Monetary Values: Double Trouble or Dollars and Sense?

Aaron Tenenbaum  
Dept of CIS, Brooklyn College  
2900 Bedford Avenue  
Brooklyn, NY 11210  
1-718-951-5657  
tbaum@sci.brooklyn.cuny.edu

Gerald Weiss  
Dept of CIS, Brooklyn College  
2900 Bedford Avenue  
Brooklyn, NY 11210  
1-718-951-5657  
weiss@sci.brooklyn.cuny.edu

David Arnow  
Dept of CIS, Brooklyn College  
2900 Bedford Avenue  
Brooklyn, NY 11210  
1-718-951-5657  
arnow@acm.org

ABSTRACT
It is curious but there is widespread misuse of floating-point types to represent monetary amounts in introductory programming classes. This is evident from a survey of popular CS1 textbooks. It is instructive to examine how dollars and cents and other currency values are represented in actual practice and how we can utilize those techniques in computer science education, for both the sake of correctness and to illustrate important general principles. Furthermore, it is particularly interesting to review the history of CS education and determine how this situation came to pass. The interplay between technology, academic cultures and institutional development turns out to be quite significant.

Categories and Subject Descriptors
D.3.3 [Programming Languages]: Language Constructs and Features – data types and structures, input/output  E.2 [Data]: Data Storage Representations – primitive data items  K.3.2 [Computers and Education]: Computer and Information Science Education – Computer science education, Curriculum

General Terms
Design, Languages

Keywords
Monetary values, currency representation, numerical types

1. BACKGROUND
A survey of introductory programming texts reveals that in the treatment of accounting problems (for example, bank accounts, hours and wages, and sales, to mention a few), the variables holding the monetary data are usually declared as being of type real, float or double. If any rationale is provided at all, it is usually that, since the monetary numbers are not whole numbers of dollars but often have cents or pence connected with them, they require a fractional part. Numbers with fractional parts are not integers. Since the only other kind of primitive numerical type available is floating-point, it is real, float or double that is selected to do the job, at least in the classroom.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republic, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SIGCSE ’10, March 10–13, 2010, Milwaukee, Wisconsin, USA.
Copyright 2010 ACM 978-1-60558-885-8/10/03...$10.00.

However, reals are almost never used in actual practice to represent monetary values, because of rounding errors and binary representation. [1,22] For example, if .01 is added 10000 times using real numbers, the result is almost never exactly 100. Sometimes, explicit rounding is used but sometimes even that is inaccurate. Other times, the instructor relies on output statements to produce the correct value in printouts and output files.

In general, floating-point type values are meant to represent either a) measurements and quantities that derive from measurements that have an inherent limited precision, or b) approximations of irrational numbers. On the other hand, integer type values represent quantities that are counted and can be put into a 1-to-1 correspondence with the integers. Money, of course, is counted, not measured. There is a count of dollars and there is a count of cents. Using a measuring stick to determine an amount of money is quite a rare occurrence indeed (although we often hear about the national budget or debt reaching from here to the moon if piled into a linear line of dollar bills). The English language itself reflects this: the discipline of tracking money is called “accounting”.

Programming languages and systems that support accounting applications take care to provide non-floating-point numerical types that are suitable for handling money. For example, both MySQL and SQL Server provide a DECIMAL type that can represent as many as 38 decimal digits with a fixed decimal point. This permits to-the-penny representations of the worst projections of the U.S. national debt in Paraguayan Guarani (currently 5000 per USD) for years to come [23,24]. Multiple digits to the right of the decimal point are often needed to facilitate currency conversion or to avoid or minimize roundoff error.

Professional programmers working with languages such as C++ and Java, whose primitive types don’t provide this kind of representative power, make use of suitable integer-based classes that they develop themselves or acquire from third parties [29]. These classes are implemented using two or more integers, as needed for their precision. A good discussion of this issue and one solution for actual representation of currency in Java is described in [1]. Systems such as Microsoft’s Excel and Access also have a currency data type that operates on currency correctly. The operations are no doubt implemented by a scheme such as in [1].

