

CS 101: Computer Programming and Utilization

Jul-Nov 2017

Umesh Bellur
(cs101@cse.iitb.ac.in)

Lecture 10: Functions

About These Slides

- Based on Chapter 9 of the book
An Introduction to Programming Through C++
by Abhiram Ranade (Tata McGraw Hill, 2014)
- Original slides by Abhiram Ranade
 - First update by Varsha Apte
 - Second update by Uday Khedker
 - Third update by Sunita Sarawagi

Can We Define New Commands?

- We already have many commands, e.g.
 - `sqrt(x)` evaluates to the square root of x
 - `forward(d)` moves the turtle forward d pixels
- Can we define new commands? e.g.
 - `gcd(m,n)` should evaluate to the GCD of m,n
 - `dash(d)` should move the turtle forward, but draw dashes as it moves rather than a continuous line
- *Function*: official name for command

Outline

- Examples of defining and using functions
- How to define a function in general
- How a function executes
- **Contract** view of functions
- Passing parameters by reference

Why Functions?

Write a program that prints the GCD of 36, 24, and of 99, 47

Using what you already know:

Make 2 copies of code to find GCD. Use the first copy to find the GCD of 36, 24 Use the second copy to find the GCD of 99, 47

Duplicating code is not good


May make mistakes in copying. What if we need the GCD at 10 places in the program?

This is inelegant. Ideally, you should not have to state anything more than once

```
main_program{  
    int m=36, n=24;  
    while(m % n != 0){  
        int r = m%n;  
        m = n;  
        n = r;  
    }  
    cout << n << endl;  
    m=99; n=47;  
    while(m % n != 0){  
        int r = m%n;  
        m = n;  
        n = r;  
    }  
    cout << n << endl;  
}
```

Using a Function (exactly how it works, later)

- A complete program
= function definitions
+ main program
- **Function definition:**
information about
 - function name
 - how it is to be called
 - what it computes
 - what it returns
- **Main program:**
calls or invokes functions
 - gcd(a,b) : call/invocation
 - gcd(99,c) : another call
 - Values supplied for each call:
arguments or parameters to
the call



```
int gcd(int m, int n){  
    while(m % n != 0){  
        int r = m%n;  
        m = n;  
        n = r;  
    }  
    return n;  
}  
  
main_program{  
    int a=36, b=24, c=47;  
    cout <<gcd(a,b) << endl;  
    cout <<gcd(99, c)<< endl;  
}
```

Form of Function Definitions

```
return-type name-of-function (  
    parameter1-type parameter1-name,  
    parameter2-type parameter2-name,  
    ...)  
{ function-body }
```

- return-type**: the type of the value returned by the function,
e.g. `int`

Some functions may not return anything
(discussed later)

- name-of-function**: e.g. `gcd`

- parameter**: variables that to hold the values of the
arguments to the function. `m,n` in `gcd`

- function-body**: code that will get executed

Function Execution

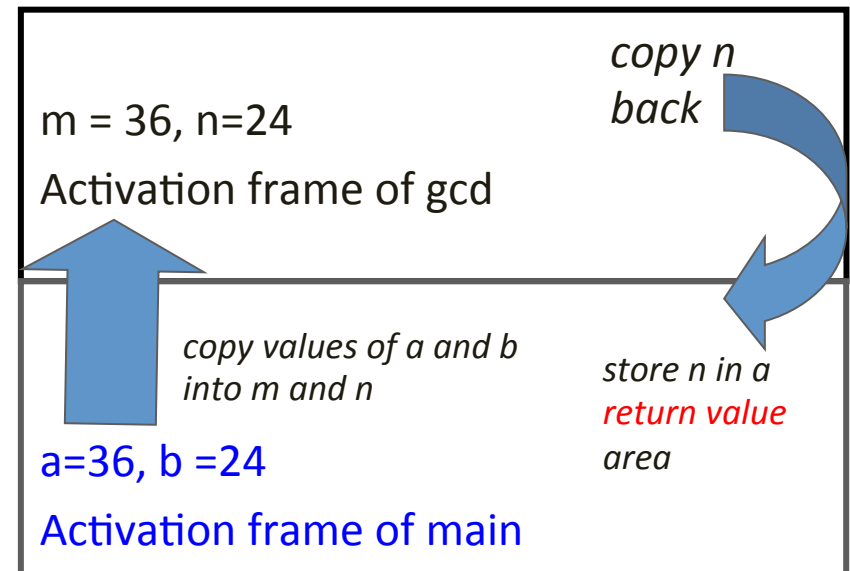
```
int gcd(int m, int n) {  
    while(m % n != 0){  
        int r = m%n;  
        m = n;  
        n = r;  
    }  
    return n;  
}  
  
main_program{  
    int a=36,b=24;  
    cout << gcd(a,b) << endl;  
    cout << gcd(99,47)<< endl;  
}
```

