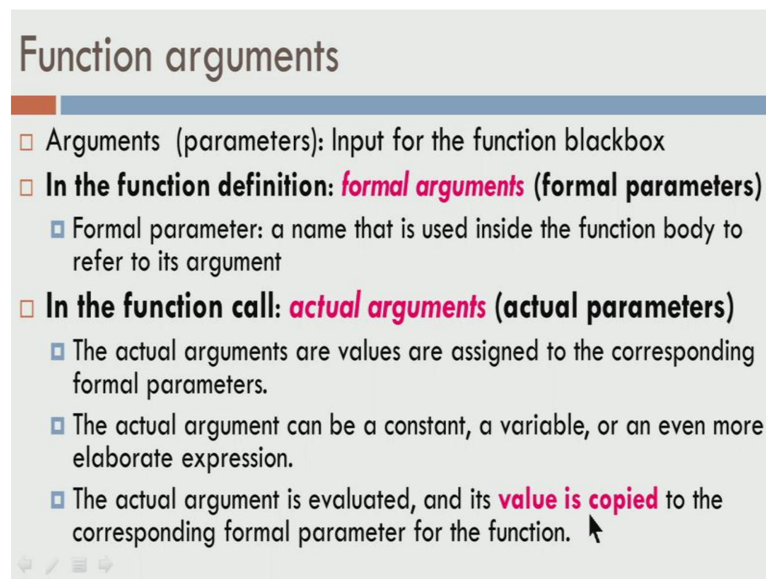
Programming, Data Structures and Algorithms
Prof. Shanker Balachandran
Deportment of Computer Science and Engineering
Indian Institute of Technology, Madras

Module - 10C
Lecture - 22
Function arguments and local variables
Automatic local variables
Scope of local variables
Call by value versus call by reference
Returning multiple values from a function

We saw the notion of arguments, these are inputs to the black box called the function.

(Refer Slide Time: 00:12)



When we talk about arguments, there are two kinds of arguments, this terminology called formal arguments or a formal parameter. So, our formal parameter is a name that is used inside the function body to refer to the arguments and the function call is actually made with actual arguments or actual parameters. So, there is a name that you used for things within the function and you may have names, which have different in the caller.

So, the caller may have names with the parameters which are different from, what is in the callee. So, the actual arguments or actually values that are assigned to the corresponding formal parameters, this actual argument could be a constant or a variable or even a complicated in elaborate expression.

(Refer Slide Time: 01:12)

Calling Power Function with i=3

printf("%d %d %d\n", i, power(3,i), power(-4,i);}

```
int power (int base, int n) {
        int i, p = 1;
        for ( i = 1; i <= n ; i ++)
                p = p * base;
        return p;
}
```

```
int power (int base, int n) {
        int i, p = 1;
        for ( i = 1; i <= n ; i ++)
                p = p * base;
        return p;
}
```

So, in our printf instead of giving the integers directly here, instead of giving the integers directly, we actually put an expression that is permissible. And the actual argument is evaluated and the value of that is copied to the corresponding formal parameter for the function. So, we will see this in a little more detail with the next example.

(Refer Slide Time: 01:30)

Example: arguments

```
// Function to calculate the nth triangular number
#include <stdio.h>
void calculateTriangularNumber ( int n )    formal argument
{
   int i, triangularNb = 0;        local variables
   for ( i = 1; i <= n; ++i )
      triangularNb += i;
   printf ("Triangular number %i is %i\n", n, triangularNb);
}
int main (void)
{
   calculateTriangularNumber (10);   actual argument
   calculateTriangularNumber (20);
   calculateTriangularNumber (50);
   return 0;
}
```

So, let us say I have a small program which calculates the triangular number, we saw the notion of triangular number in one of the earlier lecturers. So, I have a main function and the main function has int main void. So, what this means is, main is a function which does not take any parameters, you say that by using the keyword called void. So, void means you are not using any input, you are not taking any input and you are returning

