**Introduction to Programming in C**
**Department of Computer Science and Engineering**

We have been talking about designing programs using functions. And the general philosophy is that, you have a large task that you want to accomplish and you break it in to sub task, may be each of those sub task are split it in to smallest sub task and so on.

(Refer Slide Time: 00:19)

A general principle of program development

1. Break up your task into smaller sub-tasks, and those sub-tasks into still smaller sub-tasks and so on until each sub-task is easily solvable in a function.
2. Write a function for each of the sub-tasks.
3. Design your program from the top-down, big task to the small tasks.
   I. Debug/test your program bottom-up. Debug functions that perform
   II. elementary tasks, and then move to testing more complex functions.

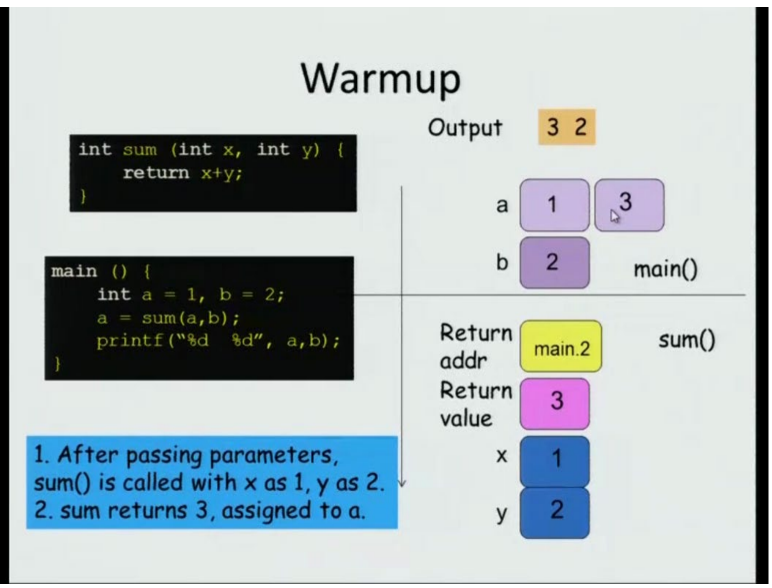
So, break them until some sub task can be easily solved by single function. And then, you put all these function together in order to solve the whole problem. So, design your program from top down, big task decomposing into small task and so on. And debug your program or make sure that they are free of errors from the bottom up. So, test each functions thoroughly and then test the overall program.

(Refer Slide Time: 00:50)

- How does C pass arguments to functions? [Call-by-value]
  - Evaluation order
  - Side Effects
- Returning values

In this, we will discuss a few technical details about how C executes its functions. In particular we will see how C passes arguments to it is functions, and also how does a return values. When passing arguments will talk about issues like evaluation order, in what order are arguments evaluated, if there are multiple arguments. And we will discuss what are known as side effects.

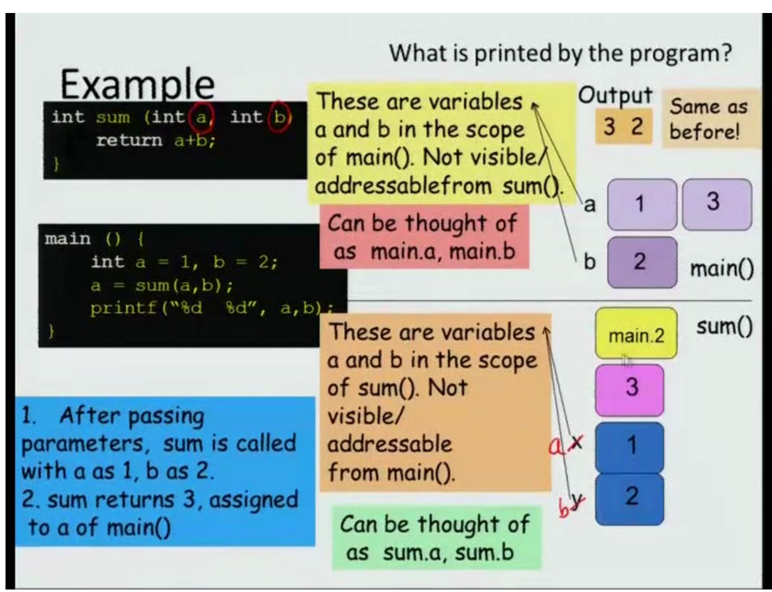
(Refer Slide Time: 01:23)




