Functions

Functions

- A function is a self-contained block of statements that perform a coherent task of some kind.
- Functions break large computation into smaller ones.
- Some common tasks written in the form of functions can be reused.

A simple function

```
#include<stdio.h>
float max(float x, float y);
main( )
{
 float a,b,m;
 scanf("%f %f", &a, &b);
 m =  max(a,b);
 printf("Max : %f\n", m);
}
float max(float x, float y)
{
    if(x > y) return x;
    else return y;
}
```

Function declaration

Function call

Function definition

One more example

```
#include <stdio.h>
int square( int  y);   /* function prototype */
int main( )
{
   int x;
   for ( x = 1; x <= 10; x++ )
      printf( "%d  ", square( x ) );
   printf( "\n" );
   return 0;
}
int square( int y )
{
   return y * y;
}
```

Main is also a function.
It returns an int

If unspecified, then by default a function returns int

What we should learn

1. What is the control flow when a function is called.
2. Function declaration or prototype.
3. Function call.
4. Function definition.

Control Flow

- Execution starts with `main( )`, when a function call like `z = func(exp1, exp2, …);` is encountered then

i. exp1, exp2, … are evaluated
ii. The values of exp1, exp2, … are passed to func( )
iii. main is suspended temporarily
iv. func( ) is started with the supplied values
v. func( ) does the processing
vi. func( ) returns a value which is assigned to z
vii. main( ) is resumed.

Function prototype

- Every function should have a declaration which specifies the function name, its arguments types and its return type.
  - This feature was borrowed by ANSI C committee from the developers of C++.
  - A function declaration (prototype) tells the compiler regarding the arguments type, their order, return type.
  - This can allow the compiler to detect errors in function call and function definition.
  - Remember this feature was not there before ANSI C and for compatibility purposes, still the compiler may not give any errors when prototype is missing.
- Eg: `float func(int j, float k);`
- Optionally j, k could be omitted, that is, `float func(int, float);` is also valid.
- This says that `func` is a function which takes two arguments (first one is `int` and second one is `float`) and return a `float`.

void

- What if a function does not return any value at all.
- In this case the function returns a void and hence the return type should be `void`.
- Eg: `void f1(void);`
- The function `f1` does not take any arguments and does not return any value too!!
- Why you need such functions?

void

```
void nothing(void);
main( )
{
        nothing( );
}
void nothing(void)
{
        printf("when you need to do some fixed thing\n"
        "then you can use a func. like this \n");
        return;
}
```

Function definition

- Return-type function-name(parameter-list)
  {
  declarations
  statements
  }

To find maximum of three ints

```
#include <stdio.h>
int maximum( int, int, int );   /* function prototype */
int main()
{
   int a, b, c;
   printf( "Enter three integers: " );
   scanf( "%d%d%d", &a, &b, &c );
   printf( "Maximum is: %d\n", maximum( a, b, c ) );
   return 0;
}
int maximum( int x, int y, int z )
{
   int max = x;
   if ( y > max )  max = y;
   if ( z > max )  max = z;
   return max;
}
```

You cannot use max in main( )

Declaration. When this function returns max disappears !!

max is called a local variable for this function.

Let us swap

This does not work !

```
void swap(int, int);
 main( )
{
        int a, b;
        a = 10, b = 20;
        swap(a, b);
        printf("a = %d, b = %d\n", a, b);
}
 void swap(int x, int y)
{
        int t;
        t = x; x = y; y = z;
        return;
}
```

```
$./a.out
 a = 10, b = 20
$
```

Why swap failed !?

- when the function is called the values 10 and 20 are passed.
- In swap( ), x = 10 and y = 20.
- swap( ) actually swapped x and y
- But this doesn't mean that a and b are swapped.

Is there a way for the swap to correctly work?

