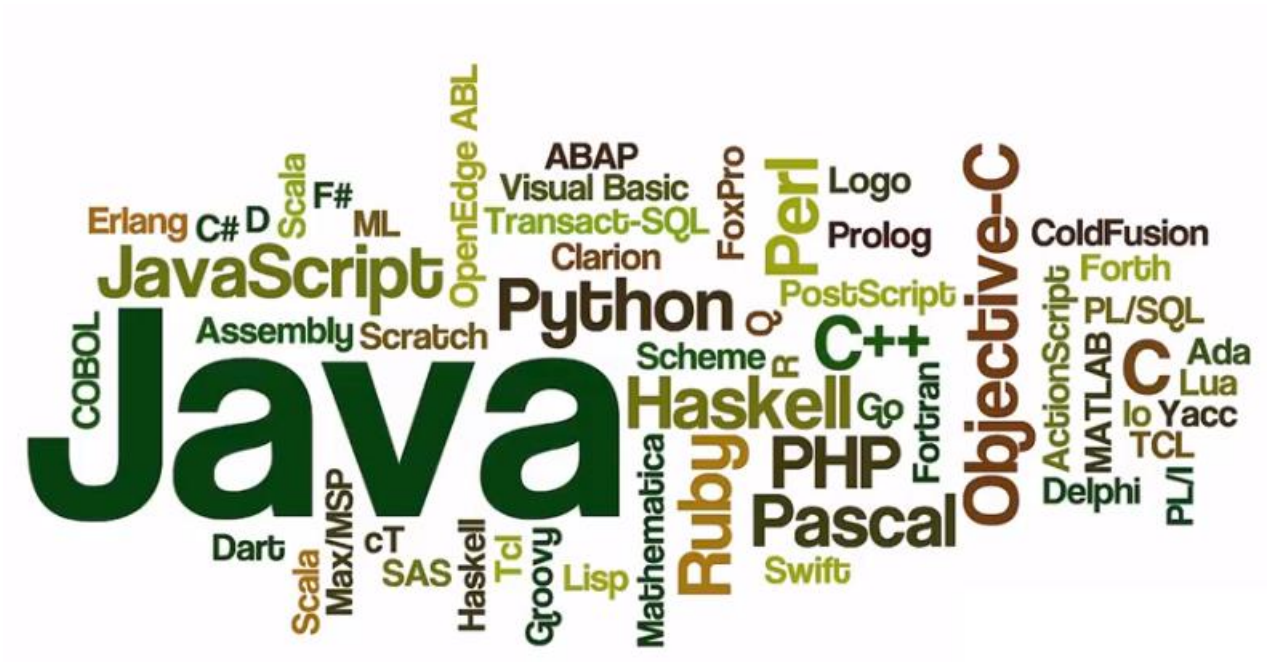
College of Computer Science and information Technology

Computer Programming I



Lecturer: Dr. Raidah Salim

Course Information	
Course Title	Computer Programming I
Credits	4 Hours
Teaching Method	3 Hour of Lecture + 2 Hours Lab

Learning Outcomes

Course Description: This course covers fundamentals of algorithms, flowcharts, problem solving, programming concept, control structures and functions.

Course Outcomes: At the end of this course, students should be able to :

- Develop algorithms to solve "computer-solvable" problems.
- Test algorithms.
- Translate algorithms to C++ programs.
- Debug, run and test C++ "procedural" programs.

Topics

- Problem solving
- Algorithms
- What is programming?
- Basic elements of C++
- General Form of a C++ Program
- Comments and Reserved Words
- Identifiers , Variables and constant
- Data Types
- Arithmetic Operators and Operator Precedence
- Expressions
- Assignment Statement
- Declaring and Initializing Variables
- Input and output
- Control Structures
- Relational Operators and precedence
- Selection: if and if...else
- Compound (Block of) Statements
- Multiple Selections: Nested if
- Selection: Switch case
- Repetition: for Looping Structure
- User-defined functions
- Function declarations and call
    - Scope rule of an Identifier

Textbook

1. Problem solving with c++ by Walter Savitch, 7th edition,2009.
2. C++: The Complete Reference by Herbert Schildt, 4th edition, 2003.

