Functions

Functions are everywhere in C

Pallab Dasgupta
Professor,
Dept. of Computer Sc & Engg

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`".

```
#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!
```

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.

Use of functions: *Area of a circle*

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

/* Function to compute the area of a circle */
float   myfunc (float r)
{           float   a;
            a = PI * r * r;
            return (a);      /* return result */
}
main()
{      float   radius, area;
       float   myfunc (float radius);

       scanf ("%f", &radius);
       area = myfunc (radius);
       printf ("\n Area is %f \n", area);
}
```

Macro definition

Function definition

Function argument

Function declaration
(return value defines the type)

Function call

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);
}
```

BODY

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.

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

```
/* 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);
}
```

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);
}
```

Scope of Variables

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

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

Output:

```
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 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.

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.

Header Files

Header files

- Contain function prototypes for library functions.
- `<stdlib.h>` , `<math.h>` , etc
- Load with: `#include <filename>`
- Example :

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

Custom header files

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

Parameter passing: by Value and by Reference

Used when invoking functions.

<u>Call by value</u>

- **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.**

<u>Call by reference</u>

- **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***

"C supports only call by value"

Example: Random Number 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);
```

#define: Macro definition

Preprocessor 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

```
#include <stdio.h>
#define PI 3.1415926
main()
{
  float r = 4.0, area;
  area = PI * r * r;
}
```

$\Rightarrow$

```
#include <stdio.h>
main()
{
  float r = 4.0, area;
  area = 3.1415926 * r * r;
}
```

#define with arguments

#define statement may be used with arguments.

- Example: #define   sqr(x)   x*x
- How will macro substitution 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:

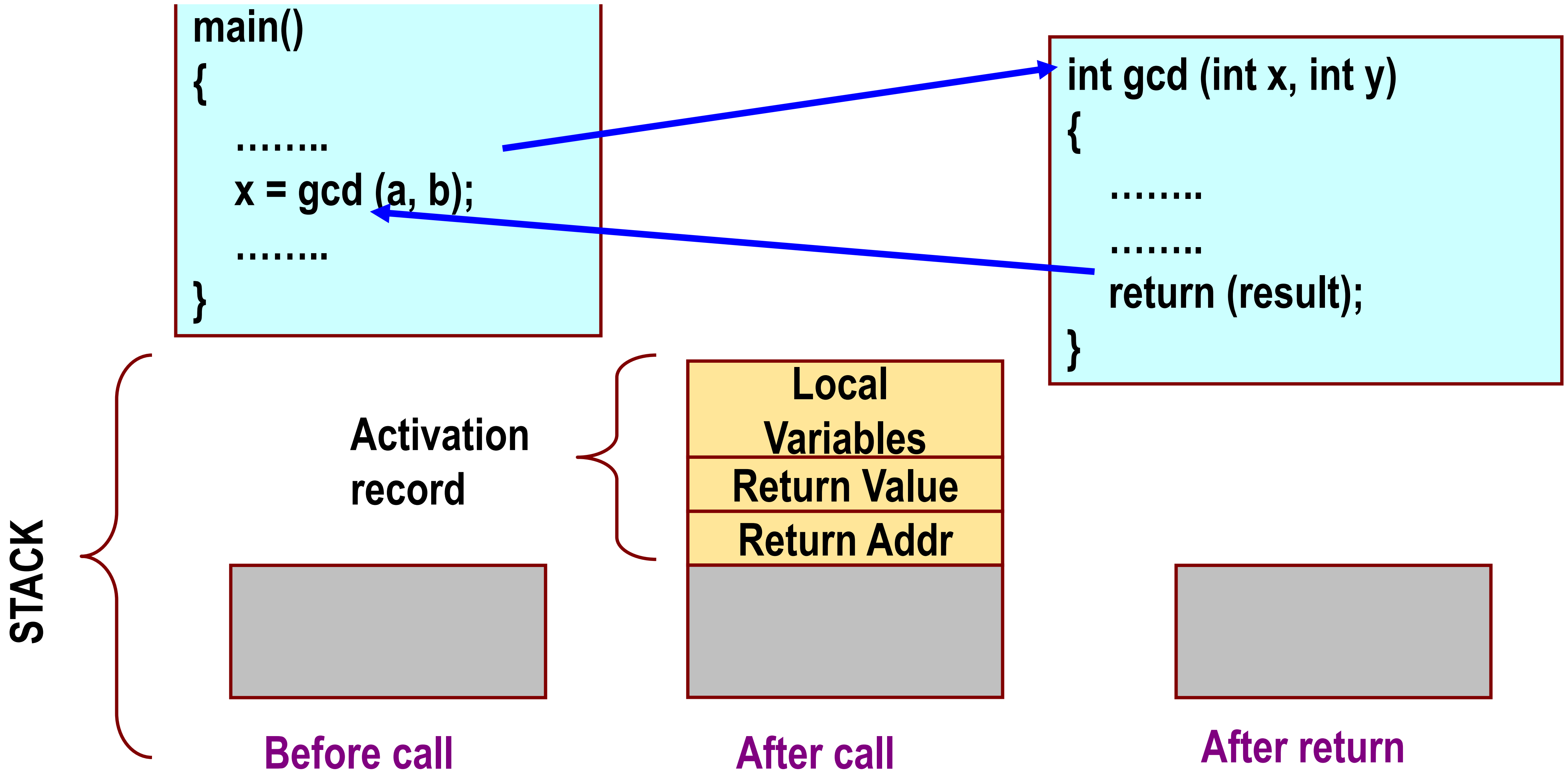
```
#define  sqr(x)  (x)*(x)
r = sqr(a+b);                →    r = (a+b)(a+b);
```

How are function calls implemented?

The following applies in general, with minor variations that are implementation dependent.

- **The system maintains a stack in memory.**
  - **Stack is a last-in first-out structure.**
  - **Two operations on stack, push and pop.**

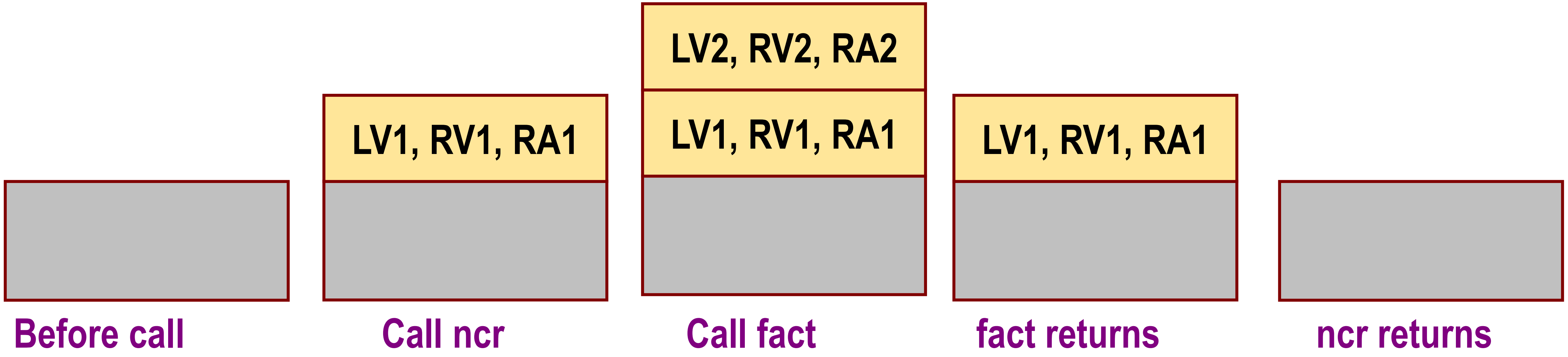
- **Whenever there is a function call, the activation record gets pushed into the stack.**
  - **Activation record consists of the return address in the calling program, the return value from the function, and the local variables inside the function.**


main()
{
....
x = gcd (a, b);
....
}
int gcd (int x, int y)
{
....
....
return (result);
}
Activation record
STACK
Local Variables
Return Value
Return Addr
Before call
After call
After return

