FUNCTIONS

CS10003: PROGRAMMING AND DATA STRUCTURES

Introduction

Function

- A program segment that carries out some specific, well-defined task.
    - Example
        - A function to add two numbers
        - A function to find the largest of n numbers
- A function will carry out its intended task whenever it is called or invoked
    - Can be called multiple times

- 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`”.
- Functions may call other functions (or itself) as instructions

Function Control Flow

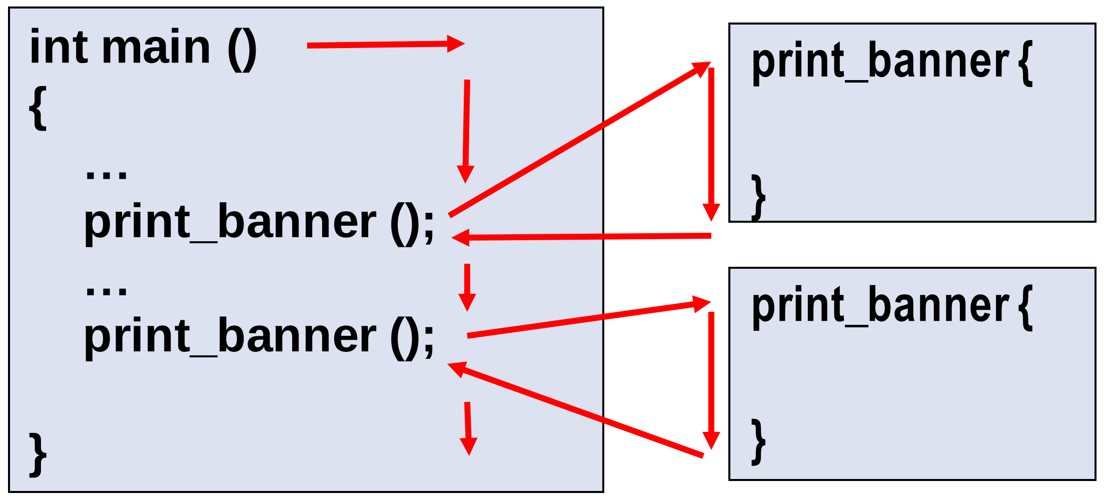
Code

```
void print_banner ( )
{
      printf(“************\n”);
}
```

```
int main ( )
{
    . . .
    print_banner ( ) ;
    . . .
    print_banner ( ) ;

}
```

Execution



```
#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!
```

- 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`”.

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

Why Functions?

Functions

- Allows one to develop a program in a modular fashion.
    - Divide-and-conquer approach.
- Use existing functions as building blocks for new programs
- Abstraction: hide internal details (library functions)
    - 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.

Defining a Function

A function definition has two parts:

- The first line.
- The body of the function.

General syntax:

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

return value type

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

body

Return value

- A function can return a value

    Using return statement

- Like all values in C, a function return value has a type
- The return value can be assigned to a variable in the caller

```
int x, y, z;
scanf("%d%d", &x, &y);
z = gcd(x,y);
printf("GCD of %d and %d is %d\n", x, y, z);
```

- Sometimes a function may not return anything
- Such functions are void functions

<u>Example</u>: 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;
}
```

- The return type is void
- The return statement for void functions is optional

The return statement

In a value-returning function (result type is not void), return does two distinct things

- specify the value returned by the execution of the function
- terminate that execution of the callee and transfer control back to the caller

A function can only return one value

- The value can be any expression matching the return type
- but it might contain more than one return statement.

In a void function

- return is optional at the end of the function body.
- return may also be used to terminate execution of the function explicitly before reaching the end.
- No return value should appear following return.