Reference

1. A first book of c++ by Gary Bronson, 4th edition, 2012 by Gary Bronson

Functions

A program can be thought of as consisting of subparts, such as obtaining the input data, calculating the output data, and displaying the output data. These subparts are called ***functions.***

Functions come in two types :

1- **Predefined functions (or built in functions) :** They can be defined by the built in as part of the compiler package .
2- **User –defined functions:** They can be defined by the user.

Top –Down Design

The top-down design strategy is an effective way to design an algorithm for a program. In this strategy you divide the program's task into subtasks and then implement the algorithms for these subtasks as *functions*.

- top-down design would make the program easier to understand, easier to change if need be, and as will become apparent, easier to write, test, and debug.
- One of the advantages of using functions to divide a programming task into subtasks is that different people can work on the different subtasks.

- C++, like most programming languages, has facilities to include separate subparts inside of a program. In other programming languages these subparts are called ***subprograms***, ***procedures***, or ***methods,*** while in C++ these subparts are called **functions.**

Predefined Functions (or built-in functions)

The Built-in functions are declared in **header files** using the form:

#include <directive which contain predefined function *file*>

e.g. for **common mathematical calculations** we include the file math with the following statement:

#include <math.h> //directive which contains the *function prototypes* for the mathematical functions in the math library.

Mathematical functions

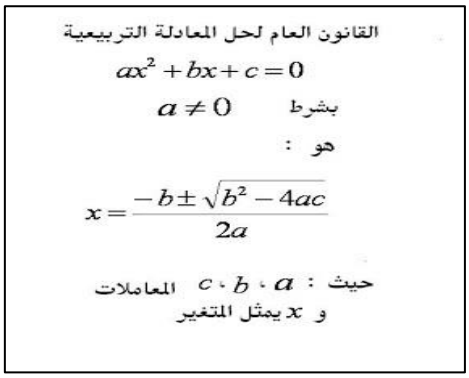
Math library functions allow the programmer to perform a number of common mathematical calculations:

Function	Description
sqrt(x)	square root
sin(x)	trigonometric sine of x (in radians)
cos(x)	trigonometric cosine of x (in radians)
tan(x)	trigonometric tangent of x (in radians)
exp(x)	exponential function
log(x)	natural logarithm of x (base e)
log10(x)	logarithm of x to base 10
abs(x)	absolute value (unsigned)
ceil(x)	rounds x up to nearest integer
floor(x)	rounds x down to nearest integer
pow(x,y)	x raised to power y

Example 1: By using predefined function, Write program that find x values from the equation :

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

where:



Example 2 : write program that find and print z value as fallow :

$$z = \begin{cases} sin(a) + cos(b) & x = 1 \\ \sqrt{a} - e^b + 2 & x = 2 \\ a^b & x = 3 \\ |a| + |b| & x = 4 \end{cases}$$

User- defined functions

A function is a group of statements that together perform a task. Every C++ program has at least one function, which is main.

To understand functions we must understand the following terms

1- **Function definitions**
2- **Function Declarations**
3- **Calling a Function**
4- **Function Arguments**
5- **Default Values for Parameters**

Function definitions

There are two type of *user-defined functions:*

- *First type:* **Return one value by its name function**

Uses: *use when we need a function that calculate and return just ONE value*

[there are no input (cin) or output (cout) statements]

Define:

type of data that the function returns

Identifier of function name

List of formal parameters names and their types that receive the values of the arguments

```
return-type function-name(parameter list)
   {
        // body of the function
     return data;
  }
```

Note: : The return _type of the function may be (int, float, char, etc.)

