19.3 NUMBERS – A MORE COMPLEX FORM OF DATA

Bitmaps are all very well, but you don't use them that often. So, we need an example of a class that might get used more widely.

How about *integers* . You can add them, subtract them, multiply, divide, compare for equality, assign them, ... You may object "*C ++ already have integers*". This is true, but not integers like

```
96531861715696714500613406575615576513487655109
```

or

```
-3344455556666667777777888888888999999
```

or

```
1768903465689120298563508127645397063678932557896556343736815447
4538966508771954347888099842255478686968873654661499873498742216
9350583273568437852654327890623572325643485 6
```

The integers we want to have represented by a class are LARGE integers; though just for our examples, we will stick to mediumish integers that can be represented in less than 100 decimal digits.

Why bother with such large numbers? Some people do need them. Casual users include: journalists working out the costs of pre-election political promises, economists estimating national debts, demographers predicting human population. The frequent users are the cryptographers.

There are simple approaches to encrypting messages, but most of these suffer from the problem that the encryption/decryption key has to be transmitted, before the message, by some separate secure route. If you have a secure route for transmitting the key, then why not use it for the message and forget about encryption?

Various more elaborate schemes have been devised that don't require a separate secure route for key transmission. Several of these schemes depend in strange ways on properties of large prime numbers, primes with 60...100 digits. So cryptographers often do get involved in performing arithmetic with large numbers.

Representing large integers

The largest integer that can be represented using hardware capabilities will be defined by the constant `INT_MAX` (in the file limits.h):

```
#define INT_MAX         2147483647
```

but that is tiny, just ten decimal digits. How might you represent something larger?

One possibility would be to use a string:

```
"123346753245098754645661"
```

Strings make some things easy (e.g. input and output) but it would be hard to do the arithmetic. The individual digits would have to be accessed and converted into numeric values in the range 0…9 before they could be processed. The index needed to get a digit would depend on the length of the string. It would all be somewhat inconvenient.

A better representation might be an array of numbers, each in the range 0…9, like that shown in Figure 19.2. The array dimension defines largest number that can be represented, so if we feel ambitious and want 1000 digit numbers then

```
fDigits[1000];
```

should make this possible.

Each digit would be an unsigned integer. As the range for the digits is only 0…9 they can be stored using "`unsigned char`". So, a large integer could be represented using an array of unsigned characters.

0	0	0	0	0	9	0	8	5	7	6	6	3	0	4	2	1	
																	+
0	0	1	0	2	0	0	0	4	0	0	0	8	0	0	7	0	
																	=
0	0	1	0	2	9	0	8	9	7	6	7	1	0	4	9	1	

Figure 19.2 Large integers represented as arrays.

Arithmetic operations would involve loops. We could simply have the loops check all one hundred (or one thousand) digits. Generally the numbers are going to have fewer than the maximum number of digits. Performance will improve if the loops only process the necessary number of digits. Consequently, it will be useful if along with the array each "Number" object has details of its current number of digits.

Of course there is a problem with using a fixed size array. Any fixed sized number can *overflow* . Overflow was explained in Part I where binary hardware representations of numbers were discussed. Figure 19.3 illustrates the problem in the context of large integers. The implementation of arithmetic operations will need to include checks for overflow.

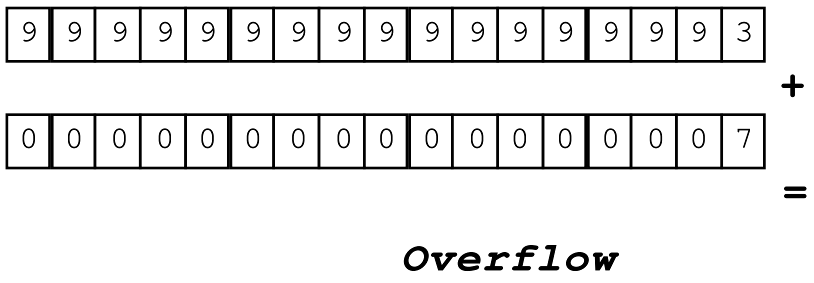

Figure 19.3 "Overflow" may occur with any fixed size number representation.

Signed numbers

There has to be a way of representing signed numbers. The smart way would be to use a representation know as "tens-complement". But that is getting really way out of our depth. So instead, we can use "sign and magnitude". The array of digits will represent the size (magnitude) of the number; there will be a separate data member that represents the sign.

Sign and magnitude representations do have a problem. They complicate some arithmetic operations. For example, you have to check the signs of both numbers that you are combining in order to determine whether you should be doing addition or subtraction of the magnitude parts:

```
+ve number      PLUS     +ve number   =>   Do addition
+ve number      PLUS     -ve number   =>   Do subtraction
-ve number      PLUS     +ve number   =>   Do subtraction
-ve number      PLUS     -ve number   =>   Do addition
+ve number      MINUS    +ve number   =>   Do subtraction
+ve number      MINUS    -ve number   =>   Do addition
...
```

As well as performing the operation, you have to determine the sign of the result.

