Variables, Data Types, and Arithmetic Expressions

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

Dept. of Computer Science
Faculty of Science and Technology

Lecturer No:	**2**	**Week No:**	**1 (1X1.5 hrs)**	**Semester:**	
Lecturer:	*Name & email*				

Lecture 2: Outline

- **Variables, Data Types, and Arithmetic Expressions**

- Working with Variables
  - Understanding Data Types and Constants
    - The Basic Integer Type int
    - The Floating Number Type float
    - The Extended Precision Type double
    - The Single Character Type char
    - The Boolean Data Type _Bool
    - Storage sizes and ranges
    - Type Specifiers: long, long long, short, unsigned, and signed
  - Working with Arithmetic Expressions
    - Integer Arithmetic and the Unary Minus Operator
    - The Modulus Operator
    - Integer and Floating-Point Conversions
  - Combining Operations with Assignment: The Assignment Operators
  - Types _Complex and _Imaginary

Variables

- Programs can use symbolic names for storing computation data
- **Variable: a <u>symbolic name</u> for a memory location**
  - programmer doesn’t have to worry about specifying (or even knowing) the value of the location’s address
- Variables have to be ***declared*** before they are used
  - Variable declaration: [symbolic name(*identifier*), type]
- Declarations that reserve storage are called ***definitions***
  - The definition reserves memory space for the variable, but doesn’t put any value there
- Values get into the memory location of the variable by ***initialization*** or ***assignement***

Variables - Examples

```
int a;          // declaring a variable of type int

int sum, a1,a2; // declaring 3 variables

int x=7; // declaring and initializing a variable


a=5;  // assigning to variable a the value 5

a1=a; // assigning to variable a1 the value of a
```

L-value ***R-value***

```
a1=a1+1;  // assigning to variable a1 the value of a1+1
          // (increasing value of a1 with 1)
```

Variable declarations

Data type Variable name

Which data types are possible

Which variable names are allowed

Variable names

- ❑ Rules for valid variable names (*identifiers*) in C :
- ❑ Name must begin with a letter or underscore (_) and can be followed by any combination of letters, underscores, or digits.
- ❑ Any name that has special significance to the C compiler (*reserved words*) cannot be used as a variable name.
- ❑ Examples of ***valid*** variable names: `Sum`, `pieceFlag`, `I`, `J5x7`, `Number_of_moves`, `_sysflag`
- ❑ Examples of ***invalid*** variable names: `sum$value`, `3Spencer`, `int`.
- ❑ C is case-sensitive: `sum`, `Sum`, and `SUM` each refer to a different variable !
- ❑ Variable names can be as long as you want, although only the first 63 (or 31) characters might be significant. (Anyway, it’s not practical to use variable names that are too long)
- ❑ Choice of meaningful variable names can increase the readability of a program

Data types

- **Basic data types in C: `int`, `float`, `double`, `char`, and `_Bool`.**
- Data type `int:` can be used to store integer numbers (values with no decimal places)
- Data type type `float:` can be used for storing floating-point numbers (values containing decimal places).
- Data type `double:` the same as type `float`, only with roughly twice the precision.
- Data type `char:` can be used to store a single character, such as the letter *a*, the digit character *6*, or a semicolon.
- Data type `_Bool:` can be used to store just the values 0 or 1 (used for indicating a true/false situation). This type has been added by the C99 standard (was not in ANSI C)

Assigning values to char

```
char letter;  /* declare  variable letter of type char */

letter = ‘A';   /* OK */
letter = A;     /* NO! Compiler thinks A is a variable */
letter = “A";   /* NO! Compiler thinks “A" is a string */
letter = 65;    /* ok because characters are really
                    stored as numeric values (ASCII code),
                    but poor style */
```

Storage sizes and ranges

- **Every type has a *range* of values associated with it**.
- This range is determined by the amount of storage that is allocated to store a value belonging to that type of data.
- In general, that amount of storage is not defined in the language. It typically depends on the computer you're running, and is, therefore, called *implementation*- or *machine*-dependent.
  - For example, an integer might take up 32 bits on your computer, or it might be stored in 64.You should never write programs that make any assumptions about the size of your data types !
- The language standards only guarantees that a minimum amount of storage will be set aside for each basic data type.
  - For example, it's guaranteed that an integer value will be stored in a minimum of 32 bits of storage, which is the size of a "word" on many computers.

