**ITCS 1212L**

**Lab lessons 2**

**Sequence, Flowcharts, Identifiers, Fundamental data types, Declaration of variables, Initialization of variables, assignment operators, arithmetic operators, compound assignments, increase and decrease, standard input and output**

The first of the four basic concepts of structured programming we are learning is sequence. This is a basic concept, many students laugh when we talk about it. But, many students get caught up in not remembering this concept when writing their programs. The basic concept of sequence is:

# The order you write the statements in your code is the order they will be executed in.

# All programs must have a function named main( )

This sounds simple enough, but many times when you see your output is wrong you may be tempted to just shuffle the statements and hope it turns out better the next time you run the program. Actually, this game of chance is a lot harder than remembering the concept of sequence.

Think of baking a cake and the sequence you must follow if you want something edible after your work. You must first decide what ingredients are needed. Then gather the appropriate amount of each ingredient. Preheat the oven, grease and flour the baking pans. Then mix the eggs, butter, vanilla and sugar in one bowl and the flour and baking soda in another. Then combine the two bowls, place in a pan and bake for 35 minutes - in that order.

You cannot bake the flour and take it out of the over then try to mix in the moist ingredients. You cannot put everything in the oven and then after 35 minutes, take the pan out and turn the oven on. The sequence is VERY important.

Remember when you were young before you could read, the teacher gave you pictures of a story to put in order in the correct sequence? This is the same concept.

The same is true of the statements you write in a program. You must specify the order in which they are executed. The order the statements appear in your code is the order in which they will be executed.

To help you in your first programs think of these three words:

**Input –> Process –> Output**

This will help you write your programs in the proper sequence.

First think of what **input** is needed from the user (ingredients needed), then think of what you need to do to this data to get the desired output (**process**). Then, finally, display the **output**.

A program can be designed in graphical method using Flowcharts. In flowchart the program is divided into small process and each of the process is represented using different geometrical diagrams. Arrows are used between these geometrical diagrams represents the sequence.

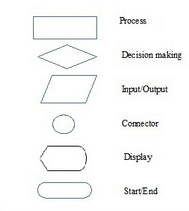
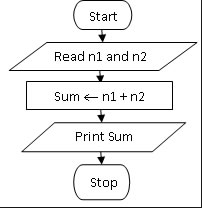


Fig 1. Basic symbols of flowchart

Some of the basic symbols used in flowchart are as shown above. Each symbol has its own function within the program. The beginning or the end of program can be represented by Start/End symbol.

Input/Output symbols are used to represent the input and output operations. Display symbol signifies that information is displayed to the user. The process symbols lets you show how the program is functioning. The decision making symbol is used when you need to make decisions based on specified criteria.



**Fig2. Flowchart for adding two numbers**

**Variable:** *A variable is a named location in the computer’s memory whose contents can change as the program is running.* (**You need to know this definition)!**

* you must declare all variables before you put any data in the variable
* variables get values two ways - using the assignment operator or with a **cin** statement

As we use names to refer to a person we use identifiers to refer to memory location (variables). As you saw in lab lesson 1 fig 3, the memory locations that were referred for data were referred to as x,y,z w.

**Data types:** When programming, we store the variables in our computer's memory, but the computer has to know what kind of data we want to store in them, since it is not going to occupy the same amount of memory to store a simple number than to store a single letter or a large number, and they are not going to be interpreted the same way.

Next you have a summary of the basic fundamental data types in C++, as well as the range of values that can be represented with each one:



Fig3. Basic data types in C++

**Identifiers**: A valid identifier is a sequence of one or more letters, digits or underscore characters (\_). Neither spaces nor punctuation marks or symbols can be part of an identifier. Only letters, digits and single underscore characters are valid. In addition, variable identifiers always have to begin with a letter. They can also begin with an underline character (\_ ), but in some cases these may be reserved for compiler specific keywords or external identifiers, as well as identifiers containing two successive underscore characters anywhere. In no case they can begin with a digit.

Another rule that you have to consider when inventing your own identifiers is that they cannot match any keyword of the C++ language nor your compiler's specific ones, which are *reserved keywords*. The standard reserved keywords are:

asm, auto, bool, break, case, catch, char, class, const, const\_cast, continue, default, delete,

do, double, dynamic\_cast, else, enum, explicit, export, extern, false, float, for, friend, goto,if, inline, int, long, mutable, namespace, new, operator, private, protected, public, register,reinterpret\_cast, return, short, signed, sizeof, static, static\_cast, struct, switch, template,this, throw, true, try, typedef, typeid, typename, union, unsigned, using, virtual, void,volatile, wchar\_t, while

Additionally, alternative representations for some operators cannot be used as identifiers since they are reserved words under some circumstances:

and, and\_eq, bitand, bitor, compl, not, not\_eq, or, or\_eq, xor, xor\_eq

**Arithmetic operators:** The first sets of operators we will use are the arithmetic operators. They are the same operators you learned in elementary math courses and follow the same rules of precedence (order) that you use in math.

