Program Design

Lecture 3 & 4

Functions

- We already know functions such as??
- main(), printf()
- We also know that we pass parameters to the function
- printf("enter numbers") -- parameter
- main() – no parameter in this case
- We can split the main program also into functions

Program without function

```
main(){
        int a,b,sum;
        printf("enter numbers");
         scanf("%d %d",&a,&b);
         sum =a+b;
        printf("sum is %d",sum);}
```

- Suppose a=1 b=2
- sum is 3

Program with functions

```
#include<stdio.h>
//function definition
addition(int a,int b){
 int sum;
 sum =a+b;
 printf("sum is %d",sum);}
main(){
  int a,b;
  printf("enter numbers");
  scanf("%d %d",&a,&b);
 //function call with parameters a & b
  addition(a,b);}
```

Working of a function

- Execution starts from main function
- When the function call is seen,
  the control flow goes to the function definition
- All the statements in the function
  will be executed either to the last statement
  (if there is no return statement in the function),
  or up to the return statement
- Control returns to the statement in the
  main function just after the main function or
  on to the statement where function is called
  if there is some pending works to be done
- Execution of main function is continued
- Activation record

```
#include<stdio.h>
//function definition
addition(int a,int b){
 int sum;
 sum =a+b;
 printf("sum is %d",sum);}
main(){
  int a,b;
  printf("enter numbers");
  scanf("%d %d",&a,&b);
 //function call with parameters a & b
  addition(a,b);}
```

Advantages of functions

- Some codes that must be executed multiple times in different places in a program.
- Same code is needed in multiple different programs.
- Frequently-used code into functions makes it easy to re-use the code
- Separating code out into functions – independent attention to each function
- Well-written functions should be general
- **Functions can be stored in libraries for later re-use**.
- Eg: log( ), sqrt( ), abs( ), cos( ), etc.

General syntax of function

- return_type function_name(arg_type argument, ...) {
  local_variable_type local_variables;
  executable statement(s);
  return return_value; }
- Example:

```
int add( int a, int b ) {
        int sum;
        sum = a + b;
        return sum; }
```

Return type

- The return type indicates what kind of data the function returns.
- In the example above, the function returns an int.
- In C, functions are not required to return a value.
- Way to indicate that a function does not return a value is to use the return type void.
- If no return type is given, the function returns an int.

Function name & formal parameter list

- The function name is an identifier by which this function will be known
- Same naming rules as applied to variable names
- Following the function name are a pair of parentheses containing a list of the formal parameters, (arguments) which receive the data passed to the function.

Formal parameter list

- int add(int a, int b) {
  int sum;
  sum = a + b;
  return sum; }
- Even if several parameters are of the same type, each must have its type given explicitly.
- If a function takes no parameters, the parameters may be left empty or write void inside the parentheses
- Eg: int add( )

Function body

- The body of the function is enclosed within curly {} braces
- The variables inside the function body are known as local variables to the function
- int add(int a, int b) {

```
int sum;    //local variable
sum = a + b;
return sum; }
```

- The return statement exits the called function and returns control back to the calling function.

Return statement

- A single return value (of the appropriate type specified along with the function name) is returned.
- Parentheses are allowed but not required around the return value.
- A function with a void return type will not have a return value after the return statement.
- **Is more than one return statement allowed in a function?**
- **but only one will ever be executed by any given function call.**
- int add(int a, int b) {

```
int sum;    //local variable
sum = a + b;
if (sum>30)
   return sum;
else
 return sum-2;
}
```

Function Prototypes / Function Declarations

- When a call to a function is executed: compiler checks to see that the correct number and types of data items are being passed to the function
- Make certain that all functions appear earlier in a file than any calls to them

  OR

- Use **function prototypes** ie. way of declaring to the compiler what data a function will require, without actually providing the function itself

Example of function prototype

```
void add( double x, double z, double b );
int main( void ){
        double x =1.0 , y= 2.0 , z=3.0 ;
        add( x, y, z); //calling a function
        return 0;}
void add (double x, double z, double b){
   double sum;
    sum= x+z+b; printf( " %d ", sum );}
```

Another Example of a function

```
double product( double x, double z, double b );
int main( void ){
        double x =1.0 , y= 2.0 , z=3.0, p ;
        p=product( x, y, z); //calling a function
        printf( "%f ",p);
         return 0;}
double product (double x, double z, double b){
        return( x*z*b);}
```

- A function returns a single value by return statement.
- Any variable changes made within a function are local to that function.
- A calling function's variables are not affected by the actions of a called function.