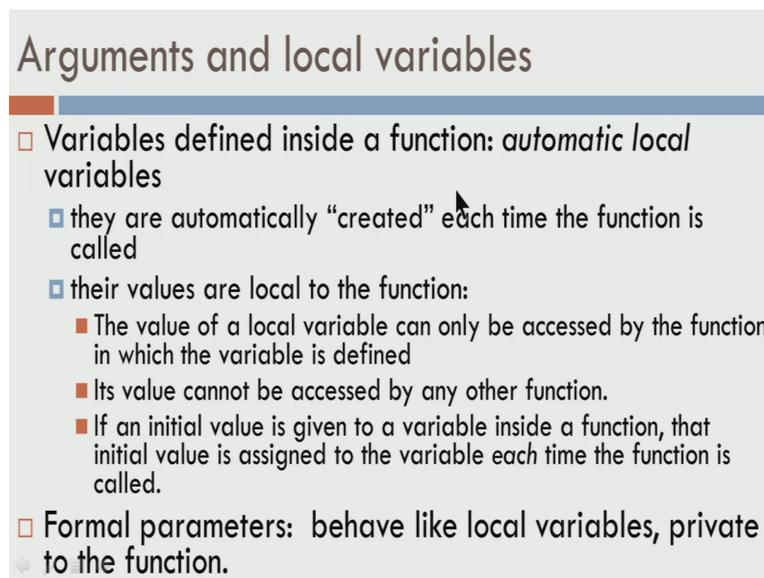
integer.

So, calculate triangular number 10. So, this is a function call and the function description is here. So, this program is actually missing the function prototype, there should be an function prototype which says, void calculate triangular number of int. So, anyway, So, we have a function call calculate triangular number of 10. So, this 10 is an actual argument, because in the caller you have the value 10.

This is the formal argument, we call it by the name n, the formal argument for calculate triangular number is n and i and triangularNb are two local variables. So, these are two local variables, i is used for the iteration and triangularNb is used for the summation. So, this is a formal argument. So, n is a formal parameter, 10, 20 and 50 are actual parameters and i and Nb are local variables. So, this is a distinction that we have to keep in mind.

So, even though there is one formal argument, the actual arguments are actually changing. So, you notice this piece of code, actual parameter is 10 at this function call, the actual parameter is 20 for this function call and the actual parameter is 50 for this function call and the formal argument is always by the name n.

(Refer Slide Time: 03:32)



So, let us see a little bit about arguments and local variables. So, the variables are actually defined inside a function. So, you go and look at this, i and triangularNb are actually defined within the function, same thing in power. We had i and p which were designed inside the function, we call them what are called automatic local variables. We

call them automatic local variables, the reason we call them automatic local variables is that they are automatically created by the compiler, each time the function is called.

So, the key thing to notice is that they are created, each time the function is called. The values that you have for these variables are local, that is why we called it local. So, these are variables, they are automatically created, but they are local to the functions. So, the value of a local variable can be accessed only within the function, in which the variable is defined. So, for example, from this function, I cannot directly use triangularNb or i.

So, in fact, this one is only printing the triangular number on the screen. Let us say, I wanted that triangular number, it is not that I want to print it on the screen, let us say I want to use the triangular number. Then, this function cannot be used directly, because it is calculating triangular number that is printed on the screen, but triangular number is a local variable, it is printed on the screen and that is it. You have not returned triangularNb here, therefore, we cannot see the value first of all and there is no way to refer to triangularNb from the main function.

So, the values of the local variables are all local to the function, you can access it only within the function and if an initial values given to the variable, every time the function is called, it is going to be assigned. So, this is particularly useful. For example, let us say calculate triangular number of 10. So, that is the first function call, you calculate triangularNb, at the end of this, the value would have been 55. So, we are adding 1, 2, 3, 4 and so, on upto 10, that is 55. So, let us say triangularNb is 55.

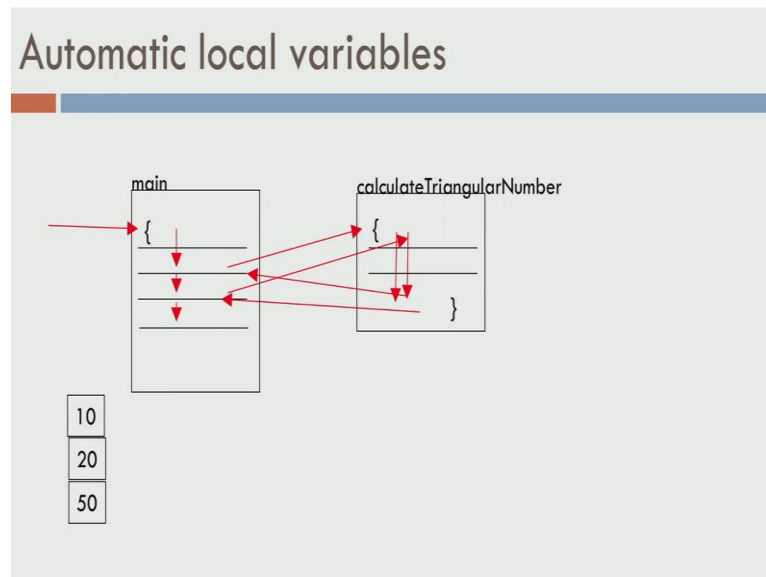
The next time I calculate the triangular number, since I outsourced it. I assume that you are going to take care of the initialization. So, when the function comes in next time, since triangularNb is again initialized to 0. So, the value 55 is completely lost. In fact, the triangularNb is not even present, once the first call returns before the second call, the triangularNb and i are not accessible any more.