```
void compute_and_print_itax ()
{
   float income;
   scanf (“%f”, &income);
   if (income < 50000)   {
                printf (“Income tax = Nil\n”);
                return;  /* Terminates function execution */
   }
   if (income < 60000)   {
                printf (“Income tax = %f\n”, 0.1*(income-50000));
                return;  /* Terminates function execution */
   }
   if (income < 150000) {
                 printf (“Income tax = %f\n”, 0.2*(income-60000)+1000);
                return ; /* Terminates function execution */
   }
   printf (“Income tax = %f\n”, 0.3*(income-150000)+19000);
}
```

Calling a function

- Called by specifying the function name and parameters in an instruction in the calling function

- When a function is called from some other function, the corresponding arguments in the function call are called actual arguments or actual parameters
  - The function call must include a matching actual parameter for each formal parameter
  - Position of an actual parameters in the parameter list in the call must match the position of the corresponding formal parameter in the function definition
  - The formal and actual arguments must match in their data types

Example

```
int main ()
{
    double x, y, z;
    char op;
    . . .
   z = operate (x, y, op);
              . . .
}
```

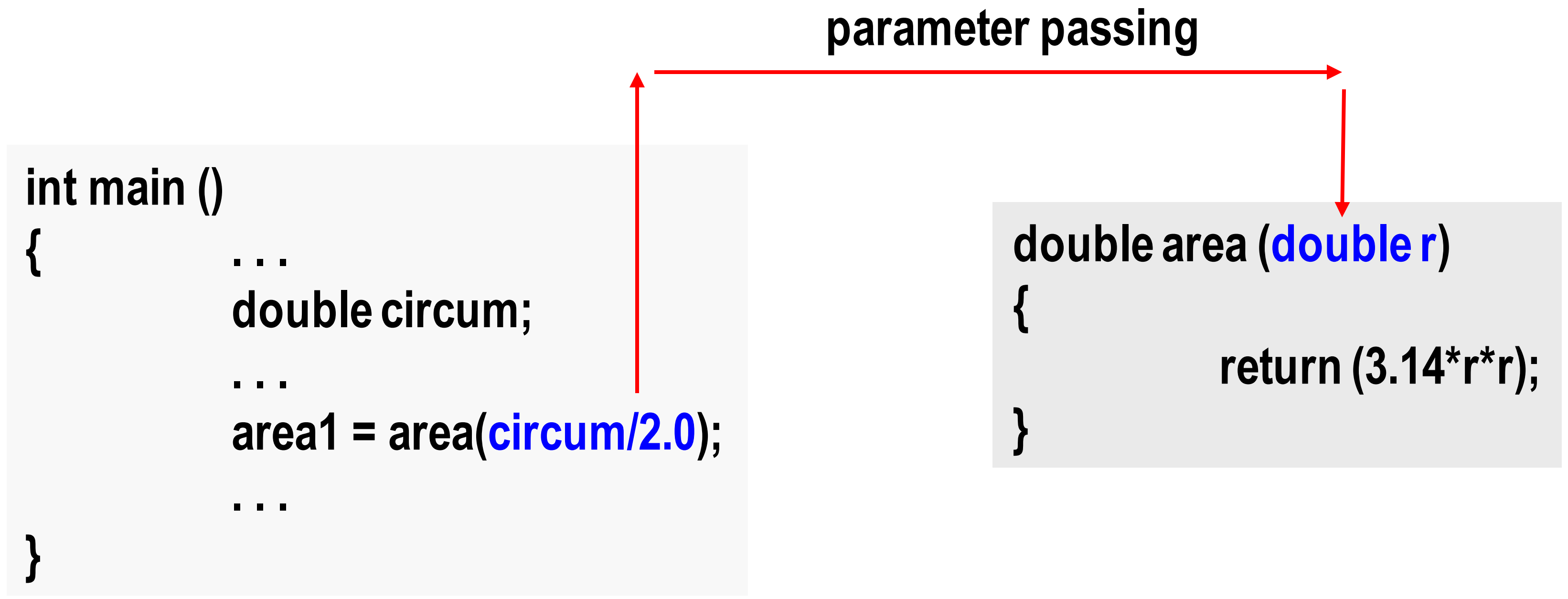
Actual parameters

Formal parameters

```
double operate (double x, double y, char op)
{
            switch (op) {
               case '+' : return x+y+0.5 ;
               case '~'  : if (x>y)
                                    return x-y + 0.5;
                              return y-x+0.5;
               case 'x' : return x*y + 0.5;
               default : return –1;
            }
}
```

Parameter Passing

When the function is executed, the value of the actual parameter is copied to the formal parameter



```
int main ()
{           . . .
            double circum;
            . . .
            area1 = area(circum/2.0);
            . . .
}
```

```
double area (double r)
{
            return (3.14*r*r);
}
```

Another Example: What is happening here?

