PROGRAMMING
MIEIC – 2016/2017

LECTURE MATERIAL

==========

C++ BASICS:
PROGRAM STRUCTURE, TYPES, VARIABLES, EXPRESSIONS, ..., CODING STYLE

==========

Program structure

- A program is basically a collection of functions, one of which must be named **main()**.
- Program execution begins with **main()**.
- If necessary, **main()** can call other functions.

==========

```
/*
FILE    : p001.cpp                    ← OPENNING DOCUMENTATION / COMMENTS
DATE    : 2016/09/23
AUTHOR : JAS
PROGRAM PURPOSE:
- To salute the user, on the screen
*/

#include <iostream>  //  ← COMPILER DIRECTIVE(S)
                     //       FOR INSERTING THE CONTENTS OF A HEADER FILE

int main()           //  ← EXECUTION STARTS HERE
{
      std::cout << "Hello !" << std::endl;  // PROGRAM STATEMENT(S)

      return 0;      // VALUE 0 IS RETURNED TO THE COMMAND INTERPRETER
                     // 0 is the 'exit code' of the program (SEE LATER)
                     //to see 'exit code' on the console: > echo %ERRORLEVEL%
}
```

==========

```
/*
FILE    : p001.c        //  ← The same code in "pure C"
...
*/
#include <stdio.h>

int main()
{
  printf("Hello!\n");
  return 0;
}
```

==========

Notes

- **C** is **case-sensitive**.
- Each **statement** must be terminated by a **semicolon**.
- **`std::cout`** is a **C++ object** of **class** **`ostream`**
  that represents the standard output stream oriented to characters (of type char).
  It corresponds to the C stream **`stdout`**.

==========

Compiler directives

- The most commonly used compiler directives are the **#include directives**.
- They are instructions for the preprocessor
  to insert the contents of the specified file(s) at this point.
- **`#include <filename>`** - for standard libraries
- **`#include "filename"`** - for programmer defined libraries
- The files included are called **header files**.
  - In C language, header file names usually end with **`.h`**
- These files contain the **declarations** of **functions**, **constants**, **variables**.
- *The use of programmer defined libraries will be introduced later.*

==========

```
/* .....
*/

#include <iostream>

using std::cout;  //avoids the need of writing std::... in the statements below
using std::endl;

int main()
{
      cout << "Hello friends !" << endl;

      return 0;
}
```

==========

```
/* ..... for simplicity, comments will be supressed in most of the examples
*/

#include <iostream>

using namespace std;

int main()
{
      cout << "Hello friends !\n"; // '\n' can be used instead of endl

      return 0;
}
```

==========

Namespaces

- Are a means of organizing typenames, variables and functions so as to avoid name conflicts.
- Usually we want to use the standard libraries that are in the **namespace** named **std**.
- This theme will be treated later.

Comments

- Two kinds of comments are allowed in C/C++
- *// comment* - continues to end of line
- */*
  comments - may extend over several lines
  **/*

Standard libraries

- `iostream` – basic interactive input/output (I/O) .
- `iomanip` – format manipulators.
- `fstream` – file I/O.
- `string` – standard C++ strings.
- `cstring` – C strings type.
- `cmath` – math functions.
- `climits` – max and min values of integer types.
- `cfloat` - max and min values of float types.
- ... and many other

Identifiers and Keywords

- **Identifiers** – a letter (or underscore) followed by any number of letters, digits or underscores.
  - identifiers starting with an underscore are usually reserved for a special purpose
- **C/C++ is case sensitive**
  - **sum**, **Sum**, and **SUM** are different identifiers
- Identifiers may not be any of the language keywords:
  - some examples of **keywords**:
    char, double, float, int, unsigned, long, short, bool, true, false, const,
    if, else, switch, case, default,
    while, do, for, break, continue, goto, return,
    class, typename, friend, operator, public, protected, ...
  - For a complete list consult a C++ manual.

- Recommended naming conventions (example):
  - `string  bookTitle ;` OR
  - `string  book_title;`

==========

Coding style

- Take a look at the links available at the course page in Moodle;
  many others are available in the web.
- Just "a couple" of **recommendations**:
  - choose adequate / meaningful identifier names
  - place a comment with each variable declaration
    explaning the purpose of the variable
  - write one statement in each line
  - indent the code
  - complex sections of code and any other parts of the program
    that need some explanation should have comments
    just ahead of them or embedded in them.

