

# Problem Solving With C++

**Mohammad A. Rajabi**

Dept. of Geomatics Eng.

University of Tehran

Cell: 0912 132 5823

Email: [marajabi@ut.ac.ir](mailto:marajabi@ut.ac.ir)

<http://www.marajabi.com>

# Why is a Computer Programming Language Needed?

# Complexity!!!



uses

High level language

Source Code

Compiler

Executable Code

Executed on

Computer

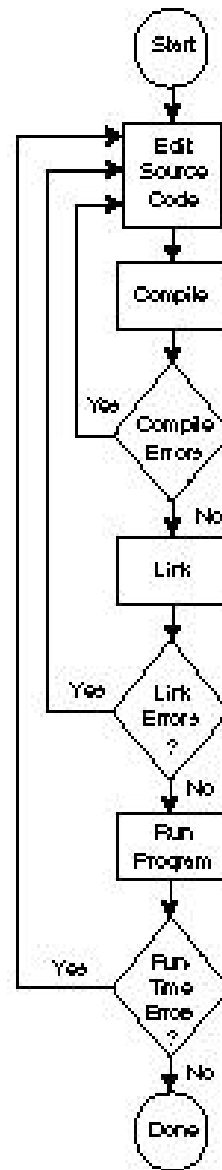


Direct input

Executable Code

Executed on

Computer



# Why C++

- C is a language designed by and for programmers
- C++ is an expanded and enhanced version of C Programming Language
- C++ supports Object Oriented Programming (OOP)
  - Encapsulation (code + data)
  - Polymorphism
  - Inheritance
- It is the language of choice for professional programmers worldwide

# Why C++

- C++ is not the easiest programming language to learn.
- It is, however, the best programming language to learn.

# Why C++

- Once mastered, C++ will give you complete control over the computer
- C++ is, above all, the most powerful programming language ever invented!!!

# How to do well?

- The best way to learn is to try the examples yourself, experimenting with your own variations.
- Have Fun!!!

# Review

```
/* Program #1 - A first C++ program.  
  
   Enter this program, then compile and run it.  
*/  
  
#include <iostream>  
using namespace std;  
  
// main() is where program execution begins.  
int main()  
{  
    cout << "This is my first C++ program.";  
  
    return 0;  
}
```

# Assignment

- Assignment of a value to a variable
- A variable is a named memory location that may be assigned a value
- The content of a variable is changeable, not fixed

# Example of Assignment

```
// Program #2 - Using a variable

#include <iostream>
using namespace std;

int main()
{
    int value; // this declares a variable

    value = 1023; // this assigns 1023 to value

    cout << "This program prints the value: ";
    cout << value; // This displays 1023

    return 0;
}
```

# cin ... console input

```
// This program converts gallons to liters.

#include <iostream>
using namespace std;

int main()
{
    int gallons, liters;

    cout << "Enter number of gallons: ";
    cin >> gallons; // this inputs from the user

    liters = gallons * 4; // convert to liters

    cout << "Liters: " << liters;

    return 0;
}
```

# float ... a new data type

## 3.4E-38 to 3.4E+38

```
/* This program converts gallons to liters using
   floating point numbers. */

#include <iostream>
using namespace std;

int main()
{
    float gallons, liters;

    cout << "Enter number of gallons: ";
    cin >> gallons; // this inputs from the user

    liters = gallons * 3.7854; // convert to liters

    cout << "Liters: " << liters;

    return 0;
}
```

# Data Types

- char, 8 bit, -128 to 127
- wchar, 16 bit, 0 to 65535
- int, 16 bit, -32768 to 32767
- int, 32 bit, -2147483647 to 2147483647
- float, 32 bit, 3.4e-38 to 3.4e38
- double, 64 bit, 1.7e-308 to 1.7e308
- bool, true or false

# ASCII

- American Standard Code for Information Interchange
- Computers really work with numbers, not characters.
- To store characters, we store numbers, but we understand them to represent characters.
- Interpret a bit pattern as an integer if declared as `int` and as a character if declared as `char`.

