SESSION

5

Variable Types

Session Checklist

- ✔ Declaring variables
- ✔ Dealing with the limitations of an integer variable
- ✔ Using floating point variables
- ✔ Declaring and using other variable types

30 Min. To Go

One problem with the Conversion program from Part I is that it deals only with integers. This integer limitation is not a problem for daily use — it is unlikely that someone would enter a fractional temperature such as 10.5___A worse problem is conversion round off. Most integer Celsius values convert to a fractional Fahrenheit reading. The Conversion program takes no heed of this — it simply lops off the fractional part without any warning.

Note

It may not be obvious, but this program had to be carefully written to minimize the effect that the rounding off of fractions would have on the output.

This chapter examines the limitations of integer variables. It goes on to examine other variables types, including those designed to limit rounding-off error, and to look at the advantages and disadvantage of each.

Decimal Numbers

Integers are the counting numbers that you are most accustomed to: 1, 2, 3, and so on plus the negative numbers –1, –2, –3, and so on. Integers are most useful in our everyday life; however, they cannot easily express fractions. Integer fractional values such as $\frac{2}{3}$ or $\frac{15}{16}$ or $3\frac{111}{126}$ are clumsy to work with. It is difficult to compare two fractional numbers if their denominator is not the same. For example, when working on the car, I have a hard time knowing which bolt is larger — $\frac{3}{4}$ or $\frac{25}{32}$ (it's the latter).

Note

I have been told, but I cannot prove, that the problem of integer fractions is what leads to the otherwise inexplicable prominence of 12 in everyday life. Twelve is the smallest integer that is divisible by 4, 3, and 2. One-fourth of a unit is accurate enough for most purposes.

The creation of decimal fractions proved to be a great improvement. It is clear that 0.75 is less than 0.78 (these are the same values as earlier, expressed as decimal values). In addition, floating-point math is much easier because there is no need to find least-common denominators and other such nonsense.

The limitations of int's in C++

The variable type `int` is the C++ version of an integer. Variables of type `int` suffer the same limitations that their counting integer equivalents in math do.

Integer round off

Reconsider the problem of calculating the average of three numbers. (This was the extra credit problem in Session 4.)

Given three `int` variables, `nValue1`, `nValue2`, and `nValue3`, an equation for calculating the average is the following:

```
(nValue1 + nValue2 + nValue3) / 3
```

This equation is correct and reasonable. However, let's consider the equally correct and reasonable solution:

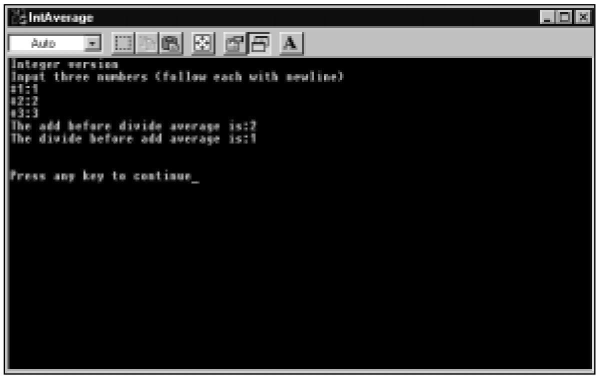
```
nValue1/3 + nValue2/3 + nValue3/3
```

To see what affect this might have, consider the following simple averaging program which utilizes both algorithms:

```
// IntAverage - average three numbers using integer
//              data type
#include <stdio.h>
#include <iostream.h>

int main(int nArg, char* pszArgs[])
{
    int nValue1;
    int nValue2;
    int nValue3;

    cout << “Integer version\n”;
    cout << “Input three numbers (follow each with newline)\n”;
    cout << “#1:”;
    cin  > nValue1;

    cout << “#2:”;
    cin  > nValue2;

    cout << “#3:”;
    cin  > nValue3;

    // the following solution does not suffer from round off as much
    cout << “The add before divide average is:”;
    cout << (nValue1 + nValue2 + nValue3)/3;
    cout << “\n”;

    // this version suffers from serious round off
    cout << “The divide before add average is:”;
    cout << nValue1/3 + nValue2/3 + nValue3/3;
    cout << “\n”;

    cout << “\n\n”;
    return 0;
 }
```

This program retrieves the three values `nValue1`, `nValue2`, and `nValue3` from `cin`, the keyboard. It then outputs the average of the three numbers, using the add-before-divide algorithm followed by the divide-before-add algorithm.

