There is a program named "backfract" on the web (its link: Backfract. Its author also released a Go game front end and named "cogban", which can be found by Baidu search engine while backfract can't). It can render Mandelbrot set fractals. This program can display various areas of Mandelbrot set in full screen automatically and the visual effect is very good (here I list some of them backfract examples). It runs on unix/X. On its homepage it is said that the program "randomly" selects "interesting" areas to display, but does not purely zoom in, sometimes it zooms in, sometimes it zooms out next. There are also some videos on the web alleged to be able to zoom in by a factor of 10's power "N". Usually they finish playing back in several minutes. I consider whether I myself can zoom it in infinitely, at least as most possible.

In mathematics, there are two kinds of definitions for Mandelbrot set. In the book Chaos Fractals And The Applications and the book The Computer Images of Fractals And Its Applications, roughly it speaks like this: The Mandelbrot set is related to another kind of set: Julia set. Notate the quadratic mapping with point c as its parameter as \( f_c = z^2 + c \). for any point c on the complex plane there is a corresponding Julia set. Some c points' corresponding Julia sets are connected while others are not, by this, there is below definition for Mandelbrot set.

**Mandelbrot set** \( M \) is a set of parameters \( c \) that makes the Julia set corresponding to the \( f_c \) mapping of it a connected set, i.e.

\[
M = \{ c \in \mathbb{C} \mid J(f_c) \text{ is a connected set} \}
\]

If we plot every \( c \) that is accompanied by a connected Julia set on the complex plane, then the set of \( c \)'s is \( M \) set fractal. This definition is not good for computer processing, luckily, another equivalent definition of Mandelbrot set has been proved in mathematics:

\[
M = \{ c \in \mathbb{C} \mid \lim_{n \to \infty} Z_n \to \infty \}
\]

in which,

\[
Z_0 = 0 \\
Z_{n+1} = Z_n^2 + c
\]

This web page (The Mandelbrot Set) talked about handling of some problems met in the actual plotting:

(1) Judgement of infinity
In mathematics, it can be proved that the Mandelbrot set is fully enclosed inside the closed circle which takes the origin as its center and is of radius 2, that is,

\[
M \subset \{ c \in \mathbb{C} \mid |c| \leq 2 \}
\]

So if one intermediate result of the iteration is of a modulus larger than 2, then it will definitely go to infinity if the iteration continues. So we use modulus 2 as a threshold value.

(2) Times of iteration
The author of the above web page gave an example applying 50 times. But I ever met the case of different results between 50 times iteration and 200 or 1000 times iteration. So I haven't been clear about the mathematical basis of this question yet.

(3) Coloring
In fact the Mandelbrot set is only of the form of the middle (black) part. All the points outside of it do not belong to Mandelbrot set. It is not very visually appealing by simply plotting a black and white image marking whether each point belongs to Mandelbrot set or not. The various colorful details we have usually seen depend on coloring. One common method is to select the color according to the modulo arithmetic of iteration times of the "escaped" point. (One other method is similar to contour line plotting, which is not used here.) But the effect of a rather simple coloring cannot compare with that produced by a program like backfract.

(The pseudo-code that calculates whether the point of coordinate (i, j) in the drawing window belongs to the M set or not and then colors it):

```c
x0 = left_coord + delta*i;  
y0 = above_coord - delta*j;  
x = x0;  // or x = 0  
y = y0;   // or y = 0  
int k = 0;  
while( k < MAX_ITERATION )  
{  
  real = x*x - y*y + x0;  
  imag = 2*x*y + y0;  
  if( |z|^2 > 4 )  
    break;  
  else  
    ( z = real; y = imag;  
    k ++;  
  }  
if( k < MAX_ITERATION  
SetPixel hdc, i, j, color[(k % 50)];  
else  
SetPixel hdc, i, j, RGB(0, 0, 0);  
```

The floating point computation got the IEEE 754 standard, and books on computer architecture generally will talk about it. There are also books such as Floating Point Programming Principles Implementation And Application by Chun-Gen Liu (in Chinese) that illustrate it. And there is a famous article on the web: What Every Computer Scientist Should Know About Floating-Point Arithmetic.

