```
printf("%d", S);
```

This piece of code allows to get integer numbers from the keyboard and sum only the even ones. Indeed, the condition of the `if` statement is true when the remainder of the division by two of the inserted number is 1. In this case, the `continue` statement allows to skip the remaining lines and return to the evaluation of the control expression `(n > 0)`. Also this construct can easily be substituted. It is enough to rewrite the condition with a different structure:

```
int n = 1, S = 0;
while (n > 0) {
   scanf("%d", &n);
   if (!(n % 2)) {
      S += n;
   }
}
printf("%d", S);
```

In this way everybody considers it a good program snippet.

4.5 A rounding problem

In this section we discuss an apparently trivial problem which turns out to be particularly instructive: adding N numbers x_j, $j = 0 \ldots N-1$. It is not a coincidence that we postponed this problem till the end of this chapter. Indeed, the numerical problems we want to discuss occur when the result implies performing many iterations.

```
#include <stdio.h>

#define N 10000000

main() {
  float S = 0., x = 7.;
  unsigned int i, iS = 0, ix = 7;
  for (i = 0; i < N; i++) {
    S += x;
    iS += ix;
  }
  printf("Using floats   : %.0f x %d = %.0f\n", x, N, S);
  printf("Using integers: %d x %d = %d\n", ix, N, iS);
}
```

Listing 4.11 A program with unexpected results.

To simplify things we assume the numbers we want to add are all equal to each other. In particular, we consider the case where $N = 10\,000\,000$ and $x_j = 7$, $\forall j$. The expected result S is trivially $S = 70\,000\,000$. Let us consider the Listing 4.11.

Before discussing the algorithm, note the format of the `printf` statement on line 12: `%.0f x %d = %.0f`. The standard behavior of the format specifiers can be changed by placing modifiers between the `%` character and the specifier (`f` in this case). The modifier of rational variables' specifiers generally have the form `n.m`, where `n` is the minimum number of characters required to print the result (including the possible decimal point) and can be omitted. In case the number consists of less than `n` characters, blank spaces are automatically added to its left such that it occupies exactly `n` characters. If `n` is omitted, the necessary digits are printed and aligned on the left. Instead, `m` represents the maximum number of digits that are printed after the decimal point. In this case, we are asking the rational values to be printed without any digits after the decimal point. In case of integers, the modifier is usually an integer number `n` representing the minimum number of characters that are to be printed. The possibilities offered by the modifiers are obviously many more and we refer the reader to the references for more details.

If we compile and execute the program of Listing 4.11, we get the following result:

```
Using floats   : 7 x 10000000 = 77603248
Using integers: 7 x 10000000 = 70000000
```

What happened? How come the sum S is wrong by more than 10 percent in case we compute it with rational numbers? The answer is easy: due to rounding errors (Section 1.4). Contrary to what happens with integers, the computer memory representation of rational numbers is not exact, as only a limited number of digits are available. Remember that rational numbers are represented as IEEE 754 floating-point numbers.

Following the program step by step, we discover that the first deviation from the true value already occurs at $i = 2396\,746$, when the variable `S` representing the sum S should be $16\,777\,229$. At the previous step, `S` = $16\,777\,222$ which in standard IEEE 754 notation is represented as

$$\texttt{0 1001 0111 (1) 000 0000 0000 0000 0000 0011}$$

The number between parentheses is implicit (i.e. not represented), the mantissa consists of 23 digits, the exponent of 2 is obtained by interpreting the 8 bits on the left in excess–127 and the sign is given by the first bit. In this case the exponent is 24 (151 − 127) and the mantissa is obtained by summing $1 + 2^{-22} + 2^{-23}$, from which $\mathtt{S} = 2^{24}(1 + 2^{-22} + 2^{-23})$. Instead, the number 7 is simply written as $1.75 \times 2^2 = (1 + 0.5 + 0.25) \times 2^2$, i.e.,