```
main()
{
    ........
    x = ncr (a, b);
    ........
}
```

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

```
int fact (int n)
{
    .........
    return (result);
}
```



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.

```
#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 );
}
```

```
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.

Static Variables

```
void test( int x )
{
    static int count = 0;

    if ( x == 0 ) {
            if ( count == 2 ) {
                printf ("Three consecutive zeros");
            }
            else count++;
    }
    else count = 0;
    return;
}
```

Initialization will happen only the first time the function is called

```
main( )
{
    int k, marks;

    for ( k=0;  k < 100; k++ )
    {
            scanf("%d", &marks) ;
            test( marks );
    }
}
```

Function that detects three consecutive zeros in a stream of 100 marks.

Register Variables

These variables are stored in high-speed registers within the CPU.

- Commonly used variables may be declared as register variables.
- Results in increase in execution speed.
- The allocation is done by the compiler.

For example:

```
register float y;  // Instructs the compiler to allocate some register to y
```

External Variables

They are not confined to single files.

Their scope extends from the point of definition through the remainder of the program.

- They may span more than one file.
- They too are global variables.

External Variables

Informs the compiler that k is an integer variable but space is not to be allocated for it.

FILE a.c

```
extern int k;
void f ( void )
{
    k++;
 }
```

FILE b.c

```
int k=0;
extern void f ( void );
void g ( void )
{
    f( );
    printf( "%d\n", k );
 }
```