# ASCII

- Example: An ASCII to decimal converter

```
#include <iostream.h>
int main()
{
    char c;
    int i;
    cout << "Enter a letter: ";
    cin >> c;
    i = c;
    cout << "The character '" << c
<< "' = " << i << "." << endl;
    return 0;
}
```

# ASCII

- Example run:
- Enter a letter: *B*
- The character 'B' = 66.



# A really weird example using char

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

int main()
{
    char letter;

    for(letter = 'Z'; letter >= 'A'; letter--)
        cout << letter;

    return 0;
}
```

# Variable Declaration

- `type variable_list;`
  - `int i, j, k=10;`
- variable name: 1024 long character, case sensitive
- Scope
  - Local
  - Global
  - Formal
- Type casting
  - `(type) variable;`
  - `(int) i;`

# Arithmetic Operators: +, -, \*, /, %

- Addition (+): Adds two numbers
- Subtraction (-): Subtracts two numbers
- Multiplication (\*): Multiplies two numbers
- Division (/): Divides two numbers
- Modulus (%): Remainder of integer division.

Examples:

$$5 \% 2 = 1$$

$$17 \% 7 = 3$$

# Arithmetic Operators: +, -, \*, /, %

- Type of result will be int if all operands are int; otherwise it will be double.

Examples:

$$3 / 2 = 1$$

$$4.2 / 2 = 2.1$$

$$3.0 / 2 = 1.5$$

$$3 / 2.0 = 1.5$$

- `x++` is executed after it is used

- `x = 7;`

- `y = x++; // x = 8, y = 7`

- `++x` is also equivalent to `x = x+1`

- `++x` executed before it is used

- `x = 7;`

- `y = ++x; // x = 8, y = 8`

- `--x` is equivalent to `x = x-1`

- `--` works just like `++`, but with subtraction instead of addition

# Precedence and Associativity

- Order of mathematical operations is important.

Examples:

$$(3+2)*4 = 5*4 = 20$$

$$3+(2*4) = 3 + 8 = 11$$

- \* and / evaluated before + and -

Example:  $3+2*4$  evaluated as  $3+(2*4)$

# Precedence and Associativity

- If precedence is equal, then evaluate from left to right.

Examples:

$$3+2+5 = 5 + 5 = 10$$

$$3*2/5 = 6/5 = 1$$

- Parentheses enforce evaluation order

Example:  $(3+2)*4 = 5*4 = 20$

- $a = ++b*4/3+4/2*5$
- $a = b+++*4/3+4/2*5$

# Unary operators: +, -

- **Unary operators: +, -**
  - -3, +17 allowed
  - Example:  $4 * -3 = -12$

# Modulo operation

- $17 / 5$  evaluates to 3.
- $17 \% 5$  evaluates to 2.
- $-17 / 5$  evaluates to -3.
- $-17 \% 5$  evaluates to -2.
- This differs from the modulo "operation" as mathematicians define it. For mathematicians, the result of a modulo operation is always positive.

# Shortcuts

- `n++`  
equivalent to `n = n + 1`  
read as "add 1 to n"
- `n--`  
equivalent to `n = n - 1`  
read as "subtract 1 from n"
- `s += n`  
equivalent to `s = s + n`  
read as "add n to s"
- `s -= n`  
equivalent to `s = s - n`  
read as "subtract n from s"

# Shortcuts

- In general:
- variable *OP*= expression
- is equivalent to
- variable = variable *OP* expression
- Example
- `x = 4;`
- `x *= 5; // x = 20`

# Boolean expressions

- Logical AND is represented by `&&`
  - `(score >= 0) && (score <= 10)`
  - is true if score is between 0 and 10 (including 0 and 10).
- Logical OR is represented by `||`
  - `(score < 0) || (score > 10)`
  - is true exactly in the cases where the previous condition would be false.

# Boolean expressions

- Logical NOT is represented by an exclamation mark: !
  - `(score < 0) || (score > 10)`
  - is equivalent to
  - `! ( (score >= 0) && (score <= 10) )`
- Pitfall: `&` and `|` are also operators in C++. However, they do something different!
- Pitfall: comparison operator for equality is `==`
  - `x == y`
  - is true if x has the same value as y.
  - `x = y`

Mohammad A. Rajabi Problem Solving With C++ on the other hand, **assigns** the value of y to the variable x.

# Boolean expressions

- **Pitfall:** Be careful with precedences. When in doubt, use parentheses.
- **Pitfall:** Do not use strings of inequalities. They may be legal C++ code but will not do what you (for now) expect them to do.