**Note: There is no exponent operator in C++.**

\*, /, %, +, - multiplication, division, modulus, addition and subtraction. This means in an expression that contains several of these operators the multiplication, division and modulus operators will be evaluated first from left to right as they appear in the expression. Once all these operators are applied then addition and subtraction will be applied, from left to right.

**Example:**

15 – 3 \* 4 \* 6/2 + 5

How it is evaluated?

1. 15 – 12 \* 6/2 + 5
2. 15 – 72/2 + 5
3. 15 – 36 + 5
4. -21 + 5
5. -16

Integer division is different from floating point division. **Integers are whole numbers have NO decimal places.**

**Integer division**: 6/4 = 1, 8/5 = 1, 4/8 = 0

**Floating point division**: 6.0/4 = 1.5, 8/5.0 = 1.6, 4.0/8.0 = .5

We will go over modulus (%), Modulus gives the remainder after integer division. Look at these examples:

5 % 3 = 2 5 divided by 3 is 1 with **remainder** 2

7 % 2 = 1 7 divided by 2 is 3 with **remainder** 1

10 % 3 = 1 10 divided by 3 is 3 with **remainder** 1

12 % 5 = 2 12 divided by 5 is 2 with **remainder** 2

28 % 5 = 3 18 divided by 5 is 5 with **remainder** 3

**Order of Precedence:**

When writing complex expressions with several operands, we may have some doubts about which operand is evaluated first and which later. For example, in this expression:

a = 5 + 7 % 2

We may doubt if it really means:

a = 5 + (7 % 2) // with a result of 6, or

a = (5 + 7) % 2 // with a result of 0

The correct answer is the first of the two expressions, with a result of 6. There is an established order with the priority of each operator, and not only the arithmetic ones (those whose preference come from mathematics) but for all the operators which can appear in C++. From greatest to lowest priority, the priority order is as follows:

|  |  |  |  |
| --- | --- | --- | --- |
| **Level** | **Operator** | **Description** | **Grouping** |
| 1 | :: | scope | Left-to- right |
| 2 | () [] . -> ++ -- dynamic\_cast static\_cast reinterpret\_cast const\_cast typeid | postfix | Left-to- right |
| 3 | ++ -- ~ ! sizeof new delete | unary (prefix) | Right-to- left |
| \* & | indirection and reference (pointers) |
| + - | unary sign operator |
| 4 | (type) | type casting | Right-to- left |
| 5 | .\* ->\* | pointer-to-member | Left-to- right |
| 6 | \* / % | multiplicative | Left-to- right |
| 7 | + - | additive | Left-to- right |
| 8 | << >> | shift | Left-to- right |
| 9 | < > <= >= | relational | Left-to- right |
| 10 | == != | equality | Left-to- right |
| 11 | & | bitwise AND | Left-to- right |
| 12 | ^ | bitwise XOR | Left-to- right |
| 13 | | | bitwise OR | Left-to- right |
| 14 | && | logical AND | Left-to- right |
| 15 | || | logical OR | Left-to- right |
| 16 | ?: | conditional | Right-to- left |
| 17 | = \*= /= %= += -= >>= <<= &= ^= |= | assignment | Right-to- left |
| 18 | , | comma | Left-to- right |

Table 1. Operators and order of precedence

**Assignment operator:** The purpose of the assignment operators is to store a value in a memory location (i.e. - a variable). The assignment operators works from right to left. The value of the expression on the right-hand-side of the assignment operator is stored in the variable on the left-hand-side of the assignment operator. The only thing that can be placed to the left of the assignment operator is a declared variable.

This is the assignment operator: **=**

The arithmetic operations on the Right-hand-side of the assignment operator are executed and the result is a value. This value is then stored in the variable on the left right-hand-side of the assignment operator.

**Compound assignments:** When we want to modify the value of a variable by performing an operation on the value currently stored in that variable we can use compound assignment operators.

For example:

value += increase; is equivalent to value = value + increase;

The compound assignments are: (+=, -=, \*=, /=, %=, >>=, <<=, &=,^=, |=)

**Increment and decrement: T**he increase operator (++) and the decrease operator (--) increase or

reduce by one the value stored in a variable. They are equivalent to +=1 and to -=1, respectively. Thus:

c++;

c+=1;

c=c+1;

are all equivalent in its functionality: the three of them increase by one the value of c.

**Standard input and output:** By default, the standard output of a program is the screen, and the C++ stream object defined to access it is cout. cout is used in conjunction with the *insertion operator*, which is written as << (two "less than" signs). For example:

cout << "Output sentence"; // prints Output sentence on screen

cout << 120; // prints number 120 on screen

cout << x; // prints the content of x on screen

The standard input device is usually the keyboard. Handling the standard input in C++ is done by applying the overloaded operator of extraction (>>) on the cin stream. The operator must be followed by the variable that will store the data that is going to be extracted from the stream. For example:

int age;

cin >> age;

The first statement declares a variable of type int called age, and the second one waits for an input from cin (the keyboard) in order to store it in this integer variable.