Example : int function-name (*parameters-list*)

- Second type: **Not return any value by its name (Void Function)**

Uses: *use when we need a function that calculate and return more than ONE value or do subtask, we can here use input (cin) or output (cout) statements.*

Define:

Identifier of function name

List of formal parameters names and their types that receive the values of the arguments

```
void function-name(parameter list)
   {
        // body of the function
   }
```

Example: void function-name (parameters-list)

Example : Following is the source code for a function called **max()**. This function takes two parameters num1 and num2 and returns the maximum between the two:

```
// function returning the max between two numbers
int max(int num1, int num2)
 {
  // local variable declaration
  int result;
  if (num1 > num2)
    result = num1;
  else
    result = num2;
  return result;
}
```

Function Declarations

A function **declaration** tells the compiler about a function name and how to call the function. The actual body of the function can be defined separately.

A function declaration has the following parts:

```
return_type   function_name( parameter list );
or
void    function_name( parameter list );
```

For the above defined function max(), following is the function declaration:

```
int  max(int num1, int num2);
```

Notes :

- Parameter names are not important in function declaration only their type is required, so following is also valid declaration:

```
int max(int, int);
```

- Function declaration is required when you define a function in one source file and you call that function in another file or you define function after main function. In such case, you should declare the function at the top of the file calling the function.

- There are two ways to write the declaration and definition of function depending on their place in program

```
# include  <iostream.h>
  //Function declaration  ;
 void main()
  {
    ....
   //  function call
  }
//Function definition
```

```
# include  <iostream.h>
 // Function definition
 void main()
  {
    ....
   //function call
  }
```

Calling a Function

While creating a C++ function, you give a definition of what the function has to do. To use a function, you will have to call or invoke that function.

- When a program calls a function, program control is transferred to the called function.
- A called function performs defined task and when its return statement is executed or when its function-ending closing brace is reached, it returns program control back to the main program.
- To call a function, you simply need to pass the required parameters along with function name, and if function returns a value, then you can store returned value. For example:

Using way1

```
#include <iostream>
    int max(int , int  );
 int main ()
 {
   // local variable declaration:
   int  a = 100;
   int b = 200;
   int  ret;
  // calling a function to get max value.
  ret = max(a, b);
  cout << "Max value is : " << ret << endl;
  return 0;
}
int  max(int  num1, int  num2)
 {
   // local variable declaration
  int result;
  if (num1 > num2)
    result = num1;
  else
    result = num2;
  return result;
}
```

Using Way 2

```
#include <iostream>
int  max(int  num 1, int  num2 )
 {
   // local variable declaration
  int result;
  if (num1 > num2)
    result = num1;
  else
    result = num2;
  return result;
}
 int main ()
 {
   // local variable declaration:
   int  a = 100;
   int b = 200;
   int  ret;
  // calling a function to get max value.
  ret = max(a, b);
  cout << "Max value is : " << ret << endl;
  return 0;
}
```

After running the source code it would produce the following result:

```
Max value is : 200
```

Function Arguments

If a function is to use arguments, it must declare variables that accept the values of the arguments. These variables are called the ***formal parameters*** of the function.

The formal parameters behave like other local variables inside the function and are created upon entry into the function and destroyed upon exit.

While calling a function, there are two ways that arguments can be passed to a function:

Call Type	Description
Call by value ***(IN-ONLY)***	This method copies the actual value of an argument into the formal parameter of the function. In this case, changes made to the parameter inside the function have **no effect** on the argument.
Call by reference ***(IN-OUT)***	This method copies the reference of an argument into the formal parameter. Inside the function, the reference is used to access the actual argument used in the call. This means that changes made to the parameter affect the argument.

Note: By default, C++ uses **call by value** to pass arguments. In general, this means that code within a function **cannot change** the arguments used to call the function .

- A function definition has a name, parentheses pair containing zero or more parameters and a body.
- For each parameter, there should be a corresponding declaration that occurs before the body. Any parameter not declared is taken to be an integer by default.