==========

```
*
FILE   : p002.cpp
DATE   : 2016/02/16
AUTHOR : JAS
PROGRAM PURPOSE:
- Read 2 integers and
- compute their sum, difference, product and quotient

BAD CODING STYLE.
*/

#include <iostream>
using namespace std;
int main() {
int x, y, s, d, p; double q;  // VARIABLE DECLARATIONS
cout << "x ? "; cin >> x; cout << "y ? "; cin >> y; s = x + y; d = x - y; p
= x * y; q = x / y; cout << "s = " << s << endl << "d = " << d << endl <<
"p = " << p << endl << "q = " << q << endl; return 0;}
```

==========

Variables

- C/C++ is a strongly-typed language,
  and requires **every variable** to be **declared** with its type before its first use.
- This informs the compiler the size to reserve in memory for the variable
  and how to interpret its value.

==========

```
/*
...
A BETTER CODING STYLE.

BUT SOME PROBLEMS WITH quotient
TEST PROGRAM WITH PAIRS (4,2) (100,5) (357,7)  ... AND (1,3)
*/

#include <iostream>

using namespace std;

int main()
{
      int operand1, operand2;  // input operands
      int sum;                 // sum of input operands
      int difference;          // difference of operands
      int product;             // product of operands
      double quotient;         // quotient of operands

      // input 2 integers
      cout << "operand1 ? ";   // C-style
      cin >> operand1;         // scanf("%d",&operand1) for input
      cout << "operand2 ? ";   // printf("operand 2 ? ") for output
      cin >> operand2;

      //compute their sum, difference, product and quotient
      sum = operand1 + operand2;
      difference = operand1 - operand2;
      product = operand1 * operand2;
      quotient = operand1 / operand2;

      //show results
      cout << "       sum = " << sum << endl;
      cout << "difference = " << difference << endl;
      cout << "   product = " << product << endl;
      cout << "  quotient = " << quotient << endl;

      return 0;
}
```

NOTE THE RELEVANCE OF TESTING ADEQUATELY YOUR PROGRAM

==========

Fundamental data types

- Variables must be declared before they are used.
- The type of the variable must be chosen
- **Integers**: `int`
    - integer variations:
      `short` (*OR* `short int`), `long` (*OR* `long int`),
      `unsigned` (*OR* `unsigned int`) , `long long`
- **Reals**: `float`, `double`, `long double`
- **Characters**: `char` (*also has some variations*)
- **Booleans**: `bool` (*logical values:* `true` *OR* `false`)
- **Void type**: `void`

```
/*
...
Solving the quotient of 2 integers problem
*/

#include <iostream>

using namespace std;

int main()
{
      int operand1, operand2;  // input operands
      int sum;                 // sum of input operands
      int difference;          // difference of ...
      int product;             // ...
      double quotient;         // ...

      // input 2 integers
      cout << "operand1 operand2 ? ";
      cin >> operand1 >> operand2;

      //compute their sum, difference, product and quotient
      sum = operand1 + operand2;
      difference = operand1 - operand2;
      product = operand1 * operand2;
      quotient = static_cast<double> (operand1) / operand2;
      //OR
      // quotient = (double) operand1 / operand2; // C-style

      //show results
      cout << "       sum = " << sum << endl;
      cout << "difference = " << difference << endl;
      cout << "   product = " << product << endl;
      cout << "  quotient = " << quotient << endl;

      return 0;
}
```

==========

Declarations

- Variables declarations have the forms:
  - *type variable_name*: **`int sum;`**
  - *type list_of_variables*: **`int operand1, operand2;`**
- **VERY IMPORTANT NOTE**:
  When the variables in the above example are declared,
  they have an undetermined value
  until they are assigned a value for the first time.
- But it is possible for a variable to have a specific value
  from the moment it is declared; this is called **variable initialization**:
  - `int x = 0, y = 1;  // C-style initialization`
  - other forms of initialization are possible:
    - `int x(0); // C++ constructor-style initialization`
    - `int x{0}; // C++11 uniform initialization`

```
/*  ...
The results of the 4 operations are always computed.
Usually one only wants the result of one of the operations.
*/
#include <iostream>

using namespace std;

int main()
{
      int operand1, operand2;  // input operands
       // NOTE THE DIFFERENCE FROM PREVIOUS EXAMPLE: other var.s not declared

      // input 2 integers
      cout << "operand1 ? ";
      cin >> operand1;
      cout << "operand2 ? ";
      cin >> operand2;

      //compute and show results
      cout << "       sum = " << operand1 + operand2 << endl;
      cout << "difference = " << operand1 - operand2 << endl;
      cout << "   product = " << operand1 * operand2 << endl;
      cout << "  quotient = " << operand1 / operand2 << endl;

      return 0;
}
```

