

**Introduction to Programming in C**  
**Prof. Satyadev Nandakumar**  
**Department of Computer Science and Engineering**  
**Indian Institute of Technology, Kanpur**

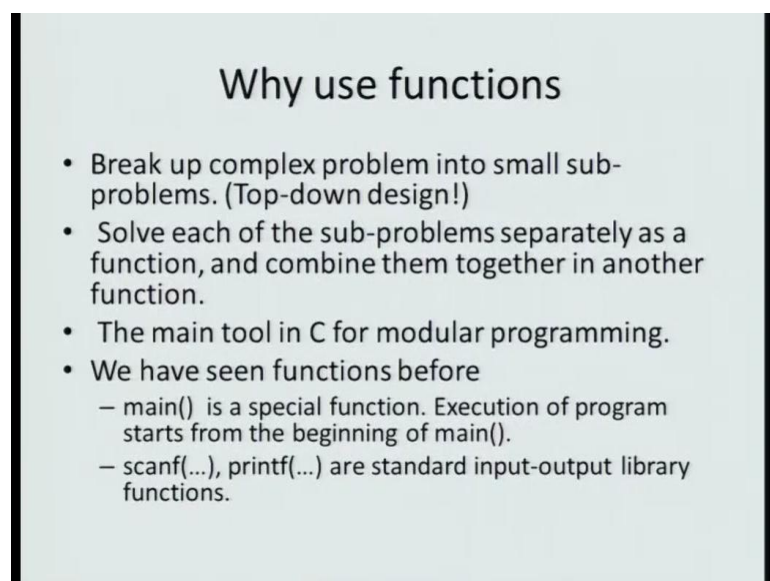
**Lecture - 25**

(Refer Slide Time: 00:11)



In this session we are going to introduce a new concept of programming in C called functions. So, initially, let us just try to motivate why we need functions, and then we will try to see whether programming becomes easier, if we have functions.

(Refer Slide Time: 00:26)



So, let us say that, why do we need functions? There are essentially 2 different reasons

for it. And I will mention these reasons one after the other. The first reason is to break up a complex problem into simple sub problems. All of us, for example, like to drop to college, less saying that these are the things I wish to accomplish today. So, step 1, you know, get to college, step 2 - attend classes, step 3 - finish home work or something like that. And then each of those main task will have several sub task. In order to get to college, maybe you need to renew the ticket subscription, get on the bus, get to college, and so on.

So, each of those higher level task involves several search smaller sub task. And conceptually, it is cleaner to say that these are the big level things that I want to do. Each of those big level task have several sub tasks, so that I can think of it, what I want to accomplish in a layer wise manner. So, this is something that we do intuitively. We always break up complex problem into simpler sub problems so that we can analyze the simpler sub problem and perform it completely, and then come back to the bigger problem. So, we need to solve it each separately.

And the main tool for this programming in C which allows you to accomplish breaking up a complex sub problem into simpler sub problems is what is known as functions. So, this enables you to do what is known as modular programming in c. And functions are not new. We have already seen 3 functions in particular - main was a function that we always wrote, and then we have print f and scan f which we use for outputting and inputting respectively. So, let us just motivate the notion of functions by using the second motivation that I was talking about.

(Refer Slide Time: 02:48)

$${}^n C_k = \frac{n!}{k!(n-k)!}$$

```
main()
{
  int a, b, c; float result;
  [ ] ← n!
  // a = n! */
  [ ] → k!
  // b = k! */
  [ ] → (n-k)!
  // c = (n-k)! */
  result = a/(b*c);
}
```

