A Simple Function for Formatting Currency

Formatting currency is a nightmarishly difficult problem. Consider this correct value that would be used in India: 3,25,84,729.25 This function gives you fine-level control over currency formats and makes it simple to switch between dollars, shekels, rupees, and yen without writing separate routines.

By Adolfo Di Mare
March 26, 2012
URL:http://drdobbs.com/cpp/232700238

Programmers are often called upon to present numeric quantities formatted as money. Conceptually, this is not a tough job. It's a simple matter of generating a formatted string that corresponds to a value stored in a numeric variable.

There are complexities, of course. For example, sometimes the fractional part is truncated and sometimes it is rounded according to specific rules. There are many international currency types and many different symbols to represent them. You can't even be sure how many decimal places to use. European currencies are customarily formatted with two decimal places, but many Arab currencies require three decimals for fractional parts.

In America, commas group digits in threes and the dot is used to separate whole dollars from cents. In much of Europe, the punctuation symbols are reversed. But that's just the beginning. In India, digits are grouped in pairs, and sometimes in trios. The user who sees `Rs3,25,84,729.25` on an e-banking display screen has no trouble parsing the amount as *3 crores, 25 lakhs, 84,729 rupees, and 25 paise*.

Programmers must also contend with different ways of representing negative amounts and even the placement of currency symbols before or after numeric digits — with or without a whitespace buffer.

Clearly, programmers who deal with multiple currencies must be prepared to contend with substantial complexity. In the C and C++ world, they logically start with locales. Locales are not to my taste, however. My experience is that they lead to low-level solutions that can be hard to write, read, debug, and maintain. That's what inspired me to search for an alternate solution to the money-formatting problem.

Learning from the Past

The first programming languages I learned were Fortran and Cobol. I frequently used picture clauses in my programs. They are simple to define and use, and they provide many formatting options. Picture clauses have worked well for decades, but for some reason, they are not part of the C++ or C library. I decided to base my C/C++ currency-formatting function on Cobol's picture clauses.

The first challenge I faced was to choose a name for my function. I settled on `mnyfmt`, which is a sort of tribute to `strlen`. Like `strlen`, `mnyfmt` is a six-letter name that starts by specifying the object the function acts upon, then describes the action it performs.

Next, I had to figure out how to round numbers. I discovered that truncation is one of more than a dozen options. I learned that bankers use a special rounding scheme called "round half to even," in which rounding goes to the closer integer number, but halves always round to an even number. For example, -12.63 rounds to -13 and -12.50 rounds to -12, but +13.50 rounds to 14.

Handling so many rounding schemes is overwhelming…so I decided to decide to set the problem aside. My function does not accept a floating-point value as an argument, but instead accepts two integer parameters. One is the integer part of the number to format, and the other is the fractional part. Programmers must do their own rounding before calling my function. The invocation reads `mnyfmt(12,50)` instead of `mnyfmt(12.50)` — note the comma that separates the two numbers. That's not a decimal point.

My first implementation was a C++ template function that received a `std::string` argument. However, I discovered that I was not using any templates, nor any `std::string` functionality, so I replaced the C++ `std::string` with a regular zero-terminated character array. I also added a parameter to mark the fractional separator. In my test programs, I used commas and dots, but there are situations where other separators could be useful.

At this point, the function prototype looked like this:

```
char* mnyfmt(char *fmtstr, char dec, long intpart, int fractpart);
  {
    { // test.overwrite
      typedef struct struct_overwrite
        {
          char bytes_8[8]; // 64 bits: probably aligned
          int int_neg; // -1 usually has all its bits equal to 1
        } overwrite;
      overwrite o = {'1','2','3','4','5','6','7','\0',-1};
      assertTrue(8-1==strlen(o.bytes_8) && o.int_neg == -1);
      strcpy(o.bytes_8, "1234567.."); // 2 more bytes...
      assertTrue(9==strlen(o.bytes_8));
      assertFalse(o.int_neg == -1 && "Adjacent memory overwritten ");
      assertTrue(o.int_neg != -1);
      assertTrue(CHAR_BIT == 8 && "8 bits bytes required");
    }
  }
```

I decided to use only one `char*` argument because I know that C strings are problematic when dealing with limited memory. For example, field `bytes_8` can hold only eight characters. When 10 are copied into it, the last two overwrite whatever values are stored after `bytes_8` — in this case, changing the value of the field `int_neg`. This type of error is particularly difficult to catch.