The Return one value by its name function:

Parameters type

- (call by value): copies the value of argument (actual parameter) into formal parameter of the function. In this case, changes made to the parameter have no effect on the arguments.

Call :

- We can write the function name with its arguments in any place can put a variable of the same type.
- Syntax:

```
Function_name(argument 1, argument 2,..., argument n);
```

Identifier or function name **actual parameters**

Return Statement :

- The keyword return is used to terminate function and return a value to its caller.
- The return may also be used to exit a function without returning a value.
- Its may or may not include on expression.
- Its general syntax is:

return ;

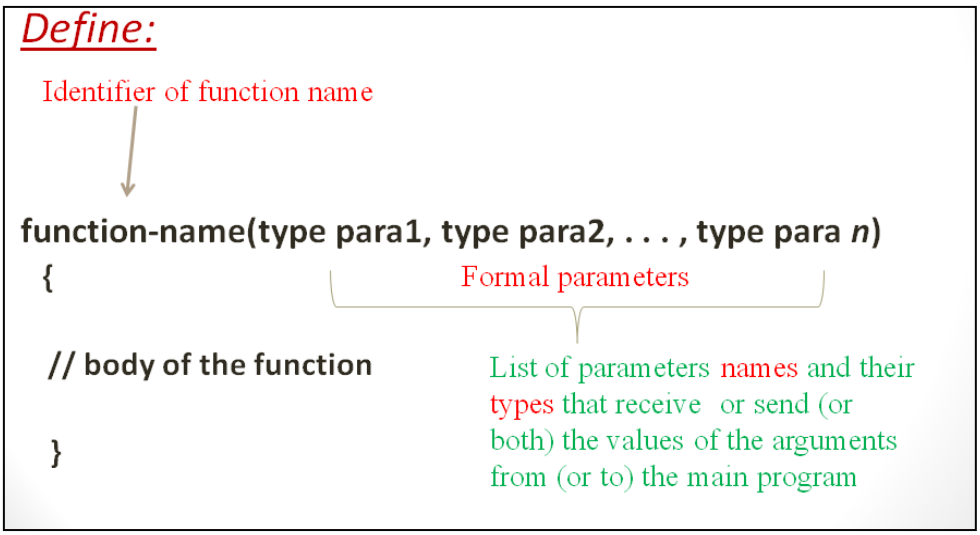
return (exp);

The return statements terminate the exaction of the function and pass the control back to the calling environment.

Call Examples :

- **Assignment statement :** a = sum (n, m);
- **If statement :** if (sum (n, m) >=4)
- **Output statement:** cout << sum (n, m);

➢ **Not return by its name (Void Function) :** It uses when we need a function that calculate and return more than one values or do special subtask



Parameters type:

1. **call by value (IN-only):** copies the value of argument (actual parameter) into formal parameter of the function. In this case, changes made to the parameter have no effect on the arguments.

2. **call by reference(INO-out):** the address of an argument is copied into the parameter. Inside the function, the address is used to access the actual argument used in the call. This means that changes made to the parameter affect the argument.

Call :

- We can write the function name with its arguments in any place can put a variable of the same type.

- Function_name(argument 1, argument 2,..., argument n);

Exercies:

1. **Write C++ program that use function named powfun() that raises an integer number passed to it to a positive integer power and returns the result as an integer.**
2. **Write C++ program can swap between two variables using function:**
3. **Write C++ program using a functions to read sequence of positive number, then print each number with it factorial.**
4. **Write C++ program using a functions to read n of integer number, then print the summation and average of even and odd numbers.**
5. **Write C++ program using function to calculate the average of two numbers entered by the user .**
6. **Write C++ program to calculate the squared value of a number passed from main program. calculate the square of numbers from 1 to 10.**

Functions Questions

1. Design the findMax() function accepts two double arguments (number1 and number2) and return the max number.

2. Design a function named findAbs() that accepts a double number passed to it and returns that number's absolute value.

3. Design a function named mult() that accepts two floating-point numbers as parameters, multiplies these two numbers, and returns the result.

