Lecture 2.1

Topics

---

1. Data and Object – `rvalue, lvalue`

In a way, objects in computer programming refer to data.

In object-oriented programming, objects have both data and operations. Data will be moved and modified using handles, identifiers, variables, or objects.

- Data are stored in memory.
- One may or may not be able to access to these memory locations, and
- One may or may not be able to change the data at these locations.

Two terms are of interests in managing data: **`rvalue`** and **`lvalue`**.

1.1 `lvalue`

An **`lvalue`** is a name or expression that identifies a particular object (memory block).

Not all objects can have their values changed, for example, constant objects. While, a modifiable **`lvalue`** refers to memory blocks that can have the stored value changed.

A variable or object on the left-hand side of an assignment must be an **`lvalue`** in a correct statement.

1.2 `rvalue`

An **`rvalue`** refers to quantities that can be assigned to modifiable some **`lvalue`**.

An **`rvalue`** can be constant, variable/object, or result of some expression. In particular, **`rvalues`** can be combined in a multiple assignment statement as follows,

```
int iA;
int iB;
int iC;

iA = iB = iC = 9;
```

The above lines are also called C++ expressions and statements.

In general, a statement is an expression that is terminated with a semicolon. More discussion on expressions will be given later.

2. `cout` Object – Brief (Again)

In general, Input/Output is a sequence of bytes that is called a stream of bytes or just I/O stream. The bytes are usually characters (others may be binary for graphics or proprietary formatted data such as digital speech data). Thus, a stream can be thought of as a sequence of characters.

2.1 Input and Output Streams

There are two types of streams: input streams and output streams.

Input Stream – A sequence of characters from an input device to computer.

Output Stream – A sequence of characters from computer to an output device.

The `iostream` header file (or just header) has an `iostream` class, which is the derived or sub class from the two `istream` class and `ostream` class. In this header file, there are two object declarations, one for **`cout`** (common output, which is of type `ostream`), and one for **`cin`** (common input, which is of type `istream`) as follows,

```
ostream cout;
istream cin;
```

These two objects are also call stream variables. A stream variable is either an input stream variable or an output stream variable.

Since `cout` and `cin` are already defined and having specific usage, to avoid confusion we should not redefine these objects in programs.

2.2 Output and Formatting Output

Displaying the results is one of a programmer's goals. The output display is performed through cout and the insertion operator. The general form is given as follows,

```
cout << expression or manipulator [ << expression or manipulator… ];
```

where,

- **`expression`** is evaluated and its value is displayed, and
- **`manipulator`** is used to format the output.

Note that the expressions enclosed in the square brackets are optional.

There are many manipulators to control the formatting of the output. We will only look at a few that are of our interest: `endl`, `setprecision`, `fixed`, `showpoint`, `setw`, and `flush`.

3. Expressions – Primary, Binary, Assignment, Unary

Expressions are formed with combinations of **sub-expressions**, which can be identified as **primary group**, and **operator-operand combination group**.

We will look at these groups; and we also limit the discussion in the operator-operand group to just a few – binary, assignment operators, and some unary operators for now. Other operators will be introduced during the course of the semester.

3.1 Primary Expressions

Primary expression is the most basic type of expression. There are three primary expressions:

Identifiers	Any variable, function, defined name, object names **`iVar   cVar   compObj   getValue()`**
Constants	Any literal (face-valued) constants **`5   9.5   ‘a’   “Monday”`**
Parenthetical Expressions	Any complex expression can be enclosed in parentheses **`( 3 + 4 * 5 )   ( iVar = 5 * 4 – 1 )`**

3.2 Binary Expressions

A binary expression has an operator that requires two operands – the left operand (**`lOp`**) and right operand (**`rOp`**). The set of binary arithmetic operators may include addition, subtraction, multiplication, division, and modulo. There are other binary operators such as assignments, comparisons, logical, etc.

For examples,

```
iVar + 9

iVar % 2
```

3.3 Assignment Expressions

There are simple assignment and compound assignments. They are easy to understand as given in the following statements.

```
iVar = 5;
iVar = iVar + 9;
iVar += 15;              /* iVar = iVar + 15 */
iVar += iCount + 5;      /* iVar = iVar + ( iCount + 5 ) */
```

3.4 Unary – Postfix Expressions

There are two postfix expressions that are of our interest -- postfix increment and postfix decrement. The general expressions are given as follows,

```
variable++

variable--
```

The explanation is best through examples.