Using non-floating-point numerical types does not remove all concerns for numerical error in accounting. But it reduces them to a controllable set. In some cases, notably interest calculations, it is beneficial to do some intermediate computations using floating point arithmetic. But the fact remains that using non-floating-point types for currency values is industry practice and rightly so.

To get a sense of the pitfalls of using floating-point representation in the context of values that require exact representation (such as
currency), consider printing the monetary quantity $93.585 (such a quantity could arise for example from a $110.10 value discounted 15%). Following the customary rules of rounding, this value displayed with two digits should be $93.59. As it turns out, the \texttt{printf} format specifier does follow the customary rules of rounding to \textit{n} digits. For example,

\begin{verbatim}
printf("%8.2f", 93.58501) yields 93.59. Unfortunately,
printf("%8.2f", 93.585) yields 93.58. We can readily see why, since
printf("%19.15f", 93.585) yields 93.584999999999994. The issue is "representation error" (though using the term "error" here is a bit unfair to the float type which never held out the promise of exact representation of decimals: float promises to represent numerical values up to a certain precision and that is what it has done here). It is tempting to correct for this representational deviation by writing
\end{verbatim}

\begin{verbatim}
printf("%8.2f", 93.585+0.005)
\end{verbatim}

which yields 93.59 but an attempt to do this programatically only leads to more problems:

\begin{verbatim}
double x=93.585;
double y=0.33;
printf("%8.2f ", x+0.005);
printf("%8.2f\n", y+0.005);
\end{verbatim}

yields

\begin{verbatim}
93.59  0.34
\end{verbatim}

instead of

\begin{verbatim}
93.59  0.33
\end{verbatim}

Furthermore, this can arise quite naturally from simple, innocently posed "first week" CS1 problems. For example:

\begin{quote}
\textit{A store is having a sale for the holidays: buy any item and get the second at a 15\% discount. Prompt the user for the full price of the item and produce an invoice containing:}
\begin{itemize}
\item The full price of the first item
\item the discounted price of the second item
\item the total price of the purchase
\end{itemize}
\end{quote}

\begin{quote}
Standard rules for rounding are to be used.
\end{quote}

Apart from using floating point type, here is a reasonable solution:

\begin{verbatim}
int main() {
    double firstItem, secondItem, total;
    printf("Price of 1st item? ");
    scanf("%le",\&firstItem);
    secondItem = 0.85 * firstItem;
    total = firstItem + secondItem;
    printf("1st item: %8.2f\n", firstItem);
    printf("2nd item: %8.2f\n", secondItem);
    printf("          --------\n");
    printf("Total   : %8.2f\n", total);
}
\end{verbatim}

Here are the results of two sample executions:

\begin{verbatim}
Price of 1st item? 110.20
1st item: 110.20
2nd item: 93.67
          --------
Total   : 203.87
\end{verbatim}

\begin{verbatim}
Price of 1st item? 110.10
1st item: 110.10
2nd item: 93.58
          --------
Total   : 203.69
\end{verbatim}

We have already seen that the error in the second execution is caused by the internal representation of the value of \texttt{secondItem}. This is not a unique example, and to make matters worse, there is no straightforward way to anticipate problematic values like 93.585. Indeed, to concoct the above example, we generated multiple cases of varying discount percentages and prices until we encountered a situation where the roundoff error produced a discrepancy to two places.

Thus, it is not merely the discrepancy that is problematic, but the fact that the discrepancy seems to arise in an arbitrary fashion (at least at the level of understanding of an intro student). Furthermore, the student’s by-hand calculations produce a different result, undermining their understanding of the situation.

