**Function**

Mathematically, a function is a subset of relation. Relation is a subset of cartesian product of sets. A function associates a sequence of values – called arguments, with a value – called the result of the function application or the call. In Mathematics, the arguments will not change when the function is applied. Not only that, each time the function is applied to the same sequence of arguments, we get the same result. Such a function is said to be a pure function. It does not depend on Rahu kaala or gunika kaala. It does not depend on the gender or the mood of the person applying it!

In programming, a function is a sequence of code. When the function is called or invoked with arguments, that sequence of code will be executed and the result is given back to the caller- one who invoked the function – by the callee – one who got called.

We use functions in programming for a number of reasons.

* Divide and conquer:
  + We can deal with complex problems in one go. We may want to divide the problem into small parts and solve each one of them
* sharing:
  + We can develop solutions to commonly occurring problem. These solutions can be grouped together in what is called a library. We can avoid reinventing the wheel.
* Maintenance:
  + A code may be required at many places. By putting in a function, we avoid redundancy of code. This helps in maintenance. Any updates need be done at only one place.
* Reliability:
  + Functions which become part of libraries are normally well tested and are made flexible and efficient. We can use these functions with the guarantee of correctness and efficiency. We are not normally very sure of our functions !
* Reuse
  + A function may be used again and again. We do not have to keep writing the function.

**Function Definition:**

The structure of function definition is as follows.

returntype fnname(<list of parameters with type>)

{

<stmt>

return <expr>

}

A function should have a return type. A function always returns a value of a particular type. The compiler should provide a temporary location of this type to hold the result before it is consumed by the caller. The compiler may also convert the expression of return to that of the return type if they do not match.

If a function is not required to return a value at all, then the function return type is made void.

The function name follows the rules of an identifier.

Following the function name, with in parentheses, we specify the parameter list. A parameter list has a list of parameters comma separated. Each parameter is declared with respect its type. When a function is invoked, these parameters are created on an activation record or stack frame. When the function returns, this stack frame or activation record is removed or popped out.

Parameters are always variables. They are initialized on the function call with arguments being copied to the parameters. Parameters exist only within the function call. There is no concept of closure as in Python here.

A function may have no parameters – not at all common. Even then the parentheses are required.

The body of the function follows the parameter list.

Refer to ex0.c for some examples of function definitions and function calls.

int f1(int x)

{

return x \* x;

}

int f2(int x, int y)

{

return x + y;

}

double f3(int x)

{

return sqrt(x);

}

void f4()

{

printf("f4 says hello\n");

}

int main()

{

// calling f1

int a = 25; int b = 10;

int res;

res = f1(a); printf("res : %d\n", res);

res = f1(11); printf("res : %d\n", res);

res = f1(11 + a); printf("res : %d\n", res);

// res = f1(int); // error

res = f2(a, b); printf("res : %d\n", res);

printf("sqrt : %lf\n", f3(25.0));

printf("sqrt : %lf\n", f3(a));

f4();

}

**Function call:**

fnname(arguments …)

A function call has the function name followed by parentheses within which we have a comma separated list of arguments. Arguments are expressions.

When a function is called, the following actions take place.

* An activation record or a stack frame is created. This contains the following.
  + Return address – a place to return when the function execution is complete.
  + Parameters
  + local variables
  + temporary variables
  + location for return value – of return type
* Arguments are evaluated. Arguments are always expressions. The order of evaluation of arguments is not defined.

a = 10; f(a++, a); may result in any of the following calls.

f(10, 10) or f(10, 11)

* Arguments are copied to the corresponding parameters.
* The control is transferred to the called function or the callee.
* The function body(called) is executed.
* When the function executes a return statement or reaches the end of the function body, the callee returns to the caller.
* If the function has a return statement of the form return <expr>, the value of the expression following return is evaluated and is copied to the return location.
* The caller picks up the return value. That is the value of the function call. A function call is an expression.
* If a function is a void function, the control is returned from the callee to the caller and the last two steps do not happen.

