LAB 05: Named Constants and Type Casting

CLOs: CL01, CLO4

Named constants

A named constant is like a variable, but its content is read-only, and cannot be changed while the program is running. Here is a definition of a named constant:

const double INTEREST_RATE = 0.129;

It looks just like a regular variable definition except that the word const appears before the data type name, and the name of the variable is written in all uppercase characters. The key word const is a qualifier that tells the compiler to make the variable read-only. Its value will remain constant throughout the program's execution. It is not required that the variable name be written in all uppercase characters, but many programmers prefer to write them this way so they are easily distinguishable from regular variable names.

Example:

```
// This program calculates the area of a circle.
// The formula for the area of a circle is PI times
// The radius squared. PI is 3.14159.
#include <iostream>
#include <math.h> // needed for pow function
using namespace std;
int main()
{
    const double PI = 3.14159;
    double area, radius;
    cout << "This program calculates the area of a circle.\n";
    cout << "What is the radius of the circle? ";
    cin >> radius;
    area = PI * pow(radius, 2.0);
    cout << "The area is " << area << endl;
    return 0;
}
```

Output:

```
This program calculates the area of a circle.
What is the radius of the circle? 2
The area is 12.56
```

#define preprocessor directive

The #define preprocessor directive creates symbolic constants. The symbolic constant is called a macro and the general form of the directive is −

#define macro-name replacement-text

When this line appears in a file, all subsequent occurrences of macro in that file will be replaced by replacement-text before the program is compiled. For example −

```
#include <iostream>
using namespace std;
#define PI 3.14159
int main () {
  cout << "Value of PI :" << PI << endl;

  return 0;
}
```

Formatting output:

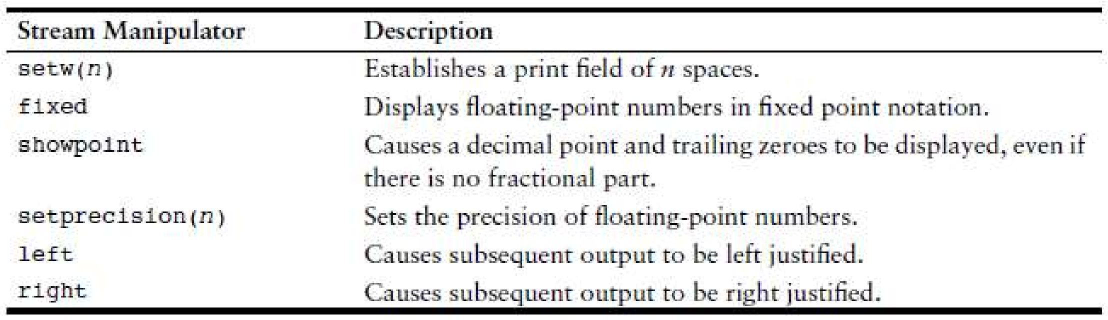
The same data can be printed or displayed in several different ways. For example, all of the following numbers have the same value, although they look different:

720
720.0
720.00000000
7.2e+2
+720.0

The way a value is printed is called its formatting. The *cout* object has a standard way of formatting variables of each data type. Sometimes, however, you need more control over the way data is displayed. For formatting manipulator functions we use <*iomanip*> header file in C++ program. There are different types of manipulator functions:

setw
setprecision
fixed
showpoint
left and right
setfill

Table 3-12



Stream Manipulator	Description
`setw(n)`	Establishes a print field of *n* spaces.
`fixed`	Displays floating-point numbers in fixed point notation.
`showpoint`	Causes a decimal point and trailing zeroes to be displayed, even if there is no fractional part.
`setprecision(n)`	Sets the precision of floating-point numbers.
`left`	Causes subsequent output to be left justified.
`right`	Causes subsequent output to be right justified.

Example:

```
 incude <iostream>
#include <iomanip>

using namespace std;

void main( )
{
        double no=100.56;
        cout<<setw(10)<<setfill('*')<<"R"<<endl;
        cout<<fixed<<setprecision(8)<<no <<endl;
        system("pause");
}
```

Output:

```
*********R
100.56000000
```

Overflow and underflow:

Because data types do have a set minimum to maximum range and a set maximum precision, you cannot represent or store every possible number in standard computer variables.

Overflow

Overflow is the situation where you try to store a number that exceeds the value range for the data type. When you try to store too large of a positive or negative number, the binary representation of the number (remember that all values are stored as a 0 and 1 pattern) is corrupted and you get a meaningless or erroneous result.

Underflow

Underflow occurs in floating point numbers and is the situation where in numbers very close to zero; there are not enough significant digits to represent the number exactly.

The example below includes two macro constants, INT_MAX from climits and DBL_MIN from cfloat.

