ESC101N
Fundamentals of Computing

Arnab Bhattacharya
arnabb@iitk.ac.in

Indian Institute of Technology, Kanpur
http://www.iitk.ac.in/esc101/

1st semester, 2010-11

Tue, Wed, Fri 0800-0900 at L7

Functions

- Functions are useful
  - To repeat the same task
  - To modularize the program and increase the understanding
  - To correct errors (called "bugs") in the program
- They are defined using the following format

```
return_type function_name(parameter_list) // function
    header
{
  ... // function body
}
```

- They are called (from other functions) using

```
variable = function_name(argument_list)
```

- Example

```
int sum(int a, int b)
{
  ...
}
z = sum(x, y);
```

Parts of a function

- `function_name` is the name of the function
  - Follows the same rules as variable names
- `return_type` is the type of value that the function returns
  - If a function does not return anything, it is specified as `void`
- `parameter_list` is a comma-separated list that describes the parameters of the function including their name and type
  - Even arrays can be parameters
- Function body may contain any valid C code
  - If there is a return type other than `void`, the function must have a `return` statement

Summation function

- Function to add two integers

```
#include <stdio.h>

int main()
{
    int x, y, z;

    scanf("%d", &x);
    scanf("%d", &y);

    z = sum(x, y);

    printf("%d\n", z);
}

int sum(int a, int b)
{
    int c;

    c = a + b;

    return c;
}
```

- What if doubles need to be added?

Summation of two doubles

- New function to add two doubles

```c
#include <stdio.h>

int sum_i(int a, int b)
{
    int c;
    c = a + b;
    printf("Inside sum_i: %d + %d = %d\n", a, b, c);
    return c;
}

float sum_f(float a, float b)
{
    float c;
    c = a + b;
    printf("Inside sum_f: %f + %f = %f\n", a, b, c);
    return c;
}

int main()
{
    float a = 4.2, b = 3.6, c;
    int x = 4, y = 3, z;

    c = sum_f(a, b);
    z = sum_i(x, y);

    printf("%f + %f = %f\n", a, b, c);
    printf("%d + %d = %d\n", x, y, z);
}
```

- Function name must be different

Scope of variables

- For blocks of code
  - A variable retains its visibility *only* in the block where it is declared
    - Declaration of a variable is done using the format

      ```
      <type> <name>[= value];
      int i;
      ```
  - Variable declared in an outer block *is* visible in all inner blocks
  - Variable declared in an inner block *is not* visible in outer blocks
  - If inner block declares a variable having the same name as that of a variable in the outer block, the one in the outer block is *not* visible in the inner block
    - Variable in the outer block becomes visible once more when the inner block is finished
    - Inner block variable may even have a different type

Scope of variables

- For blocks of code
    - A variable retains its visibility *only* in the block where it is declared
        - Declaration of a variable is done using the format

          ```
          <type> <name>[= value];
          int i;
          ```
    - Variable declared in an outer block *is* visible in all inner blocks
    - Variable declared in an inner block *is not* visible in outer blocks
    - If inner block declares a variable having the same name as that of a variable in the outer block, the one in the outer block is *not* visible in the inner block
        - Variable in the outer block becomes visible once more when the inner block is finished
        - Inner block variable may even have a different type
- For functions
    - Rules are same for blocks in the function
    - There is no inner and outer functions
    - So, no variable of a function is visible in any other function
    - This includes variables in the parameter list

Call by value

- Functions get access to the values of the variables
- They do not access the variables passed to them
- Important: Functions *cannot* change the value of a variable passed to it

```c
#include <stdio.h>

void swap(int a, int b)
{
    int t = a;
    a = b;
    b = t;
}

int main()
{
    int x = 3, y = 4;
    printf("x = %d, y = %d\n", x, y);

    swap(x, y);

    printf("x = %d, y = %d\n", x, y);
}
```

