The solution that the Gaddis book uses for the Monthly Payments problem is incorrect because, by using doubles, it does not assume that the bank will be making its calculations in terms of whole pennies. Doubles and floats are both too "accurate" and too inaccurate for this scenario. They are too accurate because C++ will make calculations that involve doubles and floats using every single digit that can be held in memory. This is dozens more digits than are necessary or useful when you are talking in terms of money (at least for the purposes of this problem). We really only need the values that represent dollars and cents, meaning that doubles and floats are holding more information than is necessary for this scenario. Which leads to the second problem of inaccuracy. The floating-point data types do not hold an infinite number of digits: when they hit their limit on space the number ends and the last number is rounded up or down appropriately. This means that the number that is saved in the double or float data type will be slightly larger or smaller than it actually is.

The excess digits that are rounded off are a problem in the current situation because the numbers that were used in the calculations were bigger or smaller than required. All numbers that represented money should have been cut off after the 100^{th} decimal place, and, when they weren't from the very start, it meant that all subsequent results were a little less correct. This wasn't really a noticeable issue, though, until the calculations compounded and larger numbers became involved. So we didn't see that there was an issue until two slightly inaccurate numbers were multiplied to give more inaccurate numbers. Here's a breakdown of what happened:

- inRate was slightly smaller than it should have been (0.009999…98 vs 0.01)
- temp was slightly smaller than it should have been since it multiplied the smaller inRate
- mnPmt was slightly larger than it should have been because the inaccurate inRate and temp were used for both multiplying and dividing
- totPmt and inPaid were noticeably larger than they should have been because the slightly larger mnPmt was multiplied by a large number (36)
- All of these values were made slightly more inaccurate when they were rounded off: this was exacerbated when floats were used, probably because rounding may have had a greater effect if the last digit was a 9 that was rounded up and because the numbers were rounded closer to the decimal point

To account for these issues and provide more accurate payment values, you turned mnPmt (the monthly payment amount in dollars) into a pseudo integer, thereby increasing its practical accuracy. You accomplished this by first adding a half of a cent ($0.005) to the value of mnPmt, thus ensuring that the digit in the 100^{th} place (where the pennies lie) would be accurate, even if the digits after it were not (they were already wrong, so messing them up in a different way wouldn't matter). You then converted the value from dollars to cents and placed them in a new variable. Since there is nothing smaller than a penny/cent, you made it an integer and truncated all the digits after the decimal point. And, ta da!, suddenly you had the correct monthly payment amount. Only it was in cents rather than dollars, so you placed it back in the original float variable and simultaneously converted it back to dollars. You now had the correct value for mnPmt, which meant that, when totPmt was calculated using mnPmt and inPaid using totPmt, you got results that were far more accurate than what the book gave.

What I don't get is why the loan value remained at 10,000 but the interest rate was changed from 0.01 to 0.009999999….8. They were both the same data type (float or double), so why was only one changed?