

COSC1101 – Programming Fundamentals

Maham Khan

Lecture – 11&12

Writing a function

- You need to decide that how the function will *look* like:
 - Return type
 - Name
 - Types of parameters (number of parameters)
- You have to write the body (the actual code).

Function parameters

- The parameters are *local variables* inside the body of the function.
 - When the function is called they will have the values *passed in*.
 - The function gets *a copy* of the values passed in (we will later see how to pass a *reference* to a variable).

Sample Function

```
int add2nums( int firstnum, int secondnum )  
{  
    int sum;  
  
    sum = firstnum + secondnum;  
  
    // just to make a point  
    firstnum = 0;  
    secondnum = 0;  
  
    return(sum) ;  
}
```

Testing add2nums

```
int main(void)
{
    int y,a,b;

    cout << "Enter 2 numbers\n";
    cin >> a >> b;

    y = add2nums(a,b) ;

    cout << "a is " << a << endl;
    cout << "b is " << b << endl;
    cout << "y is " << y << endl;
    return(0) ;
}
```

What happens here?

```
int add2nums(int a, int b)
{
    a=a+b;
    return(a) ;
}
```

...

```
int a,b,y;
```

...

```
y = add2nums(a,b) ;
```

Local variables

- Parameters and variables declared inside the definition of a function are *local*.
- They only exist inside the function body.
- Once the function returns, the variables no longer exist!
 - That's fine! We don't need them anymore!

Block Variables

- You can also declare variables that exist only within the *body* of a compound statement (*a block*):

```
{  
  int foo;  
  ...  
  ...  
}
```


Global variables

- You can declare variables outside of any function definition – these variables are *global variables*.
- Any function can access/change global variables.
- Example: flag that indicates whether debugging information should be printed.

Scope

- The *scope* of a variable is the portion of a program where the variable has meaning (where it exists).
- A global variable has global (unlimited) scope.
- A local variable's scope is restricted to the function that declares the variable.
- A block variable's scope is restricted to the block in which the variable is declared.


A note about Global vs. File scope

- A variable declared outside of a function is available everywhere, but only the functions that follow it in the file know about it.
- The book talks about *file scope*, I'm calling it *global scope*.

Block Scope

```
int main(void) {  
    int y;  
    {  
        int a = y;  
        cout << a << endl;  
    }  
  
    cout << a << endl;  
}
```

*Error – a doesn't exist
outside the block!*

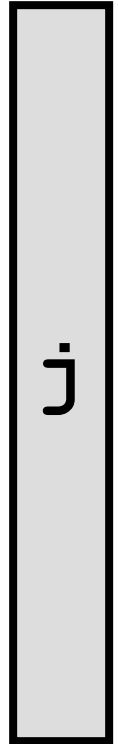


Nesting

- In C++:
 - There is no nesting of function definitions.
 - You don't need to know who calls a function to know the scope of its variables!
 - There is nesting of variable scope in blocks.

Nested Blocks

```
void foo(void) {  
    for (int j=0;j<10;j++) {  
        int k = j*10;  
        cout << j << "," << k << endl;  
        {  
            int m = j+k;  
            cout << m << "," << j << endl;  
        }  
    }  
}
```



Storage Class

- Each variable has a *storage class*.
 - Determines the period during which the variable exists *in memory*.
 - Some variables are created only once (memory is set aside to hold the variable value)
 - Global variables are created only once.
 - Some variables are re-created many times
 - Local variables are re-created each time a function is called.

Storage Classes

- **auto** – created each time the block in which they exist is *entered*.
- **register** – same as **auto**, but tells the compiler to make as fast as possible.
- **static** – created only once, even if it is a local variable.
- **extern** – global variable declared elsewhere.

Specifying Storage Class

```
auto int j;
```

```
register int i_need_to_be_fast;
```

```
static char remember_me;
```

```
extern double a_global;
```

Practical Use of Storage Class

- Local variables are **auto** by default.
- Global variables are **static** by default.
- Declaring a local variable as **static** means it will *remember* its last value (it's not destroyed and recreated each time its scope is entered).