Arrays and functions

- Arrays *cannot* be returned
- Arrays are not called by value
- Since only the array name is passed, its contents are *not* copied
- Functions receive the original array
- Important: Functions *can* change the values of array elements

```
#include <stdio.h>

void swap(int a[])
{
    int t = a[0];
    a[0] = a[1];
    a[1] = t;
}

int main()
{
    int x[] = {3, 4};
    printf("x[0] = %d, x[1] = %d\n", x[0], x[1]);

    swap(x);

    printf("x[0] = %d, x[1] = %d\n", x[0], x[1]);
}
```

Passing one-dimensional arrays to functions

- Size of array is not required in the parameter list
- Suppose size of array in parameter list is p
- Suppose size of actual array that is passed is a
  - If $p == a$, no issues
  - If $p < a$, only the *first* p elements of array should be accessed
  - If $p > a$, extra $(p - a)$ elements of array in function is filled up with random values

```c
#include <stdio.h>
void p_equal(int a[2])
{
    printf("Equal: a[0] = %d, a[1] = %d\n", a[0], a[1]);
}
void p_less(int a[1])
{
    printf("Less: a[0] = %d\n", a[0]);
}
void p_more(int a[3])
{
    printf("More: a[0] = %d, a[1] = %d, a[2] = %d\n", a[0], a[1], a[2]);
}
int main()
{
    int x[2] = {3, 4};
    printf("Original: x[0] = %d, x[1] = %d\n", x[0], x[1]);
    p_equal(x);
    p_less(x);
    p_more(x);
}
```

Searching an array

```c
#include <stdio.h>

int search(int q, int a[], int size)
{
    int i;

    for (i = 0; i < size; i++)
        if (q == a[i])
            return i;

    return -1;
}

int main()
{
    int a[8], b[4];
    int q;
    int i;
    int pa, pb;

    for (i = 0; i < 8; i++)
        scanf("%d", &a[i]);
    for (i = 0; i < 4; i++)
        scanf("%d", &b[i]);

    scanf("%d", &q);

    pa = search(q, a, 8);
    pb = search(q, b, 4);

    printf("Position of %d in a is %d\n", q, pa);
    printf("Position of %d in b is %d\n", q, pb);
}
```

Passing two-dimensional arrays to functions

- Second size, i.e., number of columns is required in the parameter list
- For multi-dimensional arrays, all sizes except the first are required

```
void mul(int a[][3], int b[][2], int c[][2], int size)
{
  int i, j, k;
  for (i = 0; i < size; i++)
    for (k = 0; k < 2; k++)
    {
      c[i][k] = 0;
      for (j = 0; j < 3; j++)
        c[i][k] = c[i][k] + a[i][j] * b[j][k];
    }
}
```

- Can be called using

```
mul(a, b, c, 2);
```

Example: matrix multiplication I

```c
#include <stdio.h>

void mul(int a[][3], int b[][2], int c[][2], int size)
{
    int i, j, k;

    for (i = 0; i < size; i++)
        for (k = 0; k < 2; k++)
        {
            c[i][k] = 0;
            for (j = 0; j < 3; j++)
                c[i][k] = c[i][k] + a[i][j] * b[j][k];
        }
}

int main()
{
    int a[2][3], b[3][2], c[2][2];
    int i, j, k;

    for (i = 0; i < 2; i++)
        for (j = 0; j < 3; j++)
            scanf("%d", &a[i][j]);

    for (j = 0; j < 3; j++)
        for (k = 0; k < 2; k++)
            scanf("%d", &b[j][k]);

    mul(a, b, c, 2);
```

Example: matrix multiplication II

```
    printf("Matrix\n");
    for (i = 0; i < 2; i++)
    {
        for (j = 0; j < 3; j++)
            printf("%d\t", a[i][j]);
        printf("\n");
    }
    printf("multiplied with\n");
    for (j = 0; j < 3; j++)
    {
        for (k = 0; k < 2; k++)
            printf("%d\t", b[j][k]);
        printf("\n");
    }
    printf("produces\n");
    for (i = 0; i < 2; i++)
    {
        for (k = 0; k < 2; k++)
            printf("%d\t", c[i][k]);
        printf("\n");
    }
}
```