So, let us start by considering a very simple example. I have very simple function called sum which just adds up two variables x and y which are integers. Therefore, the return

value is also an integer. Now, inside the main program I will call a = sum(a,b). A is 1 and b is 2. And then, you will sum this up and the return value is assigned to a. So, this is suppose to do 1 plus 2 3 and a is assigned the value 3. So, after passing the parameters sum is called with x as a, which is 1 and y as b which is 2. So, sum returns 3. So, the return value is 3 and the 3 is assigned back to a. So, when you print it, the output will be 3 followed by 2. So, this is simple enough.

(Refer Slide Time: 02:32)



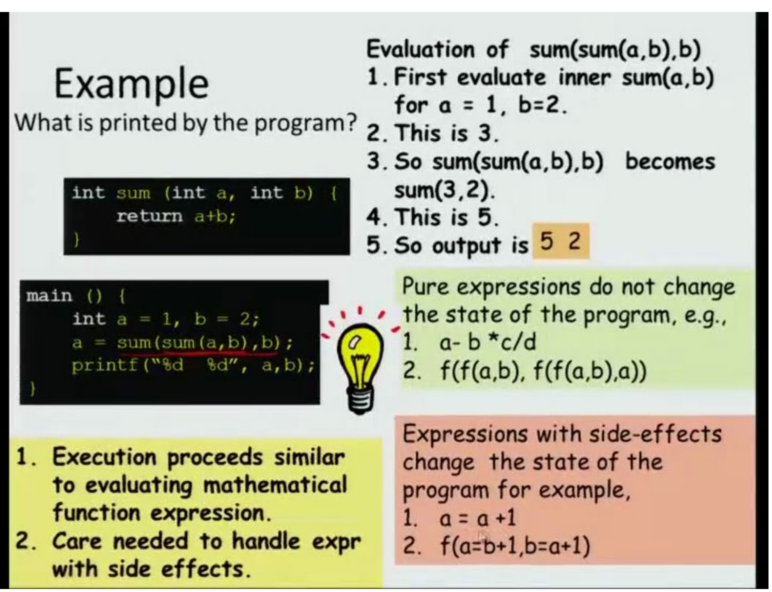

Now, let us try, slightly more tricky example. So, here is the novelty in this example. Some instead of being declared with x and y are now being declared with two variables called a and b. The main program also has two variables, named a and b. So, what will happen here? The output is the same as before. So, 3 2 if you compile the program and execute it, it will be same as before. So, what really happened here?

After passing the parameters sum is call with a as 1 and b as 2. And so it returns a value 3 assigned to the a of mean. Now, these variables are called a and b in main and they are called a and b in sum as well. Now, the variables a and b inside sum are different from the variables a and b inside main. So, the scope of these variable is mean and the scope of these variables a and b is the sum function.

So, in other words the a and b inside sum has scope just this function, they are not visible or addressable outside especially in main. So, if you want to think of it, you can think of them as sum.a, sum.b. So, they are the a variable belonging to sum and the b variable

belonging to sum. So, even though you would think that this a and this a may get confused they are actually different variables. One is the a variable belonging to main, and the other is the a variable inside sum and they are different, even though they have a common name.