$$\mathsf{0\ \ 1000\,0001\ \ (1)\ \ 110\,0000\,0000\,0000\,0000\,0000}$$

When this number is to be added to $\mathtt{S} = 16\,777\,222$, it must be expressed as a multiple of 2^{24}. The mantissa's bits (including the implicit bit 1) must be translated by 22 places. In this way, the rightmost digit is lost as there is no room for it in the 4 bytes provided for a `float`. The result is

$$\mathsf{0\ \ 1001\,0111\ \ (0)\ \ 000\,0000\,0000\,0000\,0000\,0011} = 6$$

Summing the two numbers, we have

$$\begin{array}{lr} \mathsf{0\ \ 1001\,0111\ \ (1)\ \ 000\,0000\,0000\,0000\,0000\,0011} = & 16\,777\,222\,+ \\ \mathsf{0\ \ 1001\,0111\ \ (0)\ \ 000\,0000\,0000\,0000\,0000\,0011} = & 6 = \\ \mathsf{0\ \ 1001\,0111\ \ (1)\ \ 000\,0000\,0000\,0000\,0000\,0110} = & 16\,777\,228 \end{array}$$

We obtain a number which differs by one unit from the correct result. Though the difference is quite contained, the final result is disastrous. This is because the error is accumulated many times during the iterations and finally leads to an error of the size shown. From what we just saw, one might expect the final result to be smaller than the correct one. Indeed, the loss of bits should lead to summing smaller numbers (first, several times 6 and then 2) to the variable `S`. However, modern CPUs contain an FPU (*Floating Point Unit*) responsible for treating floating-point numbers. To reduce the possibility that these type of errors occur, the FPU uses 80 bits to represent results internally. When they are returned to the memory, these are truncated to 32 bits, in case of a `float`.

i	R7	S	P
2396 745	16 777 222	16 777 222	0
2396 746	16 777 229	16 777 228	1
2396 747	16 777 235	16 777 236	1
2396 748	16 777 243	16 777 244	1
2396 749	16 777 251	16 777 252	1

Let us have a look at Table 4.5 listing the values of the variables `i` and `S`, the contents of the FPU register `R7`, keeping the result of the last performed operation, and the value of the FPU bit `P`, indicating whether a precision error occurred. When `i` $= 2396\,745$, `R7` $= 16\,777\,222$, representing the correct value, is copied in the variable `S`. As shown above, during the next iteration the last bit is lost. However, the result in the FPU is correct as it has 80 bits, namely `R7` $= 16\,777\,229$. When this value is copied into the memory, the truncation to 32 bits causes the variable `S` to take on the value $16\,777\,228$ and the bit `P` is assigned the value 1. During the next iteration the (wrong) value of `S` is copied into the FPU. Adding the value 7, the register `R7` takes on the value $16\,777\,235$. Again, this value is not representable in an exact way with 32 bits, and the variable `S` takes on the value $16\,777\,236$, which is the closest one near it. Due to rounding, `P` again equals 1. The successive iterations have an analogous effect. At each step, the value of `S` resulting from the previous iteration is copied into `R7`, to which 7 is added. The result is never representable with just 32 bits and must be approximated by the closest value when expressed in terms of powers of two, i.e., the correct one increased by 1. Analyzing all successive steps, we discover that almost always the value 8, and in one sole case, the value 4, is added to the variable `S`, instead of 7. Finally, the obtained result is

$$
\begin{aligned}
\mathtt{S} &= 7 \times 2396\,746 + 4 + 6 + 8 \times (10\,000\,000 - 2396\,746 - 1 - 1) = \\
&= 16\,777\,222 + 4 + 6 + 8 \times 7603\,252 = \\
&= 77\,603\,248\,.
\end{aligned}
$$

The essential cause of this unexpected result is that the addends are too different from each other ($16\,777\,222$ against 7). Obviously, the same might occur if the numbers x_j which are to be added were all different from each other, but small. The only difference is that in the latter case the result is more difficult to check and understand. The danger can be avoided if

$$
\log_2 S - \log_2 x_j \ll p\,,
$$

where p is the mantissa's precision in numbers of bit. In the case under examination $p = 23$, $N = 10^7$ and $x_j = 7, \forall j$, and therefore

$$
\log_2 7 \times 10^7 - \log_2 7 \simeq 23.253\,,
$$

which is of the order of p. Using a `double` variable, the problem would have emerged for larger N, of the order of 10^{16}. Still, it would not have disappeared. Note that integers instead are never approximated. However, as the interval of representable numbers is highly reduced, it is not always possible to use them.