Doing the arithmetic

How does the arithmetic get done?

The algorithms that manipulate these large integers are going to be the same as those you learnt way back in primary school. We are going to have to have

```
addition with "carry" between columns,
subtract with "borrow" between columns,
long multiplication,
long division.
```

Figure 19.4 illustrates the mechanism of addition with "carry". This can be encoded using a simple loop:

```
carry = 0;
for(int i=0; i < lim; i++) {
    int temp;
    temp = digit[i] of first number +
                  digit[i] of second number +
                         Carry;
    if(temp>=10) {
           Carry = 1; temp -= 10;
           }
    else Carry = 0;
    digit [i] of result = temp;
    }
```

The limit for the loop will be determined by the number of digits in the larger magnitude value being added. The code has to check for overflow; this is easy, if `Carry` is non-zero at the end of the loop then overflow has occurred.

The process of subtraction with "borrow" is generally similar. Of course, you must subtract the smaller magnitude number from the larger (and reverse the sign if necessary).

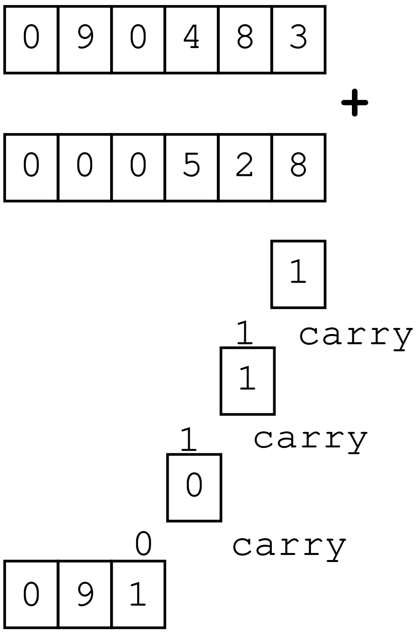

Figure 19.4 Mechanism of addition with carry.

One could do multiplication by repeated addition, but that is a bit slow. Instead a combination of "shifts", simple products, and addition steps should be used. The "shifts" handle multiplying by powers of ten, as shown in Figure 19.5. The process would have to be coded using several separate functions. An overall driver routine could step through the digits of one of the numbers, using a combination of a "times ten" and a "product" routine, and employing the functions written to handle addition.

```
for(int i=0;i<limit; i++)
    if(digit[i] != 0) {
           temp = other number * 10^i
           use product function to multiply temp
                  by digits[i]
           add product into result
           }
```

(
Check the outline multiplication code; you should find that this is the algorithm that you use to do "long multiplication".)

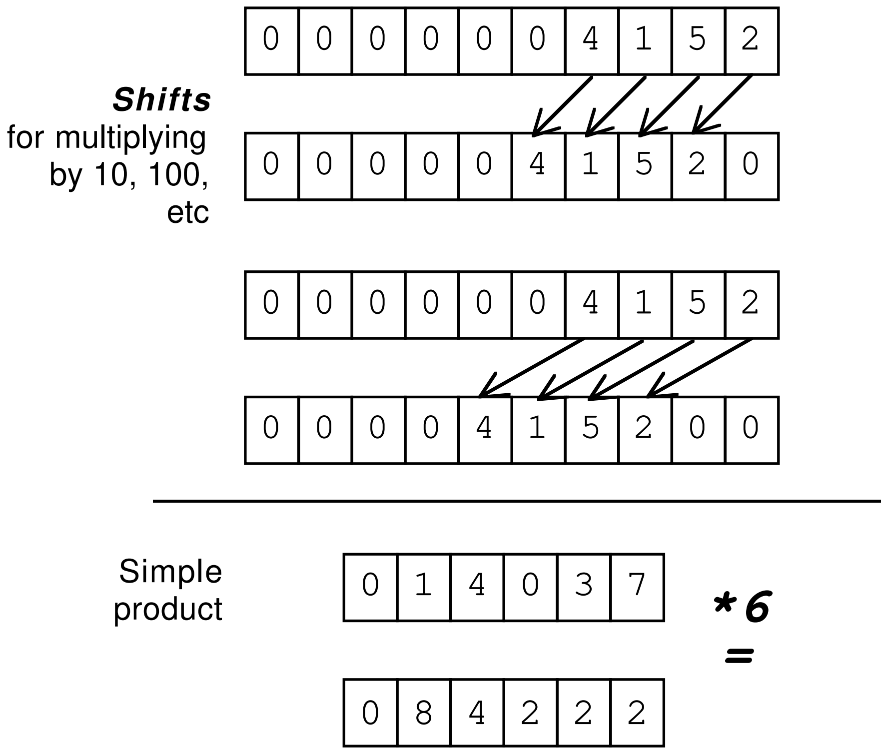

Figure 19.5 Shift and simple product sub-operations in multiplication.

The hardest operation is division. A form of "long division" has to be coded. This is actually quite a hard algorithm to code. But you don't have to. It is standard, so versions are available in subroutine libraries.