So, you have 20, when you go back here, triangularNb is again created, i is again created, you initialize triangularNb to 0, you do the computation, the result is printed on the screen and when you return from the function, i and triangularNb can be thought of as, something that vanishes also automatically. So, automatic variables are both created and destroyed automatically. So, the formal parameters are a slightly different.

So, let us look at this variable n. So, from the point of view of this function n is also a local variable, it gets created every time the function is called and n gets destroyed, every

time you return from the function, the variable called n is destroyed also. So, formal parameters essentially behave like local parameters, that completely private to the function.

(Refer Slide Time: 07:27)



So, let us look at how this sequence works. So, I have the main program, the control flow starts there, you are ready to calculate triangular number of 10. So, at this point n, i, and triangularNb are all created. This 10, the value of 10 is actually copied to n. So, think of each of these boxes has separate memory locations. So, 10 is already in some memory location, 20 is in some memory location, 50 is in some other memory location.

The value 10 is actually copied to n and then you go through whatever is supposed to happen and that happens and then you return the value or you print the things on the screen and you return to the column at that point. So, if you carefully watch the animation n, i, and triangularNb evolve vanished. So, there is no way to access n or i or triangularNb anymore. So, then the function goes to the second main function, now moves to the second function call.

At that point, again it goes to the first line of calculate triangular number, again you would create n, i and triangularNb. The value 20 gets copied and triangularNb would get the initial value of 0, i anyway gets initialized in the loop, i becomes 1. Whatever work you have do there that gets done and you return the control back to the main function and finally, you have the last call and the same thing happens once more. So, this is something that you have to remember that, every time a function is called, you would

create the variables which are local to the function as well as the formal parameters.

You do whatever work is going to happen there and when you return back from it. All these local variables and the formal parameters are lost. The only thing that can potentially be retained is the return value. If the function had a return of a variable that value gets copied back to the right side of an expression. So, for the power function, we return p. So, the value of p is comes to the right side of the expression and that can be useful for assignment to the left side, namely num1 and num2.

If you do not have return of a variable, all the calculations that you have done are completely lost. All the variables that you declared are completely lost. May be you printed something on the screen, but you can never use the 10th triangular number in any calculation, you can only print thing on the screen, with this function that we have. So, in this context it is good to understand the notion of life time and scope. So, life time... So, this is something that people get confused about, very often.

So, life time is the period of time, when a memory location is allocated. So, if you go back to this example that we had earlier ((Refer Time: 10:35)). So, the life time of i triangularNb and n are these variables are said to be alive, only when this function is called. Once the function returns, you can treat them as variables that are dead. So, life of the variable is only between the functions first line and the functions last line. These variables and the formal parameters are dead otherwise. So, we have the notion of life time.

(Refer Slide Time: 11:05)

Lifetime and Scope

- **Lifetime**: Period of time when memory location is allocated
- **Scope**: Region of program text where declaration is visible
- Scope: local variables and formal parameters => only in the body of the function
  - Local variable i in function calculateTriangularNumber is different from a variable i defined in another function (including main)
  - Formal parameter n in function calculateTriangularNumber is different from a variable n defined in another function

Scope is slightly different and it is an important concept to know also. So, scope is the region of program text, where declaration is visible. So, we have not seen this in any detail earlier, we will see that now. So, all the local variables and formal parameters are not only alive within the function, their scope is also only within the body of the function.

So, local variable i in function is different from any other variable i, you declare anywhere else and formal parameter n is also different from any other n that you are declared or used anywhere else. So, it is not only that i and n are not alive outside that, you are free to use i and n in other places, in other functions including the caller.