This article got three related programs:

The first one uses machine float, that is, the double data type in c/c++ and plots Mandelbrot set in VC++. Besides, it lets the user select an area to zoom in various parts by "rubber band".

The second one adds into the above code my class implementation for floating-point number, and compare the result to that of the above machine float. But I haven't modified and finished the function of "rubber band" yet.

The third one doesn't take a GUI, it purely compare the computation result of my floating-point implementation with that of GMP fraction implementation so to check their consistency.

The three programs were developed under WinXP + VC++45 on VMware3 at the very beginning. Later they are updated to Win7 + vs2015(MFC) installed on VMWare6, with minor modifications. Both versions are provided here:  
VC++6 version  
vs2015 version
The 1st program

This code part of operation is nearly the same as the above pseudo-code:

```c++
for(i = 0; i < w; i++)
    for(j = 0; j < h; j++)
    {
        b_is_boundary = FALSE;
        if(delta_h > delta_b)
            if( j == (int)((h*delta - (old_above_coordinate -
                                    old_below_coordinate))/(2*delta)))
                b_is_boundary = TRUE;
        else
            if( i == (int)((w*delta - (old_right_coordinate -
                                    old_left_coordinate))/(2*delta)))
                b_is_boundary = TRUE;
        double x0 = left_coordinate + delta * (double);
        double y0 = above_coordinate - delta * (double);
        double x = x0;
        double y = y0;
        int k = 0;
        #define MAX_ITERATION 1000
        while(k < MAX_ITERATION)
            {
                double real_part = x*x - y*y + x0;
                double imaginary_part = 2*x*y + y0;
                double x_square = real_part*real_part + imaginary_part*imaginary_part;
                if(x_square > 4.0)
                    break;
            }
        else
            {
                x = real_part;
                y = imaginary_part;
            }
        k++;
        if(k < MAX_ITERATION)
            {
                //unsigned long colors[16] =
                //unsigned long colors[50] =
                //RGB(0x0D , 0x88 , 0x87),
                //RGB(0x15 , 0x19 , 0x70 )
                SetPixel(hdc, i, j, colors[k%16]);
                SetPixel(hdc, i, j, colors[k%50]);
                if(b_is_boundary)
                    SetPixel(hdc, i, j, RGB(255, 255, 255));
            }
        else
            {
                SetPixel(hdc, i, j, RGB(0, 0, 0));
                if(!b_is_boundary)
                    SetPixel(hdc, i, j, RGB(255, 255, 255));
            }
    }
```

Its running result is like below (To save the space, below pictures displayed are those zoomed out by the browser. To browse them in their original size, click this link. Or you can right click below pictures and save them to your local storage for watching.)
The proportion of the area selected to zoom in by the user is very likely not the same as that of the drawing window. So I made some extension to it to adapt to the drawing window so as to make every pixel point inside the drawing window being plotted. The inner part within the two white lines in each of the above images is the area selected by the user (the visual effect maybe is poor for small pictures, you can turn to the original ones), which may appear too flat or too high for the drawing window. The judgement processing of “b_is_boundary” in the above code is for this reason.

(Two intermediate text files draw_coordinates bit and arguments bit are generated after you run the program. You need to delete them manually before the next run.)

The machine double float is of a binary precision of 53 bits, roughly 16 decimal places of accuracy. The starting horizontal and vertical coordinates are both of a magnitude of about (±2.2). If each time the dimension of the selected area for zooming in is about one tenth of that of the original image, we may find that after about 16 times of selection the image went “mesiac.” Bare 16 times, this is far less than infinitely zooming in. Besides, we will see behind that after about 40 times of iteration, the log already shows that the accumulation of error from machine float operation leads to obvious fault.