After building and executing the program, I entered the values 1, 2, and 3 expecting to get an answer of 2 in both cases. Instead, I got the results shown in Figure 5-1.



```
Integer version
Input three numbers (follow each with newline)
#1:1
#2:2
#3:3
The add before divide average is:2
The divide before add average is:1


Press any key to continue_
```

Figure 5-1
The divide-before-add algorithm displays significant rounding-off error.

To understand the reason for this strange behavior let's enter the values of 1, 2, and 3 directly in equation 2.

```
1/3 + 2/3 + 3/3
```

Because integers cannot express fractions, the result of an integer operation is always the fraction rounded toward zero.

This rounding down of integer calculations is called *truncation*.

With integer truncation in mind, the above equation becomes

```
0 + 0 + 1
```

or 1.

The add-before-divide algorithm fairs considerably better:

```
(1 + 2 + 3)/3
```

equals 6/3 or 2.

Even the add-before-divide algorithm is not always correct. Enter the values 1, 1, and 3. The answer returned by both algorithms is 1 instead of the correct 1⅔.

Note

You must be very careful when performing division involving integers because truncation is likely to occur.

Limited range

A second problem with the `int` variable type is its limited range. A normal `int` variable can store a maximum value of 2,147,483,647 and a minimum value of –2,147,483,648—more or less, plus 2 billion to minus 2 billion.

Note

Some older (actually, "very older") compilers limit the range of an `int` variable to the range -32,768 to 32,767.

20 Min.
To Go

Solving the truncation problem

Fortunately, C++ understands decimal numbers.

C++ refers to decimal numbers as *floating-point numbers*, or simply as *floats*. The term floating point stems from the decimal point being allowed to float back and forth as necessary to express the value.

Floating point variables are declared in the same way as `int` variables:

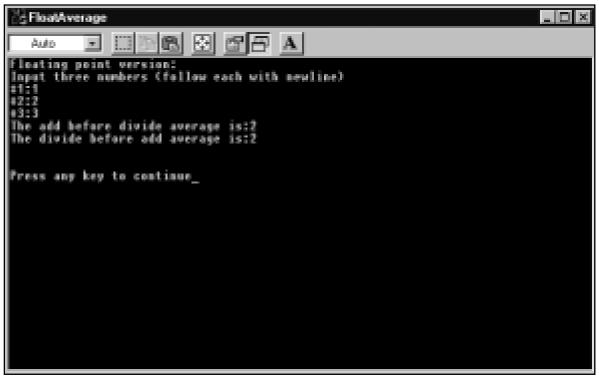
```
float fValue1;
```

To see how floating point numbers fix the rounding-off problem inherent with integers, I converted all of the `int` variables to `float` in the earlier IntAverage program. (The resulting program is contained on the enclosed CD-ROM under the name FloatAverage.)

`FAverage` converted the numbers 1, 2, and 3 to the proper average of 2 using both algorithms as shown in Figure 5-2.

Tip

If you intend to perform calculations, stick with floating-point numbers.



```
Floating point version:
Input three numbers (follow each with newline)
#1:1
#2:2
#3:3
The add before divide average is:2
The divide before add average is:2


Press any key to continue_
```

Figure 5-2
Floating-point variables calculate the proper average for the values 1, 2, and 3.

Limitations of floating point

While floating-point variables can solve many calculation problems, such as truncation, they have a number of limitations.

Counting

For one thing, floating-point variables cannot be used in applications where counting is important. This includes C++ constructs that require counting capability. This is because C++ can't be sure exactly which whole number value is meant by a given floating-point number. For example, it's clear that 1.0 is 1, but what about 0.9 or 1.1? Should these also be considered as 1?

C++ simply avoids the problem by insisting on using `int` values when counting is involved.