**Declaration and Definition:**

This is a function definition.

int f2(int x, int y)

{

return x + y;

}

Can we call the function before definition?

Can we call a function defined in another file?

How would the compiler know

* the number of arguments to pass
* the types of arguments to pass
* the order of arguments to pass
* the return type?

The compiler does not. Normally by default the return type is assumed to be an int. The compiler cannot check for any of these unless it knows what the function expects. We provide this information by declaring the function before calling.

int f2(int x, int y);

This is a function declaration. It is also called prototype or signature or specification of the function. This concept came to C from C++.

There are two terms which we should understand clearly.

**Interface and Implementation.**

The function declaration is the interface. It tells us what the function expects. It does not tell us how the function works. If we change the function definition, the user or the client of the function is not affected.

int f2(int x, int y)

{

int temp = x+ y;

return temp;

}

**Parameter Passing mechanism:**

In ‘C’, the argument is copied to the corresponding parameter. The parameter is not copied back to the argument. This is not possible if the argument is not a l-value.

Changing the parameter will not affect the corresponding argument.

Let us examine the code from ex2.c.

void foo(int x); // declaration

int a = 10;

foo(a);

printf("a : %d\n", a); // 10

Observe that the variable a has not changed. It is impossible for the function foo to change the argument a.

void foo(int x); // declaration

double b = 2.5;

foo(b);

printf("b : %lf\n", b); // 2.5

In this case, as the type of argument does not exactly match the type of parameter, the argument is cast into an integer.

foo((int)b);

But the value of the variable b is not affected either by casting or by the function call.

**How to change arguments by calling function?**

As the parameter passing is only by value, we may pass explicitly a pointer to an argument. Then the parameter becomes a pointer to the argument. Dereferencing the pointer gives us lvalue of the argument. By assigning to the dereferenced pointer, we can change the corresponding argument.

int a = 10;

int \*p = &a; // pointer variable

what(p);

printf("a : %d \*p : %d p : %p\n", a, \*p, p); // 100 100 ...

void what(int\* q)

{

int temp = 100;

\*q = temp;

}

Remember that you pass pointer to variables in scanf so that the variables are changed before scanf returns.

What if we just change the pointer? Let us look at this piece of code.

void bar(int\* q)

{

int temp = 100;

q = &temp;

}

int a = 10;

int \*p = &a; // pointer variable

bar(p);

This will not change a. q gets a copy of pointer to a. That copy is changed to point to a local variable. So, clearly the variable a and variable p are not affected. If we want to change a pointer by calling a function, we should pass a pointer to a pointer. The corresponding parameter will be a pointer to pointer.

**Return mechanism:**

The return is always through a temporary. The expression of return is computed and copied to the temporary variable by the callee. The type of return comes into play here. If the expression of return is not same as the return type of the function, the expression is cast to the return type before copying to the temporary location. The caller will pick up the return value from this temporary and use it the way it wants.

The functions in ‘C’ return a single value. It could also be a pointer. Let us look at a couple of examples.

This function receives a pointer to a variable and returns the same. Not a very meaningful example. This shows how we can return a pointer from a function.

int\* one(int\* x)

{

return x;

}

int c = 1000;

int \*p = one(&c);

printf("%d \n", \*p); // 1000

How about this?

int\* two()

{

int x = 111;

return &x;

}

p = two();

printf("%d \n", \*p);

x is a local variable. Local variables will be part of the stack frame which is created when the function is called and destroyed when the function returns. We are returning a pointer to such a variable. The pointer p in the caller will point to a variable which does not exist. This is a classical example of dangling pointer. The pointer exists; the location to which it points does not. There is no location; but we have access to it.

Dereferencing such a dangling pointer is extremely dangerous. This clearly is an undefined behaviour.

We will discuss function returning pointers later in the course again.