**Function**

Let us continue our discussion of functions with the following example from 1\_swap.c.

// Does not swap the arguments!!

void swap(int x, int y)

{

int temp = x; x = y; y = temp;

}

int main()

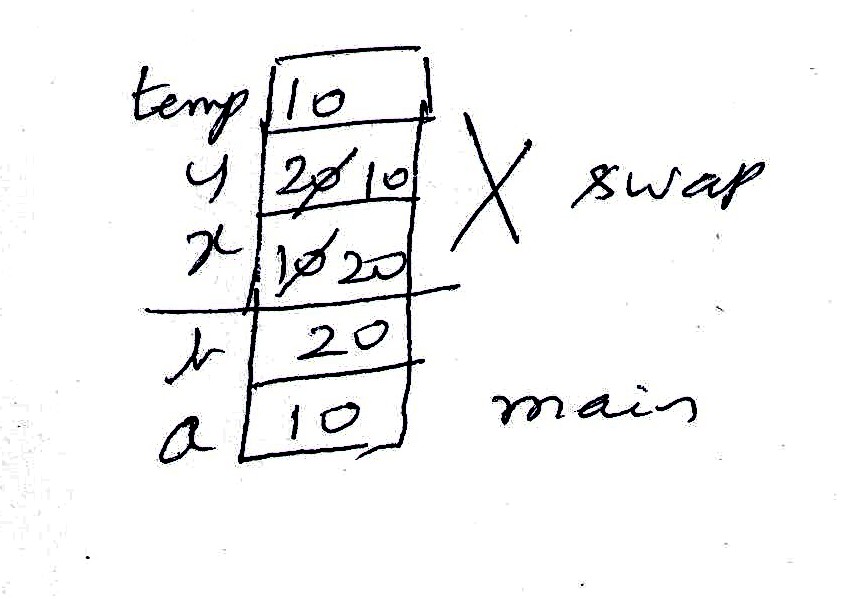
{

int a = 10; int b = 20;

swap(a, b);

printf("a : %d b : %d\n", a, b);

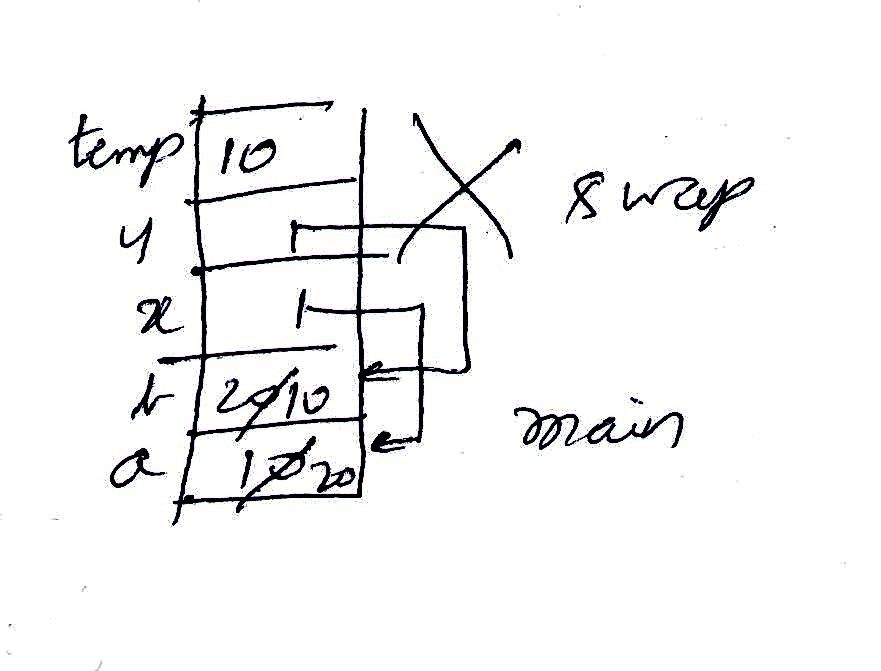
}



We know that the parameter passing in ‘C’ is by value. So, the copies of a and b get swapped; but not a and b.