==========

Input / Output (I/O) expressions

- I/O is carried out using **streams** that connect the program and I/O devices (keyboard, screen) or files.
- **Input expressions**
  - input / extraction operator: **>>**
  - The expression *instream >> variable*
    - extracts a value of the type of *variable* (if possible) from *instream*
    - stores the value in *variable*
    - **returns *instream*** as its result (if sucessful, else 0)
  - This last property, along the left-associativity (*see later*) of **>>** makes it possible to chain input expressions:
    - *instream >> variable1 >> variable2 >> .... >> variableN;*
- **Output expressions**
  - output / insertion operator: **<<**
  - The expression *outstream* **<<** *value*
    - inserts *value* into *instream;*
      - *value* may be a constant, variable or the result of an expression
      - **returns *outstream*** as its result
  - *outstream << value1 << value2 << ....<< valueN;* is also possible

Literals / Constants

- **Integers**:
  - use the usual decimal representation: `25, -3, 1000`
  - octal numbers begin with `0` (zero)
  - hexadecimal numbers begin with `0x`
  - suffixes may be appended to specify the integer type:
    - `u` *or* `U` for `unsigned`: `75U`
    - `l` *or* `L`, for `long`: `1000000000L`
    - `ll` *or* `LL`, for `long long`
- **Reals**:
  - use the usual decimal representation and `e` or `E`: `19.5, 2e-1, 5.3E3`
- **Characters**:
  - single chars are enclosed in single quotes: `'A', 'd', '8', ' '`
  - escape sequences are used for special character constants:
    - `'\n'` : newline
    - `'\''` : single quote
    - `'\\'` : backslash
    - `...`
- **Strings**:
  - strings are enclosed in double quotes: `"Programming course"`

==========

Operators

- **Assignment operator** :
  - `int x, y;`
  - `x = 5;`
  - `y = x;`
  - `y = x = 5; //x equals to 5 and the result of (x=5) is 5, so ...?`
  - `y = 2 + (x = 3);  // POSSIBLE!!! BUT NOT RECOMMENDED ...`
    - is equivalent to:
      - `x = 3;`
      - `y = 2 + 3;  // (x = 3) evaluates to 3`
- **Arithmetic operators**:
  - `+` - addition
  - `-` - subtraction
  - `*` - multiplication
  - `/` - division (NOTE: be careful when both operands are integers!)
  - `%` - modulo (rest of integer division)
- **Relational and comparison operators:**
  - `==` - equal to (NOTE THIS! BE CAREFUL!!!)
  - `!=` - not equal to
  - `>` - greater than
  - `<` - less than
  - `>=` - greater than or equal to
  - `<=` - less than or equal to
- **Logical operators:**
  - `!` - Boolean NOT
  - `&&` - Boolean AND
  - `||` - Boolean OR
- ***To be introduced latter:***
  - Compound assignment :
    - `+=, -=, *=, /=, %=, &=, ^=, |=,` ...
  - Increment and decrement (by one):
    - `++, --`
  - Conditional ternary operator:
    - `? :`
  - Comma operator:
    - `,`
  - Bitwise operators:
    - `&, |, ^, ~, <<, >>`
  - Explicit type casting operator
  - `sizeof`

THERE ARE OTHER OPERATORS (SEE NEXT PAGE)

Precedence of operators

Level	Precedence group	Operator	Description	Grouping
1	Scope	`::`	scope qualifier	Left-to-right
2	Postfix (unary)	`++ --`	postfix increment / decrement	Left-to-right
		`()`	functional forms	
		`[]`	subscript	
		`. ->`	member access	
3	Prefix (unary)	`++ --`	prefix increment / decrement	Right-to-left
		`~ !`	bitwise NOT / logical NOT	
		`+ -`	unary prefix	
		`& *`	reference / dereference	
		`new delete`	allocation / deallocation	
		`sizeof`	parameter pack	
		`(type)`	C-style type-casting	
4	Pointer-to-member	`.* ->*`	access pointer	Left-to-right
5	Arithmetic: scaling	`* / %`	multiply, divide, modulo	Left-to-right
6	Arithmetic: addition	`+ -`	addition, subtraction	Left-to-right
7	Bitwise shift	`<< >>`	shift left, shift right	Left-to-right
8	Relational	`< > <= >=`	comparison operators	Left-to-right
9	Equality	`== !=`	equality / inequality	Left-to-right
10	And	`&`	bitwise AND	Left-to-right
11	Exclusive or	`^`	bitwise XOR	Left-to-right
12	Inclusive or	`|`	bitwise OR	Left-to-right
13	Conjunction	`&&`	logical AND	Left-to-right
14	Disjunction	`||`	logical OR	Left-to-right
15	Assignment-level expressions	`= *= /= %= += -= >>= <<= &= ^= |=`	assignment / compound assignment	Right-to-left
		`?:`	conditional operator	
16	Sequencing	`,`	comma separator	Left-to-right