```
if (x < y < z)    // WRONG!  
    cout << "y is between x and z" <<  
    endl;
```

– Instead, you have to write:

```
if ((x < y) && (y < z)) // CORRECT!  
    cout << "y is between x and z" <<  
    endl;
```

# Boolean expressions

- Given  $x=3$  and  $y=4$ . What's the value of
- $! ( ( (x < y) \ \&\& \ (x > y-2) ) \ \&\& \ (x*2 < y) )$
- What's wrong with these?
  - $(y > x) \ \& \ (z > x)$
  - $(z !>= y)$
  - $(a < x < b)$
  - $!(x = 3)$

# Boolean expressions

- Both the relational and logical operators are lower in precedence than the arithmetic operators.
- Programming help:

Instead of `x == 3` use `3 == x`. Then, the compiler will catch the error if you write only one `=`

# xor Example

```
// This program demonstrates the xor() function.
#include <iostream>
using namespace std;

bool xor(bool a, bool b);

int main()
{
    int p, q;

    cout << "Enter P (0 or 1): ";
    cin >> p;
    cout << "Enter Q (0 or 1): ";
    cin >> q;

    cout << "P AND Q: " << (p && q) << '\n';
    cout << "P OR Q: " << (p || q) << '\n';
    cout << "P XOR Q: " << xor(p, q) << '\n';

    return 0;
}

bool xor(bool a, bool b)
{
    return (a || b) && !(a && b);
}
```

# Casts (force an expression to be of a specific type)

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

int main() // print i and i/2 with fractions
{
    int i;

    for(i=1; i<=100; ++i )
        cout << i << "/ 2 is: " << (float) i / 2 << '\n';

    return 0;
}
```

# The if statement

- The "if" statement allows us to execute a piece of code or not, based on the value of the expression we pass to it.

- Pseudocode:

```
if (<condition>) {  
    <statements>  
}
```

- If <condition> is TRUE (non-zero), then <statements> are executed.
- If <condition> is FALSE (zero), then <statements> are not executed.

# If ... example

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

int main()
{
    int a, b;

    cout << "Enter first number: ";
    cin >> a;
    cout << "Enter second number: ";
    cin >> b;

    if(a < b) cout << "First number is less than second.";

    return 0;
}
```

# Caution !!!

- comparison operator for equality is `==`, not `=`.

# If: selective execution

- **If: choosing between two alternatives**

- Pseudocode:

```
if (<condition>) {  
    <true_statements>  
} else {  
    <false_statements>  
}
```

- If <condition> is TRUE (non-zero), then <true\_statements> are executed.
- If <condition> is FALSE (zero), then <false\_statements> are executed.

# If: selective execution

- Example: Bank withdrawal checker

```
int balance;
int withdrawal;
cout << "Please enter the withdrawal amount: ";
cin >> withdrawal;

if(withdrawal <= balance) {
    cout << "Here is your money" << endl;
}
else {
    cout << "Sorry, insufficient balance" <<endl;
}
```

# If: selective execution

- Pitfall: Indentation by itself is not enough.

```
int age;
cout << "Please enter your age: ";
cin >> age;
if (age < 0)
    cout << "Wow, you are really young!"; //wrong
    cout << "Your age is negative.";
    cout << endl;
cout << "Your age is " << age << endl;
```

# If: selective execution

- What's wrong with this?

```
if (x > y) ;  
    x = y;  
cout << x;
```

# If: selective execution

- Pitfall: There is no semi-colon after the if-condition!

# The *for* loop

- *for(initialization, condition, increment)*  
{  
.  
.  
• C++ statements  
.  
}

# for loop ... example

```
// A program that illustrates the for loop.

