Lecture 4
Functions

Dr. Hacer Yalım Keleş

Ref: Programming in ANSI C, Kumar

OVERVIEW

Let us consider a function that computes the sum of the cubes of the digits of a given positive integer. The function is defined as follows:

```
int cubesum(int number)
  {
    int sum, residue, digit;

    residue = number;
    sum = 0;

    do
      {
        digit = residue % 10;/* rightmost digit */
        sum += digit * digit * digit;
        residue /=10;/* after removing this digit */
      }
    while (residue != 0);

    return sum;
  }
```

The *type* of this function is `int` as indicated by the type specification just before the *function name,* `cube sum`. The function name is followed by the *parameter declarations* enclosed in parentheses. This function has only one *parameter,* `number`, which is of type `int`. The opening brace { following the parameter declarations marks the beginning of the *function body.* The function body consists of the declarations of *local variables* and *function statements.* There are three local variables: `sum`, `residue`, and `digit`. These variables are only accessible in the body of `cubesum`, and not in any other function. The function statements following the variable declarations compute the sum of the cubes of the digits of `number`. The `return` statement terminates the execution of the function and communicates the value `sum` computed by the function to the calling function. The closing brace } marks the end of the function body.

All that a calling function has to knowabout `cubesum` is that it can compute the sum of the cubes of the digits of a given number; the calling function does not have to know the details of how this sum is computed. Any function that requires this sum can call `cubesum`.

FUNCTION DEFINITION

A *function definition* introduces a new function by declaring the type of value it returns and its parameters, and specifying the statements that are executed when the function is called. The general format of the function definition is:

```
function-type function-name (parameter-declarations)
    {
        variable-declarations

        function-statements
    }
```

```
float interest (float prin, float rate, int yrs)
  {
    ...
  }
```

defines `interest` to be a function that returns a value of type `float`, and has three parameters: `prin` and `rate`, which are of type `float`, and `yrs`, which is of type `int`.

Function-type specifies the type of the function and corresponds to the type of value returned by the function.

A function that does not return any value, but only causes some side effects, is declared to be of type `void`.

Function-name is the name of the function being defined.

Parameter-declarations specify the types and names of the *parameters* (also called *formal parameters*) of the function, separated by commas.

If a function does not have any parameters, the keyword `void` is written in place of parameter declarations.

```
void initialize(void)
  {
    ...
  }
```

defines `initialize` to be a function that neither returns any value nor takes any arguments; and the function definition

```
quotient (int i, int j)
  {
    ...
  }
```

defines `quotient` to be a function that returns an integer value and has two integer parameters `i` and `j`. Note that the type of `quotient` is taken to be `int`, since its type has not been explicitly specified.

The *function-body* consists of *variable-declarations* followed by *function-statements,* enclosed within the opening brace { and the matching closing brace }.

Variable-declarations specify types and names of the variables that are local to the function.

A *local variable* is one whose value can only be accessed by the function in which it is declared.

Variables declared local to a function supersede any identically named variables outside the function.

Parameters are treated as if they were declared at the top of the function body.

Thus, functions may be developed independently,

without any concern for variable names used in other functions.

For example, the functions `lcm` and `gcd`

```
int lcm (int m, int n)
 {
   int i;
   ...
 }

int gcd(int m, int n)
  {
    int i;
    ...
  }
```

have identically named parameters and local variables,

but any reference to m, `n`, or `i` in `gcd` has nothing to do with

`m`, `n`, or `i` in `lcm`.

Function-statements can be any valid C statements that are to be executed when the function is called.

The execution of the function terminates when either

- the execution reaches the closing brace at the end of the function body,
- a `return` statement is encountered.

return Statement

A `return` statement can be of one of the following two forms:

- `return` *expression;*

  the value of the *expression* is returned to the calling function.

- `return;`

  returns no value to the calling function.

If the `return` statement is of the first form, the value of the *expression* is returned to the calling function. For example, the function `smaller`

```
double smaller(double x, double y)
  {
    return x < y ? x : y;
  }
```

returns the value of the smaller of the two arguments in the function call.

If the type of the *expression* does not match the type of the function, it is converted to the type of the function.

For example, in the function

```
int trunc(void)
  {
    return 1.5;
  }
```

the `return` statement is equivalent to

```
return (int) 1.5;
```

and returns 1 to the calling function.

The `return` statement of the second form returns no value to the calling function.