(2) my floating-point class implementation

Since the calculation is only concerned with addition and multiplication operation, we should be able to do completely accurate calculation for the terminating decimals. The arithmetic taught in primary school should suffice. In order that we only need to compute terminating decimals, we only use starting coordinates and step values with finite fraction part. The latter is achieved by specifying the width and height of the drawing window as some particular values such as 500/512/600/1000/1024 so that the gap value between two pixel points is also a simple terminating decimal. That’s why I modified the code in View.cpp to support only these particular values.

```
my_mandel_test_my_floats_v2015/my_mandel_test_my_floats_v2015View.cpp
```

The reader maybe still needs to modify the code in View.h, setting the width and height of the program window that suit your display area and meet the above specification.

```
my_mandel_test_my_floats_v2015/my_mandel_test_my_floats_v2015View.h
```

The starting coordinates are: x: -2.5 to 1.5, y: -2.0 to 2.0:

```
my_mandel_test_my_floats_v2015/my_mandel_test_my_floats_v2015View.h
```

Three new files (my_floats.h, my_float1.cpp and my_float2.cpp) that implement my floating-point class and several helper functions were added. (Sometimes it reported error when I built the project under VC6 but passed after I re-added those files to the project. And it could never pass building under vs2015, reporting re-definitions found in nativedlib/Unsourced.lib and libcmtd.lib. To solve this issue, one maybe need to adjust the project. First, ignore these two libraries in the “input” entry of the “link” option of the project, then, add these two libraries as per the above order to the “Other library dependencies.”) Two different kinds of data structure for floating-point were implemented successively. Because it occurred to me that the latter new one maybe would be better than the former one that had been implemented, I did them one after another.

The first implementation:

```
my_floats.h
```

```
my_mandel_test_my_floats_v2015/my_mandel_test_my_floats_v2015View.cpp
```

The second implementation:

```
my_floats.h
```

```
my_mandel_test_my_floats_v2015/my_mandel_test_my_floats_v2015View.cpp
```

Sometimes abridged as:
Both classes use the member "minus" to represent whether the floating-point number is positive or negative, so the digits string of the number (member "number" or "digit") all go without symbol of sign. The first difference between the two classes is that the decimal point was reserved in the digits string in the previous class, but it vanished in the latter class. The second difference is the mark of the place of the decimal point in the previous class it is counted from left to right (point_pos), in the latter class, however, is counted from right to left (point_pos). Thus the implementation code of the latter class is more concise. The computation results of these two implementations can be compared to each other to verify the code. This is a second way for verification. But the cores of the algorithm of these two classes are the same, so probably its function for verification is limited. A complete verification relies on the third program below which implements the contrast of the computation results herein to that of the GMP calculation. That is a third way for verification, also can be said as the real verification. (A first way for verification is a rather local one, it only verifies addition, i.e. do a reverse operation back after an addition finishes and see if the result can be restored correctly. This way is little used.)

Floating-point classes being accomplished, take the class implementation as an example, the code is changed to roughly like this (The actual code will interleave a little because of the need for result comparison. The example here is of an early version of code.)

```cpp
#define MAX_ITERATION 100/10/100
while (k < MAX_ITERATION)
{
    if (k < MAX_ITERATION)
    {
        y0I = fi_above_coord;
        if (y0I < threshold)
        {
            break;
        } else {
            x0I = fi_below_coord;
            y0I = fi_above_coord;
        }
    }
    if (k < MAX_ITERATION)
    {
        SetPixel(hdc, i, j, colors[k+8]);
    } else {
        SetPixel(hdc, i, j, RGB(0, 0, 0));
    }
    y0I = y0I - deltaI;
    x0I = x0I + deltaI;
}
```

The `fi` class code is similar, basically it just changes "i" to "II" or appends a suffix "ii".

The illustrations showing the running result (Now one can see the program is drawing the pixels of the screen one by one. Below are two images, an early one on the left and a late one on the right.)
Below we give a graphic illustration for the log file:
(The blue wireframes orderly marked the coordinates in the window, the coordinates in the complex plane (the strings are in two kinds of form: f form and fill form), iteration order, total iteration times when the computation loop ends.)