To change a variable of the caller, the callee should receive pointer to the variable.

This works!



void swap(int \*x, int \*y)

{

int temp = \*x; \*x = \*y; \*y = temp;

}

int main()

{

int a = 10; int b = 20;

swap(&a, &b);

printf("a : %d b : %d\n", a, b);

}

**Recursion:**

A function calling itself directly or otherwise is said to be recursive. We may use if and recursion instead of a looping structure. In recursion, we try to express the solution to a problem in terms of the problem itself, but of a smaller size. We also require solution for one or more simple cases – called escape hatch or base case.

Let us examine how to trace a recursive function.

In this example, there are two places where recursive calls are made. We should know where these recursive calls will return. We can mark the position of the call with the number of the call – so that we will know where to resume the following of the code.

This function finds the number of 1s in the binary representation of the number.

If the number is odd, we count 1 and remove the unit bit and call the function on the changed number.

If the number is even, we remove the unit bit then we call the function on the changed number.

If the number is 0, we return 0.

int what(int n)

{

if(n == 0)

{

return 0;

}

else if(n % 2)

{

return 1 + what(n / 2);

// 2 => 2

// 5 => 1

// 6 => 0

}

else

{

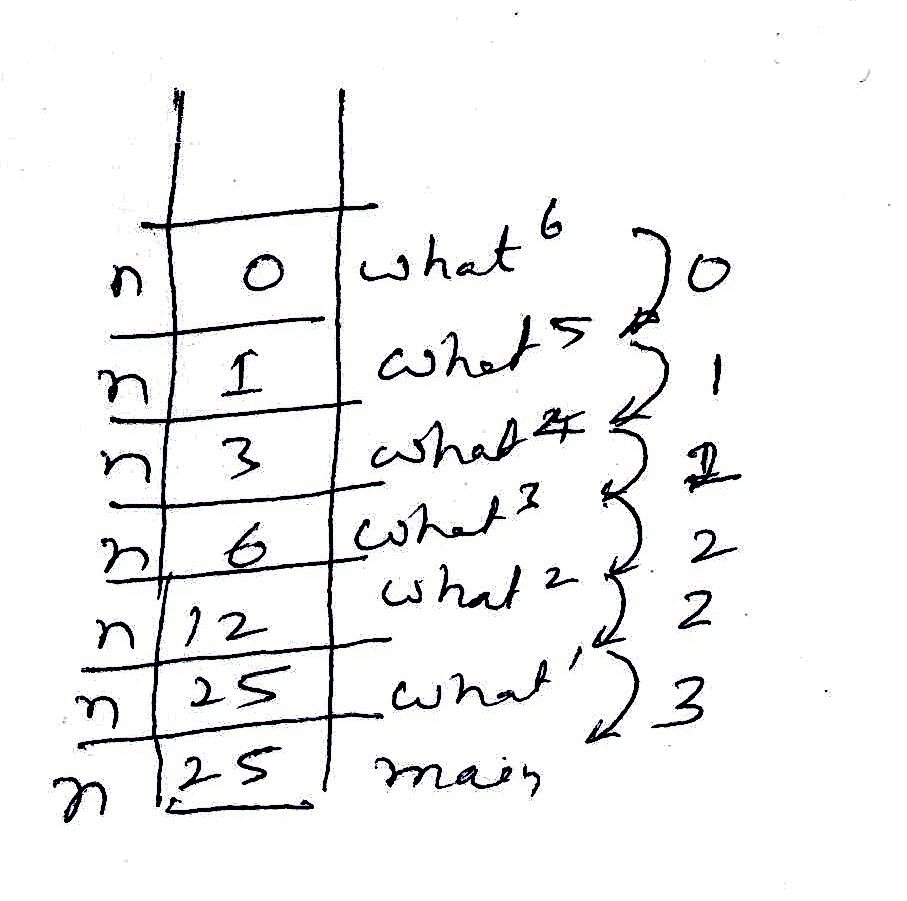
return what(n / 2);

// 3 => 2

// 4 => 2

}

}



int main()

{

int n = 25;

printf("val : %d\n", what(n));

// 1 => 3

}

**Recursion: Another example:**

This is the file 3\_what.c.