(Refer Slide Time: 04:47)




So, now let us a try a slightly more elaborate program, what happens if you have sum(sum(a,b),b), this is the program. In this case what will happen? So, first evaluate the in a program, in a function sum of a b. So, a is 1 and b is 2. So, that will return 3, then you add b again to it b is 2, you have 5 as the total sum. So, the total the complete output is a will be assigned 5 and b is still 2.

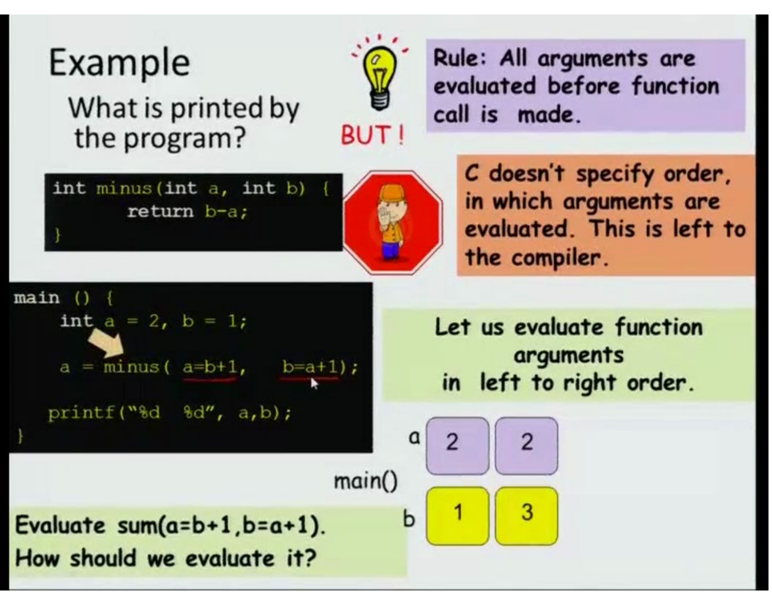
So, this is similar to evaluating a normal mathematical expression. One thing that we need to take care of is to handle expression with side effects. Now, what are expression with side effects? So, let us classify expression into two kinds, one is what are known as pure expressions. So, they are the normal mathematical expression, like a- b * c/d and so on. Similarly evaluating function, these normally do not have any effect other then returning you the value.

So, they will they will be correctly evaluated and they will return some value, other than that, they have no effect. Now, expressions with side effects change the state of the program. For example, when I execute an expression a = a +1. Now, this is an expression, it has a value. So, let us say that a was 1 before a = a +1. A plus 1 has value 2

and a is assign the value 2. The state of the program involves, for example, what values are stored in the variables. When you execute the expression a = a +1, the value of the variable a changes. Contrast this with previous expression, like a minus b star c slash d. You can see that, unless you assign to something no variables value is changing, it will just evaluated and the value will be return. Here, the value will be returned also variable a is changing. Here, in this second function you have two arguments, two function f. The first is the expression a = b +1, the second is an expression b = a + 1.

This might sound like a very strange way to code. But, you know that any expression can be given as arguments. So, in particular assignment expressions can be given as arguments. For example, a = b +1 is an assignment expression, which is given as an argument to the function. Now, such expressions are called expressions with side effects, because, the change the state of the program.

(Refer Slide Time: 08:01)