(The red wireframes orderly marked the real part, the imaginary part and the square of the modulus of z computed using machine floating-point.)

(The green wireframes orderly marked the real part, the imaginary part and the (high-precision approximation value of) the square of the modulus of z computed using my class codae (f and fill, which have the same result values.)

It can be seen that the point that lies at (0 0) exits the loop in just one time of iteration, which is outside of M set. If then goes to compute the next point (0 1).

Below we give one more example, the pixel point (16 125):

The contents marked by the purple wireframes at the end of the digit strings are different. This is not because the computation results are different. It is because that I only print up to 200 digits at most while the length of the string of the result value is larger than 200. The string in fill form got one more character, i.e. the decimal point, than that string in fill form, so its output got one less digit than the latter.

It can be seen that the point lies at (16 125) is on the real axis and belongs to M set. So the iteration reached the maximum (here it is 10 times) and the value still didn't exceed the threshold. The code did no more calculation and went to next point (16 126).

It brings dense computation to conduct accurate calculation using my floating-point classes. In my code, the calculation code was simply placed inside the OnDraw() function, this resulted in that the program didn't respond to external messages. This no responding maybe doesn't matter much under Win XP yet, one can still see the program is still drawing on the screen one pixel by one pixel. But it becomes worse under Win 7 the program draws no more than a few pixels and then just displays "The program does not respond."

To avoid blocking message loop because of dense computation (Blocking message loop is not blocking process, OnDraw() is located in the main
Being able to conduct accurate calculation using my floating-point classes, it may be found that how the error from machine floating-point (i.e. double type) computation which is intrinsically approximate could accumulate. Below is the log example. (log files for download)(Below extracted part is at the very end of the log file)

Since the iteration times is large, here it is the approximation in some certain precision of the calculated result using my floating point class that is compared to the calculated result using machine floating-point, of course, this precision is highly better than that of double type.

Here it is the iteration of the point of coordinate (16, 125) that is logged:

(The value inside the red rectangle is machine floating-point result. The value inside the green rectangle is my floating-point result.)

(P.S. In the path showed in below image, the name of the directory that log.txt lies in is in Chinese, which means "logs comparison that shows machine floating-point got obvious error after more than 40 times of iteration"; see the sub-directory name of the above log file for download.)

When \( k = 0 \), the results are the same:

When \( k = 40 \), the difference isn't large:

When \( k = 47 \), there is difference at the first decimal place,
When $k = 52$, already we cannot think they are close.

Since $k = 53$, the results might have got different plus or minus sign!
(It seems that "backfrac" uses "long double" type instead of "double". Still this is too little.)

The function to select a region to zoom in is not implemented in this version of code. Besides, the zooming-in of M set is endless. I need to consider how to zoom it in on earth. Manual selection? Or automatic selection? How should I present the image patterns of the M set?

The 3rd program

(G) GMP implementation

GMP library is the GNU "big number" library. The GMP library I used was built under MinGW on WinXP on VMWare3, it can then be directly linked into the applications developed under VC++. It also included the more libraries of MinGW and Windows. When I upgrade, I update my program on Win 7+ vs2015 on VMWare6, the library itself is still usable, only that the printf() function cannot be used for the reason of libswprintf of MinGW. So I replaced all the printf() with "write()’. For running, some VC libraries from VS and some libraries from MS SDK that are prompted to be in need are added.