What does this recursive function do? You may want to pause and think about this function.

#include <stdio.h>

int what(int a, int n)

{

if(n == 0)

return 1;

else if(n % 2)

return a \* what(a \* a, n / 2);

else

return what(a \* a, n / 2);

}

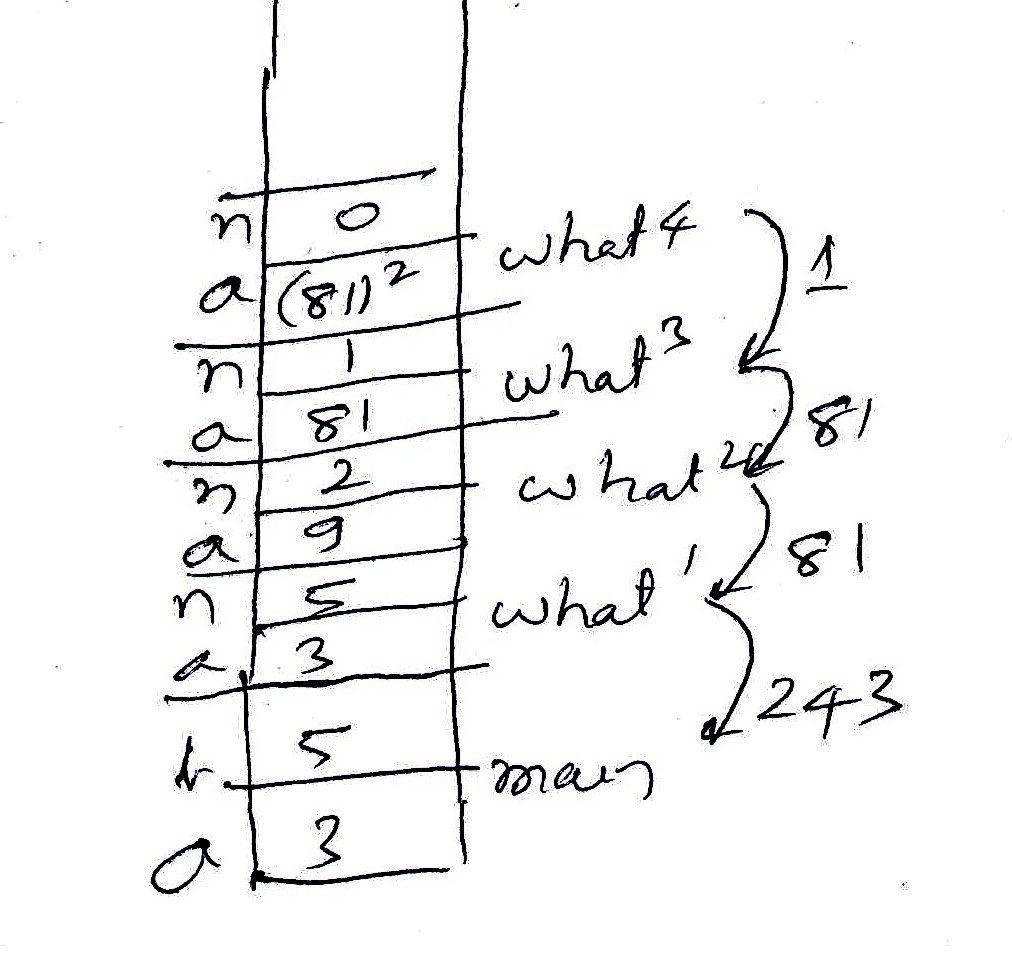
int main()

{

int a = 3; b = 5;

printf("%d\n", what(a, b));

}



It finds a to the power of n. The definition is as follows.

