Functions

Course Code: CSC1102 &1103 Course Title: Introduction to Programming

Dept. of Computer Science
Faculty of Science and Technology

Lecturer No:	**8**	**Week No:**	**6 (2X1.5 hrs)**	**Semester:**	
Lecturer:	*Name & email*				

Lectures 8: Outline

- **Functions**
- Defining a Function
  - Arguments and Local Variables
    - Automatic Local Variables
  - Returning Function Results
  - Declaring a Function Prototype
  - Functions and Arrays
    - Arrays as parameters
    - Sorting Arrays
    - Multidimensional Arrays
  - Global Variables
  - Automatic and Static Variables
  - Recursive Functions

What is a function

- A function is a self-contained unit of program code designed to accomplish a particular task.
- The concept has some equivalent in all high-level programming languages: *functions*, *subroutines*, and *procedures*
- The use of a function: a "black box"
  - defined in terms of the information that goes in (its input) and the value or action it produces (its output).
  - what goes on inside the black box is not your concern, unless you are the one who has to write the function.
  - Think on how you used functions printf, scanf, getchar !
- What kind of “output” comes out from a function black box ?
  - Some functions find a **value** for a program to use. Example: getchar() returns to the program the next character from the standard input buffer.
  - Some functions cause an **action** to take place. Example: printf() causes data to be printed on the screen
  - In general, a function can both produce actions and provide values.

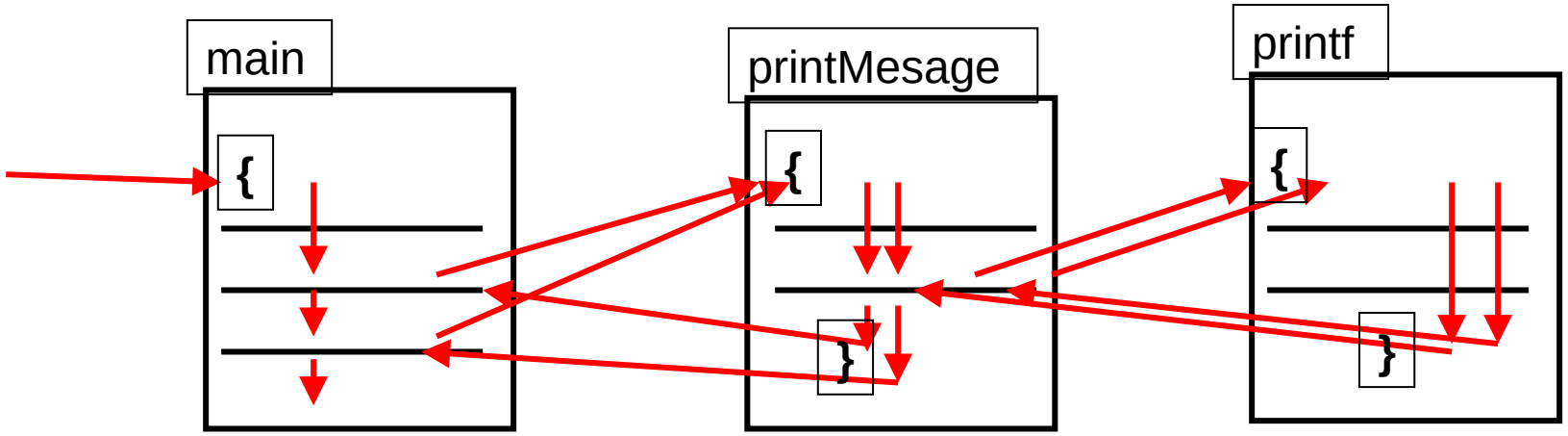
Defining a function

```
#include <iostream>
using namespace std;

void printMessage (void)
{
        cout<<"Programming is fun"<<endl;
}

int main (void)
{
        printMessage ();

        printMessage ();
        return 0;
}
```

Function Definition
-occurs ONE time for all
-outside other functions

Function calls (invocations)
-occurs ANY (0-N) times
-statement inside (other) functions body

Transfer of control flow



When a function call is executed, program execution is transferred directly to the indicated function. After the called routine is finished (as signaled by the closing brace) the program *returns* to the calling routine, where program execution continues at the point where the function call was executed.

