**Implicit Type Conversion (Dale & Weems text Chapter 3 pgs 99-101)**

**CONCEPT:** When an operator’s operands are of different data types, C++ will automatically convert them to the same data type. This can affect the results of mathematical expressions.

If a floating-point value is assigned to an int variable, what value will the variable receive? If an int is multiplied by a float, what data type will the result be? What if a double is divided by an unsigned int? Is there any way of predicting what will happen in these instances? The answer is yes. C++ follows a set of rules when performing mathematical operations on variables of different data types. It’s helpful to understand these rules to prevent subtle errors from creeping into your programs.

**Data Type Ranking**long double   
double  
float  
unsigned long   
long  
unsigned int  
int  
unsigned short   
short  
char

One exception to the ranking in Table 3-7 is when an int and a long are the same size. In that case, an unsigned int outranks long because it can hold a higher value.

When C++ is working with an operator, it strives to convert the operands to the same type. This implicit, or automatic, conversion is known as type coercion. When a value is converted to a higher data type, it is said to be promoted. To demote a value means to convert it to a lower data type. Let’s look at the specific rules that govern the evaluation of mathematical expressions.

**Rule 1: char, short, and unsigned short are automatically promoted to int.**

Anytime these data types are used in a mathematical expression, they are automatically promoted to an int \*

**Rule 2: When an operator works with two values of different data types, the lower-ranking value is promoted to the type of the higher-ranking value.**

In the following expression, assume that years is an int and interestRate is a double:

years \* interestRate

Before the multiplication takes place, the value in years will be promoted to a double.

**Rule 3: When the final value of an expression is assigned to a variable, it will be converted to the data type of that variable.**

\* The only exception to this rule is when an unsigned short holds a value larger than can be held by an int. This can happen on systems where a short is the same size as an int. In this case, the unsigned short is promoted to unsigned int.

**Explicit Type Conversion**

**CONCEPT:** Type casting allows you to explicitly perform data type conversion.

A type cast expression lets you manually promote or demote a value. The general format of a type cast expression is  
static cast<DataType> (Value)  
where Value is a variable or literal value that you wish to convert and DataType is the data type you wish to convert it to. Here is an example of code that uses a type cast expression:

double number = 3.7; int val;  
val = static cast<int>(number);// type\_casting2.cpp - This program uses a type cast to avoid an integer division.

#include <iostream>

using namespace std;

int main()

{

int books, months;

double booksPerMonth;

// Get user inputs

cout << "How many books do you plan to read? ";

cin >> books;

cout << "How many months will it take you to read them? ";

cin >> months;

// Compute and display books read per month

booksPerMonth = static\_cast<double>(books) / months;

cout << "That is " << booksPerMonth << " books per month.\n";

return 0;

}

//Program typeconversion.cpp illustrates the prestandard C++ form of type casting

#include <iostream>

#include <iomanip> // For setw() and setprecision()

using namespace std;

int main()

{

float someFloat = 12.0;

float anotherFloat = 5.5;

int someInt = 4;

cout << fixed << showpoint; // Set up floating pt. output format

someFloat = 3 \* someInt + 2; //type coercion

cout << someFloat <<endl;

someFloat = float (3\*someInt + 2); //type casting or conversion

cout << someFloat <<endl;

someInt = 5.2 /someFloat - anotherFloat; //type coercion

cout <<someInt <<endl;

someInt = int(5.2 /someFloat - anotherFloat); //type casting or conversion

cout <<someInt <<endl;

system("PAUSE");

return 0;

}