Recursion

- When a function is defined in terms of itself, it is called a recursive function and the process is called recursion
- Between calls, the argument must change
- There must be at least one base case in the recursive function
- The argument values must proceed towards the base case
- If these are not satisfied, the recursive function will not terminate
- The program will throw a segmentation fault error

```
int fact(int n)
{
  if ((n == 1) || (n == 0))
    return 1;

  return n * fact(n - 1);
}
```

Example: factorial

```
#include <stdio.h>

int fact(int n)
{
    //printf("factorial of %d called\n", n);
    if ((n == 1) || (n == 0))
        return 1;

    return n * fact(n - 1);
}

int main()
{
    int n;
    scanf("%d", &n);

    printf("Factorial = %d\n", fact(n));
}
```

Mutual recursion

- When two functions are defined in terms of each other, they are called mutually recursive functions and the process is called mutual recursion

```
int odd(int n)
{
  if (n == 1)
    return 1;

  return even(n - 1);
}

int even(int n)
{
  if (n == 1)
    return 0;

  return odd(n - 1);
}
```

Example: odd or even

```
#include <stdio.h>

int odd(int n)
{
    printf("odd  %d called\n", n);
    if (n == 1)
        return 1;
    return even(n - 1);
}

int even(int n)
{
    printf("even %d called\n", n);
    if (n == 1)
        return 0;
    return odd(n - 1);
}

int main()
{
    int n;
    scanf("%d", &n);
    if (n <= 0)
    {
        printf("%d must be a positive integer\n");
        return -1;
    }
    if (odd(n)) // odd(n) == 1
        printf("%d is odd\n", n);
    else
        printf("%d is even\n", n);
}
```

Double recursion

- A function may call itself recursively twice (or multiple times)

```
#include <stdio.h>

int choose(int n, int k)
{
    printf("choose called for %d and %d\n", n, k);
    if ((k == 0) || (n == k))
        return 1;
    return choose(n - 1, k) + choose(n - 1, k - 1);
}

int main()
{
    int n, k;
    scanf("%d", &n);
    scanf("%d", &k);
    printf("%d choose %d is %d\n", n, k, choose(n, k));
}
```

Example: quicksort I

```c
#include <stdio.h>

void swap(int a[], int i, int j)    // interchange a[i] and a[j]
{
    int t = a[i];
    a[i] = a[j];
    a[j] = t;
}

int partition(int a[], int s, int e)    // partition a[s] to a[e]
{
    printf("partition called for %d to %d\n", s, e);
    int l;  // l denotes last element of left partition
    int i;

    swap(a, s, (s + e) / 2);    // move pivot from middle to 0
    l = s;  // set size of left partition to 0

    for (i = s + 1; i <= e; i++)    // examine all elements
        if (a[i] < a[s])    // a[i] should go to left partition
        {
            l++;    // make left partition larger by incrementing l
            swap(a, l, i);  // move a[i] to left partition
        }

    swap(a, s, l);  // put pivot to its correct position l

    return l;   // return demarcation of partitions
}
```

Example: quicksort II

```c
void quicksort(int a[], int s, int e)   // sort a[s] to a[e]
{
    printf("quicksort called for %d to %d\n", s, e);
    int l;

    if (s >= e) // array contains 0 or 1 element
        return; // array is already sorted

    l = partition(a, s, e); // partition

    quicksort(a, s, l - 1); // sort left partition
    quicksort(a, l + 1, e); // sort right partition
}

int main()
{
    int x[5];
    int i;

    for (i = 0; i < 5; i++)
        scanf("%d", &x[i]);

    quicksort(x, 0, 5 - 1);

    for (i = 0; i < 5; i++)
        printf("%d ", x[i]);
    printf("\n");
}
```