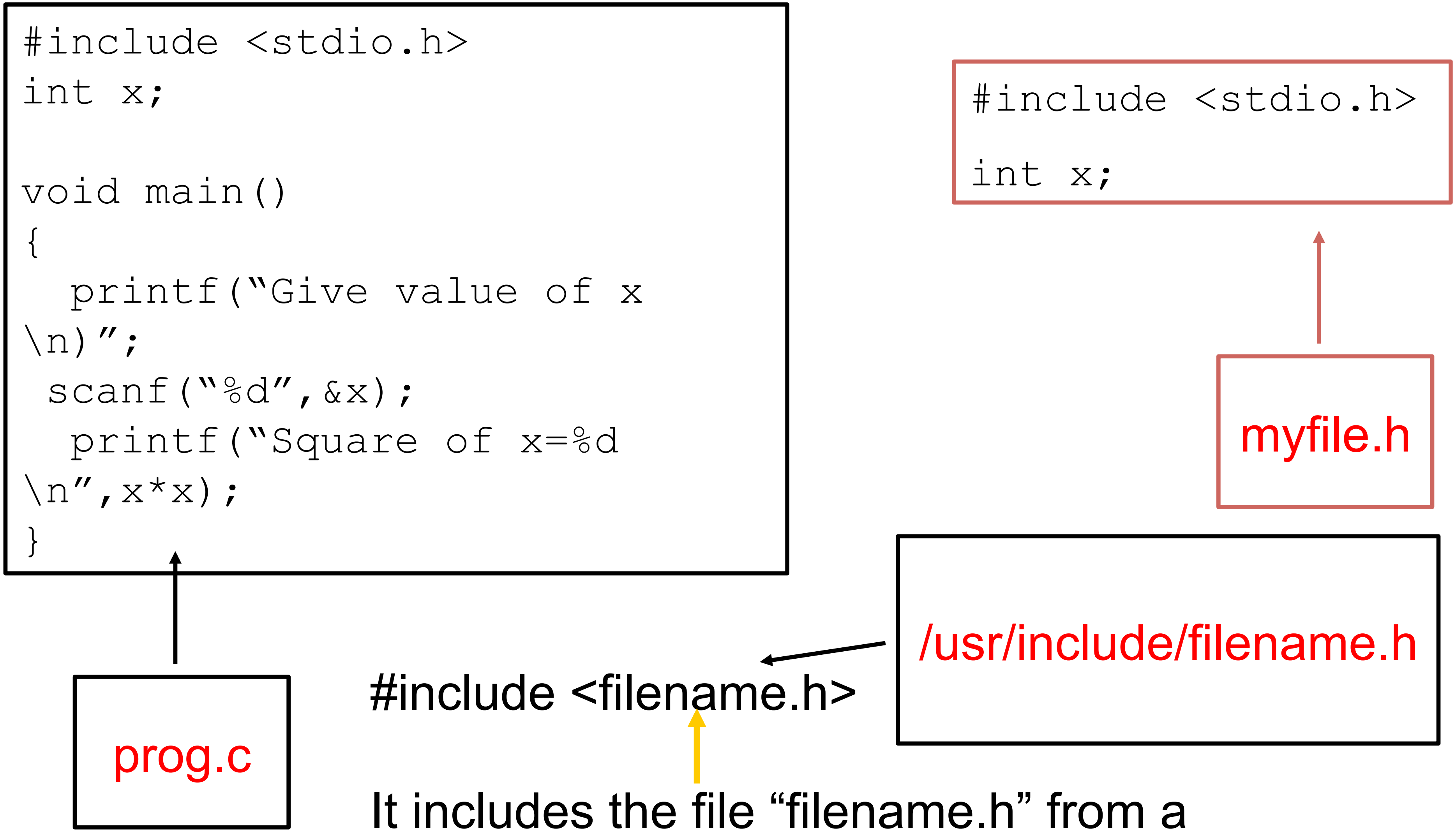
  #include “filename”

- Filename could be specified with complete path.

  #include “/home/pralay/C-header/myfile.h”

- The content of the corresponding file will be included in the present file before compilation and the compiler will compile thereafter considering the content as it is.

#include: Revisited



```
#include <stdio.h>
int x;

void main()
{
  printf(“Give value of x \n)”;
 scanf(“%d”,&x);
  printf(“Square of x=%d \n”,x*x);
}
```

prog.c

```
#include <stdio.h>

int x;
```

myfile.h

/usr/include/filename.h

#include <filename.h>

It includes the file “filename.h” from a specific directory known as include directory.