GNU big number libraries are composed of GMP, MPFR, MPC. GMP is a big integer and fraction number library. It also included function part for floating-point operation but that now is replaced by MPFR (said in GMP manual). MPFR is arbitrary precision big floating point number library. It is based on GMP. MPC is big complex number library and it is based on the former two libraries. (It is interesting that GCC building is dependent on these libraries.) Besides, there is another library MPFR branching from GMP. It is said that one of MPFR’s goals is to become a big number library that can be built with VS under Windows because GMP cannot. Here MPFR is not used. I ever thought I would need MPFR to realize the function needed, thus “mpfr” is used in the file name (“mandel_with mpfr_test”) but later I found that mere the fraction part of GMP library could suffice. And the latter calculates in absolute precision, not limited to the arbitrary precision of the former. Yet the file names are not modified. Actually nearly my whole program has nothing to do with MPFR except only at one place I use it to print a log record.

I installed a MinGW-MSYS offline-install package downloaded from the web on WinXP on VMWare. As the manual and "configure" help guided. I configured and compiled GMP and MPFR (The source package of MPFR got a file (INSTALL) talking about three ways for building it under Windows (MinGW, Cygwin, and vs2015)). There is nothing peculiar. But the library to be built can only be specified either as static or as shared/dynamic each time. I just built static ones. The two static libraries (lbcmpf.a and lbcmpf.a) obtained can directly be used by VC++6 and vs2015.

All other lib files are MinGW libraries that are needed when building the program. Besides, to run the program one need lbcmpf_a.dll of MinGW and some dlls of Windows.

GMP is implemented in c. It got a C++ wrapper library that could be configured and built. But the manual said that this c++ library generally can only be used by the application built with the same (c++) compiler. In development I found that I must use c compiler instead of c++ compiler to compile one source file that includes gmp.h, so my mpfr_lib.c has to be c file instead of cpp file.

Link order is required for MPFR library, first lbcmpf.a, then libcmpl.a. And the manual says that in some cases one must specify /MD option for the Windows cl compiler to build c++ application using GMP library.

(There is an open source project on fractals named "FractalsNow". It got MPFR computation joined into its code since version 8.0. This project is dependent on Qt. I built this project under both Ubuntu and WinXP MinGW. It got a debian package in the Ubuntu repository and was rather easy for building. On Windows, however, the Qt 4.8 depends on includes GCC4.4 for building. I could only search for and download one MinGW package with GCC4.4, then copy the earlier MSYS directory, moreover, download the "pthread-w32" package, thus accomplished its building.)

Running effect:
As to the log file, it got some more contents, mainly the GMP calculating results and the MPFR print-out.

The first image below is the case for the point of coordinate (22, 0). (If you don’t modify the program source downloaded, most probably you’ll get the log with a start like this here. This is because I skipped the points ahead and started from column 22 when debugging. The code hasn’t been cleaned up, so this is not removed.)

The meaning of the blue, green and red marking wireframes is the same as before. The ochrous wireframes marked the results calculated using GMP while printing MPFR. What is more, thickened wireframes in corresponding color are newly added. What they marked are absolutely accurate values and not high-precision approximate values. Orly, the exact value of the real part, the approximate value of the real part, the exact value of the imaginary part, the approximate value of the imaginary part, the exact value of the square of z’s modulus, the approximate value of the square of z’s modulus calculated using my floating-point implementation are verified against GMP implementation.

It finished while looping only once in the above illustration and got a short log. Let’s further observe the k=0 time of iteration of the point of coordinate (22, 125). Now the log is very long and can’t be all displayed in just one image. And here I can only replace many middle lines of digits with one “*” line. In the first illustration below it accomplished the comparison between the exact value of the real part and the approximate value of the real part.
In this second illustration below it accomplished the comparison between the exact value of the imaginary part and the approximate value of the imaginary part. For this point is on the real axis with an imaginary part of 0, the log is short.

Finally in this third illustration below it accomplished the comparison between the exact value of the square of z's modulus and the approximate value of it.

The unmarked text part with words like "FRACTION ... / ... I" in these illustrations is the representation in fraction form that GMP used, i.e., numerator / denominator, and it is irreducible.