Calculation speed

Historically, a computer processor can process integer arithmetic quicker than it can process floating-point arithmetic. Thus, while a processor might be capable of adding 1000 integer numbers in a given amount of time, the same processor might be capable of performing only 200 floating-point calculations.

Calculation speed has become less and less of a problem as microprocessors increase in capability. Most modern processors contain special calculation circuitry enabling them to perform floating-point calculations almost as fast as integer calculations.

Loss of accuracy

Further, even floating-point variables don't solve all computational problems. Floating-point variables have a limited precision: about six digits.

To see why this is a problem, consider that ⅓ is expressed as 0.333. . . in a continuing sequence. The concept of an infinite series makes sense in math, but not to a computer. The computer has finite accuracy. Thus, a floating-point version of ⅓ is roughly 0.333333. When these 6 digits are multiplied by 3 again, the processor computes a value of 0.999999 rather than the mathematically expected 1. The loss of accuracy in computations due to the limitations of the floating-point data type is called *round-off error.*

C++ can correct for many forms of round-off error. For example, in output, C++ can determine that instead of 0.999999, that the user really meant 1. In other cases, however, even C++ cannot correct for round-off error.

"Not so limited" range

The `float` data type also has a limited range, although the range of a `float` is much larger than that of an integer. The maximum value of a floating-point variable is roughly 10 to the 38th power. That's 1 followed by 38 zeroes.

Note

Only the first 6 digits have any meaning because the remaining 32 digits suffer from floating point round off error. Thus, a floating-point variable can hold the value 123,000,000 without round-off error but not the value 123,456,789.

Other Variable Types

C++ provides other variable types in addition to `int` and `float`. These are shown in Table 5-1. Each type has its advantages as well as its limitations.

Table 5-1
Other C++ Variable Types

Name	Example	Purpose
`char`	'c'	A single `char` variable can store a single alphabetic or digital character. Not suitable for arithmetic. (The single quotes indicate the single character *c*.)
`string`	"this is a string"	A string of characters; a sentence. This is used to contain phrases. (The double quotes indicate the string of characters *this is a string*.)
`double`	1.0	A larger, economy-size floating-point type with 15 significant digits and a maximum value of 10 to the 308th power.
`long`	10L	A larger integer type with a range of 2 billion to –-2 billion.

Thus, the statement

```
// declare a variable and set it to 1
long lVariable;
lVariable = 1;

// declare a variable of type double and set it to 1.0
double dVariable;
dVariable = 1.0;
```

declares a variable `lVariable` to be of type `long` and sets it equal to the value 1, while `dVariable` is a `double` type variable set to the value 1.0.

Tip

It is possible to declare a variable and initialize it in the same statement:

```
int nVariable = 1;  // declare a variable and
                    // initialize it to 1
```

Note

There is no particular significance of the name of the variable to the C++ compiler.

A `char` variable can hold a single character, whereas a string holds a string of characters. Thus, `'a'` is the character *a* whereas `"a"` is a string containing just the letter *a*. (String is not actually a variable type, but for most purposes you can treat it as such. You will learn more details about string in Chapter14.)

Never

The character 'a' and the string "a" are not the same thing. If an application requires a string, you cannot provide it a character, even if the string contains only the single character.

Both `long` and `double` are extended forms of `int` and `float` respectively — long stands for long integer, and double stands for double-precision floating point.

Types of constants

Notice how each data type is expressed.

In an expression such as `n = 1;`, the constant 1 is an `int`. If you really meant for 1 to be a long integer then you need to write the statement as `n = 1L;`. The analogy is as follows, 1 represents a single ball in the bed of a pickup truck, while `1L` is a single ball in a dump truck. The ball is the same, but the capacity of its container is much larger.

Similarly 1.0 represents the value 1 but in a floating-point container. Notice, however, that the default for floating-point constants is `double`. Thus, 1.0 is a `double` number and not a `float`.

Special characters

In general, you can store any printable character you want in a `char` or string variable. There is also a set of nonprintable characters that are so important that they are allowed as well. Table 5-2 lists these characters.

You have already seen the newline character. This character breaks a string into separate lines.