Working with arithmetic expressions

- Basic arithmetic operators: +, -, *, /
- **Precedence**: one operator can have a higher priority, or *precedence*, over another operator.
  - Example: * has a higher precedence than +
  - a + b * c
  - if necessary, you can always use parentheses in an expression to force the terms to be evaluated in any desired order.
- **Associativity**: Expressions containing operators of the same precedence are evaluated either from left to right or from right to left, depending on the operator. This is known as the *associative* property of an operator
  - Example: + has a *left to right* associativity
- In C there are many more operators -> later in this course !
- (Table A5 in Annex A of [Kochan]: full list, with precedence and associativity for all C operators)

Working with arithmetic expressions

```
#include <iostream>
using namespace std;
int main (void)
{
   int a = 100;int b = 2;
   int c = 25;int d = 4;
   int result;
   result = a - b; // subtraction
   cout<<"a - b = “<< result<<endl;
   result = b * c; // multiplication
   cout<<"b * c =  “<< result<<endl;;
   result = a / c; // division
   cout<<"a / c =  “<< result<<endl;
   result = a + b * c; // precedence
   cout<<"a + b * c = “<<result<<endl;
   return 0;
}
```

Integer and Floating-Point Conversions

- Assign an integer value to a floating variable: does not cause any change in the value of the number; the value is simply converted by the system and stored in the floating
- Assign a floating-point value to an integer variable: the decimal portion of the number gets truncated.
- Integer arithmetic (division):
  - int divided to int => result is integer division
  - int divided to float or float divided to int => result is real division (floating-point)

The Type Cast Operator

- `f2 = (float) i2 / 100; // type cast operator`
- The type cast operator has the effect of converting the value of the variable i2 to type float for purposes of evaluation of the expression.
- This operator does NOT permanently affect the value of the variable i2;
- **The type cast operator has a higher precedence than all the arithmetic operators except the unary minus and unary plus.**
- Examples of the use of the type cast operator:
  - `(int) 29.55 + (int) 21.99` results in `29 + 21`
  - `(float) 6 / (float) 4` results in `1.5`
  - `(float) 6 / 4` results in `1.5`

The assignment operators

- The C language permits you to join the arithmetic operators with the assignment operator using the following general format: `op=`, where `op` is an arithmetic operator, including +, –, ×, /, and %.
- `op` can also be a logical or bit operator => later in this course
- Example:
- `count += 10;`
  - Equivalent with:
- `count=count+10;`
- Example: precedence of op=:
  - `a /= b + c`
  - Equivalent with:
- `a = a / (b + c)`
  - addition is performed first because the addition operator has higher precedence than the assignment operator

Declaring variables

- Some older languages (FORTRAN, BASIC) allow you to use variables without declaring them.
- Other languages (C, Pascal) impose to declare variables
- **Advantages of languages with variable declarations:**
  - Putting all the variables in one place makes it easier for a reader to understand the program
  - Thinking about which variables to declare encourages the programmer to do some planning before writing a program (What information does the program need? What must the program to produce as output? What is the best way to represent the data?)
  - The obligation to declare all variables helps prevent bugs of misspelled variable names.
  - **Compiler knows the amount of statically allocated memory needed**
  - **Compiler can verify that operations done on a variable are allowed by its type (*strongly typed languages*)**

Declaration vs Definition

- Variable declaration: [Type, Identifier]
- Variable definition: a declaration which does also reserve storage space (memory) !
  - Not all declarations are definitions
  - In the examples seen so far, all declarations are as well definitions
  - Declarations which are not definitions: later in this semester !

Executing a program

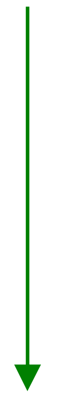
- Program = list of statements
  - ***Entrypoint***: the point where the execution starts
  - ***Control flow***: the order in which the individual statements are executed

```
Statement1
Statement2
Statement3
Statement4
Statement5
Statement6
Statement7
Statement8
```

Structure of a C++ program

Entry point of a C++ program

```
int main ()
{
   int value1, value2, sum;
   value1 = 50;
   value2 = 25;
   sum = value1 + value2;
   cout<<"The sum = " <<sum<<endl;
   return 0;
}
```



Controlling the program flow

- Forms of controlling the program flow:
  - Executing a sequence of statements
  - Repeating a sequence of statements (until some condition is met) (looping)
  - Using a test to decide between alternative sequences (branching)

```
Statement1
Statement2
Statement3
Statement4
Statement5
Statement6
Statement7
Statement8
```