Variable number of arguments

- General form:

```
scanf (control string, arg1, arg2, …, argn);
printf (control string, arg1, arg2, …, argn);
```

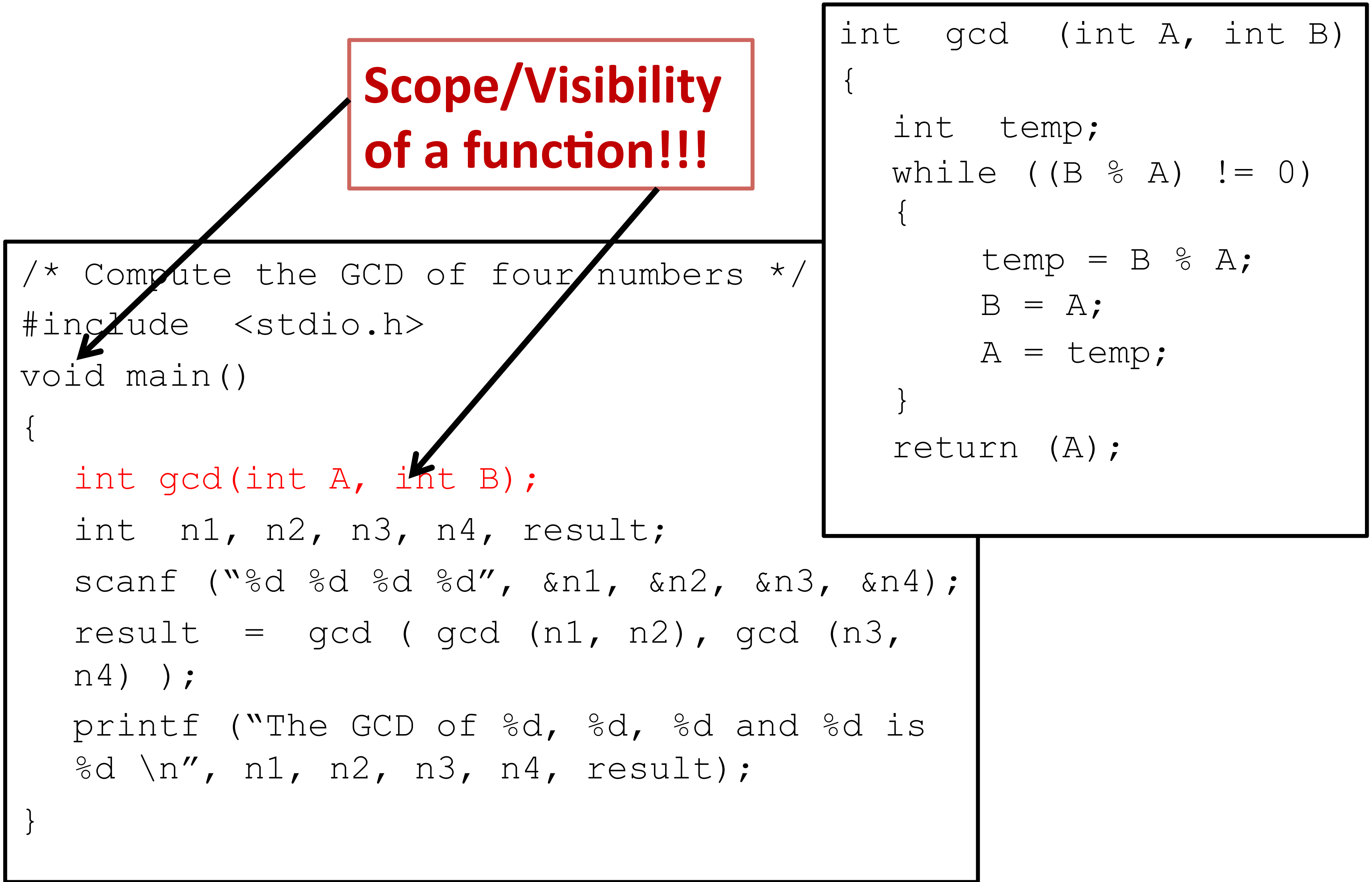
How is it possible?

Example: GCD calculation

```
/* Compute the GCD of four numbers
#include  <stdio.h>
int gcd(int A, int B);
void 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);
}
```

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

Example: GCD calculation



Scope/Visibility of a function!!!

```
/* Compute the GCD of four numbers */
#include  <stdio.h>
void main()
{
   int gcd(int A, int B);
   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);
}
```

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