Refining the Function

When the string that contains the picture clause is also the place where the formatted value will reside, the programmer needs to ensure that this variable is large enough to hold the resulting value. For most applications, a string of 96 or 128 bytes will be big enough. For what it's worth, you can represent the U.S. national debt of $14 trillion with just 16 digits, including two for the cents.

Best practices call for the construction of tests along with code. That's why my code is peppered with `assertTrue` and `assertFalse` statements. They mimic the assertions used in JUnit, the test framework for Java. In the code shown here, they simply output the condition tested when the assertion fails.

Earlier versions of `mnyfmt` returned a pointer to the formatted string, mimicking the behavior of `strcpy`. But it turns out that it is more useful if a pointer to the first significant digit is returned, so the same picture clause can be used to format a small value like 1235.87 or a huge one like 123456789.88. `mnyfmt` always uses the hyphen as the negative sign, and it replaces a 9 in the format string. The only character in the format string that ever gets changed is a 9.

As `mnyfmt` replaces each 9 in the format string with the corresponding digit, sometimes the result begins with a decimal separator. This special case should be handled by the programmer as there is no generalized solution that can be applied by `mnyfmt`.

Putting `mnyfmt` Through its Paces

The implementation of `mnyfmt` is quite straightforward. It locates the decimal fractional character and backs up, swapping the format character 9 with the corresponding digit. Then it does the same moving forward with the decimal digits. Any other characters remain untouched. If the operation fails for any reason, `mnyfmt` returns `(char*)(0)`. As the following code demonstrates, every invocation of `mnyfmt` should ensure that a non-null value is returned.

```
// test.example
char *sgn, fmtstr[96], buffer[96];
strcpy(buffer, "USD$ "); // picture clause
strcpy(fmtstr , "99,999,999.99999");
if ((sgn = mnyfmt(fmtstr, '.', -102455,87)))
  {
    assertTrue(eqstr(fmtstr, "0-,102,455.87000"));
    if ((*sgn=='-') && (','==*(sgn+1)))
      {
        ++sgn; *sgn='-';
      }
    assertTrue(eqstr(sgn, "-102,455.87000"));
    strcat(buffer, sgn);
    assertTrue(eqstr(buffer, "USD$ -102,455.87000"));
  }
else
  {
    assertFalse("ERROR [???]: " "-102,455.87000");
  }
```

In the following code, `mnyfmt` returns a null pointer `(char*)(0)`. The integer part of the value to format, 2455, requires space for four digits, but the picture clause has only three format characters before the decimal separator. The `fmtstr` remains unchanged.

```
// test.too.small
char *sgn, fmtstr[96];
strcpy(fmtstr, "999.99999");
if ((sgn = mnyfmt(fmtstr, '.', 2455,87)))
  {
    // never executed ==> buffer too small
    // 2455 has 4>3 digits [999.]
  }
assertTrue(sgn == 0);
assertTrue(eqstr(fmtstr, "999.99999"));
```

Due to the flexibility of the picture buffer approach, formatting can be quite adaptable. In the following code, a variable called `buffer` is used to put parentheses around a formatted value if it is negative.

```
// test.parentheses
char *sgn, fmtstr[96], buffer[96];
strcpy(buffer, "USD$ ");
strcpy(fmtstr, "9,999,999.999");
```

```
if ((sgn = mnyfmt(fmtstr, '.',-102455,87)))
  {
    if (*sgn=='-')
      {
        // put parentheses around the formatted value
        if (','==*(sgn+1))
          {
            // skip comma
            ++sgn; *sgn='-';
          }
        strcat(buffer, "(");
        strcat(buffer, sgn);
        strcat(buffer, ")");
        assertTrue(eqstr(buffer, "USD$ (-102,455.870)"));
      }
    else
      {
        strcat(buffer, sgn);
      }
  }
```