This third program goes without GUI and got only pure calculation. The code now lies in main.cpp instead of View.cpp. All the newly added functions for GMP implementation begin with "GMP_" and lie in two files (mpfr.h and mpfr_lib.c) as above said, named after "mpfr". I illustrate it below what to do to turn the pseudo-code before to GMP code.
The only modification to my floating-point code this third program made is when an approximation is made to the (intermediate) calculated result of my floating-point implementation, to continue to compare the results and check the consistency of the two implementations. I need to make an corresponding approximation to the (intermediate) calculated result of GMP implementation. That is, I add one more parameter(p_pos) to the truncate_float() function to tell GMP implementation side which decimal place I truncated at here, so that you must truncate correspondingly at that same decimal place of your value there.

Code before and after modification:

The complete code of the above right side:

The corresponding modification to my_floats.cpp will be seen soon below.

Currently truncate_float() is not modified because the results of floatI and floatII are the same, I only need to use one of them, currently I use floatI.

After GMP implementation was added, I found and corrected one (preliminary level, -.-) bug in my_floats.cpp/my_floatII.cpp is the same):

Why we need approximation yet? Because with absolutely accurate computation (not a bit approximation) the need for storage for the intermediate decimal result is incrementing exponentially by 2 to the power of the times of iteration. It counts in K for 10 times of iteration, and counts in M for 20 times of iteration. For 30 times of iteration it counts in G already. Besides, with computation like this on PC one becomes to feel a little slow already after about 10 times of iteration (i.e., a magnitude of 1000 digits times 1000 digits). So I have to do some approximation in the actual computation, for example, to let the need for storage increment linearly with the times of iteration. And take the speed of getting result into consideration.

Presently the algorithm for doing approximation I considered is roughly like this:
Step (0): Specify a requirement for the promotion of precision, e.g., 16 more decimal places of accuracy than the old ones;
Step (1): Specify the decimal places of accuracy, need to take these into consideration:
(A) the precision of starting coordinate,
(B) the precision of the step value between coordinates,
(C) incrementing with the times of iteration?\)
Step (2): Does the calculated result exceed the precision requirement? If not, do not modify the value of the result;
Step (3): If the answer is "yes" in step (2), count to the wanted decimal place of accuracy, then
(A) if there is already significant digit in its front, then add at most 16 more decimal places after it
(B) if there isn't any significant digit e. all are '0's) in its front, unless being of value 0, find the first significant digit(non-zero one) after it, then add at most 16 more decimal places after that digit.

Its implementation (taking floatI as the example) ([highlighted lines of code is the modification of this function as said above])

mandel_with_mpfr_test_vs2015/my_floatII.cpp:

```cpp
extern floatII truncate_floatII(int x_precision, int y_precision, int delta_precision,
int min_precision_to_truncate, floatII f, int add_to_tail_precision,
int* p_pos); //
```
The modification needed for the code that includes approximation:

```c
for(j = 0; j < w; j++)
{
    y0 = above;
    for(i = 0; i < h; i++)
    {
        x = 0;
        y = 0;
        while(k++ < MAX_ITERATION)
        {
            real = x*x - y*y + x0;
            imag = 2*x*y + y0;
            z_square = real*real + imag*imag;
            if(z_square > threshold) break;
            else
            {
                x = real;
                y = imag;
            }
        }
    }
}
```
The implementation of GMP truncating function in it: taking GMP_truncate_real() as an example, the GMP_truncate_insg() code just replace “real” with “insg”:

```c
void GMP_truncate_real(char* str_numerator, char* str_denominator,
                        char* str_gmp mpfr, int* gmp, int mpfr_precision,
                        int len, int Trunc_pos)
{
    mpz_t numerator;
    mpz_t denominator;
    mpz_t a;
    mpfr_t f1, f2;
    mpfr_t exp;
    mpz_init(numerator);
    mpz_init(denominator);
    mpz_init(a);

    if(mpz_set_str(numerator, str_numerator, 10) != 0)
    {
        return;
    }

    int n;
    char* str;
    char p;
    mpz_set(denominator, mpz_densref(gmp_real_approx));
    //gmp_printf("denominator is %dn", denominator);
    mpz_set_si(a, 10, 1);
    n = 0;
    while((mpz_cmp_si(denominator, 1) != 0)
    
```
So far, we've haven't looked at the addition and multiplication code yet. Though it's only primary school arithmetic, below we'll take a rough look at it. Although on the above the above we've showed it is really powerful so maybe to use GMP and MPFR to do more computation will be a better choice later (because it allows people to use fractions, that is, repeating decimals, and other functions etc. that I myself haven't got). It might be better to have one more kind of implementation in parallel when available. However, if not available, I can just use GMP.

