MECH215
Lecture Notes 1

Assignment Operator In C++

The assignment operator in C++ is the = symbol. The assignment operator STORES a result into a variable. The general form of the assignment operator is the following:

variable = expression;

expression can be a literal value, another variable, or any syntactically correct expression in C++. Several examples are:

```
int x = 0;  // declare an integer x and initialize it to 0
x = 10;     // give x a new value of 10
```

A more complex example is:

```
float grossSalary;
float taxRate;
float takeHomePay;
float giveToGovermentToPayForSocialPrograms;

giveToGovermentToPayForSocialPrograms = taxRate * grossSalary;
takeHomePay = grossSalary - giveToGovermentToPayForSocialPrograms;
```

Assignment Conversions

In an assignment statement, the left and right hand operators should be of the same type, but they do not have to be.

```
int x;
float y;
double z;

x = 6;
y = 2.3;
z = 5.678;
```

In the above expressions, the targets of the three assignment statements match the values which are being assigned to them. (6 is an integer literal, 2.3 and 5.678 are valid float and double literal expressions respectively).

In the case where the value of an assignment does not match the target type, the compiler will attempt to convert the right hand value so that its type MATCHES that of the target.

Consider the following program:

```
// double2Integer.cpp
#include <iostream>
#include <string>

using namespace std;

int main() {

    int x;          // declare an integer variable called x
    x = 4.56;       // assign a floating point value to an integer

    cout << "The value of x after conversion is " << x << endl;
    return 0;

}
```

The program produces the following output:

```
The value of x after conversion is 4
```

Since 4.56 is a double constant, the assignment operator first converts the 4.56 into its integer equivalent 4 (by truncating the fractional portion, it does not round).

Assignment conversion can lead to some unexpected results when the value of the right hand operand is larger than can fit into the left-hand target:

```
// integer2Short.cpp
#include <iostream>
#include <string>

using namespace std;

int main() {
    short someShort;
    int   someInteger;

    someInteger = 234456;
    someShort   = someInteger;

    cout << "someInteger = " << someInteger << endl;
    cout << "someShort   = " << someShort   << endl;

    return 0;
}
```

It produces the following output:

```
someInteger = 234456
someShort   = -27688
```

If we give the following initial values:

```
someInteger = 65536;
someShort   = someInteger;
```

and recompile and re-run the program, the following output results:

```
someInteger = 65536
someShort   = 0
```

The reason is the following. The size of an int is 4 bytes and the size of a short is 2 bytes. In binary the value of 65536 is

```
00000000 00000001 00000000 00000000
```

This is stored in memory as:

```
--------
00000000    someInteger
--------
00000000
--------
00000000
--------
00000001
```

The conversion of the int to a short takes the first 2 bytes, so someShort is assigned the value of 0. Please note that the actual ordering of the bytes stored in memory depends upon the compiler and the underlying computer hardware upon which the program is run. The above example is only for illustrative purposes and may not correspond the the exact way the binary information is stored in a particular's computer memory.

Assignment Precedence and Associativity

C++ allows for expressions of the following form:

```
int firstVar, secondVar, thirdVar;
thirdVar = 1;

firstVar = secondVar = thirdVar + 2;
```

How do we interpret the above expression? The answer lies in the precedence and associativity of the assignment (=) operator. Notice that the operand thirdVar is surrounded by two different operators (the = operator and the + operator). Since the PRECEDENCE of the + operator is higher than that of the = operator, the addition is performed
first:

```
firstVar = secondVar = (thirdVar + 2);
```

Next, we note that secondVar is surrounded by the = operator on both sides. The associativity of the assignment operator determines which one is to be performed first. Unlike the arithmetic operators, the assignment operator is RIGHT associative. This means the the original expression is to be interpreted as:

```
firstVar = (secondVar = (thirdVar + 2));
```

The final values of the variables are:

firstVar = 3
secondVar = 3
thirdVar = 1

This interpretation of the associativity of the assignment operator is quite natural, since if the = operator were to be left associative we would arrive at the following nonsensical expression:

```
(firstVar = secondVar) = (thirdVar + 2);
```

Compound Operators

In programming it is very common to perform operations of the sort:

```
i = i + 1;
j = j - 5;
k = k * 4;
l = l / 2;
m = m % 2;
```

The above 5 examples all share the following common feature, an arithmetic operator is being performed on a variable and then the result is stored back in the variable.