(Refer Slide Time: 11:55)

Example: Scope of local variables

```
#include <stdio.h>
void f1 (float x)  {
        int n=6;
        printf("%f \n", x+n);
}
int f2(void) {
        float n=10;
        printf("%f \n",n);
}
int main (void)
{
   int n=5;
   f1(3);
   f2();
   return 0;
```

So, let us see a small example of what the scope is. So, let us first look at the main function. It has a variable called n which takes the value 5 and you call f1 of 3 f2, which has no parameter and return 0. So, this is not really doing anything with n, this example is only there to show you, what the notion of scope is. So, as a control you will start with int n equals 5. So, 5 gets the value of n and then you have f1 of 3.

So, 3 is the actual parameter, f1 is the function name, you go and look at f1, f1 expects a floating point called x. So, x is the formal name. Here, you have another declaration called int n equals 6. So, you may wonder whether this n and this n are actually the same. So, the answer to that is, they are not. Remember, whenever a function is called, there are variables that are created. So, when you come inside, you create a new variable called n and only this variable is seen in this function.

So, here n equal 6 and you called f1 with 3. So, x is 3 and n is 6 and x plus n would be 9. So, this printf percentage f would print floating point value 9 on the screen. Now, the function would return back to the caller. In this case you have f2, f2 does not have any parameter, but when the function gets called, you have float n equals 10. You again create a new variable by the name n. The life of this variable n is only within this function.

You just print n equals 10 and when you come back here, you have this return statement. So, at this point let us say between f1 and f2, if you printed n it would have been 5. After f2, if you printed n it would have been 5 also. This n and this n are not in the same scope as this n. This n is local to this main function, this n is local to this f2 and this n is local to f1. So, x and n are local to f1, they get created, when f1 is called, they get destroyed, when f1 returns.

Similarly, n gets created when f2 is called, it is destroyed when f2 is returned and these variable names are essentially local, it does not interfere with what is there in the collar. So, in this case in fact, the n was a floating point and this was an integer and this was an integer, for all you care there may be another function, where n is a character or even a pointer and so, on, it does not matter. So, within the same scope, you cannot have two declarations for a variable.

But, once you go to different scopes, namely within functions, across functions and so, on, you are opening a new scope you are allowed to declare new variable names. So, that is something that you have to keep in mind.

(Refer Slide Time: 15:12)

Call by Value

In C, function arguments are passed "by value"

- values of the arguments given to the called function in temporary variables rather than the originals
- the modifications to the parameter variables do not affect the variables in the calling function

"Call by reference"

- variables are passed by reference
  - subject to modification by the function
- achieved by passing the "address of" variables

So, one thing that we have done so, far is, we have seen what is called call by value. So, in C the function arguments are all passed by value, values are the arguments. So, the values contained in the actual parameters are copied to variables, which are in the formal parameters. So, we have the actual parameters.

When you call the function, the formal parameters get created and like any variable, initially they do not have any values and the first thing when you have for a function call is, automatically the actual parameters are copied to the formal parameters and with the formal parameters, you do all the local computations, the value gets returned and those variables are destroyed. So, the modifications to the parameter variables do not affect the variables in the calling function.

So, we already saw this example here also ((Refer Time: 16:10)). So, n became 10, n became 6 and so, on, this has nothing to do with this n, first of all and you have passed a parameter f1 of 3, that has x here. So, this x is not changed in this example, but even if you changed x, the value 3 will not change. So, there is also something that you can do which is called call by reference. So, I want to show you the difference between call by value and call by reference.

When you say call by value, we copy the contents, but when you say call by reference, we pass the reference or the address of the variables instead of the contents of the value. So, in fact, you can think of this has passing the r values and this has passing the l values of the variables.

(Refer Slide Time: 16:58)

Call by Value - an example

```
int test(int, int, int);
main( ) {
int p = 1, q = 2, r = 3, s;

...;
s = test (p, q, r); ...        /* s is assigned 9 */
}                   /* p,q,r don't change, only their copies do */

int test( int a, int b, int c)
{
   a ++; b ++; c ++;
   return (a + b + c);
}
```

Function prototype

Function call

Function definition

So, let us see this call by value using an example. We have p equals 1, q equals 2 and r equals 3 and I have a function call test. So, actually this is a function prototype, this cannot appear within this function, it has to be outside and we have call s equals test. So, test equals p q r. So, this is something that should have been outside. So, it cannot appear within another function, it should have been outside. So, this is correct.