This form of the `return` statement should be used only when the function is of type `void`;

Flowing off the end of a function, without encountering a `return` statement, is equivalent to executing a `return` statement of the second form

More than one `return` statement can be used in the same function:

```
int factorial(int n)
  {
    int i, result;

    if (n < 0) return -1;

    if (n == 0) return 1;

    for (i = 1, result = 1; i <= n; i++) result *= i;

    return result;
  }
```

```
int factorial(int n)
  {
    int i, result;

    if (n < 0) return -1;

    if (n == 0) return 1;

    for (i = 1, result = 1; i <= n; i++) result *= i;

    return result;
  }
```

The first executed `return` statement terminates the execution of the function, and the rest of the function body is not executed. Thus, if `factorial` is called with argument `0`, the function will return with the value 1, and the `for` loop will not be executed.

FUNCTION CALL

A function call is an expression of the form

function-name **(** *argument-list* **)**

where *function-name* **is the name of the function called, and** *argument-list* **is a comma-separated list of expressions that constitute the** *arguments* **(also called** *actual arguments***) to the function.**

FUNCTION CALL

the expression

```
cubesum(i)
```

is a function call that invokes the function named `cubesum`

with the argument `i`.

The type of a function-call expression is the same as the type of the function being called, and its value is the value returned by the function. A function call can occur in any place where an expression can occur, such as in

```
if (i == cubesum(i)) printf("%d\n", i);
```

A function call, followed by a semicolon, becomes an expression statement. Thus, the expression in the statement

```
printf("hello there");
```

is a call to the function named `printf` with the argument `"hello there"`.

Function calls can be embedded in other function calls. Thus, the statements

```
t = cubesum(i);
j = cubesum(t);
```

are equivalent to

```
j = cubesum( cubesum(i) );
```

Parentheses must be present in a function call even when the argument list is empty. Thus,

```
initialize() ;
```

is a call to the function named `initialize`, which does not take any argument.

The function in which the function call is contained is said to be the *calling* function and

The function named in the call is said to be the *called* function.

A function call alters the sequential execution of the program.

Upon call, the program control passes from the calling function to the called function, and execution begins from the first executable statement of the called function.

The called function is executed until a `return` or the closing brace of the function is encountered, at which point the control passes back to the point after the function call.

The calling function may choose to ignore the value returned by the called function.

When a function is called, parameters in the called function are bound to the corresponding arguments supplied by the calling function.

The names of the parameters need not be identical to those of the arguments.

C only provides *call by value* parameter passing, meaning thereby that the called function is only provided with the current values of the arguments, but not their addresses, and the corresponding parameters are assigned these values. Since the addresses of the arguments are not available to the called function,.any change in the value of a parameter inside the called function does not cause a change in the corresponding argument. This form of parameter passing is different from the *call by reference* parameter passing.

In call by reference, the address of the argument is supplied to the called function, and any change in the value of a parameter is automatically reflected in the corresponding argument.

```c
#include <stdio.h>

int main(void)
  {
    int i = 1, j = 2;
    void exchange (int, int);

    printf("main :   i = %d j= %d\n", i, j);
    exchange(i,j);
    printf("main :   i = %d j= %d\n", i, j);
    return 0;
  }.

void exchange(int i, int j)
  {
    int t;

    t = i, i = j, j = t;
    printf("exchange: i = %d j= %d\n", i, j);

  }
```

The following is the output of this program:

```
main   :  i   =  1   j=  2
exchange:  i   =  2  j=  1
main : i = 1 j= 2
```

FUNCTION PROTOTYPES

Before calling a function, it must be declared with a prototype of its parameters. The general form for a function declaration is

function-type function-name (parameter-type-list) ;

Thus, the function declaration

```
int cubesum(int n);
```

is a function prototype.

The prototype of a function must agree with the function definition and its use. However, the parameter names in the prototype can be different from the names used in the function definition. These names are effectively treated as comments, and may even be omitted. Thus, the preceding prototypes could have been equivalently written as

```
int cubesum(int);
float interest (float, float, int);
```

When a function for which a prototype has been specified is called, the arguments to the function are converted, as if by assignment, to the declared types of the parameters. Thus, the call

```
interest (prin, rate, yrs)
```

where `prin` and `yrs` are of type int and `rate` is of type `float`, is equivalent to

```
interest ((float) prin, rate, yrs)
```

and no explicit casting is necessary. It is an error if the number of arguments in the call is different from the number in the prototype, or if their types are not the same as or convertible to the types in the prototype.