source: http://www.cplusplus.com/doc/tutorial/operators/

- Examples:
  - `x = 2 + 3 * 4;` `// x = ...?`
  - `y = (2 + 3) * 4;` `// y = ...?`
  - `z = y = x + 10;` `// how is this evaluated ?`
- When an expression has two operators with the same precedence level, grouping determines which one is evaluated first: either left-to-right or right-to-left.
- Enclosing all sub-statements in parentheses (even those unnecessary because of their precedence) improves code readability.

==========

CONTROL STRUCTURES

==========

Control structures

- Are language constructs that allow a programmer to control the flow of execution through the program.
- There are 3 main categories:
  - **sequence**
  - **selection**
  - **repetition**

Sequence

- Sequential execution is implemented by **compound statements** (or **blocks**)
- They consist of statements enclosed between curly braces: `{` and `}` .

```
{
   statement1    ← REMEMBER: statements end with ;
   statement2
   ...
   statementN
}
```

- Example:

```
{
   cout << "X and Y values ? ";
   cin >> x >> y;
   cout << " x + y = " << x + y;
}
```

Selection

- Selective execution is implemented by :
  - **if** statements
  - **if ... else** statements
  - **switch ... case** statements

- **`if` statement**

```
if (boolean_expression)
   statement
```

  - The boolean expression must be enclosed in parenthesis.
  - The statement may be a compound statement.

  - Example 1:

```
if (x > 0)
  cout << "X is positive \n";
```

  - Example 2 (what is the result of this compound statement ?):

```
if (x > y)
{
  int t = y; // t only exists temporarily, inside this block
  y = x;
  x = t;
}
```

  - NOTE 1: in C/C++, the value of any variable or expression may be interpreted as true, if it is different from zero, or false, if it is equal to zero.

```
if (x)  // if x is different from zero
{       // to improve readability do write (x!=0)
  ...
}
```

  - NOTE 2: a very common error (not detected by the compiler)

```
if (x = 10)
{
  ...
}
```

    - will assign 10 to `x`
    - and the value of `(x = 10)` is 10 (`true`)
    - so... **BE CAREFUL!**

  - NOTE 3: if written in this way the compiler will detect the error ☺

```
if (10 = x)
{
  ...
}
```

- **if ... else statement**

```
if (boolean_expression)
   statement1
else
   statement2
```

  - Example 1:

```
if (x==y)
  cout << "x is equal to y \n"; // note the semicolon
else
  cout << "x is not equal to y \n";
```

- **switch ... case statement**

```
switch (integer_expression)  //NOTE: must evaluate to an integer
{
  case_list_1:
     statement_list_1
     break;    // usually a break OR return is used after each statement list
  case_list_1:
     statement_list_2
     break;
  .
  .
  .
  default:
     default_group_of_statements
}
```

  - Each *case_list_i* is made up of
    - case *constant_value* :
      OR
      case *constant_value_1* : ... : case *constant_valueN* :
  - switch can only test for equality.
  - No two case constants can have the same value
  - The break statement causes the execution of the program to continue after the switch statement.
  - The default part is optional;
    it only is executed if the value of the *integer_expression* is in no ***case_list_i***

```
/*
A NOT VERY GOOD SOLUTION. WHY ?

TEST PROGRAM USING:
 2+3, 2 /3,    5   *   10   , ...  2  3  4  ...OR  a + 2 (!!!)
*/

#include <iostream>

using namespace std;

int main()
{
      double operand1, operand2;  // input operands
      double sum;  // sum of input operands
      double difference;  // difference of ...
      double product;  // ...
      double quotient;  // ...
      char operation; // QUESTION: WHY "OPERATION" AND NOT "OPERATOR"
                      // consult the list of reserved identifiers / keywords

      // input 2 numbers
      cout << "x op y ? ";
      cin >> operand1 >> operation >> operand2;


      //compute their sum, difference, product or quotient
      if (operation == '+')
         sum = operand1 + operand2;
      if (operation == '-')
         difference = operand1 - operand2;
      if (operation == '*')
         product = operand1 * operand2;
      if (operation == '/')
         quotient = operand1 / operand2; // BOTH OPERANDS ARE double ☺

      //show results
      cout << operand1 << ' ' << operation << ' ' << operand2 << " = ";
      if (operation == '+')
         cout << sum;
      if (operation == '-')
         cout << difference;
      if (operation == '*')
         cout << product;
      if (operation == '/')
         cout << quotient;
      // TRY THE FOLLOWING with and without parenthesis in (quotient = ...)
      // if (operation == '/')
      //    cout << (quotient = operand1 / operand2);

      cout << endl;

      return 0;
}
```