Here is the same program and results, this time using integer type:

\begin{verbatim}
int main() {
    int firstItem, secondItem, total;
    printf("Price of 1st item? ");
    scanf("%d",\&firstItem);
    secondItem = (85 * firstItem + 50) / 100;
    total = firstItem + secondItem;
    printf("1st item: %8d\n", firstItem);
    printf("2nd item: %8d\n", secondItem);
    printf("          --------\n");
    printf("Total   : %8d\n", total);
}
\end{verbatim}

\begin{verbatim}
Price of 1st item? 11020
1st item: 11020
2nd item: 9366
          --------
Total   : 20386
\end{verbatim}

\begin{verbatim}
Price of 1st item? 11010
1st item: 11010
2nd item: 9358
          --------
Total   : 20368
\end{verbatim}

Although the input and output format of this program suffers from cosmetic deficiencies, it has the advantage of being correct. (The deficiencies can be corrected to the degree to which the CS1 instructor or author chooses to address simple parsing and formatting techniques.)

The problem is not confined to C and C++. The following "first week" Java program and resulting output make this clear:

\begin{verbatim}
Price of 1st item? 110.20
1st item: 110.20
2nd item: 93.67
          --------
Total   : 203.87
\end{verbatim}

\begin{verbatim}
Price of 1st item? 110.10
1st item: 110.10
2nd item: 93.58
          --------
Total   : 203.69
\end{verbatim}
public class Sale {
    public static void main(String[] a) {
        double chrg = 5.10;
        double taxrate = .05;
        double tax = taxrate*chrg;
        double total = chrg+tax;
        System.out.printf(" chrg= $%6.2f\n",chrg);
        System.out.printf(" tax= $%6.2f", tax,taxrate);
        System.out.printf("total= $%6.2f\n",total);
    }
}

charge= $ 5.10
  tax= $ 0.26
total= $ 5.35

Using output integer types and arithmetic thus seems to be the only way to go. True, there are issues of rounding as well (how much do we pay a $100/hour worker for 20 minutes?) but at least the results are uniform and can be predicted at the CS 1 level.

2. THE CS1 CHALLENGE

The languages most commonly used in introductory programming courses, Java, C, C++, C# and Python do not provide the decimal types needed for convenient correct handling of currency values. The CS1 author or instructor then must first decide whether or not to simply avoid examples that involve currency. If such examples are included in the text or course, then another decision must be made: to what extent should currency values be properly handled?

Few, as an abstract matter, would advocate intentionally taking an incorrect approach. Remarkably, however, the erroneous use of double (or other floating point types) to represent monetary quantities is quite widespread. Although we cannot cite what faculty do in the classroom, the common occurrence of this error in a great many textbooks that are otherwise of the finest quality and authored by the most esteemed colleagues in the community strongly supports this contention [8, 9, 12, 17, 18, 19, 26, 28, 32].

3. HISTORY

3.1 Before The Fall

In the United States in the 1960s and through the early 1970s, the primary high-level languages of introductory computing instruction were COBOL and FORTRAN. FORTRAN was viewed as a scientific/engineering programming language, and had no specific provisions that made it easy to represent currency values.

Commercial programming depended on COBOL. Those introductory programming courses that were commercially oriented, in information systems or business programs, were taught using COBOL as well. COBOL did have a tool for representing currency values well. It had a native decimal (as well as binary) number type, with a specification for the number of digits before and after the decimal point. Many computers had decimal numbers implemented in hardware or firmware, and the COBOL data types were designed to take advantage of those facilities. So, COBOL was built to represent monetary values, and its data types do that well. No one used a floating-point data type to represent money in COBOL. Decimal numbers with a fixed number of decimal digits were used invariably.

COBOL, however, was “inelegant.” It was wordy, non-mathematical and not very useful for scientific or systems work. Since most of those who developed the computer science discipline came from mathematics or the sciences, COBOL became viewed as a “vocational” or “data processing” language, not suitable for use in scientific higher education.