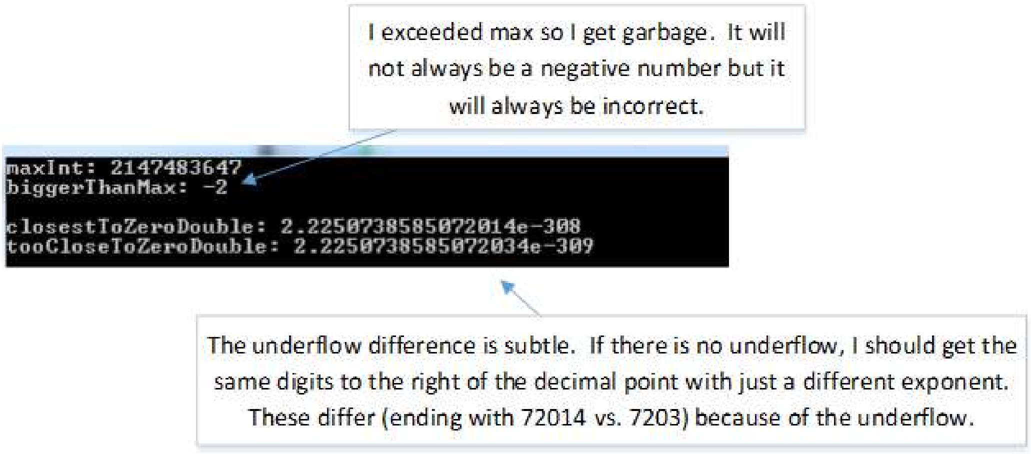
INT_MAX: maximum value for an object of type int

DBL_MIN: minimum representable floating point number for an object of type double

The program

```
#include <climits> // Added to access macro constants
#include <cfloat>
#include <iostream>
#include <iomanip>
using namespace std;
int main()
{
         int maxInt = INT_MAX; // Initialize as largest possible integer
        int biggerThanMax = maxInt * 2; // make too big of an integer
        cout << "maxInt: " << maxInt << endl;
         cout << "biggerThanMax: " << biggerThanMax << endl;

        double closestToZeroDouble = DBL_MIN; // Initialize as smallest possible
        double
        double tooCloseToZeroDouble = closestToZeroDouble / 10.0; // shift decimal
        point closer to 0 (1 to left)

        cout << endl; // insert a blank line
        cout << setprecision(30); // adjusting number of digits displayed

        cout << "closestToZeroDouble: " << closestToZeroDouble << endl;
        cout << "tooCloseToZeroDouble: " << tooCloseToZeroDouble << endl;

  return 0;
}
```

The output



I exceeded max so I get garbage. It will not always be a negative number but it will always be incorrect.

```
maxInt: 2147483647
biggerThanMax: -2

closestToZeroDouble: 2.2250738585072014e-308
tooCloseToZeroDouble: 2.2250738585072034e-309
```

The underflow difference is subtle. If there is no underflow, I should get the same digits to the right of the decimal point with just a different exponent. These differ (ending with 72014 vs. 7203) because of the underflow.

Type Casting:

Typecasting is making a variable of one type, such as an int, act like another type, a char, for one single operation. To typecast something, simply put the type of variable you want the actual variable to act as inside parentheses in front of the actual variable. (char)a will make 'a' function as a char.

```
#include <iostream>
using namespace std;
int main()
{
        cout<< (char)65 <<"\n";
        // The (char) is a typecast, telling the computer to interpret the 65 as a
        //  character, not as a number.  It is going to give the character output of
        //  the equivalent of the number 65 (It should be the letter A for ASCII).
         cin.get();
}
```

One use for typecasting for is when you want to use the ASCII characters. For example, what if you want to create your own chart of all 128 ASCII characters. To do this, you will need to use to typecast to allow you to print out the integer as its character equivalent.

```
using namespace std;
int main()
{
        for ( int x = 0; x < 128; x++ ) {
                cout<< x <<". "<< (char)x <<" ";
                //Note the use of the int version of x to
                // output a number and the use of (char) to
                 // typecast the x into a character
                // which outputs the ASCII character that
                // corresponds to the current number
         }
        cin.get();
}
```

This is more like a function call than a cast as the type to be cast to is like the name of the function and the value to be cast is like the argument to the function. Next is the named cast, of which there are four:

```
int main()
{
  cout<< static_cast<char> ( 65 ) <<"\n";
  cin.get();
}
```

static_cast is similar in function to the other casts described above, but the name makes it easier to spot and less tempting to use since it tends to be ugly. Typecasting should be avoided whenever possible. The other three types of named casts are const_cast, reinterpret_cast, and dynamic_cast. They are of no use to us at this time.

Lab Tasks:

Task 1

With the help of manipulator functions write the program that produces the following output

```
********9997
0.100
1.000e-001
```

Task 2

Write a program that converts Celsius temperatures to Fahrenheit temperatures. The formula is:

F= (9/5)*C+32

F is the Fahrenheit temperature and C is the Celsius temperature.

Task 3

Write a program that prompts the user to enter the weight of a person in kilograms and outputs the equivalent weight in pounds. Output both the weights rounded to two decimal places. (Note that 1 kilogram is equal to 2.2pounds.) Format your output with two decimal places.

Task 4

Write a program that ask user to enter the angle in degrees. Convert the angle in radians and display the output. Keep the value of PI constant.

Radians= Angle * (PI/180)