Example

```
float product( float x, float z, float b );
int main( void ){
        float x =1.0 , y= 2.0 , z=3.0, p ; int a=0;
        p=product( x, y, z); //calling a function
        printf( "%f ",p);
        printf("value of a in main :%d ", a);
        return 0;}
float product (float x, float z, float b){
       int a =20;
        printf("  value of a in product  : %d ", a);
        return( x*z*b);}
```

Another example

```
int max(int x , int y ){
        if (x > y)      return x;
        else   return y; }

int main(void){
        int a = 10, b = 20;
        int m = max(a, b);
        printf("m is %d", m);return 0;}
```

Parameter Passing mechanism

- The parameters passed to function are called ***actual parameters***. Eg: 10 & 20 in the following example
- The parameters received by function are called ***formal parameters***. For example, in the below code, x and y are formal parameters.

```
int max(int x , int y ){
        if (x > y)return x;
        else      return y; }

int main(void){
        int a = 10, b = 20;
        int m = max(a, b);
      printf("m is %d", m);return 0;}
```

Parameter passing mechanisms

- There are two most popular ways to pass parameters to a function:

- ***Pass by Value or Call by value***

- ***Pass by Reference or Call by reference***

Pass by Value or Call by value

- Values of actual parameters are copied to function's formal parameters
- These two types of parameters are stored in different memory locations
- Any changes made inside functions are not reflected in actual parameters of caller.
- void fun(int x){
  x = 30;}
- int main(void){
  int x = 20;
  fun(x);
  printf("x = %d", x);
  return 0;}
- x=20

Pass by Reference or Call by reference

- Both actual and formal parameters refer to same locations or address
- Any changes made inside the function are actually reflected in actual parameters of caller.
- In C, pointers are used for call by reference
- What is a pointer?

Brief introduction on pointers

- What is a pointer?
- Pointer is a variable that contains the address of a variable
- How do we declare a pointer variable?
- Syntax of pointer declaraion:

  data_type *pointer_name;
- Example: int *ptr;
- int a; //declaration of an integer

  int *ptr; //declaration of a pointer variable which points to an integer variable

  ptr = &a; //assigning the address of variable to the pointer variable—initialization of a pointer variable

Pointers

- * operator : indirection or dereferencing operator
- Every pointer points to a specific data type
- Does not point to an expression

```
void main() {
        int i=10;
       printf("\nValue of :%d“   ,i);
       printf("\nAddress of i :%d",&i);  }
```

Predict the output

```
main(){
        int x=10;  int  y=20;
        // variable ptr1 is declared as a pointer variable
        int  *ptr1=NULL;
         //it is initialized as  0
        printf( " First: %d %d %d", x,y, ptr1);
        ptr1 = &x;
        //ptr1 is assigned x's address
        y = *ptr1;
        // y is assigned the value ptr1 is pointing to
        printf( "Second: %d %d %d", x,y, *ptr1);
        *ptr1 =0;
          // value at ptr1 is now 0
        printf( "Third: %d %d %d", x,y, *ptr1);   }
```

Ans: First: 10 20 0 Second:10 10 10 Third: 0 10 0

Predict the output

```
main()
{
int x=5,y=15,*p1,*p2;
p1=&x; p2=&y;
*p1=6;
*p2=*p1;
p1=p2; *p1=2;
printf("%d %d", x,y);
}
```

Trace the code

```
main()
{
int x=5,y=15,*p1,*p2;
p1=&x; p2=&y; //value at p1 =5 , value at p2=15
*p1=6;  // value at p1 =6 , hence x=6
*p2=*p1;//  value at p2 =6 , hence y=6
  p1=p2; // p1 points to p2
*p1=2; // value at p1 =2, hence value at y=2
printf("%d %d", x,y);
}
```