It is useful to note that modern compilers can optimize a code making it faster or consume less memory. In some cases, this optimization may hide this type of problem. For example, compiling the program in Listing 4.11 on a Linux system, using the 3.3.3 (or higher) version of the `gcc` compiler with the option `-O1`, the problem vanishes.

The reason is that the optimization consists, among other things, in extracting the cycle invariants,i.e., it extracts all quantities that do not depend on the cycle. Indeed, the optimization directly transforms the cycle in

```
S = x * N;
iS = ix * N;
```

Note that we used a constant value for `x` to simplify our discussion, while in practice all values x_j are different from each other. In the latter case the optimization is not beneficial.

To avoid this inconvenience we must try to limit the difference between the addends. One way is to perform the additions in various steps (e.g., first summing M values x_j at a time and then summing these K results, with $K \times M = N$). The most extreme choice would then be to first sum all couples of adjacent numbers x_j and x_{j+1} ($M = 2$), then add these results by iterating this operation until we end up with one single result. To this purpose, we set $x_i = x_{2i} + x_{2i+1}, i = 0 \ldots N/2$ at each iteration[9], thus reducing the number of components which are to be added by a factor 2.

```
float sum=0., corr=0., x = 7.;
int i;
for(i=0; i<N; i++) {
  float tmp, y;
  y = corr + x;
  tmp = sum + y;
  corr = (sum - tmp) + y;
  sum = tmp;
}
sum += corr;
```

Listing 4.12 Kahan summation algorithm.

[9]Care needs to be taken when the number of addends are odd.

A more general method is the Kahan summation algorithm [Kahan (1965)], given in Listing 4.12.

Obviously, from an algebraical point of view, the sequence of operations performed by the algorithm is completely equivalent to that of Listing 4.11. In Listing 4.12, the variable `corr` represents, at each step, the correction which is to be made to the element `x` that is to be summed as it compensates for the error made during the previous step. The error on this correction value is negligible (asymptotically zero) as the numbers which are to be subtracted have the same order of magnitude.

Until the sums are exact, the value of `corr` remains zero and thus `y = x`. The variable `tmp` represents the temporary approximation to the sum. Indeed, in this variable it is possible that the precision is not maintained due to the information loss we discussed above. If the result were exact, the difference `(sum - tmp)` evaluated on line 7 would be exactly `-y` and thus `corr` would continue to remain zero. Instead, in case the result is approximated, the latter takes on a value equal to the quantity lost due to this approximation[10].

For the example discussed above, when `sum` $= 16\,777\,222$, `tmp` $= 16\,777\,228$. In this case `corr` $= (\mathtt{sum} - \mathtt{tmp}) + \mathtt{y} = (16\,777\,222 - 16\,777\,228) + 7 = -6 + 7 = 1$. At the next step, instead of the value 7, the value 8 is added to the sum, compensating for the error.

Always keep in mind that computer arithmetic is not so easy. Special care needs to be taken when treating numerical problems, particularly when a program contains many iterations.

Hands on 5 - Summing many small numbers

Write an algorithm computing the sum of 10^7 numbers x all equal to each other. Analyze how the error behaves as a function of x. For which values of x is the result exact? Why? Compute the sums again with the Kahan summation algorithm. Can you predict how the execution time increases with respect to the direct algorithm? When you know how to use arrays (Chapter 4.5), you can rewrite the algorithm such that it sums N different numbers x_j. Then compare the performance of Kahan's algorithm with the iterative one summing pairs of numbers as described above.

---

[10] This value cannot always be fully recovered. However, it can be proven[Goldberg (1991)] that the final obtainable result with the Kahan summation algorithm can be expressed as $\sum x_j(1+\delta_j) + \mathcal{O}(Np^2)\sum |x_j|$, with $|\delta_j| \le 2p$.