Function definitions

General form of function definition:

return-type function-name(*argument declarations*)
{
    declarations and statements
}

return-type *arguments*

```
void printMessage ( void )
{
        cout<<"Programming is fun";
}
```

Function prototype

- The first line of the function definition
- Contains everything that others (other functions) need to know about the function in order to use it (call it)

- `void printMessage (void)`
- `void calculateTriangularNumber (int n)`

Function prototype

return-type function-name(*argument declarations*)
{
    declarations and statements
}

Function arguments

- arguments (parameters): a kind of input for the function blackbox
- **In the function definition**: ***formal arguments*** **(formal parameters)**
  - Formal parameter: a name that is used inside the function body to refer to its argument
- **In the function call**: ***actual arguments*** **(actual parameters)**
  - The actual arguments are values are assigned to the corresponding formal parameters.
  - The actual argument can be a constant, a variable, or an even more elaborate expression.
  - The actual argument is evaluated, and its **value is copied** to the corresponding formal parameter for the function.
    - Because the called function works with data copied from the calling function, the original data in the calling function is protected from whatever manipulations the called function applies to the copies.

Example: arguments

```
// Function to calculate the nth triangular number
#include <iostream>
using namespace std;
void calculateTriangularNumber ( int n )
{
int i, triangularNumber = 0;
for ( i = 1; i <= n; ++i )
       triangularNumber += i;
cout<<"Triangular number "<<n <<" is " <<triangularNumber<<endl;
}
int main (void)
{
calculateTriangularNumber (10);
calculateTriangularNumber (20);
calculateTriangularNumber (50);
return 0;
}
```