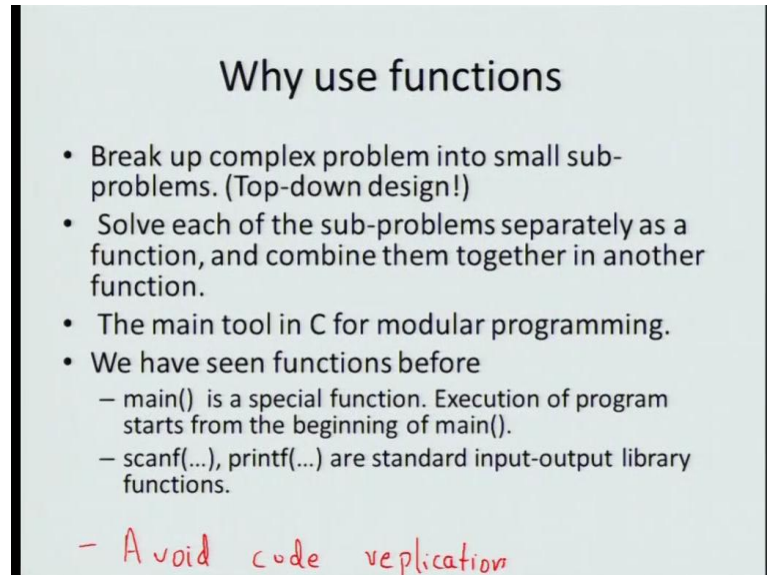
So, suppose, you have, you want to say, I want to compute  $n C k$ , which is  $n$  factorial upon  $k$  factorial  $n$  minus  $k$  factorial, correct? So, this is the definition of  $n$  choose  $k$  for  $n C k$  as it is known. Now, suppose I want to write this,  $n$  code this into C program, so I will have, let us say, a main function. And then inside the main function I will have, let us say, 3 variables –  $a$ ,  $b$ ,  $c$ , and then float result because the result of a division will be a float. So, I will have, what should I do intuitively, one way to do it is I will have a block of code which says it will calculate  $n$  factorial which is the numerator, then I will say that  $a$  equal to  $n$  factorial; at end of this, let us say, that  $a$  stores  $n$  factorial.

Then, I will have another block of code which says that I will calculate  $k$  factorial. And then this will say, let us say,  $b$  equal to  $k$  factorial. And the third block of code will calculate  $n$  minus  $k$  factorial; let us say, I will store this in  $c$ . And then I will say, ok result equal to  $a$  by  $b$  times  $c$ , some code that looks like this. And you would notice what is inconvenient about it; all these 3 blocks of code, once we complete it, will look very similar. They are all calculating the factorial of a particular number.

But, there is nothing in  $c$ , which will, that using the features that we have seen so far, which will tell us that this code, this code and this code are essentially the same, and I need to write that similar code only once. So, there is no simple way to use loops to accomplish these. So, it seems like this redundant business of writing similar code again and again can be avoided. So, this is the second motivation for introducing the notion of

functions which is basically to avoid duplication of code.

(Refer Slide Time: 05:39)



The slide is titled "Why use functions" and contains the following content:

- Break up complex problem into small sub-problems. (Top-down design!)
- Solve each of the sub-problems separately as a function, and combine them together in another function.
- The main tool in C for modular programming.
- We have seen functions before
  - main() is a special function. Execution of program starts from the beginning of main().
  - scanf(...), printf(...) are standard input-output library functions.

– Avoid code replication

So, here is a side benefit of functions, avoid code replication. We have already seen loops to some extent avoid code replication. But, here is a newer method to avoid code replication in a greater unit. So, the second reason why we write functions is to avoid writing similar code again and again.

So, let us try to write functions by motivating it with the help of an example. This example will show the benefit of how we can avoid code duplication using functions, and also how we can breakup a complex problem into simpler sub problems. So, in this I will introduce the problems similar to what have we seen before.

(Refer Slide Time: 06:41)

### An example

- Problem: first line of input is  $n$ —the number of numbers to follow in the next line. Count the number of successive pairs of numbers that are relatively prime (i.e., gcd is 1).
- Example Input 

8
4 6 16 7 8 9 10 11
- Relatively Prime Pairs are:  

16 7, 7 8, 8 9, 9 10, 10 11
-----------------------------
- The pairs 4 6 and 6 16 are not relatively prime.
- Answer 5

We have a sequence of numbers. The first number tells you how many inputs there are. And then what we need to do is to pick out the numbers which are relatively prime in these sequences. So, 2 numbers are relatively prime if their gcd is 1. So, 16 and 7 are relatively prime; 4 and 6 are not because they have a common factor of 2; 6 and 16 are not, they have a common factor of 2; 16 and 7 do not have a common factor other than 1; 7 and 8 are similarly relatively prime; 8 and 9 are relatively prime; 9 and 10 are relatively prime; and 10 and 11 are relatively prime.