The Scope of Functions

- In C++ we really talk about the scope of an identifier (name).
 - Could be a function or a variable (or a class).
- Function names have *file* scope
 - everything that follows a function definition in the same file can use the function.
- Sometimes this is not convenient
 - We want to call the function from the top of the file and define it at the bottom of the file.

Function Prototypes

- A Function prototype can be used to *tell* the compiler what a function looks like
 - So that it can be called even though the compiler has not yet seen the function definition.
- A function prototype specifies the function name, return type and parameter types.

Example prototypes

```
double sqrt( double) ;
```

```
int add2nums( int, int) ;
```

```
int counter(void) ;
```

Using a prototype

```
int counter(void) ;
```

```
int main(void) {  
    cout << counter() << endl;  
    cout << counter() << endl;  
    cout << counter() << endl;  
}
```

```
int counter(void) {  
    int count = 0;  
    count++;  
    return(count) ;  
}
```

Flow of Control in Functions

- When the program is executed (that is, run) execution always begins at the first statement in the function main no matter where it is placed in the program.
- Other functions are executed only when they are called.
- Function prototypes appear before any function definition, so the compiler translates these first.
- The compiler can then correctly translate a function call.

Flow of Control in Functions contd

- A function call statement results in the transfer of control to the first statement in the body of the called function.
- After the last statement of the called function is executed, the control is passed back to the point immediately following the function call.
- A value-returning function returns a value. Therefore, for value-returning functions, after executing the function when the control goes back to the caller, the value that the function returns replaces the function call statement.

Formal and actual arguments

```
int sum(int, int);           //declaration

void main()
{
    int a=5, b=6,ans;
    ans =sum(a , b);         //calling function
    printf ("Answer : %d",ans);
}

int sum (int x, int y)       //function definition
{
    int val;
    val = x +y;
    return val;
}
```

Formal and actual arguments

- The argument listed in the function calling statement are referred to as actual arguments.
- They are the actual values passed to a function to compute a value or to perform a task.
- The argument used in the function declaration are referred as formal arguments.
- They are simply formal variables that accept or receive the values supplied by the calling program.

Return values and their types

- We can pass n numbers of values to the called function, but the called function can only return one value per call.
- The return statement can take one of the following form:
 return;
 return(expression);
- The return only does not return any value.
- return statement with expression returns the value of the expression
- There can be more than one return statement if there is use of conditional statement.

```
// Creating and using a programmer-defined function.
```

```
#include <iostream.h>
```

```
int square( int );    // function prototype
```

Function prototype: specifies data types of arguments and return values. **square** expects an **int**, and returns an **int**.

```
int main()
```

```
{
```

```
    // loop 10 times and calculate and output
```

```
    // square of x each time
```

```
    for ( int x = 1; x <= 10; x++ )
```

```
        cout << square( x ) << " ";    // function call
```

```
    cout << endl;
```

```
    return 0;    // indicates successful termination
```

Parentheses () cause function to be called. When done, it returns the result.

```
// square function definition returns square of an integer
```

```
int square( int y )    // y is a copy of argument to function
```

```
{
```

```
    return y * y;    // returns square of y as an int
```

```
}    // end function square
```

Definition of **square**. **y** is a copy of the argument passed. Returns **y * y**, or **y** squared.

```
1  4  9 16 25 36 49 64 81 100
```

compute square and cube of numbers [1..10] using functions

```
#include<iostream.h>

int square(int); // prototype
int cube(int);   // prototype
main()
{   int i;
    for (int i=1;i<=10;i++){

        cout<< i<< "square=" << square(i) << endl;
        cout<< i<< "cube="    <<cube(i) << endl;
    } // end for
    return 0;
} // end main function
int square(int y) //function definition
{
    return  y*y; // returned Result
}

int cube(int y) //function definition
{
    return  y*y*y; // returned Result
}
```

Output
1 square=1
1 cube=1
2 square=4
2 cube=8
.
.
.
.
10 square=100
10 cube=1000

```
// Finding the maximum of three floating-point (real) numbers.
```

```
#include <iostream.h>
```

```
double maximum( double, double, double ); // function prototype
```