C++ allows for the following SHORTHAND NOTATION:

```
i += 1;
j -= 5;
k *= 4;
l /= 2;
m %= 2;
```

The two versions are equivalent as shown by this following C++ program:

```
// Author: Ted Obuchowicz and John Peach
// compoundOperator.cpp
// Example program illustrating use of compound operators

#include <iostream>
#include <string>

using namespace std;

int main() {

    // Initialize
    int i = 0;
    int j = 10;
    int k = 2;
    int l = 8;
    int m = 7;

    // Modify using the "long hand versions"
    i = i + 1;
    j = j - 5;
    k = k * 4;
    l = l / 2;
    m = m % 2;

    // Display the new values
    cout << "i = " << i << endl;
    cout << "j = " << j << endl;
    cout << "k = " << k << endl;
    cout << "l = " << l << endl;
    cout << "m = " << m << endl;

    // assign back the initial values
    i = 0;
    j = 10;
```

```
    k = 2;
    l = 8;
    m = 7;

    // modify them using the shorthand compound operators
    i += 1;
    j -= 5;
    k *= 4;
    l /= 2;
    m %= 2;

    // display the new values
    cout << "i = " << i << endl;
    cout << "j = " << j << endl;
    cout << "k = " << k << endl;
    cout << "l = " << l << endl;
    cout << "m = " << m << endl;

    return 0;

}
```

Increment and Decrement Operators

C++ has two special operators for incrementing and decrementing a variable. The increment operator is ++ and the decrement operator is --. Can you guess the origin of the name C++ (C++ is C incremented with extra features).

For example:

i++ is the same as i += 1 which is the same as i = i + 1

j-- is the same as j -= 1 which is the same as j = j - 1.

Both the increment and decrement operators come in two variations. PREFIX and POSTFIX (the two above examples are both postfix versions).

The following sample program illustrates the differences between the prefix and postfix forms of the increment and decrement operators:

```
// Author: Ted Obuchowicz and John Peach
// prepostfix.cpp
// Example program illustrating use of increment and decrement
operators in both postifx and prefix variations

#include <iostream>
```

```
#include <string>

using namespace std;

int main() {
    int i = 0;      // dummy variable
    int k = 5;      // dummy variable

    // Add 1 to i after it is output for the first time
    cout << "i is " << i++ << endl;
    cout << "i is " << i << endl;

    // Subtract 1 from k before it is output
    cout << "k is " << --k << endl;
    cout << "k is " <<  k << endl;

}
```

The program produces the following output:

```
i is 0
i is 1
k is 4
k is 4
```

One can conclude that the POSTFIX version of the increment and decrement operators first use the value, then modify it. The PREFIX version first modifies the value and then returns the modified value were it can be subsequently used.

Here is another program which illustrates the differences in the postfix and prefix notations:

```
// Author: Ted Obuchowicz and John Peach
// prepostfix2.cpp
// Another example program illustrating use of increment and
decrement operators in both postifx and prefix variations

#include <iostream>
#include <string>

using namespace std;

int main() {

    int i = 1;      // dummy variables
    int j = 4;
    int k;
```

```
    int l;
    int m = 4;

    k = i * j++;    // Will k be 1 * 4 then set j = 5 or set j=5 and
then 1 * 5
    l = i * --m;    // Will l be 1 * 4 then set m=3 or set m=3 and
then 1 * 3

    cout << "i  = " << i << " k = " << k << " j = " << j  << endl;
    cout << "i  = " << i << " l = " << l << " m = " << m << endl;

    return 0;

}
```

The program output is:

```
i  = 1 k = 4 j = 5
i  = 1 l = 3 m = 3
```

Note that in the expression k = i * j++ that the current value of j (which is 4) is first used in computing the value of i*j and this is then assigned to k. After the assignment to k, the value of j is then incremented to 4.

Similarly, in the expression l = i * --m, the value of m is first decremented from 4 to 3 and then the decremented value is used to calculate the value of 1 * 3, which is assigned to the variable l.