Sometimes the money amount must fill the whole picture clause and its leading non-significant digits must be displayed as asterisks. Here is how this is done:

```
// test.asterisks
char *sgn, fmtstr[96];
strcpy(fmtstr, "$9,999,999.999");
if ((sgn = mnyfmt(fmtstr, '.', -455,87)))
  {
    if ((*sgn=='-') && (','==*(sgn+1)))
      {
        ++sgn; *sgn='-';
      }
    assertTrue(eqstr(sgn, "-455.870"));
    for (--sgn; (sgn!=fmtstr); --sgn)
      {
        *sgn = '*'; // avoid writing over "$"
      }
    assertTrue(eqstr(fmtstr , "$*****-455.870"));
  }
```

Dealing with floating-point values is not hard, but some care should be taken. In the example below, the double value to format is split into its integer and fractional parts with the standard function `modf`. The value written in the program is 2455.87, but the fractional part calculated by `modf` is 86, not 87. It turns out that the binary representation of this number is not exact in machines that use IEEE 754 binary floating-point arithmetic. To get the expected result, a rounding strategy must be applied.

```
// test.modf
char *sgn, fmtstr[96];
double intdouble, fractdouble;
long intpart;
unsigned fractpart;
fractdouble = modf(2455.87, &intdouble);
intpart = intdouble; // 2455
fractpart = fractdouble*100; // .87
{
  assertFalse(fractpart == 87 && "???");
  assertTrue(fractpart == 86 && "!!!"); // binary rounding...
}
strcpy(fmtstr, "[[ 999,999.99999 ]]");
if ((sgn = mnyfmt(fmtstr, '.', intpart,fractpart)))
  {
    assertTrue(eqstr(fmtstr, "[[ 002,455.86000 ]]"));
    assertTrue(eqstr(sgn, "2,455.86000 ]]"));
```

```
    { // std::round_toward_infinity
      fractpart = ceil(fractdouble*100);
      strcpy(fmtstr, "[[ 999,999.99999 ]]");
      assertTrue(fractpart == 87 && "!!!");
      if ((sgn = mnyfmt(fmtstr, '.', intpart, fractpart)))
        {
          assertTrue(eqstr(sgn,  "2,455.87000 ]]"));
        }
    }
  }
```

Using the Code

You should be able to compile `mnyfmt` with any C compiler. Even older C++ compilers support the `(long long)` data type, which gets implemented as a 64-bit (at least) binary number that has enough range to represent most money quantities. To make it possible to use `mnyfmt` in compilers that do not support this data type, macro `MNYFMT_NO_LONG_LONG` is used to define `mnyfmt_long` (the type of the integer part of the number) as `(long)` instead of `(long long)`:

```
// test.limit
char *sgn, fmtstr[96];
#ifndef MNYFMT_NO_LONG_LONG
  mnyfmt_long max = LONG_LONG_MAX;
  strcpy(fmtstr, "999,999,999,999,999,999,999");
  // 9,223,372,036,854,775,807
  if (9223372036854775807LL == LONG_LONG_MAX)
    {
      if ((sgn = mnyfmt(fmtstr, ' ',  max,0)))
        {
          assertTrue(eqstr("9,223,372,036,854,775,807", sgn));
        }
    }
  else
    {
      assertFalse("BEWARE: (long long) is not 8 bytes wide");
    }
#endif
{
  mnyfmt_long max = LONG_MAX;
  strcpy(fmtstr, "999,999,999,999");
  // 2,147,483,647
  if (2147483647L == LONG_MAX)
    {
      if ((sgn = mnyfmt(fmtstr, ' ', max,0)))
        {
          assertTrue(eqstr("2,147,483,647", sgn));
        }
    }
  else
    {
      assertFalse("BEWARE: (long) is not 4 bytes wide");
    }
}
```