Here I take the float11 class as the example, the implementation code of float11 class is longer, the float11 class is more concise, both of their algorithms are the same. There isn't very much to say about addition.

```cpp
mandel_with_mpfr_test_vx2015<any_float11.cpp:

1  float11 float11::operators ( float11 & f2 )
2  {
3      // TODO: Implement
4      int sign, pos;
5
6      // align point
7      float11 f1 = *this;
8      int int_len1 = strlen(f1.digits) - f1.point_pos - 1;
9      int int_len2 = strlen(f2.digits) - f2.point_pos - 1;
10     int frac_len1 = f1.point_pos;
11     int frac_len2 = f2.point_pos;
12     int max_int_len = int_len1 > int_len2 ? int_len1 : int_len2;
13     int max_frac_len = frac_len1 > frac_len2 ? frac_len1 : frac_len2;
14
15     float11 f1_old = f1;
16     float11 f2_old = f2;
17
18     // pos
19     pos = max_frac_len;
20     int len = max_int_len + max_frac_len;
21
22     char * s1 = f1.extend_zeros( max_int_len-int_len1, max_frac_len-frac_len1 );
23     char * s2 = f2.extend_zeros( max_int_len-int_len2, max_frac_len-frac_len2 );
24
25     if(f1.minus == f2.minus)
26     {
27         // minus assign
28         sign = !f1.minus ? -1 : 1;
29         int carry = ((int) *malloc( (len+1) * sizeof(int) ));
30         char * p = (char *) *malloc( len );
31         memcpy(p, 0, (len+1)*sizeof(int));
32         char * q = s1+len;
33         char * r = s2+len;
34         int k = 0;
35
36         while(k <= s1)
37         {
38             m = chtol(q[k]) + chtol(r[k]) + carry[k];
39             if(m >= 20)
40             {
41                 carry[k+1] += 1;
42                 r[k] = itoeh(m-20);
43             }
44             else
45             {
46                 r[k] = itoeh(m);
47             }
48             q[k] = p[k] = r[k];
49             k++;
50         }
51         char * str = (char *) *malloc((len+1)); // doesn't care if more
52         int i = 0;
53         int j;
54         if(carry[i] > 0)
55         {
56             for(j = 0; j < len; j++)
57                 str[j] = r[j];
58         }
59         str[i] = '\0';
60
61     float11 str_result(str, sign, pos);
62
63     free(str); // do it in destructor?
64     free(r);
65     free(carry);
66     free(s1);
67     free(s2);
68
69     return str_result;
70   }
71   else
72   {
73       // to decide sign
74       char * p = s1;
75       char * q = s2;
76
77       while("p" != "q") // && "q" != "q"
78       {
79         if(chtol(p[0]) != chtol(q[0]))
80             break;
81         p++;
82         q++;
83       }
84       if("p" == "q")
85       {
86         free(s1);
87         free(s2);
88
89         float11 f4("0");
90         if(My_DEBUG) printf("they are equal, minus result is 0\n");
91         return f4;
92       }
93       else
94       {
95         bool t = chtol(p[0]) > chtol(q[0]) ? f1.minus : f2.minus;
96         sign = t ? 1 : 1;
97
98         char * big = ("p" > "q") ? s1 : s2;
99         char * small = ("p" > "q") ? s2 : s1;
100        p = big + len-1;
101        q = small + len-1;
102
103        for(int k = len; k >= 0; k--)
104        {
105            if(chtol(p[k]) != chtol(q[k]))
106                carry[k] = 1;
107            else
108                carry[k] = 0;
109
110            m = chtol(p[k]) + chtol(q[k]) + carry[k];
111            if(m >= 20)
112            {
113                carry[k+1] += 1;
114                r[k] = itoeh(m-20);
115            }
116            else
117            {
118                r[k] = itoeh(m);
119            }
120            q[k] = p[k] = r[k];
121            k--;
122         }
123
124         char * str = (char *) *malloc((len+1)); // doesn't care if more
125         int i = 0;
126         int j;
127         if(carry[i] > 0)
128         {
129             for(j = 0; j < len; j++)
130                 str[j] = r[j];
131         }
132         str[i] = '\0';
133
134         float11 str_result(str, sign, pos);