#include <iostream>
using namespace std;

int main()
{
    int count;

    for(count=1; count<=100; count=count+1)
        cout << count << " ";

    return 0;
}
```

# The ++ operator

- $count = count + 1$  is normally written as  
 $count++$
- that's where C++ got its name!!!

# Introducing *Functions*

- A C++ program is constructed from building blocks called *functions*
- *main()* is a function where program execution begins
- A function is a subroutine that contains one or more C++ statements and performs one or more tasks

# Functions ... example

```
/* This program contains two functions: main()
   and myfunc().
*/
#include <iostream>
using namespace std;

void myfunc(); // myfunc's protoype

int main()
{
    cout << "In main() ";
    myfunc(); // call myfunc()
    cout << "Back in main() ";

    return 0;
}

void myfunc()
{
    cout << " Inside myfunc() ";
}
MONAMMAAD A. KAJABI                PROBLEM SOLVING WITH C++
```

# Functions ... contd.

- Every function must have a name (*main()* is reserved)
- One function cannot be embedded within another function (these are separate entities)
- One function may call another

# Functions ... contd.

- Two types of functions: those written by *You* and others provided to you in the compiler's standard library

# Function Arguments

- A value passed to a function is called an *argument*
- It is passed by specifying it between the parenthesis that follow the function's name

# Example ... library function abs()

```
// Use the abs() function.
#include <iostream>
#include <cstdlib> // for older compilers, use <stdlib.h>
using namespace std;

int main()
{
    cout << abs(-10);

    return 0;
}
```

# parameters

- The variables in a function that receive arguments are called *parameters* of the function

```
// A simple program that demonstrates mul().

#include <iostream>
using namespace std;

void mul(int x, int y); // mul()'s prototype

int main()
{
    mul(10, 20);
    mul(5, 6);
    mul(8, 9);

    return 0;
}

void mul(int x, int y)
{
    cout << x * y << " ";
}
```

# Functions Returning Values

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

int mul(int x, int y); // mul()'s prototype

int main()
{
    int answer;

    answer = mul(10, 11); // assign return value
    cout << "The answer is " << answer;

    return 0;
}

// This function returns a value.
int mul(int x, int y)
{
    return x * y; // return product of x and y
}
```

# C++ function general form

*Return-type function-name(parameter list)*

{

.

*.body of function*

.

}

# Output on Next line (\n character)

```
/* This program demonstrates the \n code, which
   generates a new line.
*/
#include <iostream>
using namespace std;

int main()
{
    cout << "one\n";
    cout << "two\n";
    cout << "three";
    cout << "four";

return 0;
}
```

# Declaration of Variables

- local variables
- formal parameters
- global variables

# Local Variables ... example

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

void func();

int main()
{
    int x; // local to main()

    x = 10;
    func();
    cout << "\n";
    cout << x; // displays 10

    return 0;
}

void func()
{
    int x; // local to func()

    x = -199;
    cout << x; // displays -199
}
```

# Formal Parameters

- `int func1 (int first, int last, char ch)`  
{  
    .  
    .  
    .  
}

# Global Variables ... example

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

void func1();
void func2();

int count; // this is a global variable

int main()
{
    int i; // this is a local variable

    for(i=0; i<10; i++) {
        count = i * 2;
        func1();
    }

    return 0;
}

void func1()
{
    cout << "count: " << count; // access global count
    cout << '\n'; // output a newline
    func2();
}

void func2()
{
    int count; // this is a local variable
    for(count=0; count<3; count++) cout << '.';
}
```

Mohar  
}

# Type Modifiers

- signed
- unsigned
- long
- short

# Type Modifiers ... contd.

- Difference between signed and unsigned?



# signed, unsigned ... example

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

/* This program shows the difference between
   signed and unsigned integers.
*/
int main()
{
    short int i; // a signed short integer
    short unsigned int j; // an unsigned short integer

    j = 60000;
    i = j;
    cout << i << " " << j;

    return 0;
}
```

# Specifying constants

- Hexadecimal constants start with a 0x
- e.g 0xFF // 255 in decimal
  
- Octal constants start with a 0
- e.g. 011 // 9 in decimal



# Backslash character constants

- `\b`          backspace
- `\n`          newline
- `\t`          horizontal tab
- `\a`          alert
- .
- .
- .