4. Design a function named square() that computes and returns the square of the integer value passed to it.

5. Design A function named powfun() that raises an integer number passed to it to a positive integer power and returns the result as an integer.

6. Design a function named table() that produces a table of numbers from 1 to 10, their squares, and their cubes.

7. Design a function named check() that accept two integer numbers as parameters, the function return "ok" if the second number is factorial of first number otherwise the function return "not ok"

8. Design a function to read sequence of number (N), then print the times repeat of a specific number.

9. Design a function to read sequence of number, then print each number with it factorial.

10. Design a function to generate N term of fibo series:

1 1 2 3 5 8 13 .....

Solve All Previous Questions and Examples by using Function Concept.

Default Values for Parameters

When you define a function, you can specify a default value for each of the last parameters. This value will be used if the corresponding argument is left blank when calling to the function.

This is done by using the assignment operator and assigning values for the arguments in the function definition.

If a value for that parameter is not passed when the function is called, the default given value is used, but if a value is specified, this default value is ignored and the passed value is used instead.

Consider the following example:

```
#include <iostream.h>
int sum(int a, int b=20) {
   int  result;
   result = a + b;
   return (result); }
int main () {
  // local variable declaration:
  int a = 100;
  int b = 200;
  int result;

  // calling a function to add the values.
  result = sum(a, b);
  cout << "Total value is :" << result << endl;

  // calling a function again as follows.
  result = sum(a);
  cout << "Total value is :" << result << endl;

  return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
Total value is :300
Total value is :120
```

Scope rule of an Identifier

scope	(of an identifier) the range of program statements within which the identifier is recognized as a valid name

C++ Scope Rules

1. Every identifier must be declared and given a type before it is referenced (used).
2. The scope of an identifier begins at its declaration.
3. If the declaration is within a compound statement, the scope of the identifier ends at the end of that compound statement. We say the identifier is local to that compound statement (or block).
4. If the declaration is not within a compound statement, the scope of the identifier ends at the end of the file. We say the identifier has global scope.

Variables definition in the Functions

There are three types of variables definition:

1. **Local variable**: every variable defined in any function is called local for this function, so it is valid just in this function but not valid in the other functions.
2. **Global variable:** every variable defined in the beginning of the program before main and any function is called global. it is valid in all the program.
3. **Not-Local variable:** every variable defined outside the functions and before some functions is called Not-local. This variable is valid in all functions written after this variable definition But not valid in all functions written before.

```
# include <iostream.h>
 int v;          // v is global variable we can use v in all functions and main

void sum ( int m, int n )
{ int x;         //x is local variable we can use x in the function sum only
.....}

                 /*k is NOT local but Not global variable we can use k in
 int k;            all functions down this definition with also main */

void test (int m, int n )
  { ... }

void main ()
{ int y;         // y is local use only in the function main
}
```

```
#include <iostream.h>
 int k=10;
 void a ( )
   { int s=6;
     cout << " S = "<< s <<" ///// k = "<< k<<endl;
    }
 int v=4;
 void b()
   {
     cout << "  V = "<< v<<" ///// k = "<< k<<endl;
    }

 main ()
 {
   a();
   b();
cout << " k=  "<< k<<endl;
 }
```

k is Global Variable

S is Local Variable

v is not (Global or Local Variables)

```
S =   6 //// k =   10
V =   4 //// k =   10
  k   = 10
```

```
#include <iostream.h>

void F(double c);
const int a = 17;
int b;
int c;

int main( ) {
    int  b;
    char c;
    b = 4;
    c = 'x';
    F(42.8);

    return 0;
}
void F(double c) {
    double b;
    b = 3.2;
    cout << "a = " << a;
    cout << "b = " << b;
    cout << "c = " << c;
    int a;
    a = 42;
    cout << "a = " << a;
}
```

what is the scope of each declared identifier?