The dichotomy in programming courses was naturally reflected in introductory textbooks. While we have not undertaken an exhaustive search of the introductory programming literature before 1976, a clear division appears: FORTRAN texts, which illustrated programming using numerical, scientific, engineering, and discrete math problems [2, 7, 10, 11, 20, 21, 30], and COBOL texts, which illustrated programming using administrative, business and financial problems. In the COBOL texts, the availability and use of fixed decimal attributes made the proper handling of monetary values easy and accessible to the beginner. Since problems involving monetary amounts were never or barely discussed in the FORTRAN texts (and presumably the courses that used them), these quantities were almost never misrepresented.

An author of an early leading and influential FORTRAN text, Professor Dan McCracken of City College, wrote us:

I did not use any monetary examples in my text because the book, in 1960/61, was aimed at engineers and scientists. I was also aware of the problem with non-exact floating binary for most fractional amounts. The vast majority of authors, in my sampling, still don’t understand the issue and cheerfully use doubles, with no warnings. But no bank on the planet uses doubles for monetary amounts. Banks can’t be sending out checking account statements that are off by a penny.

Another language, ALGOL, was used primarily in Europe and in newly created computer science departments that often focused on neither scientific or business applications, but on systems and systems programming. ALGOL, like FORTRAN, did not have decimal representation support built into it, and reals were used in the very few cases where monetary examples were provided in a text.

Many texts in FORTRAN and ALGOL did not even mention an example with monetary values [20, 21, 11] or used them only in connection with modeling [2]. An exception may be found in the very popular FORTRAN text [7] on page 109.

Computer science as an educational discipline arose from users of ALGOL or FORTRAN, not from COBOL users. Examples were primarily oriented towards mathematical, engineering or systems programming. Thus the tradition arose that the few times that money was represented, it was represented by the native data types in those languages—the reals. It was “good enough” for science-bound, introductory programming students. One did not want to confuse them by going into the complexities of how money should be represented, since the “purpose” of programming for this community was not commercial applications in any case and because it added an extra layer of complexity, taking away from the main thrust of the course. Ideas of abstraction and representation did not yet have the prominence in computer science that they gained later.
And thus there were, by the early 1970s, two distinct educational worlds, business computing using COBOL and nascent computer science using FORTRAN and ALGOL—each restricting itself to problem domains that were treated correctly—for the most part.

This world was disrupted by a succession of technological advances in the 1970s. The first of these was the increased availability of PL/I, arguably the first language that, whatever its faults, was equally suitable for scientific, systems and commercial work. PL/I, like COBOL provided a native attribute, **fixed decimal**, with which monetary values could be represented in a satisfactory way—as a number with a specified number of decimal digits. PL/I saw rapid adoption as an initial instructional language. It also made it possible to have a single introductory course for students of business computing and computer science students.

At the same time, the decade-long explosion of undergraduate computer science programs and departments began. With the availability of PL/I, these programs could justifiably claim to offer introductory programming sequences that were appropriate for students of all interests. Introductory programming courses and the textbooks that served them reflected this unification, and most texts used PL/I’s **picture** and **fixed decimal** attributes to provide correctly implemented credible examples from accounting that were co-mingled with those from science and engineering [3, 27, 31, 33, 25]. This period of “currency Eden” was not to last very long.

### 3.2 After The Fall

Then, starting in the late 1970s and early 1980s, introductory programming textbooks appeared, authored by some of the giants in the world of CS education, that returned to the misrepresentation of monetary amounts as floating point values [5, 6, 14, 13]. As it turned out, the misrepresentation error in these seminal texts, which appeared during the birth-phase of most undergraduate computer science programs, has replicated itself, as we have seen, to the present day.

#### 3.3 How Could This Have Happened?

It seems that this occurred as a result of a combination of significant historical events and simple, human oversight. The advent of the microprocessor and the PC replaced the decimal arithmetic supporting mainframes that had been the mainstay of many CS departments. This shift was the death-knell to PL/I as a teaching language, but the replacement of PL/I had already begun earlier as the pendulum swung back to “small languages” like Pascal and C, neither of which supported fixed decimal arithmetic.