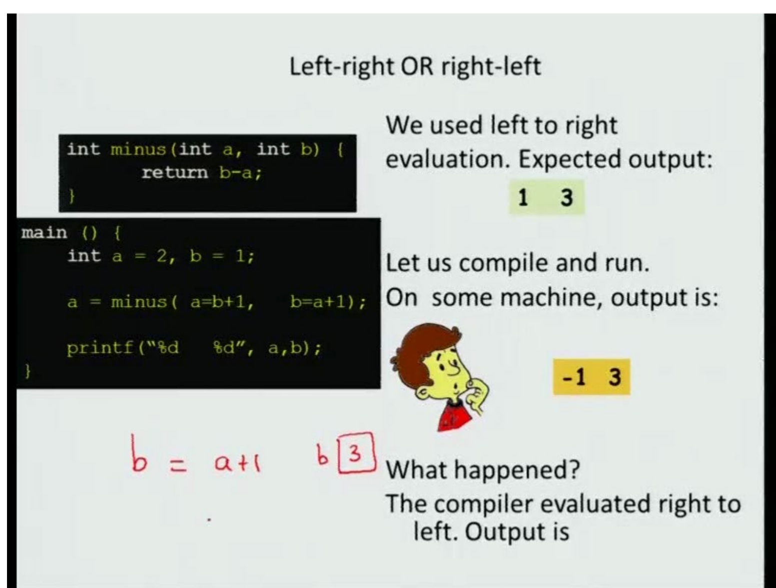



When you have side effects you should be careful. For example, what will happen in the following program? You have function int minus(int a, int b) and it returns b - a. Now, in this program main calls the minus function with two expressions as arguments a = b +1 and b = a + 1. They are expressions with side effects, because, once evaluate these arguments, you know that the variable a will change in the first expression and the variable b will change the second expression.

So, what will happen in this program? So, how should we evaluate it? The general rule is that all arguments are evaluated before the function call is made. So, before the function is executing, we know that a = b +1 and b = a + 1 both will be executed. But, and here is the major problem, we know that both have to be executed. But, C does not specify in which order they have to be executed, so, it was the left to the compiler.

So, let is evaluate it in first in left right order. So, this expression first and then b = a + 1. So, what will happen then? a = b +1 b is 1. So, a will get the value 2, b = a + 1 will be executed after that a is now 2.  So, b will get the value 3. Now, you execute minus. So, you will return 3 - 2 which is 1 and b has value 3. So, this is the expected output. But, when you run it on some machines, you may get the output -1 3.

(Refer Slide Time: 10:07)




So, what happened here? Now, this happens for example, when the compiler would evaluate it right to left. So, when you evaluate it right to left what will happen is that b = a + 1 will be executed first. So, b = a + 1, b gets the value 2 plus 1 3. And then, you will execute a = b +1, b is now 3. So, a gets the value 4. So, when you call minus of 4 comma 3 minus will return 3 -4 which is -1. So, in this case you know that b gets the value 3, a gets the value 4 and the result will be -1.

(Refer Slide Time: 11:10)

## What was the mistake?

- So what was the mistake?
- Actually, C does not specify the order in which the arguments of a function should be evaluated.
- It leaves it to the compiler. Compilers may evaluate arguments in different orders.
- Both answers are consistent with C language!! What should we do?

**Write your arguments to functions so that the result is not dependent on the order in which they are evaluated. Better still, write them so that the operand expressions are side-effect free.**

So, what was the mistake? The mistake was that we assume that both arguments will be evaluated before the function is called. But, we assume that it will be evaluated left to right. And the first expression will be evaluated before the second expression, that is the reasonable assumption to make. But, c does not guarantee you that, c leaves this decision to the compiler. Now, compilers may evaluate arguments in different orders. For example, a very common order is right to left.

So, both answers like 1 and 3 and -1 and 3 are both consistent with the c specification. Now, this is the very troubling is scenario, what should we do? So, we should write this function in such a way that, they do not depend on whether the arguments are evaluated left to right or whether they are evaluated right to left. So, write expressions in such way, that they are free of side effects, when you pass them into functions.