Choosing a Parameter Passing Mechanism

Pass-by-Reference
- use **only if** the design of the called function requires that it be able to modify the value of the parameter

Pass-by-Constant-Reference
- use if the called function has no need to modify the value of the parameter, but the parameter is very large (e.g., a string or a structure or an array, as discussed later)
- use as a **safety net** to guarantee that the called function cannot be written in a way that would modify the value passed in†

Pass-by-Value
- use in all cases where none of the reasons given above apply
- pass-by-value is safer than pass-by-reference

† Note that if a parameter is passed by value, the called function may make changes to that value as the formal parameter is used within the function body. Passing by constant reference guarantees that even that sort of internal modification cannot occur.

Parameters Pass

Pass-by-Value
- default passing mechanism except for one special case discussed later
- allocate a temporary memory location for each formal parameter (when function is called)
- copy the value of the corresponding actual parameter into that location
- called function has no access to the actual parameter, just to a copy of its value

```
. . .
int First  = 15,
    Second = 42;
int Least = FindMinimum(First, Second);
. . .
```

Variable	Value
First	
Second	
Least	

```
int FindMinimum(int A, int B) {
   if (A <= B)
      return A;
   else
      return B;
}
```

Variable	Value
A	
B	

Created when call occurs and destroyed on return.

Pass-by-Reference
- put ampersand (&) after formal parameter type in prototype and definition
- forces the corresponding actual and formal parameters to refer to the same memory location; that is, the formal parameter is then a synonym or alias for the actual parameter
- called function may modify the value of the actual parameter

```
. . .
int First  = 15,
    Second = 42;
SwapEm(First, Second);
. . .
```

Variable	Value
First	
Second	

```
void SwapEm(int& A, int& B) {
   int TempInt;
   TempInt = A;
   A       = B;
   B       = TempInt;
}
```

Variable	Value
A	
B	
TempInt	

Pass-by-Constant-Reference
- precede formal parameter type with keyword `const`, follow it with an ampersand (&)
- forces the corresponding actual and formal parameters to refer to the same primary memory location; just as in pass-by-reference
- **but**, the called function is not allowed to modify the value of the parameter; the compiler flags such a statement as an error

```
. . .
int First  = 15,
    Second = 42,
    Third;
Third = AddEm(First, Second);
. . .
```

Variable	Value
First	
Second	
Third	

```
int AddEm(const int& A, const int& B) {
   int Sum;
   Sum = A + B;
   return Sum;
}
```

Variable	Value
A	
B	
Sum	

Parameters Restrictions

With pass by value, the actual parameter can be an expression (or a variable or a constant):

```
double CalcForce(int Weight, int Height){
    . . .
}
```

```
F = CalcForce(mass * g, h);
```

With pass by reference and pass by constant reference, the actual parameter must be an *l-value*; that is, something to which a value can be assigned.

That rules out expressions and constants.

```
void getRGB(int& Red, int& Green,
                              int& Blue){
    . . .
}
```

Parameter communication Trace

```
. . .
int main( ) {                          //  1
const int W = 100;                     //  2
int  X = 10, Y = 20, Z = 30;           //  3
void Mix(int P, int& Z);               //  4

   Mix ( X, Y );                       //  5
   cout << W << X << Y << Z << endl;   //  6
   Mix ( Z,  X );                      //  7
   cout << W << X << Y << Z << endl;   //  8
   return 0;                           //  9
}

void Mix (int  P,   int& Z ) {         // 10
   int  Y = 0, W = 0;                  // 11

   Y = P;                              // 12
   W = Z;                              // 13
   Z = Z + 10;                         // 14
   cout << P << W << Y << Z << endl;   // 15
 }
```

Memory space for `main()`:

W	
X	
Y	
Z	

Memory space for `Mix()`:

P	
Z	
Y	
W	

Output