```
int main()
{
  int numb, flag, j=3;
  scanf("%d",&numb);
   while (j <= numb)
   {
       flag = prime( j );
       if (flag==0)  printf( "%d is prime\n", j );
       j++;
   }
   return 0;
 }
```

```
int prime(int x)
{
   int i, test;
   i=2, test =0;
   while ((i <= sqrt(x)) && (test ==0))
   {
       if (x%i==0) test = 1;
       i++;
     }
    return test;
}
```

Tracking the flow of control

```
int main()
{
  int numb, flag, j=3;
  scanf("%d",&numb);
  printf("numb = %d \n",numb);
  while (j <= numb)
  {
    printf("\nMain, j = %d\n",j);
    flag = prime(j);
    printf("Main, flag = %d\n",flag);

    if (flag == 0) printf("%d is prime\n",j);
    j++;
  }
  return 0;
}
```

```
int prime(int x)
{
   int i, test;
   i = 2;  test = 0;

   printf("In function, x = %d \n",x);
   while ((i <= sqrt(x)) && (test == 0))
   {
      if (x%i == 0) test = 1;
      i++;
   }
   printf("Returning, test = %d \n",test);

   return test;
}
```

PROGRAM OUTPUT

```
5
numb = 5

Main, j = 3
In function, x = 3
Returning, test = 0
Main, flag = 0
3 is prime

Main, j = 4
In function, x = 4
Returning, test = 1
Main, flag = 1

Main, j = 5
In function, x = 5
Returning, test = 0
Main, flag = 0
5 is prime
```

Function Prototypes

Usually, a function is defined before it is called

- main() is the last function in the program written
- 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 are written after 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())
- Prototypes can specify parameter names or just types (more common)
- Examples:

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

- Note the semicolon at the end of the line.
- The parameter name, if specified, can be anything; but it is a good practice to use the same names as in the function definition

Example:

```
#include <stdio.h>
int sum( int, int );
int main( )
{
        int x, y;
        scanf(“%d%d”, &x, &y);
        printf(“Sum = %d\n”, sum(x, y));
}
int sum (int a, int b)
{
        return(a + b);
}
```

Nested Functions

A function cannot be defined within another function. It can be called 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);
}
```

Local variables

A function can define its own local variables

The locals have meaning only within the function

- Each execution of the function uses a new set of locals
- Local variables cease to exist when the function returns

Parameters are also local

```
/* Find the area of a circle with diameter d */
double circle_area (double d)
{
        double radius, area;
        radius = d/2.0;
        area = 3.14*radius*radius;
        return (area);
}
```

parameter

local variables

Revisiting nCr

```
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);
}
```

The n in ncr( ) and the n in fact( ) are different

Scope of a variable

- Part of the program from which the value of the variable can be used (seen)

- Scope of a variable - Within the block in which the variable is defined
    - Block = group of statements enclosed within { }

- Local variable – scope is usually the function in which it is defined
    - So two local variables of two functions can have the same name, but they are different variables

- Global variables – declared outside all functions (even main)
    - scope is entire program by default, but can be hidden in a block if local variable of same name defined

Local Scope replaces Global Scope

```
#include  <stdio.h>
int A;   /* This A is a global variable */
void main( )
{
    A = 1;
    myProc( );
    printf ( "A = %d\n", A );
}

void myProc( )
{
    int A = 2;  /* This A is a local variable */
    while( A==2 )
    {
            printf ( "A = %d\n", A );
            break;
    }
    printf ( "A = %d\n", A );
}
```

A = 1

A = 2

A = 2

Scope of global A

Scope of local A

What happens here?

```
#include <stdio.h>
int A;   /* This A is a global variable */
void main( )
{
    A = 1;
    myProc( );
    printf ( "A = %d\n", A );    ⟶    A = 2
}

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

What happens here?