==========

```
/*
1) A LITTLE BETTER ... WHY ?

2) Try with an invalid operation (ex: X instead of *)
*/

#include <iostream>

using namespace std;

int main()
{
	double operand1, operand2;  // input operands
	double sum;  // sum of input operands
	double difference;  // difference of ...
	double product;  // ...
	double quotient;  // ...
	char operation; // operation; possible values: + - * /

	// input 2 numbers
	cout << "x op y ? ";
	cin >> operand1 >> operation >> operand2;

	//compute their sum, difference, product or quotient
	if (operation == '+')
		sum = operand1 + operand2; //NOTE the semicolon
	else if (operation == '-')
		difference = operand1 - operand2;
	else if (operation == '*')
		product = operand1 * operand2;
	else if (operation == '/')
		quotient = operand1 / operand2;

	//show results
	cout << operand1 << ' ' << operation << ' ' << operand2 << " = ";
	if (operation == '+')
		cout << sum;
	else if (operation == '-')
		cout << difference;
	else if (operation == '*')
		cout << product;
	else if (operation == '/')
		cout << quotient;

	cout << endl;

	return 0;
}

// A little better solution but still bad ...
// ... for several reasons.
// See next solutions.


=========================================================================
```

```
/*
 AN EVEN BETTER SOLUTION. WHY ?

 GOOD CODING STYLES:
 - TRY TO KEEP INPUT, PROCESSING AND OUTPUT SEPARATED FROM EACH OTHER
 - USE CONSTANTS INSTEAD OF "MAGIC NUMBERS"
*/
#include <iostream>
#include <iomanip>  //needed for stream manipulators: fixed, setprecison

using namespace std;

int main()
{
	const unsigned int RESULT_PRECISION = 3;  //for const's use uppercase

	double operand1, operand2;  // input operands
	char operation; // operation; possible values: + - * /
	double result; // result of "operand1 operation operand2"
	bool validOperation = false; // operation is not + - * or /

	// input 2 numbers
	cout << "x op y ? ";
	cin >> operand1 >> operation >> operand2;

	// compute result if operation is valid
	if (operation == '+' || operation == '-' || operation == '*' ||
operation == '/') // TO DO: remember operator precedence
	{
		//compute their sum, difference, product or quotient
		if (operation == '+')
			result = operand1 + operand2;
		else if (operation == '-')
			result = operand1 - operand2;
		else if (operation == '*')
			result = operand1 * operand2;
		else if (operation == '/')
			result = operand1 / operand2;
		validOperation = true;
	}

	//show result or invalid input message
	if (validOperation)  // equivalent to: if (validOperation == true)
	{
		cout << operand1 << ' ' << operation << ' ' << operand2 << " =
";
		cout << fixed << setprecision(RESULT_PRECISION) << result <<
endl;
	} //TO DO: search for other stream manipulators
	else
		cerr << "Invalid operation !\n"; // NOTE:stream for error output

	return 0;
}
```

==========

TO DO BY STUDENTS: search how to reset precision to default values

```
const std::streamsize oldp = cin.precision();
cout << fixed << setprecision(3) << x << endl << setprecision(oldp) << x;
```

```
/*
USING THE SWITCH STATEMENT
*/

#include <iostream>
#include <iomanip>

using namespace std;

int main()
{
	const unsigned RESULT_PRECISION = 3;

	double operand1, operand2;  // input operands
	char operation; // operation; possible values: + - * /
	double result;  // result of "operand1 operation operand2"
	bool validOperation = true; // operation is + - * or /

	// input 2 numbers
	cout << "x op y ? ";
	cin >> operand1 >> operation >> operand2;

	// compute result if operation is valid
	switch (operation)
	{
	case '+':
		result = operand1 + operand2;
		break;
	case '-':
		result = operand1 - operand2;
		break;
	case '*':
		result = operand1 * operand2;
		break;
	case '/':
		result = operand1 / operand2;
		break;
	default:
		validOperation = false;
	}

	//show result or invalid input message
	if (validOperation)
	{
		cout << operand1 << ' ' << operation << ' ' << operand2 << " =
";
		cout << fixed << setprecision(RESULT_PRECISION) << result <<
endl;
	}
	else
		cerr << "Invalid operation !\n";

	return 0;
}
```

==========