Straight-Line Computation in C++

Steven Zeil

May 25, 2013

Contents

Read chapters 1 and 2 of your text, if you have not already done so.

Chapter 2 of your text takes off at a pretty good pace. Don't worry if not everything in there is clear right now – it's really intended as an overview and we'll be revisiting a lot of it in more detail later.

At its most basic, a program is a collection of *instructions* for manipulating *data*. Data comes in many different types in C++, and you will eventually see that one of the language's strengths is in allowing you to define new types of data of your own.

Similarly, there are many different kinds of instructions that we can issue to manipulate data. More importantly, however, we can arrange instructions in varied and subtle ways. Just like when you follow someone's directions on how to drive to a destination, or try to carry out a recipe when cooking, or when following an instruction booklet to assemble something, sometimes the instructions are simple and straightforward ("Go straight on down the road. You can't miss it!", "Insert tab A into slot B") or moderately complicated ("Turn right at the schoolhouse, then go to the next light, make a U-turn, come halfway back", "Bake at 350 for one hour or until the internal temprature reaches 165 degrees") or downright complicated ("Bake for longer at high altitudes. Reduce baking time if using a glass pan.").

In programming, the simplest way to arrange instructions is in a "straight line" computation so that we do the first instruction, then the second, then the third, and so on, never varying the order. That's our goal for this section.

1 Data Types

C++ manipulates many types of data. The language provides certain basic types, sometimes called *primitive* types because they are build into the language. There are also a selection of "type builders" that let us build new types of data from existing ones, either primitives or still others of our own that we have already built.

The primitive data types in C++ are

- The integer numbers: *int*, *long int*, *unsigned int*, *unsigned long*, and *char*
  - These give us numbers like 123, −47, etc.
  - Although it may be counter-intuitive, even characters are really integers. But the usual way to write a character constant is not as its nummeric value, but to place the desired character inside apostrophes, e.g., `'A'`, `'a'`, `'!'`. But you *could* use ordinary integers if you preferred.

    This works because the most common characters have been assigned an integer value according to a scheme known as the ASCII code. These codes are arranged in ways that, for the most part, make sense. For example, the upper case letters 'A' to 'Z' are assigned 26 consecutive integer values, with the integer value of 'B' being one above the value of 'A', the value of 'C' two above the value of 'A', and so on. The lowercase letters 'a'..'z' have a similar arrangement.
- The floating point numbers: *float* and *double*
  - These give us numbers like 123.0, −4.7, 3.14159, etc.

Floating point numbers are often not exact. Just as, for example, the number 1/3 is a repeating decimal (0.333...) in base 10 and therefore cannot be written exactly in any finite number of decimal digits, there are many numbers such as 1/5 that are repeating decimals in base 2 and so cannot be written exactly in any finite number of bits.

The difference between *float* and *double* is that *double* has more bits and can therefore represent numbers to more precision than can *float*. On most modern CPUs, the calculation time for *float* and *double* calculations are identical. The storage savings offered by *float* are seldome significant, but the extra precision offered by *double* is often important. So C++ programmers tend to use *double* for nearly all floating point calculations.

- Logical values: *bool*
  - These have just the two values, `true` and `false`

The type builders in C++ will be covered later, but I will mention them briefly for now:

- *Arrays* group multiple values of the same type into a linear sequence, making it easy to process one data value after another.

  Example: Think of a bookshelf - an arrangement of books into a linear sequence.

- *Structured types* (*struct*s and *class*es) group values of different types, allowing us to give names to each component.

  Example: We might think of a "street address" as a combination of an integer (the house number) and an array of characters (the chanracters making up the street name).

- The *address types* "pointer to" and "reference to" allow us to refer to data of various types indirectly in cases when we might not have an actual name for the data we want.

  Example: Suppose you have just been asked the question, "Who do you want to speak to?". You might answer that question by giving a specific name "John Smith", or you might answer by pointing at someone and saying "That guy!". The first answer is rather like using an ordinary variable. The second is more like a pointer - a way of designating an individual thing even if you don't know its name -- or even if it doesn't have a name: "Look at *this* blade of grass."