So, these are the relatively prime pairs. And we need to write a function which given a sequence of these numbers count how many pairs, how many successively occurring numbers are relatively prime to each other. In this case there are 5 such pairs.

(Refer Slide Time: 07:43)

### Design the program

- Suppose we write a **function** `iscoprim`(a, b) that takes two positive integers a and b and returns 1 if a and b are co-prime, and returns 0 otherwise.
- The function declaration in C would be  

```
int iscoprim ( int a, int b )
```

↑            ↑            ↑            ↑  
type of return value    function name    input arguments
- This means that `iscoprim` is a function that takes two integer arguments, the first one called a, the second one called b. The function itself returns an integer value.

So, in this problem we can clearly see that there is a sub problem which is, given 2 numbers are they relatively prime? That is one sub problem. And if we have the solution to that sub problem then we can compose the solution to the whole problem as follows. Given 2 numbers, I check whether they are relatively prime. If they are relatively prime I will increment the count of the relatively prime pairs I had seen so far, otherwise I will skip to the next pair and see whether they are relatively prime. So, for each new pair of numbers I am seeing that is the sub task of checking whether they are relatively prime.

So, let us say that suppose we have a function; a function is something that we will see in a minute. Suppose we have a small component which will perform the task of testing `iscoprim a, b`. So, `iscoprim a, b`, that function will take 2 numbers a and b and check whether they are relatively prime or not. If a and b are relatively prime it evaluates to 1. It is, we say that it returns 1 if they are relatively prime; and if they are not coprime to each other, if there not relatively prime, then it has to return as 0. So, it has to evaluate to 0.

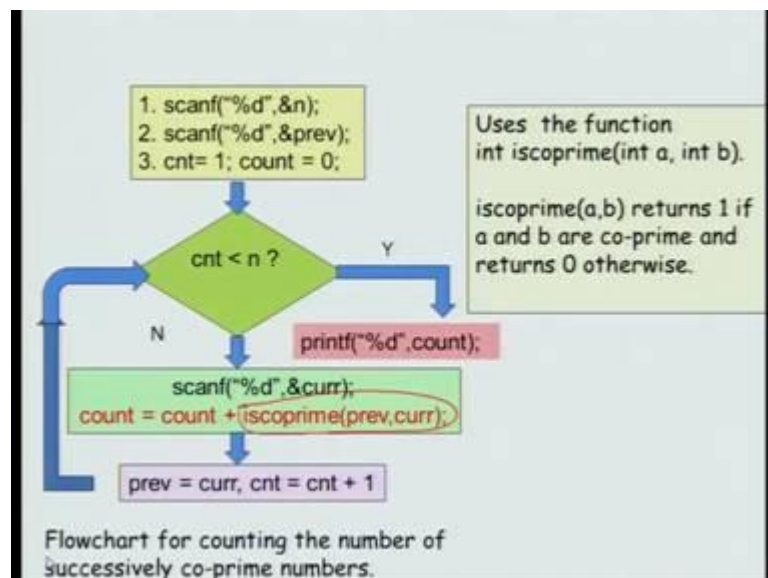
Now, associated with every function there are 3 concepts. We will see them one by one. There is this declaration of a function which says what does the function look like, what is the type of the function. So, the declaration of the function will be written in the following way. It will be written as `int iscoprim int a, int b`. This means that `iscoprim` is the function name, and then it takes 2 arguments - a and b which are of type int; so int a and int b. If we had written another function which takes a float a and int b, we would

say, function int, float a, int b.

So, in this case we are taking 2 integers as arguments, so you have to say, int a, int b. A small syntactic point that you have to notice, that, you cannot abbreviate this as int a, comma b; so that is not allowed. Each variable needs to have a separate type signature. So, these are called the input arguments. So, that is the second part of the declaration. The first part of the declaration, the first, which says that, it is an int, is actually the type of the return value. So, the return value is 1 if the pairs is coprime, and it is 0 if the pair is not coprime. So, the return value is an integer. So, we need a function name, we need a declaration of the input arguments. The arguments need to be named, and the return value of the output.