We have a function prototype which takes three integers, p equals 1, q equals 2 and r equals 3 and this is a fourth variable called s. So, this is the function prototype now. You pass s equals test. So, you pass p, q and r to test, they are received by the variables a, b and c. So, a, b and c are local variables. In this example, a is incremented by 1, b is incremented by 1 and c is incremented by 1.

So, a would start with a copy of p which is 1, but it gets incremented. So, a would be 2, b gets a copy of q that is 2 and it is incremented. So, b becomes 3, c gets a copy of r which is 3 and it is incremented so, it is 4. So, a would be 2, b would be 3 and r would be 4, c would be 4 and you add 2 plus 3 plus 4. So, the result would be 9 and that is what it returned as s. So, s get's 9, that is something that is probably clear to you by now. But, what happens is, p, q and r do not change.

Remember, we made a copy of p to a, copy of q to b and copy of r to c. We really did not do any changes to p, q and r. The variables a, b and c changed, they did not change p, q and r. So, this is called pass by value. So, every time a function is called like this, p gets copied to a, q gets copied to b and r gets copied to c. So, this is called pass by value or

call by value.

(Refer Slide Time: 19:07)

Call by Reference

```
#include <stdio.h>
void quoRem(int, int, int*, int*); /*pointers*/
main(){
    int x, y, quo, rem;
    scanf("%d%d", &x, &y);
    quoRem(x, y, &quo, &rem);
    printf("%d  %d", quo , rem);
}
void quoRem(int num, int den, int* quoAdr, int* remAdr){
    *quoAdr = num / den;   *remAdr = num % den;
}
```

Passing addresses

Does not return anything

In contrast, there is something called pass by reference. So, in this slightly loaded example, we have a few things. So, I have two integers, x and y and let us say I want to find out, what is the quotient of dividing x by y and what is the remainder of dividing x by y so, these two are integers. So, I do an integer division, I get a quotient and I get a remainder. So, I want to know both the quotient and the remainder.

So, one thing that you have noticed is that functions cannot return multiple parameters. So, if you remember the basic prototype, basic description or template of a function, you can pass more than one inputs to it, but you can only return one output. That is clearly a restriction for this problem. I want both the quotient and the remainder, but if I write a function for it, I can only get the quotient or the remainder, but not both.

So, to do that we have a small trick, here we are going to pass what is called passing by value. So, we have this function called quoRem that stands for quotient and remainder. It takes two parameters, numerator and denominator. It takes two more parameter, quotient address and remainder address and they are not integers, they are pointer to integers. So, we have two pointers that are passed on and two integers that are passed on.

So, num will get a copy of x, den will get a copy of y and quoAdr will get a copy of ampersand of quo, which means the address of quo is copied to quoAdr and address of rem is copied to rem of Adr, but then now you have the addresses of quotient and rem. Now, if I do num by den, you do integer division, the value gets truncated. But, I can

assign it to a memory location. In this case, I assign it to the location pointed to by quoAdr and num percentage den gives the remainder. The remainder is computed and it is stored in the location address by remAdr.

So, quoAdr and remAdr are both local variables, but these are pointer variables. So, when you do star quoAdr, you are not changing the pointer, but you are changing what is pointed to, you have deference quoAdr. So, the num by den, the value is stored in the location that is pointed to by quoAdr. When you come back here, since quoAdr had a copy of quo and remAdr had a copy of ampersand of rem, the pointers were copied.

So, at this point we already saw this, when you manipulate things with pointers, the memory locations are the same. So, quo and rem actually would have appropriate values from num by den and num percentage den. So, this does two things, we did not really have to change C, the language C to give us two return parameters or three return parameters and so, on. With still one parameter, we are able to get things done. Only that you have to pass pointers instead of passing values.

So, we have passed references to quo and rem, instead of passing the values of quo and rem. Because, if we are passed quo and rem directly and if we add integers here, you would get the local variable calculated, but remember when they return, since you have a void, those values get destroyed you will not see the appropriate values in the local copies, passing by reference takes care of that.

We will see this passing by reference in a lot more gory detail later, especially in the context of arrays. So, this brings us to the end of this module. In the next few modules, we look at more details related to functions.