```
#include  <stdio.h>
int A;   /* This A is a global variable */
void main( )
{
    A = 1;
    myProc( A );
    printf ( "A = %d\n", A );         → A = 1
}

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

Parameter Passing: by Value and by Reference

- Used when invoking functions

Call by value

- Passes the value of the argument to the function
- Execution of the function does not change the actual parameters
    - All changes to a parameter done inside the function are done on a copy of the actual parameter
    - The copy is removed when the function returns to the caller
    - The value of the actual parameter in the caller is not affected
- Avoids accidental changes

Call by reference

- Passes the address to the original argument.
- Execution of the function may affect the original
- Not directly supported in C except for arrays

Parameter passing & return: 1

```
int main()
{
    int  a=10, b;
    printf  (“Initially a = %d\n”, a);
    b = change (a);
    printf  (“a = %d, b = %d\n”, a, b);
    return 0;
}
int  change  (int x)
{
   printf  (“Before x = %d\n”,x);
    x = x / 2;
   printf  (“After x = %d\n”, x);
    return (x);
}
```

Output

```
Initially a = 10
Before x = 10
After x = 5
a = 10, b = 5
```

Parameter passing & return: 2

```
int main()
{
    int  x=10, b;
    printf  (“M: Initially x = %d\n”, x);
    b = change (x);
    printf  (“M: x = %d, b = %d\n”, x, b);
    return 0;
}
int  change  (int x)
{
   printf  (“F: Before x = %d\n”,x);
    x = x / 2;
   printf  (“F: After x = %d\n”, x);
    return (x);
}
```

Output

```
M: Initially x = 10
F: Before x = 10
F: After x = 5
M: x = 10, b = 5
```

Parameter passing & return: 3

```
int main()
{
    int  x=10, b;
    printf  (“M: Initially x = %d\n”, x);
    x = change (x);
    printf  (“M: x = %d, b = %d\n”, x, x);
    return 0;
}
int  change  (int x)
{
   printf  (“F: Before x = %d\n”,x);
    x = x / 2;
   printf  (“F: After x = %d\n”, x);
    return (x);
}
```

Output

```
M: Initially x = 10

F: Before x = 10

F: After x = 5

M: x = 5, b = 5
```

Parameter passing & return: 4

```
int main()
{
    int  x=10, y=5;
    printf  (“M1:  x = %d, y = %d\n”, x, y);
    interchange (x, y);
    printf  (“M2:  x = %d, y = %d\n”, x, y);
    return 0;
}

void interchange  (int x, int y)
{ int temp;
  printf  (“F1:  x = %d, y = %d\n”, x, y);
  temp= x; x = y; y = temp;
  printf  (“F2:  x = %d, y = %d\n”, x, y);
}
```

Output

```
M1:  x = 10, y = 5

F1:  x = 10, y = 5

F2:  x = 5, y = 10

M2:  x = 10, y = 5
```

How do we write an interchange function?
(will see later)

Passing Arrays to Function

Array element can be passed to functions as ordinary arguments

- IsFactor (x[i], x[0])
- sin (x[5])

An array name can be used as an argument to a function

- Permits the entire array to be passed to the function
- The way it is passed differs from that for ordinary variables

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

Whole Array as Parameter

```
const int ASIZE = 5;
float average (int B[ ])
{
        int i, total=0;
        for (i=0; i<ASIZE; i++)
                total = total + B[i];
        return ((float) total / (float) ASIZE);
}

int main ( ) {
        int x[ASIZE] ; float x_avg;
        x = {10, 20, 30, 40, 50};
        x_avg = average (x) ;
  return 0;
}
```

Only Array Name/address passed.
[ ] mentioned to indicate that it is an array.

Called only with actual array name

Arrays used as Output Parameters

```
void VectorSum (int a[ ], int b[ ], int vsum[ ], int length) {
        int i;
        for (i=0; i<length; i=i+1)
                vsum[i] = a[i] + b[i] ;
}
void PrintVector (int a[ ], int length)  {
        int i;
        for (i=0; i<length; i++) printf (“%d “, a[i]);
}

int main ()  {
        int x[3] = {1,2,3}, y[3] = {4,5,6}, z[3];
        VectorSum (x, y, z, 3) ;
        PrintVector (z, 3) ;
        return 0;
}
```

The Actual 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

Passing parameters in this way is called call-by-reference

Normally parameters are passed in C using call-by-value

Basically what does it mean?