So, let us say how do we design the higher level function? So, here is how you use functions when you program. You assume that the function is already available to you, and it does what it is supposed to do. Using that how do I build the solution to the whole program? So, in this case, let us just assume that we have written int iscoprime; we have written that function. And we are interested in, how do we build the solution to the entire problem using that?

(Refer Slide Time: 11:12)

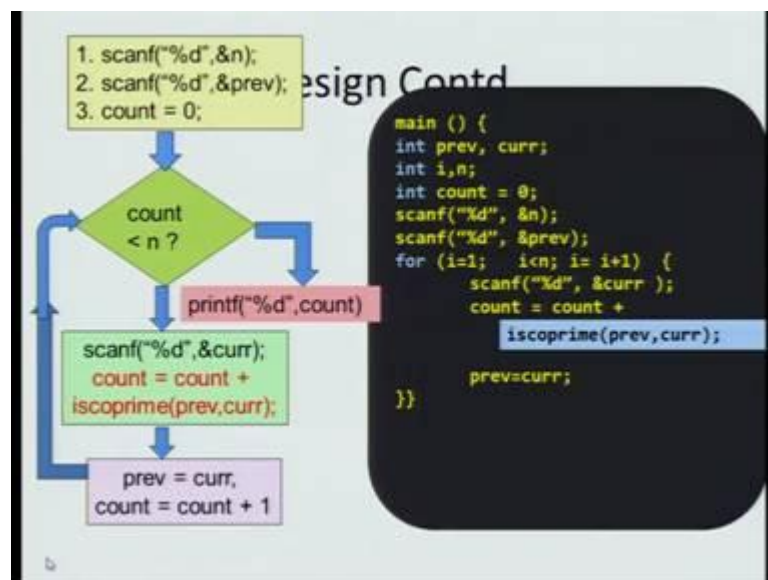


So, how do you do that? Use, have a flow chart which reach numbers one by one. And count is the number of coprime pairs that you have seen so far. So, you check whether you have seen n numbers. If you have not seen n numbers then you read the next number

and check whether the previous number and the current number form a coprime pair. So, you give `iscoprime prev current`; it will return 1, if they are coprime. So, that will get added to the count. If there not coprime, they will, it will return a 0. So, count will remain as it is.

Once you do that you say `prev` is equal to `current`, and indicate that you are going to read the next number. This is similar to other problems where we scanned this bunch of numbers and did some function based on that. The new think here is the `iscoprime` function which we just assumed that it is correctly written, and it does what it is supposed do. So, this is the function declaration.

(Refer Slide Time: 13:10)



Now, how do you code this up? You basically code this up in c, just as you did it with other program, other functions like `scanf`. You just say, `count plus iscoprime previous current`. So, this is how you can encode the flowchart including the function called as a C program.



(Refer Slide Time: 13:34)

```
int iscoprime(int a, int b) {
/* returns 1 if gcd of a and b is 1 and 0 otherwise */
/* a and b assumed > 0 */
/* write the gcd code */
    int t;
    if(a < b){t=a; a=b; b=t;} /* exchange a and b */
    while ( !(b == 0) ) {
        t = b;
        b = a%b;
        a = t;
    }
    /*a is the gcd. For co-primality, test if a is 1*/
    if (a==1)
        return 1; /* important: functions */
    else
        return 0; /* return value using
                    keyword return */
}
```

Now, let us come to the interesting part which is, how do we design the int iscoprime function? So, the top is the declaration part of the function where I say that what is its type. So, the function name is iscoprime. It takes 2 variables a and b; a is of type int, b is of type int. And it is supposed to be written in integer value. So, that much is clear from the type declaration the type signature so called of iscoprime.

Now, what you do with it? You say that, so this is the classic gcd code; you declare a t variable; if a is less than b, you swap a and b. And this part of the code is just calculating the gcd. This is code that we have seen before. And at the end of that, a, will become the gcd. If a and b are coprime then a, will be 1. If, a, is any number greater than 1, then they are not coprime. So, if, a, is equal to 1, you return 1. And for returning, you use the keyword return. So, you return the value 1; otherwise you return the value 0. So, this is how you write the function iscoprime.