(Refer Slide Time: 12:13)

For example

```
int minus(int a, int b) {
        return b-a;
}
```

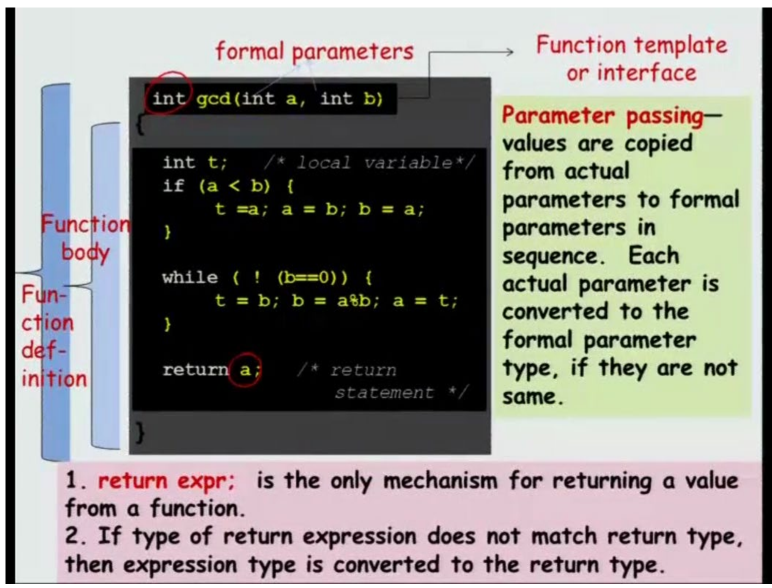
```
main () {
    int a = 2, b = 1;
    a=b+1;
    b=a+1;

    a = minus( a, b ); /* operands do not have
                          side effects */

   printf("%d  %d", a,b);
}
```

So, how do we do that? We can do the following. For example, if we really wanted the left to right order, that is if you want a = b +1 to happen first and then b = a + 1. Why not write them explicitly that order in the main function. So, first this will be executed then b = a + 1 will executed. So, a will get the value 2 and then b will get the value 3. So, minus ( a, b) will execute as 3 - 2 in which case you will get 1. So, the important thing to note is that, in this particular function called, the arguments do not have any side effects. Because, we explicitly coded them up before to specify that, this is the order in which I want. If put it here, then it is up to the compiler, the compiler can do whatever is best in for several criteria.

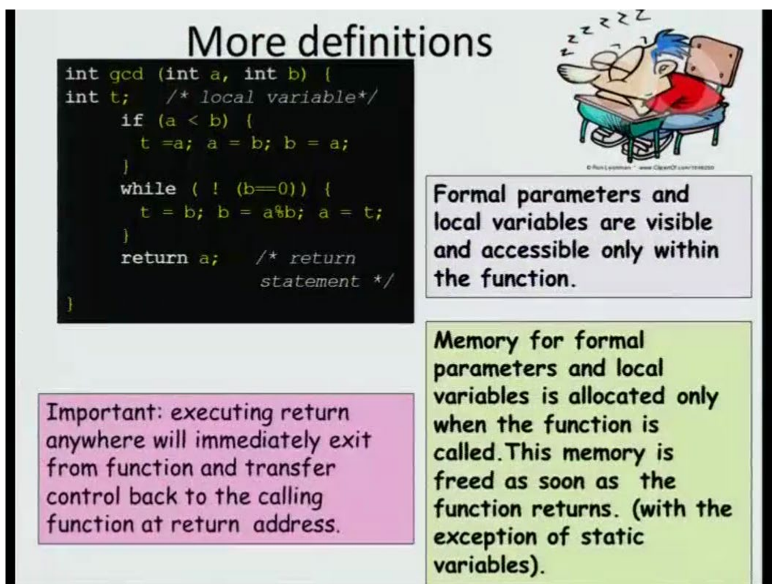
(Refer Slide Time: 13:21)




So, now let us come back to what the function, we have the function definition which is the entire function. The logic of the function is what is known as the function body. And the heading is what we call the type signature. The type signature has for example, two arguments a and b these are call the formal parameters. Now, we focus on the return expression. So, return followed by some expression is the only mechanism for returning the value from a function.