- 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

Contd.

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

Library Functions

Library Functions

- Set of functions already written for you, and bundled in a “library”
    - Example: printf, scanf, getchar,
- C library provides a large number of functions for many things
- Already seen math library functions earlier
- Will look at string library functions

String Library Functions

String library functions

- perform common operations on null terminated strings
- Must include a special header file

  `#include <string.h>`

Example

- `printf ("%f", strlen(C));`
  - C is a null-terminated string
  - Calls function strlen, which returns the number of characters in C (not counting the '\0' character)

Common string library functions

strlen – returns the length of a string

strcmp – compares two strings (lexicographic)

- Returns 0 if the two strings are equal, < 0 if first string is less than the second string, > 0 if the first string is greater than the second string
- Commonly used for sorting strings

strcat – concatenates two strings

strcpy – copy one string to another

- we will need some basic knowledge of pointers to understand how to use strcat and strcpy

Many others, but these are the ones you will know in this course

Example

```
int main()
{
    char A[20], B[20];
    int n, m, val;
    scanf(“%s%s”, A, B);
    n = strlen(A);
    m = strlen(B);
    printf(“The lengths of the strings are %d and %d\n”, n, m);
    val = strcmp(A, B);
    if (val == 0)
            printf(“The strings are the same\n”);
    else if (val < 0)
            printf(“%s is smaller than %s\n”, A, B);
    else
            printf(“%s is smaller than %s\n”, A, B);
}
```

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.

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)
```

Seeding the random generator

`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
```