The number of trailing zeroes in the fractional part does not change the formatted value, as shown below. There is no need for a negative fractional part because the sign comes in the integer part.

```
// test.fract.zero
char *sgn, fmtstr[96];
int i,tenPow;
// The fraction 643/2136 approximates
// log10(2) to 7 significant digits.
int N = ((CHAR_BIT * sizeof(int) - 1) * 643 / 2136);
tenPow = 12;
```

```
for (i=0; i<N; ++i)
  {
    strcpy(fmtstr, "999,999.999999999");
    if ((sgn = mnyfmt(fmtstr , '.', -455,tenPow)))
      {
        if ((*sgn=='-') && (','==*(sgn+1)))
          {
            ++sgn; *sgn='-';
          }
        assertTrue(eqstr(fmtstr, "00--455.120000000"));
        assertTrue(eqstr(sgn, "-455.120000000"));
        tenPow *= 10; // 12 120 1200 12000 120000 ...
      }
  }
```

The signature for the final version of `mnyfmt` looks like this:

```
#ifndef MNYFMT_NO_LONG_LONG
  typedef long long mnyfmt_long;
#else
  typedef long mnyfmt_long;
#endif
char* mnyfmt
  (
    char *fmtstr,
    char dec,
    mnyfmt_long intpart,
    unsigned fractpart
  );
```

Quirks, Insights, and Obscure Details

Many C library functions come in multiple flavors, one for each type. For example, `lround, lroundf, lroundl, llround, llroundf`, and `llroundl` are versions of a single rounding function. The name changes because various versions return different kinds of integers and accept different types of arguments. It is not necessary to have multiple versions of `mnyfmt` because conversion to the wider integer type is always provided by the compiler.

The relationship between types `char` and `wchar_t` is not always clear. The C standard does not specify the exact type for `wchar_t`. It can be two or four bytes wide, or even one byte wide. Moreover, type `char` can be signed or unsigned. Because most picture clauses for `mnyfmt` rely on characters from the 103-character portable character subset that POSIX requires in any character set, it's easiest to use the `char` type for picture clauses and perform an explicit conversion into `wchar_t` when necessary:

```
// Convert a char[] into a wchar_t[]
char *src, chBuff[128];
wchar_t *dst, *wcBuff;
strcpy (chBuff, "Convert me to (wchar_t)");
wcBuff = (wchar_t*)(malloc(strlen(chBuff)*sizeof(wchar_t)));
for (dst=wcBuff,src=chBuff; (*dst=*src); ++dst,++src) {}
// ... C++ will let you use more sophisticated stuff
free(wcBuff);
```

Most processors use two's complement binary arithmeti, which behaves strangely when dealing with the smallest negative values. That is why `mnyfmt` fails to calculate the correct formatted result for this case and returns a null pointer:

```
// test.minus.max
long long_min = -LONG_MAX-1;
assertTrue(long_min<0);
long_min = -long_min;
```

```
assertTrue(long_min<0 && "?????");
assertTrue(long_min == -long_min);
```

The behavior of `mnyfmt` is different when formatting the integer and the fractional part of the number. In the integer part, all leading non-significant format characters get replaced by 0, but only those that immediately follow the decimal separator get changed. This is why, in the next code sample, the 1 in the format string stops the substitution, leaving the remaining characters unchanged. This code also shows a bad programming practice, as there is no check to ensure that `sgn` is not null. This error could cause a program failure when the null pointer returned by `mnyfmt` gets used to change a value in memory. Enclosing every invocation of `mnyfmt` in an `if` statement is necessary to avoid this pitfall.

```
// test.stop
char *sgn, fmtstr[96];
strcpy(fmtstr, "999,999.9999919,one9.");
if ((sgn = mnyfmt(fmtstr, '.', 2455,87)))
  {
     if ((*sgn=='-') && (','==*(sgn+1)))
       {
         ++sgn; *sgn='-';
       }
      assertTrue(eqstr(fmtstr, "002,455.8700019,one9."));
      assertTrue(eqstr(sgn, "2,455.8700019,one9."));
   }
```