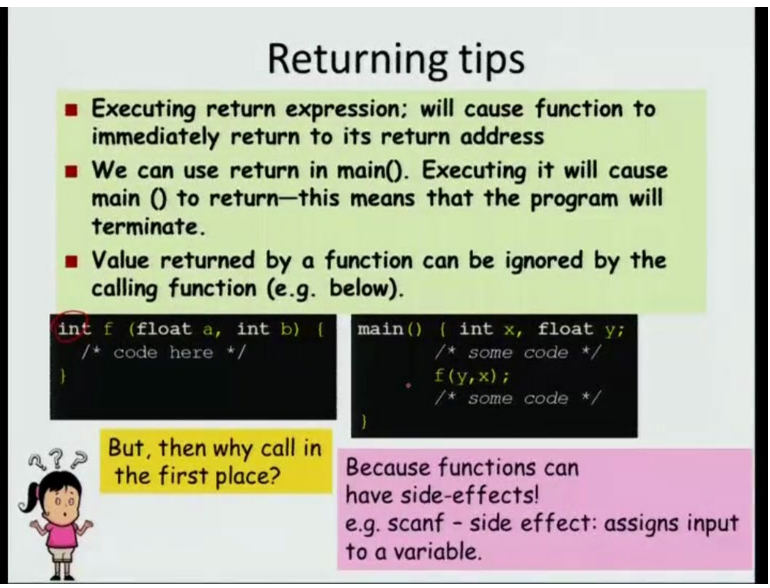
If the type of return expression does not match the declare type of the return. So, if for example, a is of a variable which is different from int. In this case they are the same, then it is fine. But, otherwise the return expression is converted to this type and then returned. So, it might lead to some undesirable variable. Now, we have discussed parameter passing's in when passing parameters in c, the values from the calling function are copied to formal parameters in the called function. So, the actual parameters are converted to the formal parameter type and separate copies made. So, this is known as call by value.

(Refer Slide Time: 15:03)




So, formal parameters and local variables are the function are accessible only within the function, we have already see in this. And memory for the formal parameters and the local variables of the called function will be erased as soon as function returns. So, executing return any where inside the function will immediately return from the function. And transfer control back to the calling function at the specified return address.

(Refer Slide Time: 15:37)




So, when you execute there are few things keep in mind. Whenever, you execute any return expression, it will cause the function to immediately return. Now, main is a

function so, we can use return statement inside main what; that means, the main will immediately stop execution. That is the whole program will stop execution. Now, when you return a particular value, the calling function may choose to ignore the value.

For example, let us say that I write some dummy function int f and it takes two argument float a and int b and we some code here. And then, I have the main function in which I have two variable int x and float y. Then, I have some code and here is the interesting thing, I call f(y,x), y is an float x is an int. So, I am find, but this function returns an integer value. But, I am not assigning it to anything. So, I am not saying something like x = f(y,x). So, this is not required.

So, if this is the case, then why call the function in the first place? This is, because the function also may have side effects. So, remember that side effects are something some expressions, we change the state of the program. So, functions may have side effects, your already seen one such function which has the side effect for examples, scanf. So, the side effect of calling scanf is that the input from the keyboard is copied into some variable. So, function may have side effect, this is why you can call the function and choose to ignore the output or the return value.

(Refer Slide Time: 17:42)

- Executing return; will cause function to immediately return to its return address (i.e., in the calling function). return value is unpredictable.

What is the output of the program?

```
float  f( int a, float  b) {
      float  t=a+b;
      return;  /* no expr given with return */
     /* return value is unpredictable: garbage!  */
}
main () {
       int a = 1;
       float b =2.0;
       printf("%d\n", f(a,b));
}
```

Printed value is unpredictable.

Now, just for curiosity sake executing return will calls the function to immediately return to the return address. Now, the return value if you omit it, then the return value is unpredictable. So, here is a example, you should in general avoid doing things like this.

But, just for completeness, I am supposed to return a float value instead if I just say a return, the program will compile. But, when you execute some unpredictable behavior may result. So, the printed value in this case can in general will not predictable.