formal argument

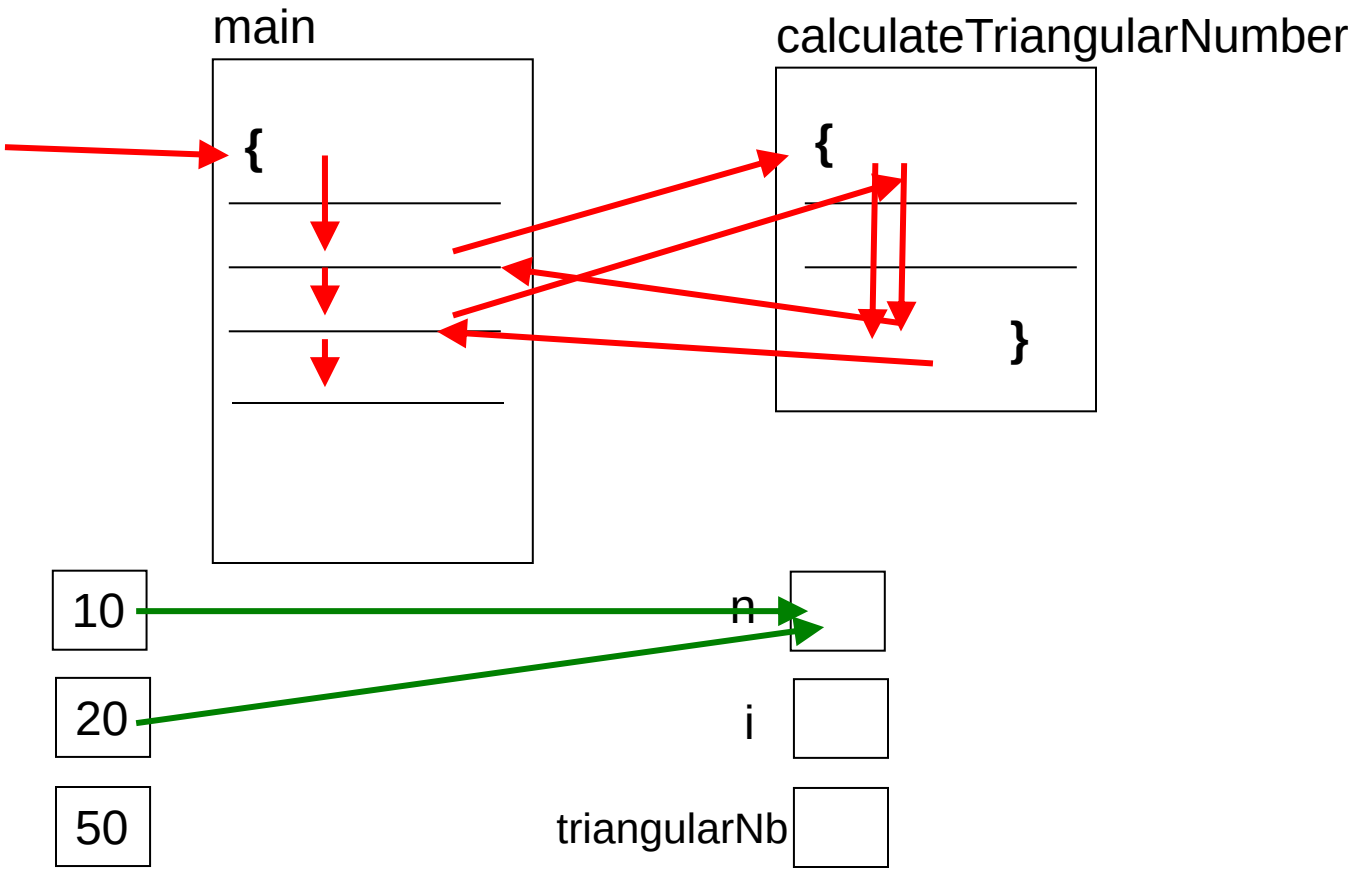
local variables

actual argument

Arguments and local variables

- Variables defined inside a function: *automatic local* variables
  - they are automatically "created" each time the function is called
  - their values are local to the function:
    - The value of a local variable can only be accessed by the function in which the variable is defined
    - Its value cannot be accessed by any other function.
    - If an initial value is given to a variable inside a function, that initial value is assigned to the variable *each* time the function is called.
- Formal parameters: behave like local variables, private to the function.
- ***Lifetime***: Period of time when memory location is allocated
- ***Scope*****:** Region of program text where declaration is visible
- Scope: local variables and formal parameters => only in the body of the function
  - Local variable i in function calculateTriangularNumber is different from a variable i defined in another function (including main)
  - Formal parameter n in function calculateTriangularNumber is different from a variable n defined in another function

Automatic local variables



Example: scope of local variables

```
#include <iostream>
using namespace std;
void f1 (float x)  {
       int n=6;
       cout<< x+n<<endl;
}
int f2(void) {
       float n=10;
       cout<<n<<endl;
}
int main (void)
{
    int n=5;
    f1(3);
    f2();
    return 0;
}
```

Arguments are passed by copying values !

- In a function call, the actual argument is evaluated, and its **value is copied** to the corresponding formal parameter for the function.
  - Because the called function works with data copied from the calling function, the original data in the calling function is protected from whatever manipulations the called function applies to the copies

Example: arguments

```
#include <iostream>
using namespace std;
void gcd (int u, int v){
    int temp;
    while ( v != 0 ) {
        temp = u % v;
        u = v;
        v = temp;
        }
        cout<<u<<endl;
}

int main (void){
    gcd (150, 35);
    gcd (1026, 405);
    gcd (83, 240);
    return 0;
}
```