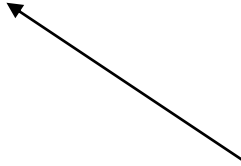
```
int main()
```

```
{  
    double number1, number2;  
    double number3;  
  
    cout << "Enter three real numbers: ";  
    cin >> number1 >> number2 >> number3;  
  
    // number1, number2 and number3 are arguments to the maximum function call  
    cout << "Maximum is: "  
        << maximum( number1, number2, number3 ) << endl;  
    return 0; // indicates successful termination  
  
} // end main
```

```
// function maximum definition. x, y and z are parameters
```

```
double maximum( double x, double y, double z )
```

```
{  
    double max = x; // assume x is largest  
    if ( y > max ) // if y is larger,  
        max = y; // assign y to max  
    if ( z > max ) // if z is larger,  
        max = z; // assign z to max  
    return max; // max is largest value  
} // end function maximum
```



Function **maximum** takes 3 arguments (all **double**) and returns a **double**.

```
Enter three real numbers: 99.32 37.3 27.1928  
Maximum is: 99.32
```

```
Enter three real numbers: 1.1 3.333 2.22  
Maximum is: 3.333
```

Function Prototypes

- Function prototype contains
 - Function name
 - Parameters (number and data type)
 - Return type (**void** if returns nothing)
 - Only needed if function definition after function call
- Prototype must match function definition
 - Function prototype

```
double maximum( double, double, double );
```
 - Definition

```
double maximum( double x, double y, double z )  
{  
    ...  
}
```

void Function takes arguments

If the Function does not RETURN result, it is called void Function

```
#include<iostream.h>
void add2Nums(int,int);
main()
{
    int a, b;
    cout<<"enter tow Number:";
    cin >>a >> b;
    add2Nums(a, b)
    return 0;
}
void add2Nums(int x, int y)
{
    cout<< x<< "+" << y << "=" << x+y;
}
```


void Function take no arguments

If the function Does Not Take Arguments specify this with EMPTY-LIST OR write void inside

```
#include<iostream.h>
```

```
void funA();
```

```
void funB(void)
```

```
main()
```

```
{
```

```
    funA();
```

```
    funB();
```

```
    return 0;
```

```
}
```

```
void funA()
```

```
{
```

```
    cout << "Function-A takes no arguments\n";
```

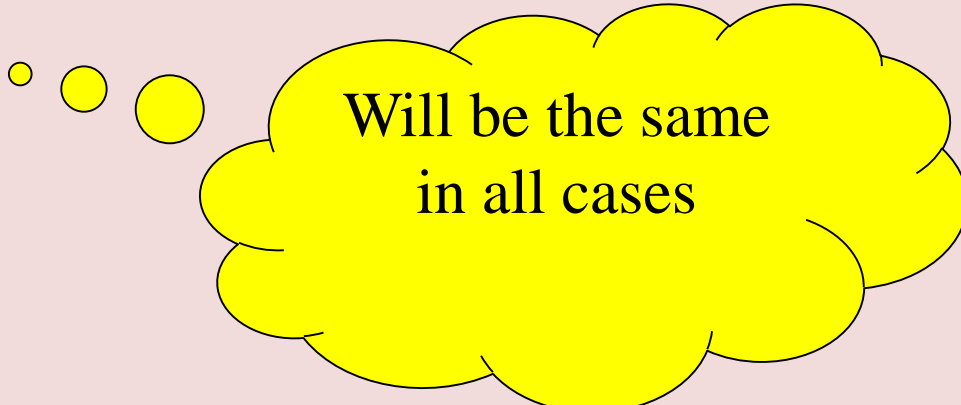
```
}
```

```
void funB()
```

```
{
```

```
    cout << "Also Function-B takes No arguments\n";
```

```
}
```



Will be the same
in all cases

Remember

- Local variables
 - Known only in the function in which they are defined
 - All variables declared inside a function are local variables
- Parameters
 - Local variables passed to function when called (passing-parameters)
- Variables defined outside and before function **main**:
 - Called global variables
 - Can be accessible and used anywhere in the entire program

Remember

- Omitting the type of returned result defaults to **int**, but omitting a non-integer type is a Syntax Error
- If a Global variable defined again as a local variable in a function, then the **Local-definition overrides** the Global defining
- Function prototype, function definition, and function call must be consistent in:
 - 1- Number of arguments
 - 2- Type of those arguments
 - 3-Order of those arguments