Table 5-2
Nonprintable but Often Used Character Constants

Character Constant	Meaning
`‘\n’`	new line
`‘\t’`	tab
`‘\0’`	null
`‘\\’`	backslash

So far, I have only placed newlines at the end of strings. However, a newline can be placed anywhere within a string. Consider, for example, the following string:

```
cout << “This is line 1\nThis is line 2”
```

appears on the output as

```
This is line 1
This is line 2
```

Similarly, the `‘\t’` tab character moves output to the next tab position. Exactly what that means is dependent on the type of computer you are using to run the program.

Because the backslash character is used to signify special characters, there needs to be a character pair for the backslash itself. The character `‘\\’` represents the backslash.

C++ Collision with MS-DOS Filenames

MS-DOS uses the backslash character to separate folder names in the path to a file. Thus, Root\FolderA\File represents *File* in *FolderA*, which is a subdirectory of *Root*.

Unfortunately, MS-DOS’s use of backslash conflicts with that of C++. The MS-DOS path Root\FolderA\File is represented by the C++ string “Root\\FolderA\\File”. The double backslashes are necessary because of the special use of the backslash character in C++.

10 Min. To Go

Mixed Mode Expressions

I almost hate to bring this up, but C++ enables you to mix variable types in a single expression. That is, you can add an integer with a double. The following expression in which `nValue1` is an `int` is allowed:

```
// in the following expression the value of nValue1
// is converted to a double before performing the
// assignment
int nValue1 = 1;
nValue1 = nValue1 + 1.0;
```

An expression in which the two operands are not of the same type is called a *mixed mode expression*. Mixed mode expressions generate a value whose type is equal to the more capable of the two operands. In this case, `nValue1` is converted to a `double` before the calculation proceeds.

Similarly, an expression of one type may be assigned to a variable of a different type as in the following statement

```
// in the following assignment, the whole
// number part of fVariable is stored into nVariable
fVariable = 1.0;
int nVariable;
nVariable = fVariable;
```

Note

Precision or range may be lost if the variable on the left side of the assignment is "smaller." In the previous example, the value of *fVariable* must be truncated before being stored in *nVariable*.

Done!

Converting a "larger" size value to a "smaller" size variable is called *promotion* while converting values in the opposite direction is known as *demotion*. We say that the value of `int` variable `nVariable1` is promoted to a `double`:

```
int nVariable1 = 1;
double dVariable = nVariable1;
```

Tip

Mixed mode expressions are not a good idea. You should make your own decisions instead of leaving it up to C++ — C++ may not understand what you really want.

Naming Conventions

You may have noticed that I begin the name of each variable with a special character that seems to have nothing to do with the name. These special characters are shown below. Using this convention, you can immediately recognize `dVariable` as being a variable of type double.

Character	**Type**
`n`	int
`l`	long
`f`	float
`d`	double
`c`	character
`sz`	String

These leading characters help the programmer keep track of the variable type. Thus, you can immediately identify the following as a mixed mode assignment of a `long` variable to an `int` variable.

```
nVariable = lVariable;
```

Remember, however, that although I might use some special characters in variable names, these characters have no significance to C++. I could just as soon have used `q` to signify *`int`* had I wanted to. Many programmers don't use any naming convention at all.

REVIEW

As you have seen, integer variables are both efficient in terms of machine time and easy to program. Integer variables have definite limitations when used in calculations, however. Floating-point variables are ideal for use in mathematical equations because they do not suffer from significant round off nor significant range limitation. On the downside, floating-point variables are more difficult to use and not as universally applicable as integers.

- Integer variables represent counting numbers, such as 1, 2, and so on.
- Integer variables have a range of –2 billion to +2 billion.
- Floating-point variables represent decimal fractional values.
- Floating point variables have a practically unlimited range.
- The `char` variable type is used to represent ANSI characters.

QUIZ YOURSELF

1. What is the range of an `int` variable? (See "Limited Range.")
2. Why don't `float` variables suffer from significant rounding off error? (See "Solving the Truncation Problem.")
3. What is the type of the constant 1? What is the type of 1.0? (See "Types of Constants.")