- Each function has a separate data space (independent scope)
- These data spaces are arranged in a data structure called **stack**
- Imagine the data spaces as data books and stacked up one on the other
- The book on the top of the stack is the one we can access
Last-In-First-Out (LIFO)

Function Execution

```
int gcd(int m, int n) {  
    while(m % n != 0){  
        int r = m%n;  
        m = n;  
        n = r;  
    }  
    return n;  
}  
  
main_program{  
    int a=36,b=24;  
    cout << gcd(a,b) << endl;  
    cout << gcd(99,47)<< endl;  
}
```

- Data space of a function is also called an **activation frame** (or activation record)



(contd.)

- Execution of the called function ends when **return** statement is encountered
- Value following the keyword **return** is copied back to the calling program, to be used in place of the expression `gcd(...,...)`
- Activation frame of function is destroyed, i.e. memory reserved for it is taken back
- `main_program` resumes execution

Function Execution

```
int gcd(int m, int n) {  
    while(m % n != 0){  
        int r = m%n;  
        m = n;  
        n = r;  
    }  
    return n;  
}  
  
main_program{  
    int a=36,b=24;  
    cout << gcd(a,b) << endl;  
    cout << gcd(99,47)<< endl;  
}
```

- Activation frame: area in memory where function variables are stored

gcd activation frame is destroyed

a=36, b =24 returned value of n
Activation frame of main

Function Execution

```
int gcd(int m, int n) {  
    while(m % n != 0){  
        int r = m%n;  
        m = n;  
        n = r;  
    }  
    return n;  
}  
  
main_program{  
    int a=36,b=24;  
    cout << gcd(a,b) << endl;  
    cout << gcd(99,47)<< endl;  
}
```

How A Function Executes

1. main_program executes and reaches gcd(36,24)
2. main_program suspends
3. Preparations made to run subprogram gcd:
 - Area allocated in memory where gcd will have its variables. **activation frame**
 - Variables corresponding to parameters are created in activation frame
 - Values of arguments are copied from activation frame of main_program to that of gcd. This is termed **passing arguments by value**
4. Execution of function-body starts

Remarks

- Set of variables in calling program e.g. `main_program` is completely disjoint from the set in called function, e.g. `gcd`
- Both may contain same name. Calling program will reference the variables in its activation frame, and called program in its activation frame
- New variables can be created in called function
- Arguments to calls/invocations can be expressions, which are first evaluated before called function executes
- Functions can be called while executing functions
- A declaration of function must appear before its **call**

Function To Compute LCM

We can compute the least common multiple of two numbers m , n using the identity

$$\text{LCM}(m,n) = m*n/\text{GCD}(m,n)$$

```
int lcm(int m, int n){  
    return m*n/gcd(m,n);  
}
```

`lcm` calls `gcd`.