Example 1

```
/**
 *Program Name:  cis25L0211.cpp
 *Discussion:    Unary Expressions -- Postfix Operators
 */

#include <iostream>
using namespace std;

int main() {
  int iNum1;
  int iNum2 = 5;

  double dNum1;
  double dNum2 = 15.56;

  cout << "Enter an integer + ENTER : ";
  cin >> iNum1;

  cout << "\nValue of iNum1 : " << iNum1
       << "\nValue of iNum2 : " << iNum2 << endl;

  iNum1 += iNum2++;

  cout << "\nAfter evaluating iNum1 += iNum2++ :"
       << "\n\tValue of iNum1 : " << iNum1
       << "\n\tValue of iNum2 : " << iNum2 << endl;
```

```
    cout << "\nEnter a floating-point + ENTER : ";
    cin >> dNum1;

    cout << "\nValue of dNum1 : " << dNum1
         << "\nValue of dNum2 : " << dNum2 << endl;

    dNum1 += dNum2++;

    cout << "\nAfter evaluating dNum1 += dNum2++ :"
         << "\n\tValue of dNum1 : " << dNum1
         << "\n\tValue of dNum2 : " << dNum2 << endl;

    return 0;
  }
```

OUTPUT

```
Enter an integer + ENTER : 9

Value of iNum1 : 9
Value of iNum2 : 5

After evaluating iNum1 += iNum2++ :
        Value of iNum1 : 14
        Value of iNum2 : 6

Enter a floating-point + ENTER : 3.6

Value of dNum1 : 3.6
Value of dNum2 : 15.56

After evaluating dNum1 += dNum2++ :
        Value of dNum1 : 19.16
        Value of dNum2 : 16.56
```

Discussion of the above output will be given in class. Note that the value of `iNum2` and `dNum2` are used in the expressions and then updated.

3.5 Unary – Prefix Expressions

There are two prefix expressions that are of our interest

- Prefix increment, and
- Prefix decrement.

The general expressions are given as follows,

```
++variable

--variable
```

Example 2

```
/**
 *Program Name:  cis25L0212.cpp
 *Discussion:    Unary Expressions -- Prefix Operators
 */

#include <iostream>
using namespace std;

int main() {
  int iNum1;
  int iNum2 = 5;

  double dNum1;
  double dNum2 = 15.56;

  cout << "Enter an integer + ENTER : ";
  cin >> iNum1;

  cout << "\nValue of iNum1 : " << iNum1
       << "\nValue of iNum2 : " << iNum2 << endl;

  iNum1 += ++iNum2;

  cout << "\nAfter evaluating iNum1 += ++iNum2 :"
       << "\n\tValue of iNum1 : " << iNum1
       << "\n\tValue of iNum2 : " << iNum2 << endl;

  cout << "\nEnter a floating-point + ENTER : ";
  cin >> dNum1;

  cout << "\nValue of dNum1 : " << dNum1
       << "\nValue of dNum2 : " << dNum2 << endl;

  dNum1 += --dNum2;

  cout << "\nAfter evaluating dNum1 += --dNum2 :"
       << "\n\tValue of dNum1 : " << dNum1
       << "\n\tValue of dNum2 : " << dNum2 << endl;

  return 0;
}
```

OUTPUT

```
Enter an integer + ENTER : 6

Value of iNum1 : 6
Value of iNum2 : 5

After evaluating iNum1 += ++iNum2 :
        Value of iNum1 : 12
        Value of iNum2 : 6

Enter a floating-point + ENTER : 3.6

Value of dNum1 : 3.6
Value of dNum2 : 15.56

After evaluating dNum1 += --dNum2 :
        Value of dNum1 : 18.16
        Value of dNum2 : 14.56
```

What do you think about the following statements?

```
int iNum;
iNum = 5;
++iNum--;
```

4. Operators – Precedence, Associativity, and Side Effects

The rules of precedence and associativity are important because they provide the instructions for identifying and evaluating expressions.

> Precedence is used to determine the order in which different operators in a complex expression are evaluated.
>
> Associativity is used to determine the order in which operators with the same precedence are evaluated in a complex expression.

4.1 Precedence

The operator precedence would resolve many of the possible confusion when trying to obtain the result of some expressions. Given an expression with combination of several operators and operands, one can use the following rules:

(i) Starting from the left, scan through the expression to identify the operators,

(ii) Mark the operators with their rankings based on the list given in previous lecture,

(iii) Starting with the operators with highest rank to lowest rank, identify their operand(s),

(iv) Use parentheses to group operator and operand(s) for evaluation,

(v) Evaluate all possible parenthetical expressions,

(vi) Repeat Steps (i) through (v) until a single result is produced.

Examples

```
2 + 3 * 4       →    ( 2 + ( 3 * 4 ) )
-iNum++         →    ( - ( iNum++ ) )
```

The above expressions have operators with different level of precedence. They can be interpreted quite easily.