(Refer Slide Time: 15:11)

```
#include <stdio.h>
int iscoprime(int a, int b)
{
    int t;
    if (a < b) {
        t = a;
        a = b;
        b = t;
    }
    while ( !(b == 0) ) {
        t = b;
        b = a%b;
        a = t;
    }
    if (a==1)
        return 1;
    else
        return 0;
}

main () {
    int prev, curr;
    int i,n;
    int count = 0;
    scanf("%d", &n);
    scanf("%d", &prev);
    for (i=0; i<n; i++) {
        scanf("%d", &curr );
        count = count +
        iscoprime(prev,curr);
    }
    prev=curr;
}
```

Putting it together

In sequence in the same file

So, now we have to put both these go together. So, I will say, include stdio dot h; this is the first line of the code. Then I will have the source code for iscoprime. So, I will write that. And afterwards write name function, so that, when main calls iscoprime function, then we already have the code for iscoprime available. First this line, then the iscoprime function, and then the main function.

(Refer Slide Time: 15:53)

```
#include <stdio.h>
int iscoprime(int a,
int b){
    int t;
    if (a < b) {
        t = a;
        a = b;
        b = t;
    }
    while (b != 0){
        t = b;
        b = a%b;
        a = t;
    }
    if (a==1)
        return 1;
    else
        return 0;
}
```

a, b are the formal parameters of iscoprime function. Specified to be of type int each. Formal parameters are viewed as variables inside function body.

return is used to pass back the function's value

So, let us look at the function in somewhat greater detail; a and b are what are called the formal parameters of the function. They are viewed as variables. Now, the formal parameters are visible only within the function. So, we say that their scope is inside the

function.

(Refer Slide Time: 16:22)

```
#include <stdio.h>
int isoprime(int a,
int b) {
    int t;
    if (a < b) {
        t = a;
        a = b;
        b = t;
    }
    while (b != 0) {
        t = b;
        b = a%b;
        a = t;
    }
    if (a==1)
        return 1;
    else
        return 0;
}
```

Functions can be called. E.g., 5, 6 are referred to as actual parameters of the call.

```
main () { int x;
    x = isoprime(5,6);
    printf("%d",x);
}
```

Once a function call is encountered:

return is used to pass back the function's value

Now, there is, this is what is known as the declared definition of the function. Every function can be called. Notice that we have already called the functions like print f and scan f. So, once you define a function you can call a function; calling a function will be evaluating that function with particular arguments; you can do that. So, when you call a function you execute the function with the given arguments. So, 5 becomes a, and 6 become b.

(Refer Slide Time: 16:55)

The slide is divided into several sections. On the left, there is a code block for a function named `iscoprime` that takes two integers `a` and `b` as input. The function uses a `while` loop to find the greatest common divisor (GCD) of `a` and `b` by repeatedly dividing the larger number by the smaller one until the remainder is 1. It then returns 1 if the GCD is 1, and 0 otherwise. A flowchart below the code shows the execution path, with arrows indicating the flow from the `return` statements back to the `main` function.

On the right, there is a text box stating "Functions can be called. E.g., 5, 6 are referred to as actual parameters of the call." Below this, a code block shows the `main` function calling `iscoprime(5,6)` and printing the result. A list of three steps explains the process of function call execution:

1. Create variables corresponding to **formal parameters**: `a`, `b`. Copy values of actual parameters to formal parameters, so `a` becomes 5, `b` is 6—**parameter passing**.
2. Store **return address** of the call. When the function finishes execution, control will jump back to this point.
3. Create a box for **return value**: the value returned by the function.

At the bottom of the slide, a blue box contains the text: "return is used to pass back the function's value".

Once a function call is encountered what happens is that formal parameters are mapped to actual parameters. So, `a` becomes 5; so the value 5 is copied to `a`, and the value 6 is copied to `b`. This process of copying values is known as parameter passing. Then what you do is, you store the return address of the call. The return address is the line of the main function where the function was called. So, let us say that it was called in the second line of `main`. Once the function finishes it has to come back to this point.