BLOCK STRUCTURE

A *block* is a sequence of variable declarations and statements enclosed within braces. C does not allow a function to be defined inside another function, but it is permissible to nest blocks and to declare variables and initialize them at the beginning of any block. The *scope* of a variable declared in a block extends from its point of declaration to the end of the block. Such a declaration hides any identically named variable in the outer blocks.

```
int factorial(int n)
  {
    if (n < 0)
        return -1;
    else if (n == 0)
        return 1;
    else
      {
        int. i, result =1;

        for (i = 1; i <= n; i++) result *= i;
        return result;
      }
  }
```

In this version, the variables `i` and `result` are declared inside the block associated with the second `else`, rather than at the beginning of the function block.

```
#include <math.h>

int primesum(int from, int to)
  {
    int i, j, sum =0;

    for (i = from; i <= to; i++)
      {
        int sqrt_i = (int) sqrt(i);

        for (j = 2; j <= sqrt_i; j++)
            if (i % j == 0) /* i is not prime */
                break;

        if (j > sqrt_i)          /* i is prime */
            sum += i;
      }

    return sum;
  }
```

Rather than computing the square root of `i` for every iteration of the inner loop, it is computed only once for a given value of `i` and saved in `sqrt_i`, a variable declared local to the block associated with the outer `for`.

EXTERNAL VARIABLES

Local variables can only be accessed in the function in which they are defined; they are unknown to other functions in the same program. Even if variables in different functions have the same name, they are not related in any way.

If a variable is defined outside any function at the same level as function definitions, it is available to all the functions defined below in the same source file, and is called an *external* variable. Technically, that part of the program within which a name can be used is called its *scope*. The scope of a local variable is the function in which its has been defined. whereas the scope of an external variable is the rest of the source file starting from its definition. Note that the scope of external variables defined before any function definition will be the whole program, and hence such variables are sometimes referred to as *global* variables.

```
int i, j; /* external variables accessible in
             input, compute, and output */

void input(void)
  {
    scanf("%d %d", &i, &j);
  }

int k; /* external variable accessible in compute and output */

void compute(void)
  {
    k = power(i, j) ;
  }

void output(void)
  {
    printf ("i = %d j = %d k = %d", i, j, k);
  }
```

A local variable definition supersedes that of an external variable. If an external variable and a local variable have identical names, all references to that name inside the function will refer to the local variable. Thus, the following definition of the function `power` can be inserted between the functions `compute` and `output` without affecting `i` in `output`:

```
int power(int base, int exponent)
  {
    int i, result; •

    for (i = 1, result = 1; i <= exponent; i++)
        result *= base;
    return result;
  }
```

STORAGE CLASSES

A variable belongs to one of the two storage classes: *automatic* and *static*. The storage class determines the lifetime of the storage associated with the variable.

Automatic Variables

A variable is said to be *automatic* if it is allocated storage upon entry to a segment of code, and the storage is deallocated upon exit from this segment. A variable is specified to be automatic by prefixing its type declaration with the storage class specifier `auto` in the following manner:

`auto` *type variable-name;*

Variables can be declared to be automatic only within a block. If the storage class has not been explicitly specified, a variable declared within a block is taken to be `auto`.

```
int factorial(int n)
  {
    int i, result;
    ...
  }
```

are equivalent to

```
int factorial(int n)
  {
    auto int i, result;
    ...
  }
```

and declare `i` and `result` to be automatic variables of type integer.

Static Variables

A variable is said to be *static* if it is allocated storage at the beginning of the program execution and the storage remains allocated until the program execution terminates. Variables declared outside all blocks at the same level as function definitions are always static. Within a block, a variable can be specified to be static by prefixing its type declaration with the storage class specifier `static` in the following manner:

`static` *type variable-name;*

Variables declared `static` can be initialized only with constant expressions. Unlike with `auto` variables, the initialization takes place only once, when the block is entered for the first time. If not explicitly initialized, `static` variables are assigned the default initial value of zero. The values assigned to `static` variables are retained across calls to the function in which they have been declared.

The following program illustrates the difference between `auto` and `static` variables:

```
♦include <stdio.h>

int main(void)
{
    int i;
    void incr(void);

    for (i =0; i < 3; i++) incr();
    return 0;
  }

void incr(void)
  {
    int auto_i = 0;
    static int static_i = 0;

    printf("auto = %d static = %d\n",
              auto_i++, static_i++);
  }
```

The output generated by this program is:

```
auto = 0 static = 0
aut.o = 0 static = 1
auto = 0 static = 2
```

Reading Assignment

Please read Section 5.8 from the book. I'll assume you have read the whole section and implemented the related illustrative examples.