The following code shows that a picture clause can be used to format rupee amounts when words are also included. This code looks for the decimal point and replaces it with a blank space to achieve a more convincing result.

```
// test.rupee
char *sgn, fmtstr[96]; char *p;
strcpy(fmtstr, "99,99,99,99,99,99,99,999.99");
// LONG_LONG_MAX == 92,23,37,20,36,85,47,758.07
// 3,25,84,729.25
// 19 digits: 12 34 56 78 90 12 34 567 89
if ((sgn = mnyfmt(fmtstr, '.', 32584729,25)))
  {
    assertTrue(eqstr(sgn, "3,25,84,729.25"));
  }
strcpy(fmtstr,"99,99,99,99,99 crores 99 lakhs 99,999 rupees.99 paise");
if ((sgn = mnyfmt(fmtstr, '.', 32584729,25)))
  {
    for (p=sgn; *p!='.'&&*p!=0; ++p)
      {
        // advance p to dec '.'
      }
    *p = ' '; // blank the dot: ".rupees" ==> " rupees"
  }
assertTrue(eqstr(sgn, "3 crores 25 lakhs 84,729 rupees 25 paise"));
// Rp3,25,84,729.25 is read as three crore(s), twenty-five lakh(s),
// eighty-four thousand, seven hundred and twenty-nine rupees and
// twenty-five paise.
```

Picture clauses can be used to format almost any type of numeric values. The following code shows how to handle dates and hours, but many more applications are possible.

```
// test.times
char *sgn, fmtstr[96];
strcpy(fmtstr, "99/99/9999");
if ((sgn = mnyfmt(fmtstr, 000, 9272002,0)))
  {
    assertTrue(eqstr(fmtstr, "09/27/2002"));
    assertTrue(eqstr(sgn, "9/27/2002"));
  }
```

```
strcpy(fmtstr ,     "99:99:99");
if ((sgn = mnyfmt(fmtstr, '?', 21435,0)))
  {
    assertTrue(eqstr(fmtstr, "02:14:35"));
    assertTrue(eqstr(sgn, "2:14:35"));
  }
```

The next code sample illustrates a mistake any programmer could make: failure to copy the format string into the formatting variable before invoking `mnyfmt`. Many programmers will choose to place their currency-formatting logic inside functions that are tailored to each application in order to prevent mistakes like this one.

```
// test.no.strcpy
char *sgn, fmtstr[96];
strcpy(fmtstr, "9,999.");
if ((sgn = mnyfmt(fmtstr, '.', 2455,87)))
  {
    assertTrue(eqstr(sgn, "2,455."));
  }
if ((sgn = mnyfmt(fmtstr, '.', 1400,87)))
  {
    // never executed: missing strcpy()
    // no char in "2,455." is a format char
  }
else
  {
    assertFalse(eqstr(fmtstr, "1,400."));
    assertTrue(eqstr(fmtstr, "2,455.")); // ???
    assertTrue("BEWARE: missing strcpy()"); // ???
    {
      strcpy(fmtstr, "9,999.");
      sgn = mnyfmt(fmtstr, '.', 1400,87);
      assertTrue(eqstr(sgn, "1,400."));
    }
  }
```

The double parentheses in the `if` statement that contains the invocation to `mnyfmt` might seem odd, but it is a good programming practice suggested by the compiler: "warning: suggest parentheses around assignment used as truth value."

A Good Enough Solution

`mnyfmt` in its current form does a "good enough" job of formatting international currencies. Further internationalization can be accomplished with a library like ISOMON, which provides simple access to ISO currency data. If you're programming in C++, you'll probably want to pack `mnyfmt` inside a wrapper that validates input and guards against other usage errors.

For further information, download the complete source code and documentation for `mnyfm`.

Adolfo Di Mare is a researcher at the Escuela de Ciencias de la Computacion e Informática, Universidad de Costa Rica, where he is full professor. He is a tutor at the Stvdivm Generale in the Universidad Autónoma de Centro America, where he is a Cathedraticum.