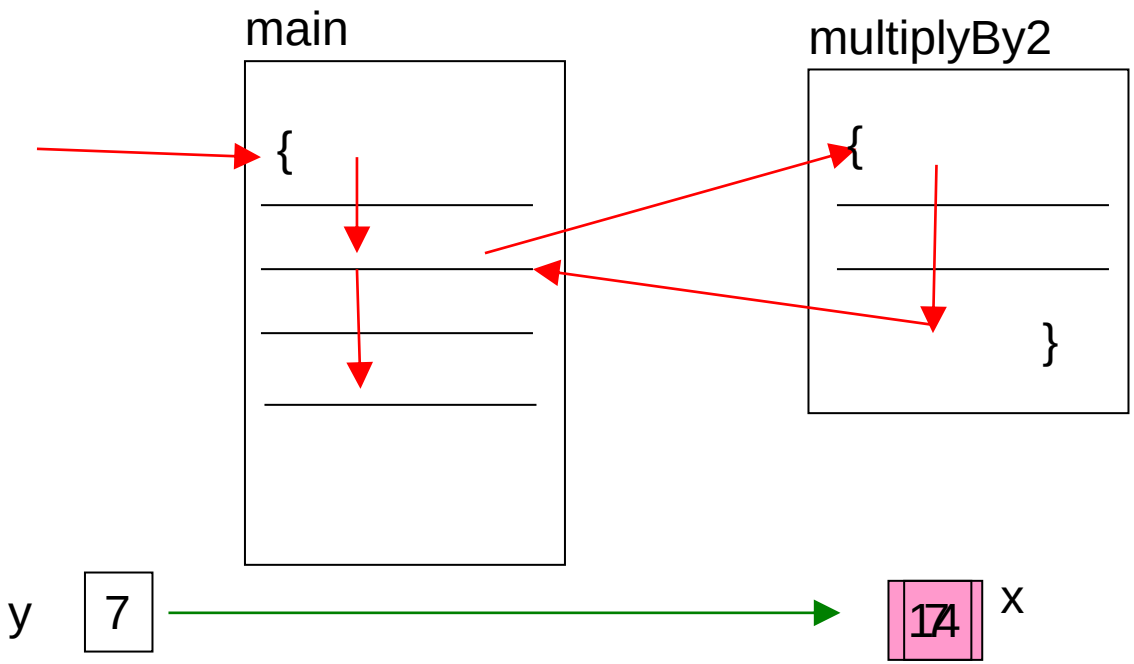
Example: arguments are passed by copying values !

```
#include <iostream>
using namespace std;
void gcd (int u, int v){
int temp;
while ( v != 0 ) {
temp = u % v;
u = v;
v = temp;
        }
    cout<<u<<endl;
}
int main (void)
{
int x=10,y=15;
gcd (x, y);
cout<<"x= "<<x<< "y= "<<y;
return 0;
}
```

The formal parameters u and v are assigned new values in the function

The actual parameters x and y are not changed !

Arguments by copying



Returning function results

- A function in C can optionally return a single value
- return *expression*;
- The value of *expression* is returned to the calling function. If the type of *expression* does not agree with the return type declared in the function declaration, its value is automatically converted to the declared type before it is returned.
- A simpler format for declaring the return statement is as follows:
- return;
- Execution of the simple return statement causes program execution to be immediately returned to the calling function. This format can only be used to return from a function that does not return a value.
- If execution proceeds to the end of a function and a return statement is not encountered, it returns as if a return statement of this form had been executed. Therefore, in such a case, no value is returned.
- If the declaration of the type returned by a function is omitted, the compiler assumes that the function returns an int !

Example: function result

```
/* Function to find the greatest common divisor of two nonnegative
integer values and to return the result */

#include <iostream>
using namespace std;
int gcd (int u, int v)
{
    int temp;
    while ( v != 0 ) {
        temp = u % v;
        u = v;
        v = temp;
    }
    return u;
}
int main (void)
{
    int result;
    result = gcd (150, 35);
    cout<<"The gcd of 150 and 35 is "<< result<<endl;
    result = gcd (1026, 405);
    cout<<"The gcd of 1026 and 405 is "<< result<<endl;
    cout<<"The gcd of 83 and 240 is "<< gcd (83, 240)<<endl;
    return 0;
}
```

Function declaration

- a function prototype—a declaration that states the return type, the number of arguments, and the types of those arguments.
- Useful mechanism when the called function is defined after the calling function
- The ***prototype*** of the called function is everything the compiler needs in order to be able to ***compile*** the calling function
- In order to produce the ***executable program***, of course that also the whole ***definition*** of the function body is needed, but this occurs later, in the process of ***linking***

Examples: function declarations

In a function declaration you have to specify the argument type inside the parentheses, and not its name.
You can optionally specify a “dummy” name for formal parameters after the type if you want.

```
int gcd (int u, int v);
```
Or
```
int gcd (int, int);

void calculateTriangularNumber (int n);
```
Or
```
void calculateTriangularNumber (int);
```