If n is 0, the result is 1.

if n is even, square a and then exponentiate the result to n / 2.

If n is odd, square a and then exponentiate the result to n / 2 and then multiply this with a.

**Parameter passing and pointers:**

This is an interesting example of understanding parameter passing.

The loop of this function finds the greatest common divisor of m and n. p and q point to the corresponding arguments. m and n are copies of \*p and \*q and have nothing to do with the arguments.

\*p /= m; \*q /= n;

This statement causes division of the arguments by the greatest common divisor.

void what(int \*p, int \*q)

{

int m = \*p; int n = \*q;

while(m != n)

{

if(m > n)

{

m -= n;

}

else

{

n -= m;

}

}

\*p /= m; \*q /= n;

}

int main()

{

int a = 25; int b = 15;

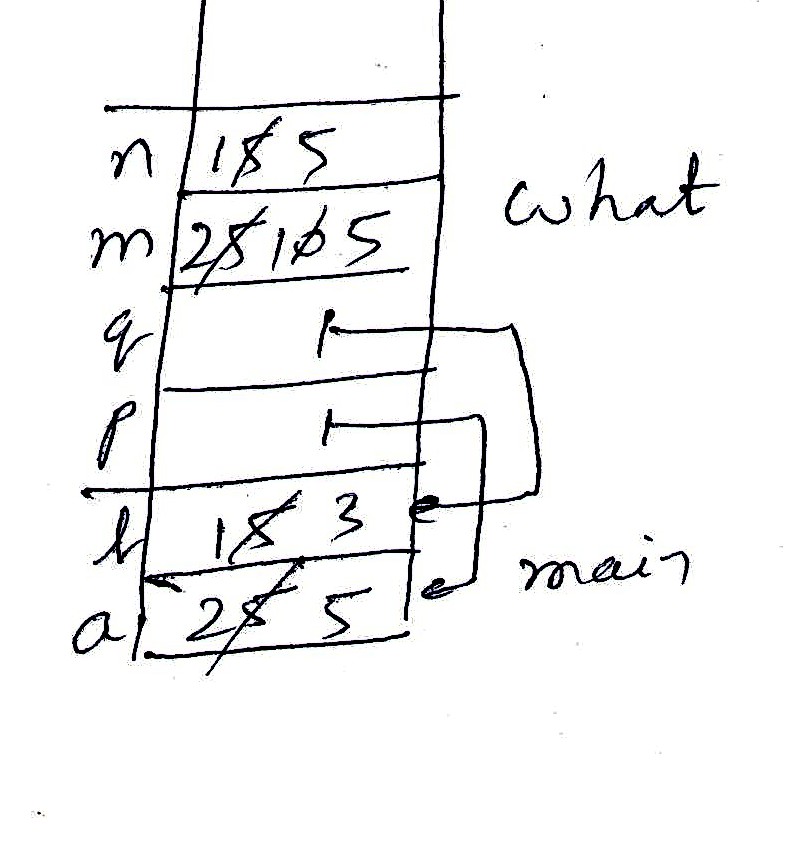
what(&a, &b);

printf("%d %d\n", a, b);

what(&a, &a);

printf("%d\n", a);

}



int a = 25; int b = 15;