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

$\xrightarrow{\text{macro pre-processing}}$

```
#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:

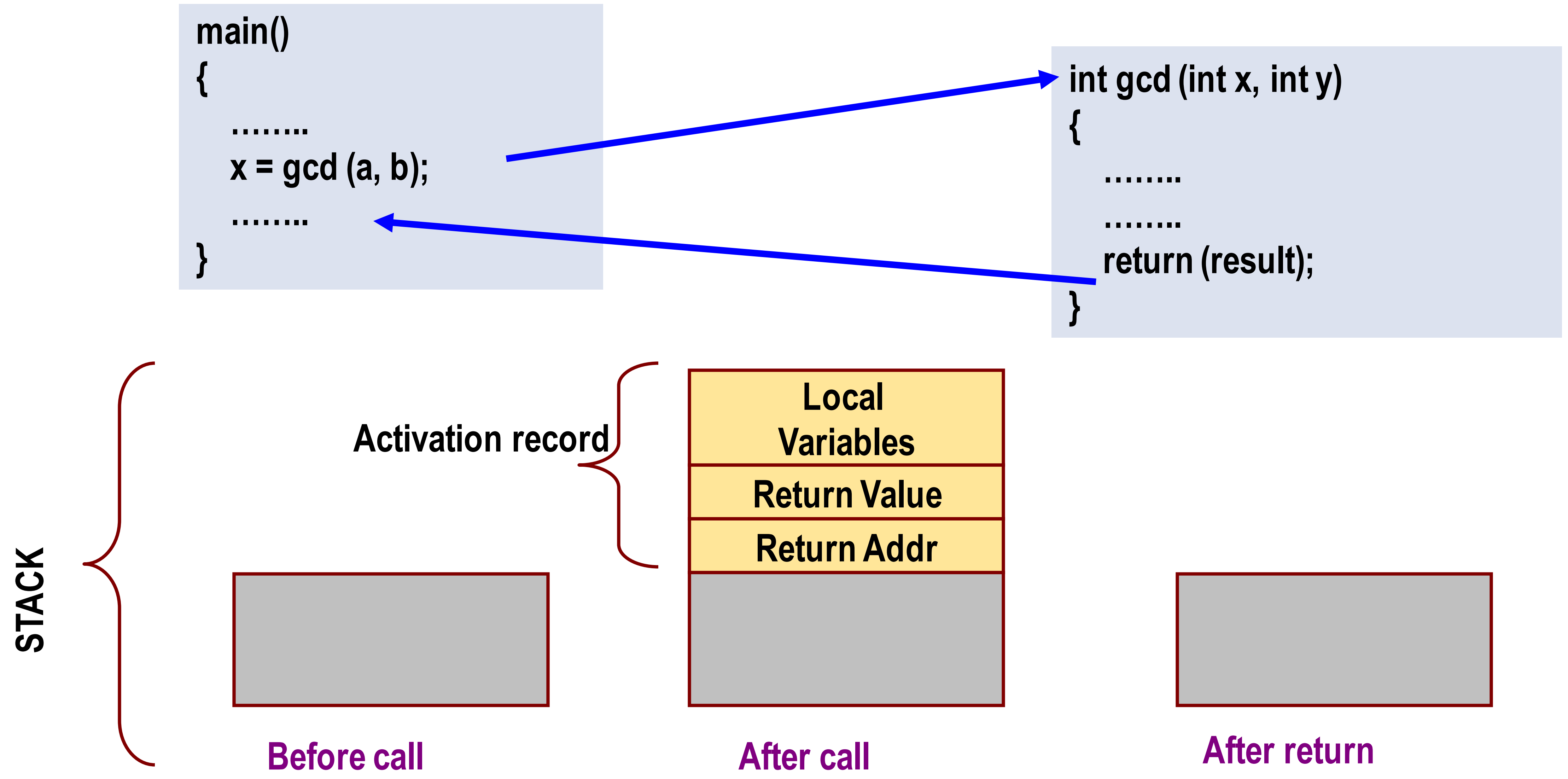
```
#define  sqr(x)  (x)*(x)
r = sqr(a+b);               →       r = (a+b)*(a+b);
```

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.**



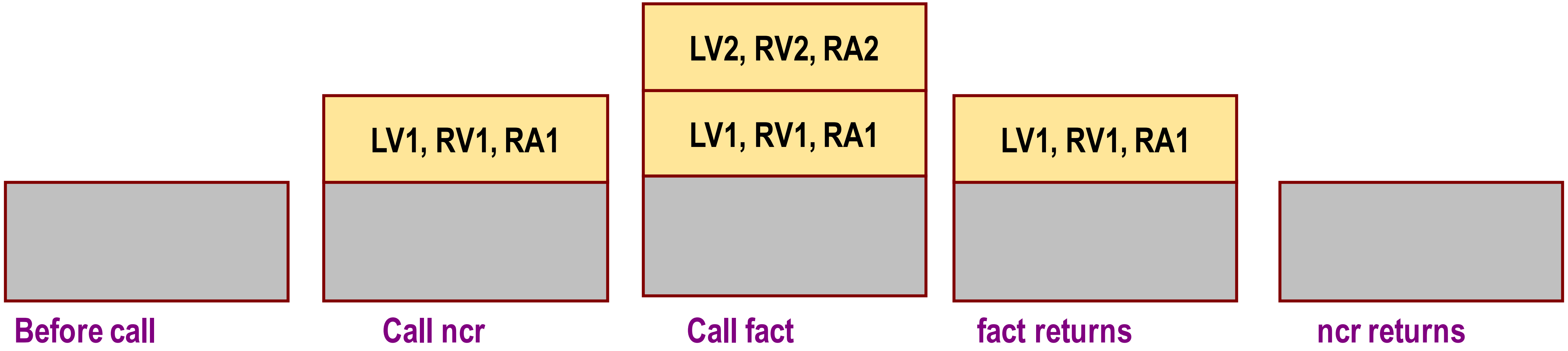
```
main()
{
    ……..
    x = gcd (a, b);
    ……..
}
```

```
int gcd (int x, int y)
{
    ……..
    ……..
    return (result);
}
```

```
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.

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 );
 }
```

Practice Problems

No separate problems needed.

- Look at everything that you did so far, such as finding sum, finding average, counting something, checking if something is true or false (“ Is there an element in array A such that….) etc. in which the final answer is one thing only (like sum, count, 0 or 1,…).
- Then for each of them, rather than doing it inside main (as you have done so far), write it as a function with appropriate parameters, and call from main() to find and print.

  - Normally, read and print everything from main(). Do not read or print anything inside the function. This will give you better practice.
  - However, you can write simple functions for printing an array.