There is little question that C and Pascal offered improvements over both PL/I and FORTRAN as teaching languages. But in terms of numerical primitive types, they both returned to the world of FORTRAN. However, the university world had moved on: introductory programming no longer lived in EE, Physics or Math departments. It existed in the context of undergraduate CS departments that had mission obligations that transcended the world of math, science and engineering. It was more difficult to take the approach of the accounting-avoiding FORTRAN textbooks and courses of the 1960s and early 1970s.

At least one author [15] did write a Pascal text that avoided all accounting problems, and there was at least one C textbook [16] that managed the same. Ken Bowles, in his text [4] gave an explicit warning about the problem of using **real** values to represent money:

---

Accuracy problems with REAL numbers are also very important in administrative computations. One of the strategies used in most implementations of COBOL, the “Common Business Oriented Language,” results in most computation being done with integers, even though the programmer thinks the numbers are really decimal fractions. COBOL provides ways for the programmer to specify how many decimal digits of precision must be carried. Unknown to the programmer, a <variable> indented to represent Dollars and Cents (2 decimal points of precision) would in fact be carried as an integer representation of cents.

But his was a lone voice. Most other authors since then have incorporated accounting examples using floating-point types for monetary values. A few authors such as Dale [8] and Deitel [9] make it clear that this is wrong in principle and dangerous in practice. Deitel’s admonishes the reader:

> Notice that lines 17-19 declare the **double** variables ...

> We did this for simplicity because we’re dealing with fractional parts of dollars, and we need a type that allows **decimal points** in its values. Unfortunately, this can cause trouble ...

and he goes on to provide an excellent, compelling numerical example of how the very code he presented as an example can give incorrect results. In his warning, he also conveys his reason for using doubles: expediency in dealing with fractions. This is probably the reason that most authors have for making this choice.

### 4. REMEDIES

There is no reason that the CS education community has to continue to make this choice. Rather than shy away from accounting problems or use inappropriate representations to solve them, CS instructors and authors can instead use monetary values as an excellent vehicle for teaching many points about programming.

The difference between floating-point and integer representations and between binary and decimal representations can easily be brought home using monetary values, something that every student is familiar with. The fact that every integer within a certain range can be represented, but not every real even when the decimal precision is fixed, is important for students to know. The ideas of precision, rounding and truncation can be introduced as well. Some of the relation of mathematics and computation can be brought home here.

Apart from numerical and mathematical issues is the issue of the meaning behind a primitive type. Reals are the numbers of measurement, integers the numbers of counting. Nothing brings this home to students better than a discussion of currency representation.

Addressing the representation of monetary values provides an excellent segue into use of existing classes (such as Java’s BigDecimal) or can motivate the development of simple class definitions that reinforce basic concepts in typing and object-oriented programming. This in turn can be related to currency data types as they exist in real software that many students are already acquainted with (such as spreadsheets), or that they will learn about. It is also a good basis for a lecture about the entire concept of reality, abstraction and representation.
5. THE LESSON
CS1 education has been remarkably successful in keeping up with the fast pace of ever-advancing computer science. The community of authors of the introductory texts have kept up with new languages, and introduced at various periods structured, modular, and O-O programming, graphical user interfaces, web technology, semi-formal methods, and other innovations. However, it is also important not to lose what was once gained. We no longer have the convenience of fixed decimal primitive types in the languages we use in introductory courses, but we need not give in to a false expediency at the expense of giving our students — who may well go on to work in accounting and financial applications — an incorrect view of monetary data. To do so does a disservice to both our students and our community.

6. ACKNOWLEDGMENTS
Our thanks to Dan McCracken for his valuable thoughts on the subject.

7. DEDICATION
In memory of our late colleague, Pat Sterbenz, who could have explained all of this so easily and more clearly.

8. REFERENCES
[29] Stein, G. 2009 private conversation with lead software project manager in a major Wall Street Brokerage.