Ans: 6 2

Back to Call by Reference

- In call by reference, we already know:
- Both actual and formal parameters refer to same locations or address
- Any changes made inside the function are actually reflected in actual parameters of caller.
- In C, pointers are used for call by reference

Simple example of call by reference

```
//function definition by call by reference
void fun(int *ptr){
    *ptr = 30; }

int main() {
  int x = 20;
  fun(&x); // call by reference
  printf("x = %d", x);
  return 0;}
```

Another example

```
#include <stdio.h>
int main () {
        int a = 100; int b = 200; /* local variable definition */
        printf("Before swap, value of a : %d\n", a );
        printf("Before swap, value of b : %d\n", b );
        swap(&a, &b); /* calling a function to swap the values */
        printf("After swap, value of a : %d\n", a );
        printf("After swap, value of b : %d\n", b );
        return 0; }

void swap(int *x, int *y) {
         int temp;
        temp = *x; /* save the value of x */
        *x = *y; /* put y into x */
        *y = temp; /* put temp into y */
         return; }
```

Recursion

- What is recursion?
- Recursion is the process of repeating itself
- If a program calls a function inside the same function, then it is called a recursive call of the function.
- Trivial way to represent a recursive function:

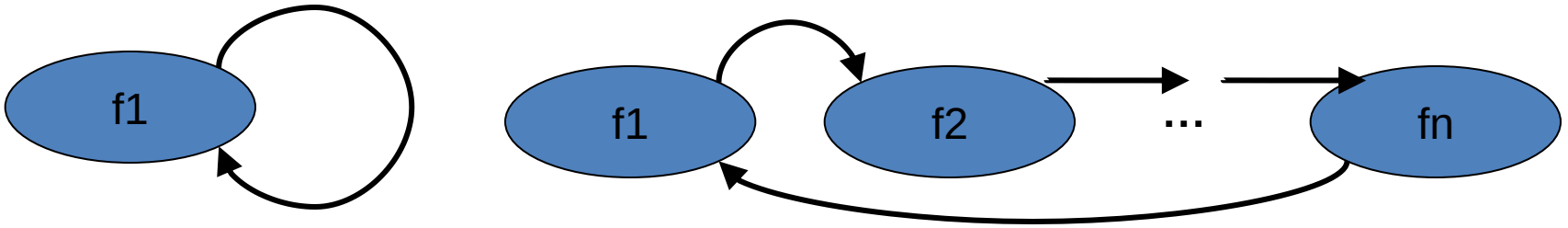
```
void recursion() {
        recursion(); /* function calls itself */ }
int main() { recursion(); }
```

- Sometimes, the best way to solve a problem is by solving a smaller version of the exact same problem first
- Recursion is a technique that solves a problem by solving a smaller problem of the same type
- a function that calls itself

  Directly or

  Indirectly (a function that is part of a cycle in the sequence of function calls.)

Pictorial representation of direct and indirect recursive calls



Direct recursive call

Indirect recursive call

Syntax

```
function_name(parameter list)
{
        …
        //'c' statements
        …
        function_name(parameter values)   //recursive call
        …
 }
```

Factorial of a number

- F**actorial** of a non-negative integer n, represented as n!
- The product of all positive integers less than or equal to n.
- For example, 5 ! = 5 × 4 × 3 × 2 × 1 = 120.
- 4! = 4 × 3 × 2 × 1
- 3! = 3 × 2 × 1
- 2!=2 × 1
- 1! = 1

Representation of factorial

- Factorial (5) = 5* factorial(4)
- Factorial (4) = 4* factorial(3)
- Factorial (3) = 3* factorial(2)
- Factorial (2) = 2* factorial(1)
- Factorial (1) = 1* factorial(0) = 1
- If n>0 , Factorial (n) = n* factorial(n-1)
- If n=0 Factorial (n) =1
- This is the base case of recursion
- Recurrence relation

Factorial Problem defined recursively

- There are many problems whose solution can be defined recursively

Example: *n factorial*

$$n! = \begin{cases} 1 & \text{,if } n = 0 \\ (n\text{-}1)!*n & \text{,if } n > 0 \end{cases}$$