Increment and Decrement Operators for float, double and long double

Both versions of the increment and decrement operators can be applied to the floating point data types: float, double, and long double. They have the effect of adding or subtracting 1.0 from a floating point variable:

```
// Author: Ted Obuchowicz and John Peach
// incrementFloatingPoint.cpp
// Example program illustrating use of increment and decrement
operators  with floating point numbers

#include <iostream>
#include <string>

using namespace std;

int main() {
    float  i = 0.2;
    double k = 5.8;
```

```
    cout << "i is " << i++ << endl;
    cout << "i is " << i << endl;
    cout << "k is "  << --k << endl;
    cout << "k is "  <<  k << endl;

}
```

The program output is:

```
i is 0.2
i is 1.2
k is 4.8
k is 4.8
```

Incrementing a char Variable

The ++ and -- operators can be applied to char variables as well with the expected results that the result is equal to the next (or previous) character in the ASCII collating sequence. Each character has a number assigned to it and it association is commonly referred to as the ASCII value. From the point of view of C++ a character is just a number.

Consider the following program:

```
// Author: Ted Obuchowicz and John Peach
// incrementChar.cpp
// Example program illustrating use of incrementing and decrementing
a character variable

#include <iostream>
#include <string>

using namespace std;

int main() {

    char letter = 'a';      // note the use of the single quotes

    cout << "letter is " << letter++ << endl;
    cout << "letter incremented once is " << letter++ << endl;
    cout << "letter incremented another time is " << letter << endl;
    return 0;
}
```

Incrementing letter once yields 'b' as the result. Incrementing yet again yields 'c'. Here is the program output:

```
letter is a
letter incremented once is b
letter incremented another time is c
```

The Cast Operator

Consider the task of calculating the average value of two integers. It seems simple enough, read in the two integers, calculate the value of their sum divided by two and assign this to another variable and output the result.

Here is a first attempt:

```
// Author: Ted Obuchowicz and John Peach
// meanNoCast.cpp
// Example program illustrating use of the cast operator
// This program reads in two integer values and computes their
average

#include <iostream>
#include <string>

using namespace std;

int main() {

    int firstInt;           // The first number that the user enters
    int secondInt;          // The second number that the user enters
    float mean;             // The mean of firstInt and secondInt

    // Read in the values from the user
    cout << "Input an integer number ";
    cin >> firstInt;
    cout << "Input a second integer number ";
    cin >> secondInt;

    // Calculate the average value
    mean = (firstInt + secondInt) / 2;

    cout << "The average value is " << mean << endl;

    return 0;
}
```

When executed, the output is:

```
Input an integer number 3
Input a second integer number 2
The average value is 2
```

What went wrong, we know that the average of 3 and 2 is 2.5 not 2. The answer lies in the method that is used to calculate the value of mean = (firstInt + secondInt) / 2;

In this expression, firstInt, secondInt, and the literal 2 are all integer values, hence the result of the expression is an integer value. The expression is truncated from 2.5 to 2. The assignment operator then converts this integer 2 into a float 2. This is not the desired result, we wish that that the arithmetic expression be somehow performed in floating point arithmetic. A solution is to force the expression to be performed in floating point arithmetic by making one of the operands a floating point expression:

```
// Author: Ted Obuchowicz and John Peach
// meanCast.cpp
// Example program illustrating use of the cast operator
// This program reads in two integer values and computes their
average

#include <iostream>
#include <string>

using namespace std;

int main() {

    int firstInt;           // The first number that the user enters
    int secondInt;          // The second number that the user enters
    float mean;             // The mean of firstInt and secondInt

    // Read in the values from the user
    cout << "Input an integer number ";
    cin >> firstInt;
    cout << "Input a second integer number ";
    cin >> secondInt;

    // Calculate the average value
    mean = (float)(firstInt + secondInt) / 2;

    cout << "The average value is " << mean << endl;

    return 0;
}
```

The result is now:

```
Input an integer number 3
Input a second integer number 2
The average value is 2.5
```

In the expression mean = (float)(firstInt + secondInt) / 2 , the (float) portion is called a CAST. It causes the compiler to convert the integer 1 into its equivalent float representation. The rules of arithmetic in C++ then cause the addition to be performed in floating point arithmetic and thus the division is also performed in floating point arithmetic yielding a floating point result.