Program To Find LCM Using Functions

gcd, lcm

Function *definitions* appear
before their calls

```
int gcd(int m, int n)
{ ...}
int lcm(int m, int n)
{
    return m*n/gcd(m,n);
}
main_program{
cout << lcm(50,75);
}
```

Function *declarations* appear
before their calls

```
int lcm(int m, int n);
main_program{
    cout << lcm(50,75);
}
int gcd(int m, int n)
{ ...}
int lcm(int m, int n)
{
    return m*n/gcd(m,n);
}
```


Execution

- main_program starts executing
- main_program suspends when the call lcm(..) is encountered
- Activation frame created for lcm
- lcm starts executing after 50, 75 copied to m,n call to gcd encountered. lcm suspends
- Activation frame created for gcd
- Execution of gcd starts after copying arguments 50, 75 to m,n of gcd.
- gcd executes. Will returns 25 as result
- Result copied into activation frame of lcm, to replace call to gcd
- Activation frame of gcd destroyed
- lcm continues execution using result. $m*n/gcd(m,n) = 50*75/25 = 150$ computed
- 150 returned to main_program, to replace call to lcm
- Activation frame of gcd destroyed
- main_program resumes and prints 15

Execution of our Program

```
int gcd(int m, int n)
{ ...}
int lcm(int m, int n)
{
    return m*n/gcd(m,n);
}
main_program{
cout << lcm(50,75);
}
```

A Function to Draw Dashes

```
void dash(int d) {  
    while(d>10) {  
        forward(10);    penUp(); d -= 10;  
        if(d<10) break;  
        forward(10); penDown(); d -= 10;  
    }  
    forward(d); penDown();  
    return;  
}  
  
main_program{  
    turtleSim();  
    repeat(4) {dash(100);    right(90);}  
}
```

Remarks

- Dash does not return a value, so its return type is `void`
- The `return` statement used in the body does not have a value after the key word `return`
- Exercise: write an invariant for the loop in dash

Contract View Of Functions

- Function : piece of code which takes the responsibility of getting something done
- Specification : what the function is supposed to do Typical form: If the arguments satisfy certain properties, then a certain value will be returned, or a certain action will happen
certain properties = preconditions
- Example: gcd : If positive integers are given as arguments, then their GCD will be returned
- If preconditions are not satisfied, nothing is promised

Contract View of Functions (contd.)

- Function = contract between the programmer who wrote the function, and other programmers who use it
- Programmer who uses the function trusts the function writer
- Programmer who wrote the function does not care which program uses it
- Analogous to giving cloth to tailor. Tailor promises to give you a shirt if the cloth is good. Tailor does not care who wears the shirt, wearer does not care how it was stitched

Contract View of Functions (contd.)

Postconditions: After the function finishes execution, does it modify the state of the program?

Example: After dash finishes its execution it might always leave the pen up (not true for the code given earlier)

Exercise: Modify the code of dash to ensure that the pen is up at the end

Post conditions must also be mentioned in the specification

Writing clear specifications is very important

Some Shortcomings

Using what we saw, it is not possible to write functions to do the following:

- A function that exchanges the values of two variables
- A function that returns not just one value as the result, but several. For example, we might want a function to return polar coordinates given Cartesian coordinates

Exchanging The Values of Two Variables, Attempt 1

```
void exchange(int a,
int b) {
    int temp = a;
    a = b; b = temp;
    return;
}
main_program{
    int a=1, b=2;
    exchange(a, b);
    cout << a << ' ' <<
        b << endl;
}
```

- Does not work. 1, 2 will get printed
- When exchange is called, 1, 2 are placed into m, n
- Execution of exchange exchanges values of m,n
- But the change in m,n is not reflected in the values of a,b of main_program

Exchanging The Values of Two Variables, Attempt 1

```
void exchange(int a,
int b) {
    int temp = a;
    a = b; b = temp;
    return;
}

main_program{
    int a=1, b=2;
    exchange(a, b);
    cout << a << ' ' <<
        b << endl;
}
```

Reference Parameters

```
void exchange(int &m, int
&n){
    int temp = m;
    m = n; n = temp;
    return;
}
main_program{
    int a=1, b=2;
    exchange(a,b);
    cout << a << ' ' <<
        b << endl;
}
```