(*recursive* solution)

$$n! = \begin{cases} 1 & \text{,if } n = 0 \\ 1*2*3*\ldots*(n\text{-}1)*n & \text{, if } n > 0 \end{cases}$$

(*closed form* solution)

(also called as iterative method)

Coding the factorial function

- Iterative implementation

```
int Factorial(int n)
{
 int fact = 1;

 for(int count = 2; count <= n; count++)
   fact = fact * count;

 return fact;
}
```

Coding the factorial function (An Example of Recursive Call)

- Recursive implementation

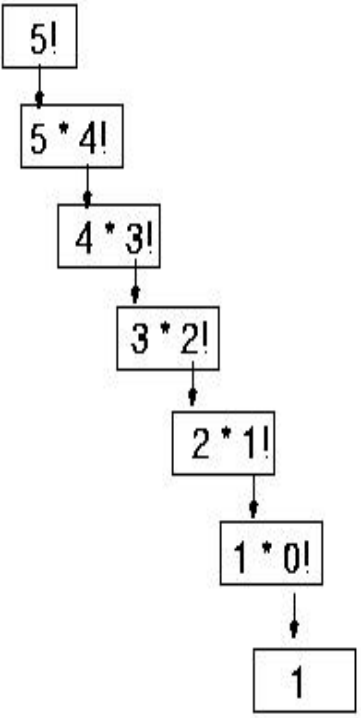
```
int Factorial(int n)
{
 if (n==0)  // base case
   return 1;
 else
   return n * Factorial(n-1);
}
```

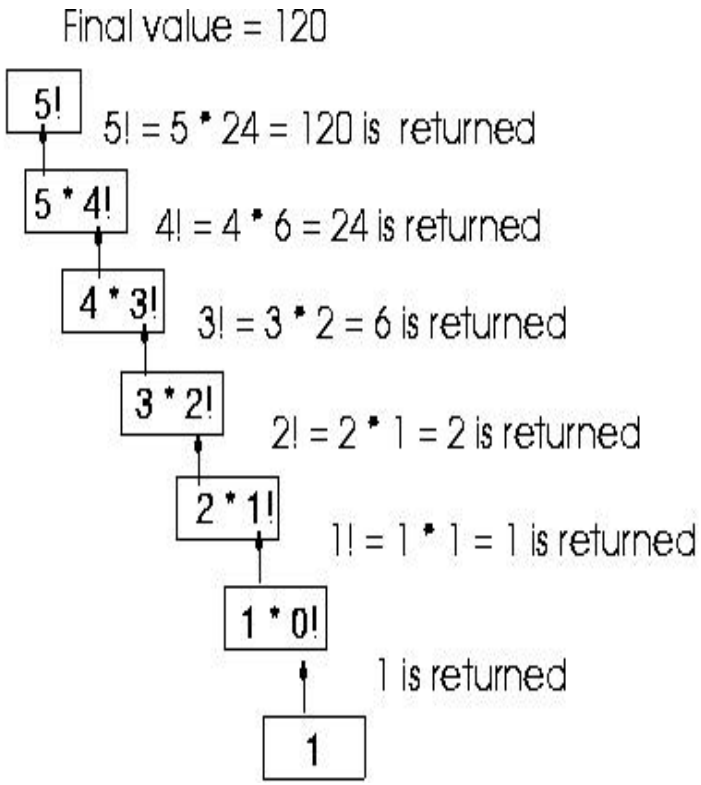
Complete code: Factorial using recursion

```
int Factorial(int n)
{
 if (n==0)  // base case
   return 1;
 else
   return n * Factorial(n-1);
}
int main() {
        int i = 5;
        printf("Factorial of %d is %d\n", i, Factorial(i));
        return 0; }
```

Coding the factorial function (cont.)

```
int Factorial(int n)
{
 if (n==0)  // base case
   return 1;
 else
   return n * Factorial(n-1);
}
int main() {
        int i = 5; int f;
       f=Factorial(i);
       printf("Factorial of %d is %d\n", i, f);
        return 0; }
```





Thank You