How swap can correctly work

- There are three ways how information is passed to a function.
  - **Passing values** : this is what our swap is doing
  - **Passing references** : In some languages it is possible to pass the actual variables, but this is not present in C
  - **Passing addresses**: Explicitly use addresses in the function, so that the function can modify the actual variable using these addresses.
- You need to pass addresses of a and b
- Based on these addresses, respective values should be swapped
- This we will see when we discuss about pointers.

Tips and traps

- Omitting the return-type in a function definition causes a syntax error if the function prototype specifies a return type other than int.
- Forgetting to return a value from a function that is supposed to return a value can lead to unexpected errors.
- Returning a value from a function whose return type is void causes a syntax error.
- Even though an omitted return type defaults to int, always state the return type explicitly. The return type for main is however normally omitted.

Recursion

- A function can call itself !!
- This is a very good strength of the language.
- Many big problems can be solved easily with recursions
- Eg: n! = n * (n-1)!
- Can we do like
  - int factorial(int n)

```
{
        int t;
        t =  factorial(n-1);
        return ( n *  t );
}
```

- Is there any problem with this?

Recursion

- If you are not careful then it is easy to fall in infinite recursion.
  - int factorial(int n)
    {
    int t;
    if (n < 0) return -1; /* an error code */
    else if (n == 0) return 1;
    else {
    t = factorial(n-1);
    return (n * t);
    }
    }
- what happens when we call

```
x = factorial(4);
```

Recursion

- Fibonacci series
  - 0, 1, 1, 2, 3, 5, 8, 13, 21, …
  - fib(0) = 0
  - fib(1) = 1
  - fib(2) = 1
  - ……
  - fib(n) = fib(n-1) + fib(n-2)

```
int fib(int n)
{
        if(n == 0 || n ==1) return n;
        else return fib(n-1) + fib(n-2);
}
```

- what happens when we call z = fib(3);

Recursion

- Recursion is useful when the problem can be defined recursively.
  - To find sum of first n natural numbers.
    - Sum(0) = 0
    - Sum(n) = n + Sum(n-1)
  - ```
    int sum(int n)
    {
    if (n == 0) return 0;
    else return n + sum(n-1);
    }
    ```
    ```

Recursion

- One of the common pit-fall is: forgetting or wrongly stating the termination condition.
- Whatever you do using recursion can be done without recursion also, but using some loop structures.
- A copy of the function is created whenever a function call occurs, and hence recursive functions can be heavy on the memory.
- Nevertheless, solution becomes conceptually simpler.

Replacing recursion

- To find sum of first n natural numbers
  - ```
    int sum( int n)
    {
      int t = 0, j;
      for(j=1; j<=n; j++)
            t = t + j; /* alternatively, t += j; */
      return ( t );
    }
    ```
    ```

Fibonacci without recursion

- Algorithm to find fib(n)
  1. n = read from key board.
  2. Let a = fib(0), b = fib(1)
  3. If n == 0 or n ==1 then output a or b and exit;
  4. Let j = 1;
  5. while (j < n) Do
     1. c = a + b;
     2. a = b;
     3. b = c;
     4. j = j+1;
  6. Output(fin(n) is c);

Fibonacci without and with recursion

```
int fib( int n)
    {
        int  a = 0,  b = 1,  c;
        int j = 1;
        if (n == 0 || n == 1) return n;
        while(j < n) {
                c = a+b;
                a = b;
                b = c;
                j ++;
        }
        return c;
    }
```

```
int fib( int n)
    {
        if (n == 0 || n == 1)
            return n;
        else
            return fib(n-1) + fib(n-2);
    }
```

Recursion

- Problems that can be defined using recursion will mostly have good recursive solutions.
- Using recursion often simplifies the solution conceptually.
- Some of the data structures (like trees) can be easily dealt with recursions.
  - Mostly these aspects you will be studying in Data Structures Course.

Functions

- Appropriate use of functions makes your program a very structured one.
- It might be easy to debug a well structured program.
- A program is normally a set of functions which communicate among themselves using arguments and return values.
  - Some times functions communicate with each other using global variables and addresses.