Local vs Global Variables

```
#include<iostream.h>
int x,y; //Global Variables
int add2(int, int); //prototype
main()
{
    int s;
    x = 11;
    y = 22;
    cout << "global x=" << x << endl;
    cout << "Global y=" << y << endl;
    s = add2(x, y);
    cout << x << "+" << y << "=" << s;
    cout<<endl;
    cout<<"\n---end of output---\n";
    return 0;
}
int add2(int x1,int y1)
{
    int x; //local variables
    x=44;
    cout << "\nLocal x=" << x << endl;
    return x1+y1;
}
```

global x=11
global y=22
Local x=44
11+22=33
---end of output---

Finding Errors in Function Code

```
int sum(int x, int y)
{
    int result;
    result = x+y;
}
```

→this function must return an integer value as indicated in the header definition (`return result;`) should be added

```
int sum (int n)
{ if (n==0)
    return 0;
  else
    n+sum(n-1) ;
}
```

→the result of `n+sum(n-1)` is not returned; sum returns an improper result, the else part should be written as:-

```
else return n+sum(n-1) ;
```

Finding Errors in Function Code

```
void f(float a);  
{  
    float a;  
    cout<<a<<endl;  
}
```

→ ; found after function definition header.

→ redefining the parameter **a** in the function

```
void f(float a)  
{  
    float a2 = a + 8.9;  
    cout <<a2<<endl;  
}
```

Finding Errors in Function Code

```
void product(void)
{
    int a, b, c, result;
    cout << "enter three integers:";
    cin >> a >> b >> c;
    result = a*b*c;
    cout << "Result is" << result;
    return result;
}
```

- According to the definition it should not return a value , but in the block (body) it did & this is WRONG.
- → Remove `return Result;`

Random Number Generator

- **rand** function generates an integer between 0 and RAND-MAX(~32767) a symbolic constant defined in **<stdlib.h>**
- You may use modulus operator (%) to generate numbers within a specifically range with **rand**.

```
//generate 10 random numbers open-range
```

```
int x;  
for( int i=0; i<=10; i++){  
    x=rand();  
    cout<<x<<" ";  
}
```

```
//generate 10 integers between 0.....49
```

```
int x;  
for( int i=0; i<10; i++){  
    x=rand()%50;  
    cout<<x<<" ";  
}
```


Random Number Generator

```
//generate 10 integers between 5...15
int x;
for ( int i=1; i<=10; i++){
    x= rand()%11 + 5;
    cout<<x<<" ";
}
```

```
//generate 100 number as simulation of rolling a
dice
int x;
for (int i=1; i<=100; i++){
    x= rand%6 + 1;
    cout<<x<<" ";
}
```

Random Number Generator

- the `rand()` function will generate the same set of random numbers each time you run the program .
- To force NEW set of random numbers with each new run use the **randomizing** process
- Randomizing is accomplished with the standard library function `srand(unsigned integer)` ; which needs a header file `<stdlib.h>`

Explanation of **signed** and **unsigned** integers:

- **int** is stored in at least two-bytes of memory and can have positive & negative values 32767 to -32768
- **unsigned int** also stored in at least two-bytes of memory but it can have only positive values 0.....65535

Randomizing with srand

```
#include<iostream.h>
#include<iomanip.h>
#include<stdlib.h>
int main()
{
    int i;
    unsigned num;
    // we will enter a different number each time we run
    cin>>num;
    srand(num);
    for(i=1; i<=5; i++)
        cout<<setw(10)<< 1+rand()%6;
    return 0;
}
```

Output for Multiple Runs

19→	6	1	1	4	2	1
18→	6	1	5	1	4	4
3→	1	2	5	6	2	4
0→	1	5	5	3	5	5
3→	1	2	5	6	3	4

Different-set of Random
numbers



without srand

```
#include<iostream.h>
#include<iomanip.h>
#include<stdlib.h>
int main()
{
    int i;

    for(i=1; i<=5; i++)
        cout<<setw(10)<< 1+rand()%6;
    return 0;
}
```

Output for Multiple Runs

5	3	3	5	4	2
5	3	3	5	4	2
5	3	3	5	4	2
5	3	3	5	4	2

Same set of numbers for
each run