- “&” before the name of the parameter: Says, do not allocate space for this parameter, but instead just use the variable from the calling program
- With this, when function changes m,n it is really changing a,b
- Such parameters are called **reference parameters**

Remark

If a certain parameter is a reference parameter, then the corresponding argument is said to be **passed by reference**

Cartesian to Polar

```
void CtoP(double x, double y, double &radius, double
&theta){
    radius = sqrt(x*x + y*y);
    theta = atan2(y, x); //arctan
return;
}
main_program{
    double x=1, y=1, r, theta;
    CtoP(x,y,r,theta);
    cout << r << ' ' << theta << endl;
}
// Because r, theta in CtoP are reference parameters,
// changing them changes the value of r, theta in
// the main program.
// Hence will print sqrt(2) and pi/4 (45 degrees)
```

Pointers

- A **pointer** is a variable that can store addresses
 - The number assigned to a byte (different from what is stored in the byte) is said to be its **address**.
 - If a computer has B bytes of memory ---- address will range from 0 to B-1.
- What we accomplished using reference variables can also be accomplished using pointers.
- Pointers will also be useful elsewhere.

How to find the address of a variable

- The operator **&** can be used to get the address of a variable. (The same & is used to mark reference parameters; but the meaning is different)

```
int t;
```

```
cout << &t << endl;
```

- This prints the address of variable t.
- Addresses are in hexadecimal (16) radix, i.e. they will consist of a sequence of hexadecimal digits prefixed by “0x”. Note: hexadecimal digits: 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F.

Variables that can store addresses

- To create a variable `v` in which you can store addresses of variables of type `int` you write:

```
int *v;    // read as “int star v”
```

- The `*` is not multiplication. Think of it as `(int*) v`; where `int*` means the type: “address of `int`”.

```
int p;  
v = &p;  
cout << v << ' ' << &p << endl;  
// both print same
```

- In general, to create a variable `w` to store addresses of variables of type `T`, write:

```
T* w;
```


The dereferencing operator *

- If `v` contains the address of `p`, then we can get to `p` by writing `*v`.

```
int *v;
```

```
int p;
```

```
v = &p;
```

```
*v = 10; // as good as p = 10.
```

- Think of `*` as the inverse of `&`.
- `&p` : the address of the variable `p`
- `*v` : the variable whose address is in `v`

Pointers in functions

```
void CtoP(double x,
double y, double *pr,
double *ptheta){
    *pr = sqrt(x*x +
y*y);
    *ptheta =
atan2(x,y);
    return;
}
main_program{
    double r, theta;
    CtoP(1,1,&r,&theta);
    cout << r << ' '
        << theta <<
endl;
}
```

- main_program calls CtoP, supplying &r, &theta as third and fourth arguments.
- This is acceptable because corresponding parameters have type double*.
- The addresses are copied into pr, ptheta of CtoP.
- *pr means the variable whose address is in pr, in other words, the variable r of main_program.
- Thus CtoP changes the variables of main_program.

• This is how it works: $\text{main_program} \rightarrow \text{CtoP} \rightarrow \text{main_program}$

Remarks

- You cannot store an address of an `int` variable into an `int` variable, nor store an `int` into a variable of type `int*`.

```
int *v, p;
```

```
v = p;    // not allowed
```

```
p = v;    // not allowed
```

- For now, assume that the only operations you can perform on a variable of type `T*` are
 - dereference it,
 - store into it a value `&v` where `v` is of type `T`,
 - store it into another variable of type `T*`
 - pass it to a function as an argument, provided corresponding parameter is of type `T*`

Concluding Remarks

- Functions allow us to divide the program into smaller parts such that each part deals with a particular functionality
- Apart from separation of computations, functions also allow separation of data spaces for computations
- This separation of concerns is a major help in understanding programs
- Functions can be seen as another control flow mechanism (apart from sequence, selection, and iteration)
- Function calls follow the LIFO (Last-In-First-Out) policy of execution of nested calls