4.2 Associativity

Associativity can be from the Left-to-Right or Right-to-Left and it should be used to apply to operators and operands of the same level of precedence in an expression.

4.2.1 Left-to-Right Associativity

The illustration is given in the following example.

```
3 * 8 / 4 % 4 * 5                                   (Expr. 1)
```

All the operators have the same precedence and left-to-right associativity. Applying the above steps to the expression will result in the following parenthesized expression:

```
( ( ( ( 3 * 8 ) / 4 ) % 4 ) * 5 )                   (Expr. 2)
```

4.2.2 Right-to-Left Associativity

There are only three types of expressions that have the right-to-left associativity: (most of) the unary expressions, the conditional ternary expression, and the assignment expressions. Let’s consider the assignment expressions here and the other expressions will be discussed in future lectures.

When several assignment operators are used in an expression, the association must start from right to left. That means the rightmost expression will be evaluated first; its result will be assigned to the left operand of the next expression for subsequent evaluation.

An example is given below,

```
a += b *= c -= 5                                    (Expr. 3)
```

The final parenthesized expression is found to be as follows,

```
( a += ( b *= ( c -= 5 ) ) )                        (Expr. 4)
( a = a + ( b = b * ( c = c – 5 ) ) )               (Expr. 5)
```

Thus, the result can be found easily from the last expression.

4.3 Side Effects

In general, a side effect is an action that would change or update a value after evaluating an expression. Changing value of a variable is a side effect. Any changes to the state of the program are also called side effects, such as displaying data on screen.

The side effects can be either pre-effects or post-effects. There are four (4) pre-effects – prefix increment, prefix decrement, function call, and the assignment. The side effect for these expressions takes place before the expression is evaluated.

There are two (2) post-effects – postfix increment, and postfix decrement. The side effect for these expressions takes place after the expression has been evaluated. Therefore, the variable value is not changed until after it has been used in the expression.

Consider the following statement:

```
iX = 4;
```

This statement has a simple expression with two primary expressions and an assignment operator. This simple expression will have value 4 with a side effect of variable `iX` receives the value 4.

The assignment statement will have a value for the expression, which is the same as the value on the right of the equal sign. This value is being assigned to the variable on the left of the equal sign. The following program will display the results.

Example 3

```
/**
 *Program Name:  cis25L0213.cpp
 *Discussion:    Simple Function
 */

#include <iostream>
using namespace std;

//Function Prototypes
void printClassInfo( void );

void evaluateExpr( void );

int main(){
  printClassInfo();

  evaluateExpr();

  return 0;
}

//Function Definitions
/**
```

```
 *Function Name: printClassInfo()
 *Description:   To print class information
 *Pre        :   Nothing (no argument required)
 *Post:      :   Displaying class information on screen
 */
void printClassInfo( void ) {
  cout << "Class Information --"
       << "\n  CIS 25 -- C++ Programming"
       << "\n  Laney College" << endl;
  return;
}

/**
 *Function Name: evaluateExpr()
 *Description:   To evaluate expression
 *Pre        :   Nothing (no argument required)
 *Post:      :   Displaying results of expressions
 */
void evaluateExpr( void ) {
  int iVar;

  cout << "\nEnter an integer + ENTER : ";
  cin >> iVar;

  cout << "\n\t(1) Value of iVar : " << iVar << endl;
  cout << "\n\t(2) Value of iVar + 4 : " << iVar + 4 << endl;
  cout << "\n\t(3) Value of iVar = iVar + 4 : "
       << ( iVar = iVar + 4 ) << endl;
  cout <<  "\n\t(4) Value of iVar : " << iVar << endl;

  return;
}
```

OUTPUT

```
Class Information --
  CIS 25 -- C++ Programming
  Laney College

Enter an integer + ENTER : 5

        (1) Value of iVar : 5

        (2) Value of iVar + 4 : 9

        (3) Value of iVar = iVar + 4 : 9

        (4) Value of iVar : 9
```

In the above example, the values of the primary and sub-expression(s) are displayed. They reflect the intermediate or final values that may be produced while evaluating the statement.

4.4 C++ Operators

A partial list of C++ operators is given below.