Passing arrays as parameters

- A whole array can be one parameter in a function
- ***In the function declaration, you can then omit the specification of the number of elements contained in the formal parameter array.***
  - The compiler actually ignores this part of the declaration anyway; all the compiler is concerned with is the fact that an array is expected as an argument to the function and not how many elements are in it.
- Example: a function that returns the minimum value from an array given as parameter
  - `int minimum (int values[10]);`
    - We must modify the function definition if a different array size is needed !
  - `int minimum (int values[]);`
    - Syntactically OK, but how will the function know the actual size of the array ?!
  - `int minimum (int values[], int numberOfElements);`

Example: Passing arrays as parameters

```
// Function to find the minimum value in an array
#include <iostream>
using namespace std;
int minimum (int values[], int numberOfElements)
{
    int minValue, i;
    minValue = values[0];
    for ( i = 1; i < numberOfElements; ++i )
       if ( values[i] < minValue )
         minValue = values[i];
    return minValue;
}
int main (void)
{
    int array1[5] = { 157, -28, -37, 26, 10 };
    int array2[7] = { 12, 45, 1, 10, 5, 3, 22 };
    cout<<"array1 minimum: "<< minimum (array1, 5)<<endl;
    cout<<"array2 minimum: "<<minimum (array2, 7)<<endl;
    return 0;
}
```

Array parameters are passed by reference !

- Parameters of non-array type: passed by copying values
- ***Parameters of array type: passed by reference***
  - the entire contents of the array is *not* copied into the formal parameter array.
  - the function gets passed information describing *where* in the computer's memory the original array is located.
  - *Any changes made to the formal parameter array by the function are actually made to the original array passed to the function, and not to a copy of the array*.
  - *This change remains in effect even after the function has completed execution and has returned to the calling routine.*

Example: Array parameters are passed by reference !

```
#include <iostream>
using namespace std;
void multiplyBy2 (float array[], int n)
{
int i;
for ( i = 0; i < n; ++i )
       array[i] *= 2;
}
int main (void)
{
float floatVals[4] = { 1.2f, -3.7f, 6.2f, 8.55f };
int i;
multiplyBy2 (floatVals, 4);
for ( i = 0; i < 4; ++i )
       cout<< floatVals[i];
return 0;
}
```

Global variables

- A ***global variable*** declaration is made *outside* of any function.
- It does not belong to any particular function. *Any* function in the program can then access the value of that variable and can change its value.
- The primary use of global variables is in programs in which many functions must access the value of the same variable. Rather than having to pass the value of the variable to each individual function as an argument, the function can explicitly reference the variable instead.
- There is a **drawback** with this approach: Because the function explicitly references a particular global variable, the generality of the function is somewhat reduced !
- Global variables do have **default initial values: zero**

Example: global variables

```
#include <stdio.h>

int x;

void f1 (void) {
        x++;
}

void f2 (void) {
        x++;
}

int main(void) {
        x=7;
        f1();
        f2();
        printf("x=%i \n",x);
}
```

Automatic and static variables

- Automatic local variables (the default case of local vars) :
  - an automatic variable disappears after the function where it is defined completes execution, the value of that variable disappears along with it.
  - the value an automatic variable has when a function finishes execution is *guaranteed* not to exist the next time the function is called.
  - The value of the expression is calculated and assigned to the automatic local variable *each* time the function is called.

- Static local variables:
  - If you place the word `static` in front of a variable declaration
  - "something that has no movement"
  - a static local variable—it does *not* come and go as the function is called and returns. This implies that the value a static variable has upon leaving a function is the same value that variable will have the next time the function is called.
  - Static variables also differ with respect to their initialization. A static, local variable is initialized only *once* at the start of overall program execution—and not each time that the function is called. Furthermore, the initial value specified for a static variable *must* be a simple constant or constant expression. Static variables also have default initial values of zero, unlike automatic variables, which have no default initial value.

Example: Automatic and static variables

```
// Program to illustrate static and automatic variables
#include <stdio.h>
void auto_static (void)
{
    int autoVar = 1;
    static int staticVar = 1;
    printf ("automatic = %i, static = %i\n", autoVar, staticVar);
    ++autoVar;
    ++staticVar;
}
int main (void)
{
    int i;
    for ( i = 0; i < 5; ++i )
        auto_static ();
    return 0;
}
```