Angles

Implementing a Data Type

An angle can be viewed as a simple numeric value, supporting most of the customary arithmetic and relational operations. However, using one of the built-in data types, such as double, to represent an angle is not really satisfactory. For one thing, representing an angle using a decimal value mires us in the intricacies and inadequacies of floating-point arithmetic on digital hardware. (See CS 3414 for details.) And, while the measure of an angle may be expressed as a single real number if radian measure is used, what if the angle is to be measured in degrees, minutes and seconds?

Micro-refresher: there are 360 degrees in a circle (or revolution), 60 minutes in a degree, and 60 seconds in a minute. Traditionally, an angle might be written in DMS form as 47° 23´ 55´´. If you want more information on geometry and trigonometry, a Google search on those two words turned up about 116 million and 8.7 million hits, respectively. Not that we’re going to go very deep into all that.

No single built-in type C++ adequately models DMS representation of an angle. And, angles have some relationships for which the built-in types have no analogs (e.g., the notion of being coterminal). So, in reality, no built-in type is adequate to model an angle, regardless of internal representation.

In order to support the degree/minute/second representation, and to provide the natural operations, you will implement a C++ class named DMSAngle.

Before implementation comes design. The design of the public interface for this class is an interesting exercise, involving identifying just what operations should be supported in order to make the type useful in contexts where angles are naturally significant. The most fundamental issue, however, is just what the internal data representation should be.

There are many choices for the internal representation. Some are obvious. Some are not. At this point, I will give only a little guidance regarding this decision:

⇒ It is important that we provide representation for a relatively wide range of values. Different standard types have different ranges (e.g., unsigned char versus short versus int versus unsigned int).
⇒ Accuracy is important.
⇒ The meaning of accuracy depends, to some extent, upon what is being represented. For representing forces within a structure, we would probably want at least several digits beyond the decimal point. For representing dollars and cents, it really only makes sense to store two digits after the decimal point.
⇒ Computational efficiency is important. Speed matters. Using standard types for the internal representation will usually lead to faster performance than if we use our own custom types. Of course, the standard types also have serious range limitations. As usual in life, goals conflict.
⇒ Storage efficiency is important. Size matters. Of course, achieving greater range will generally require that we use more space. But there are choices that can mitigate that effect, often at the expense of computational efficiency.

At this point, you don’t have enough information to justify writing much, if any, code. (Not that this will stop some of you from doing so, but I sincerely hope that any time you spend writing code, before the final draft of this assignment is posted, turns out to be completely wasted.)

You do have plenty of information to begin thinking seriously about the design of the class DMSAngle. You should be able to identify many of the operations you will eventually have to support. You should be able to identify a number of alternative data representations, and weigh the advantages and disadvantages of both. I suggest you begin your deliberations now.

Because your implementation will be subjected to automated testing, by compiling it with a test harness implemented by me, it will eventually be necessary to specify a minimal public interface that you must support. However, that time has not yet come.