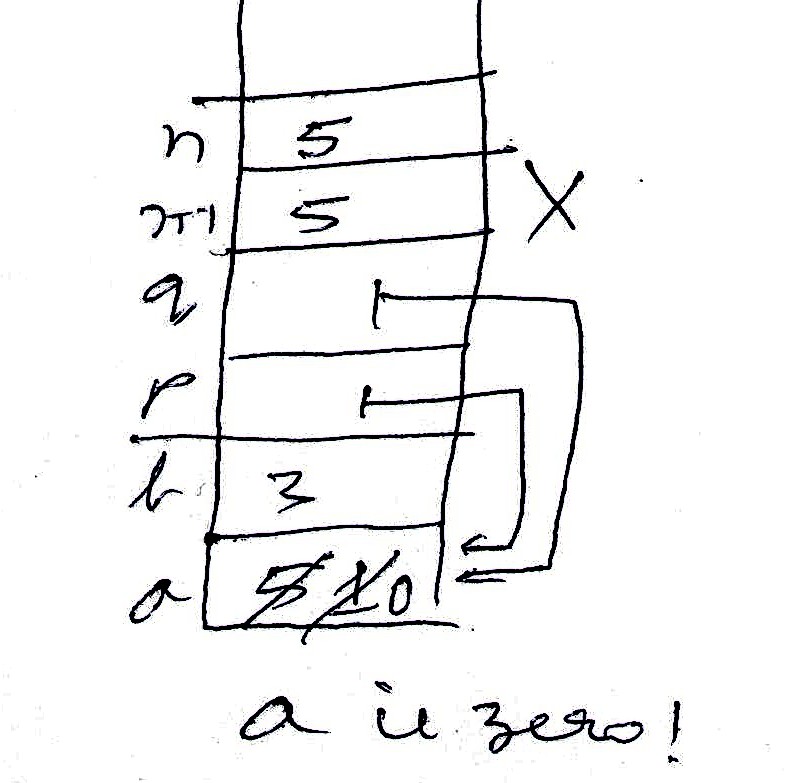
what(&a, &b);

printf("%d %d\n", a, b);

So, a and b get divided by 5. So a becomes 5 and b becomes 3.

The next part of the code is interesting. What do you expect the variable a to become. - 1 ? Be in for a surprise!

Both p and q point to the same variable. This is called aliasing. m and n become 5. The loop is not entered. We update \*p that is same as variable a by dividing by m. So a becomes 1. We update \*q that is also a by dividing by n. So, a becomes 0!!



**Separating the interface from implementation:**

Our requirement let us say is to check whether a given number is a palindrome.

We can write the function ourselves. But we could use a function in the library or written by somebody as long as it satisfies the requirement.

Let us know that there is a function with the following signature.

int is\_palindrome(int n);

Then we can write the code like this.

int n;

scanf("%d", &n);

// check whether it is a palindrome

if(is\_palindrome(n))

{

printf("%d is a palindrome\n", n);

}

else

{

printf("%d is not a palindrome\n", n);

}

The writer of the function does not know how we use. We do not know how the developer of the function has written the function. The developer may write the logic together in the same function or call some other function to reverse the number.

int rev(int n)

{

int r = 0;

while(n)

{

r = r \* 10 + n % 10;

n /= 10;

}

return r;

}

int is\_palindrome(int n)

{

#if 0

int rev = 0; int m = n;

while(n)

{

rev = rev \* 10 + n % 10;

n /= 10;

}

return m == rev;

#endif

return n == rev(n);

}

So, we separate the client code and server code. The common code is put in the header file. It contains declaration of functions. The header file acts like the interface between the client and server.

Multiple files and make file:

The next example has 3 files.

6\_server.c – provides implementation of squaring and cubing

6\_client.c – uses these functions squaring and cubing

6\_server.h : interface for these functions.

We compile both the .c files then link them to create the loadable image.

Each time we change any of the files, we should remember to recreate the files by using appropriate commands.

Can we transfer these two ideas to a tool?

- dependency of files

- command(s) to be executed when the dependent files change.

This tool called the make tool takes care of these.

This is the file : mymakefile.mk

The first line indicates the file created last in the sequence – the loadable image(executable). This is called a target. Then this line indicates which all file it depends on. They are called sources.

The next lines(s) start with a tab and any command to be executed when the sources are modified.

This sequence of dependencies and commands repeat.

The command to be executed is : make -f mymakefile.mk.

a.out : 6\_client.o 6\_server.o

gcc 6\_client.o 6\_server.o

6\_client.o : 6\_client.c 6\_server.h

gcc -c 6\_client.c

6\_server.o : 6\_server.c 6\_server.h

gcc -c 6\_server.c

Experiment changing the files and observe that only required files are re-created.