2 Assignment Statements

One of the most common statements (instructions) in C++ is the *assignment statement*, which has the form:

```
destination = expression;
```

= is the *assignment operator*. This statement means that the expression on the right hand side should be evaluated, and the resulting value stored at the desitnation named on the left. Most often this destination is a variable name, although in come cases the destination is itself arrived at by evaluating an expression to compute where we want to save the value.

Some examples of assignment statements would be

```
pi = 3.14159;
areaOfCircle = pi * r * r;
circumferenceOfCircle = 2.0 * pi * r;
```

Now, a few things worth noting:

- These statements manipulate 4 different variables: `pi`, `r`, `areaOfCircle` and `circumferenceOfCircle`.
- We have to assume that `r` already contains a sensible value if we are to believe that these assignments will do anythign useful.
- The last two only make sense if the first assignment has been performed already. Luckily, when we arrange statements into a straightline arrangement like this, they are performed in that same order.
- Note that we have reused `pi` in two different statements. We didn't *need* to do this. I could instead have written

  ```
  areaOfCircle = 3.14159 * r * r;
  circumferenceOfCircle = 2.0 * 3.14159 * r;
  ```

  but I think the original version is easier to read.

When using variables on either side of an assignment, we need to declare the variables first:

```
double pi;
double areaOfCircle;
double circumferenceOfCircle;
pi = 3.14159;
areaOfCircle = pi * r * r;
circumferenceOfCircle = 2.0 * pi * r;
```

Actually, we can combine the operations of declaring a varable and of assigning its first, or *initial* value:

```
double pi = 3.14159;
double areaOfCircle = pi * r * r;
double circumferenceOfCircle = 2.0 * pi * r;
```

Technically these are no longer assignments. Instead they are called *initialization* statements. But the effect is much the same.

I actually prefer this second, combined version, by the way.

Let me restate that. I *strongly* prefer that second version combining declarations with initialization.

One of the most common programming errors is declaring a variable, forgetting to assign it a value, but later trying to use it in a computation anyway. If the variable isn't initialized, you basically get whatever bits happened to be left in memory by the last program that used that address. So you wind up taking an essentially random group of bits, feeding them as input to a calculation, feeding that result into another calculation, and so on, until eventually some poor schmuck gets a telephone bill for $1,245,834 or some piece of expensive computer-controlled machinery tears itself to pieces.

By getting into a habit of *always* initializing variables while declaring them, I avoid most of the opportunities for ever making this particular mistake.

3 Examples of Straight-Line Computations

3.1 Farenheit to Centigrade

We can convert a temperature `F`, expressed in degrees on the Farenheit scale, to degrees Centigrade by the formula: which renders in code as

```
double f;
   ⋮
double c = (5.0 / 9.0) * (f - 32.0);
```

Note, by the way, the use of the constants "5.0" and "9.0" rather than just "5" and "9". That's actually important. `5.0/9.0` is a floating point calculation that yields a floating-point answer of approximately 0.5556. But `5/9` is an integer calculation that *truncates* any fractional part and actually returns an answer of 0.

So if we had written

```
double f;
   ⋮
double c = (5 / 9) * (f - 32);
```

then, no matter how hot it gets on the Farenheit scale, we would think it's still freezing in Centigrade!

3.2 Quadradic Formula

You may remember, from way back when in a math class you have tried hard to forget, that when you have something described by the *quadratic equation*: then you can compute the (two) solutions by the *quadratic formula*:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

We can do this in C++ like this

```
double a, b, c;
   ⋮
double x1 = (-b + sqrt(b*b - 4.0*a*c))/(2.0*a);
double x2 = (-b - sqrt(b*b - 4.0*a*c))/(2.0*a);
```


We can simplify this a bit by introducing a new variable to hold the shared part of the calculation:

```
double a, b, c;
   ⋮
double shared = sqrt(b*b − 4.0*a*c);
double x1 = (−b + shared)/(2.0*a);
double x2 = (−b − shared)/(2.0*a);
```

This might even be a little faster, as we avoid computing the same shared value twice.