Precedence	Operator	Operation	Association
1	( )	Parentheses : Parameter Evaluation, Function (Method) Invocation/Call	L to R
	[ ]	Array indexing	L to R

	.	Member reference	L to R
	++	Unary postfix increment	R to L
	--	Unary postfix decrement	R to L
2	++	Unary prefix increment	R to L
	--	Unary prefix decrement	R to L
	~	Bitwise NOT	R to L
	!	Logical NOT	R to L
3	sizeof	Size in bytes of a type	R to L
	*	De-reference	R to L
	&	Address of	R to L
	(type)	Type casting	R to L
4	*	Multiplication	L to R
	/	Division	L to R
	%	Remainder	L to R
5	+	Addition	L to R
	-	Subtraction	L to R
6	<	Less than	L to R
	<=	Less than or equal to	L to R
	>	Greater than	L to R
	>=	Greater than or equal to	L to R
7	==	Equal to	L to R
	!=	Not equal to	L to R
8	&	Boolean AND	L to R
9	^	Boolean exclusive OR (XOR)	L to R
10	|	Boolean OR	L to R
11	&&	Logical AND	L to R
12	||	Logical OR	L to R
13	? :	Ternary conditional operator	R to L
14	=	Assignment	R to L
	+=	Addition, then assignment	R to L
	+=	String concatenation, then assignment	R to L
	-=	Subtraction, then assignment	R to L
	*=	Multiplication, then assignment	R to L
	/=	Division, then assignment	R to L
	%=	Remainder, then assignment	R to L

5. Expression Evaluations

The evaluation of an expression will depend on the kind of elements in the expression. Some elements may introduce no side effects during the course of evaluating the expression; and some may introduce side effects.

The process of evaluating an expression is given in the steps described previously. Let's simplify the process a bit just to show the important steps and the result, while some of the details are left out but can be performed using the (above) given rules.

Example   Expression with **no side effects**

```
Let
     iX = 3    iY = 4    iZ = 5
Evaluate
     iX * 4 + iY / 2 – iZ * iY
     This is an expression that has no side effect.
Step 1
     Replacing all variables by their values
     3 * 4 + 4 / 2 – 5 * 4
Step 2
     Using precedence and evaluating proper operations
     12 + 2 - 20
Step 3
     Simplifying and obtaining a final value
     -6
```

Example   Expression with **side effects**

```
Let
     iX = 3    iY = 4    iZ = 5

Evaluate
     --iX * ( 3 + iY ) / 2 – iZ++ * iY
     This is an expression with two side effects.

Pre-Evaluating Substitution Rules
     (1)  Copy any prefix increment or decrement expressions, and
          place them before the expression being evaluated. Replace
          each removed expression with its variable.
     (2)  Copy any postfix increment or decrement expressions, and
          place them after the expression being evaluated. Replace
          each removed expression with its variable.

Apply Substitution Rules & Evaluate Proper Operations
     With
          iX = 3    iY = 4    iZ = 5
     Evaluate
          --iX
     Produce
          iX = 2    iY = 4    iZ = 5
          iX * ( 3 + iY ) / 2 – iZ * iY
          iZ++

     Evaluate
          iX = 2    iY = 4    iZ = 5
          2 * ( 3 + 4 ) / 2 – 5 * 4
          iZ++

     Evaluate
          iX = 2    iY = 4    iZ = 5
          -13
          iZ++

     Final Results
```

```
Expression Value : -13
Variable Values : iX = 2  iY = 4    iZ = 6
```

Note that if a variable is being modified more than one then the expression value may not be evaluated with any certainty. In this case, rewrite the expression or come up with a different logic to be expressed in another version of the expression.

6. Mixed Type Expressions

There are many expressions that involve operators and their operands, which may have different types such as integer, floating-point, etc. How would C handle these mixed type expressions?

C will follow several rules to perform the conversion of data (values) during the evaluation process: (1) Simple assignment rule, (2) Implicit data conversion, and (3) Explicit data Conversion.

6.1 Simple Assignment Rule

For an assignment expression, the final expression value must have the same type of the left operand, which is the operand that receives the (final) value.

6.2 Implicit Data (Type) Conversion

For many expressions, C++ will automatically convert any intermediate values to the "proper" type so that the expression can be evaluated. This is known as implicit data (type) conversion.

C++ uses the rule that, in all expressions except assignments, any implicit data conversions will always be made to the more general type according to the following order of promotion:

```
char          →  short     →  int  →
unsigned int  →  long int  →  unsigned long int  →
float         →  double    →  long double
```

For examples,

```
iNum     iNum / 2.0
         2.0 * iNum / 4
```

6.3 Explicit Data (Type) Conversion

The cast operator "**`( Type )`**" can be used to convert a value to a particular type explicitly.

For examples,

```
int iNum;

static_cast< double > ( iNum );
static_cast< double > ( iNum );
( static_cast< double > ( iNum ) ) / 2;
```

What about the following expression?

```
static_cast< double > ( iNum / 2 );
```