Now, in addition, we also create a box for storing the return value. At the end of function either 1 or 0 will be returned. So, we also need some space in memory to store that return value.

(Refer Slide Time: 17:53)



(Refer Slide Time: 17:55)

```
1 #include <stdio.h>
2 int iscoprime(int a, int b) {
3     int t;
4     if (a < b) {
5         t = a;
6         a = b;
7         b = t;
8     }
9     while ( !(b == 0) ) {
10        t = b;
11        b = a%b;
12        a = t;
13    }
14    if (a==1) return 1;
15    else return 0;
16 }
17
18 main () {
19     int x;        x = -1;
20     x = 20a iscoprime(9,4);
21     printf("%d",x);
22 }
```

- Steps when a function is called: `iscoprime(9,4)` in step 20a.
- Allocate space for (i) **return value**, (ii) **store return address** and (iii) **pass parameters**.
- 1. Create a box informally called "Return value" of same type as the return type of function.
- 2. Create a box and store the location of the next instruction in the calling function (main)—**return address**. Here it is 20. Execution resumes from here once function terminates.
- 3. **Parameter Passing**- Create boxes for each formal parameter, a, b here. Initialize them using actual parameters, 9 and 4.

So, to look at it in slightly greater detail, so let us say that `iscoprime 9, 4` is called in step 20 a. So, this is the address; 20 a by which I mean it is line 20 and some location a. So, now, you have to allocate the space for the return value; store the return address and pass the parameters. Now, at, when you pass the inputs, 9 and 4, the space is allocated for, a equal to 9, and b equal to 4. This is the process of parameter passing.

(Refer Slide Time: 18:36)

```
1 #include <stdio.h>
2 int iscoprime(int a, int b) {
3     int t;
4     if (a < b) {
5         t = a;
6         a = b;
7         b = t;
8     }
9     while ( !(b == 0) ) {
10        t = b;
11        b = a%b;
12        a = t;
13    }
14    if (a==1) return 1;
15    else return 0;
16 }
17 main () {
18     int x;    x = -1;
19     x = iscoprime(9,4);
20     printf("%d",x);
21 }
22 }
```

Calling iscoprime(9,4):

1. Allocate space for return value.
2. Store return address (20).
3. Pass parameters.

STACK

main	x	-1
iscoprime	Return value	
	Return Address	20
	a	9
	b	4

(Memory)

After completing iscoprime(), execution in main() will re-start from address 20.

So, we visualize the memory as a stack. So, when you start the programs you start executing from line 1 of many; so x is initialized to minus 1. And then you come to the function called iscoprime 9, 4. So, when you execute this you do the following: you allocate the space for the return value, you pass the parameters and then execute the function, and finally pass back the return value.

So, when you execute the function you imagine that the stack is now divided into a separate space. So, here is a clean separation between the memory that is required for main. So, above here is main, and below here is the memory required for iscoprime. So, in that I have stored a box for return value. I have stored the return address which is 20 a. And then I have, a equal to 9, and b equal to 4. Now, I will execute the function although memory is limited to here.

(Refer Slide Time: 20:00)

```
1 #include <stdio.h>
2 int iscoprime(int a, int b) {
3     int t;
4     if (a < b) {
5         t = a;
6         a = b;
7         b = t;
8     }
9     while ( !(b == 0) ) {
10        t = b;
11        b = a%b;
12        a = t;
13    }
14    if (a==1) return 1;
15    else return 0;
16 }
17 main () {
18     int x;    x = -1;
19     x = iscoprime(9,4);
20     printf("%d",x);
21 }
22 }
```

Calling iscoprime(9,4):

1. Allocate space for return value.
2. Store return address (20).
3. Pass parameters.

x	-1	1	main
Return value	1		
Return Address	20		iscoprime
a	9	4	1
b	4	1	0
t		4	1

So, I will declare t, and then execute the gcd algorithm. So, this is stuff that we have seen before. And finally, a is the gcd which is 1. If a is 1 we have to return 1. So, the value 1 will be copied to the return value, and that is the value that will be passed back; so x will be 1. So, the return value will be copied back to the main function.