135
136         free(str); // do it in destructor?
137         free(r);
138         free(carry);
139         free(s1);
140         free(s2);
141
142         return str_result;
143   }
It prepares to align the decimal point beginning from the highlighted line 7. Line 20 obtained the position of the decimal point. Line 24 and 25 supplement 0’s in an aligning style. Line 27 is about to do addition if the signs of the two numbers are the same. Its next line of code begins to do the addition, recording the carry value. Beginning from line 37 it prepares for the result string, which reads from left to right, while the addition itself is done from the lower to the higher. i.e. from right to left. Line 66 constructs the object of flt class using the string and returns from the function. Line 64 is about to do subtraction for the signs of the two numbers are different. First it does comparison between their absolute values to decide which one is larger and what sign the difference will be. Line 94 shows that they are equal so the code returns 0. Line 104 is about to subtract the smaller absolute value from the larger one. Next it does subtraction, borrowing, etc. When done it likewise constructs the string, then constructs the object of flt class. Thus done.

The multiplication costs two "charm" operator overloading functions, however. The multiplication of two objects of the class is transformed to first doing multiplications of one object of the class and one integer variable (i.e. a number multiplied by every digit of another number), then summing them up after shifting their decimal points correspondingly. (For example, if we got to calculate \( f_1 \times f_2 \), in which \( f_2 \) is of the form "xyz.w", then the result would be:

\[
\text{result} = f_1 \times x \times 10^n + f_1 \times y \times 10^{n-1} + f_1 \times z \times 10^{n-2} + \ldots + f_1 \times t \times 10^{-2} + f_1 \times 0 \times 10^{-3} \quad (-3=\text{max_frac_len})
\]

Thus I used two operator overloading functions.
The highlighted lines are those illustrated above. In the first function it does carrying similar to that in addition.

Having long been a c programmer up until now, I know very little about c++. It looks more natural using operator overloading in c++ to operate on the floating-point number class. However, it feels more portable for c libraries than c++ libraries when one is using GMP.

Once I found the memory usage of the third program soon went too high, and it kept increasing. After debugging I found out that some pieces of code modifying “malloc/free” at random/delete” led to this problem. This is not memory leak yet, for the memory is restored as soon as one closes the program. But under such memory usage the program will soon be not able to run. It seems the memory allocated through “new” has not been returned to the OS quickly after “delete”, while “free” does. Since the program does large amount of computation persistently, no memory leak is crucial to the program.

It is rather good that the result of my implementation is consistent with GMP computation, using not very much code. GMP uses the fraction form, this enlarges the range of operation greatly compared to using decimal form by me. Surely it got a more complicated algorithm. The algorithm of mine is only for multiplication and addition, so it is just suitable for computing Mandelbrot set( don’t know if it is well suitable for computing Julia set.) and may not be enough for computation of other fractals.

Besides Win32, I would like to implement it on Linux/GTK.

I ever thought it was best to distribute the programs on ReactOS. But I tried ReactOS on VMWare and found its compatibility and stability were not suitable for actual use yet. .......

(P.S.: The precision in MDVR means binary precision, this may not easily match the decimal precision I adopted in my floating-point implementation.)