P. Brinch Hansen recently published a version of the long division algorithm("Software Practice and Experience", June 1994). His paper gives a model implementation (in Pascal) with the long division process broken down into a large number of separate routines that individually are fairly simple. (The implementation of the `Number` class includes a C++ version of Brinch Hansen's code. You should read the original paper if you need a detailed explanation of the algorithm.)

Specification

Implement a class that represents multi digit integer numbers. This "Numbers" class should support the basic arithmetic operations of addition, subtraction, multiplication, and division. It should be possible to transfer Numbers to/from disk, and to print readable representations of numbers. Comparison functions for numbers should be defined. It should be possible to initialize a Number from an ordinary long integer value, or from a given string of digits.

Design

The data owned

First, consider the data. What data does an individual `Number` own? Each instance of class `Number` will have an array of unsigned long characters to store the digits. In addition, there has to be a "sign" data member (another unsigned character) and a length data member (a short integer). The size of the array of unsigned characters can be defined by a #define constant.

```
unsigned char      fDigits[kMAXDIGITS+1];  // +1 explained later
unsigned char      fPosNeg;
short              fLength;
```

Services provided by a Number

The next step is to decide what "services" a `Number` object should provide. As in the previous example, you make up a list of useful functions – things that a `Number` might be expected to do. The list is definitely not going to be the complete list of member functions. As already noted, functions like "Multiply" and "Divide" are complex and will involve auxiliary routines (like the "times 10" and "product" needed by "Multiply"). These extra auxiliary routines will be added to the list of member functions as they are identified during the design process. These extra functions will almost always be private; they will be simply an implementation detail.

When writing a test program to exercise a class, you may identify the need for additional member functions. Getting a class right does tend to be an iterative process.

The following functions form a reasonable starting point:

- Constructors.
  Several of these. The simplest should just initialize a `Number` to zero. The specification required other constructors that could initialize a `Number` to a given long and to a given string.

  Another constructor that would probably be useful would initialize a Number given an existing Number whose value is to be copied.

- Add, Subtract, Multiply, Divide.
  These combine two numbers. It seems reasonable to make them functions that return a Number as a result. A Number isn't too large (around 100-110 bytes) so returning a Number in the stack is not too unreasonable. A function giving a Number as a result would also be convenient for programs using Numbers.

- Zero.
  Probably useful, to have a function that tests whether a Number is zero, and another function that zeros out an existing Number.

- ReadFrom, WriteTo.
  Binary transfer to files.

- PrintOn.
  Output. Probably just print digit sequence with no other formatting. One hundred digit numbers just about fit on a line. If `Numbers` were significantly larger, it would be necessary to work out an effective way of printing them out over several lines.

- Compare.
  Can return a -1, 0, or 1 result depending on whether a `Number` is less than, equal to, or bigger than the `Number` that it was asked to compare itself with.

- ChangeSign.
  Probably useful.

The functions that combine `Numbers` to get a `Number` as a result will take the second `Number` as a reference argument.

Prototypes

We can jump immediately to function prototypes and complete a partial declaration of the class:

```
#define kMAXDIGITS = 100;

class Number {
public:
    Number();
    Number(long);
    Number(char numstr[]);
    …
```

```
        void    ReadFrom(ifstream& in);
        void    WriteTo(ofstream& out) const;
        void    PrintOn(ostream& printer) const;

        Number  Add(const Number& other) const;
        Number  Subtract(const Number& other) const;
        Number  Multiply(const Number& other) const;
        Number  Divide(const Number& other) const;
        int     Zero_p() const;
        void    MakeZero();

        int     Equal(const Number& other) const;
        void    ChangeSign();
        int     Compare(const Number& other) const;
    private:
        // private functions still to be defined
        …
        …


        unsigned char  fDigits[kMAXDIGITS+1];
        unsigned char  fPosNeg; // 0 => positive, 1 => negative
        short          fLength;
    };
```

It is worthwhile coming up with this kind of partial class outline fairly early in the design process, because the outline can then provide a context in which other design aspects can be considered. (The extra digit in the `fDigits` data member simplifies one part of the division routines.)

Error handling

There aren't many general design issues remaining in this class. One that should be considered is error handling. There are several errors that we know may occur. We can ignore i/o errors, they can be checked by the calling environment that transfers `Numbers` to and from disk files. But we have to deal with things like overflow. Another problem is dealing with initialization of a `Number` from a string. What should we do if we are told to initialize a `Number` with the string "Hello World" (or, more plausibly, "1203o5l7" where there are letters in the digit string)?

Chapter 26 introduces the C++ exception mechanism. Exceptions provide a means whereby you can throw error reports back to the calling code. If you can't see a general way of dealing with an error when implementing code of your class, you arrange to pass responsibility back to the caller (giving the caller sufficient information so that they know what kind of error occurred). If the caller doesn't know what to do, the program can terminate. But often, the caller will be able to take action that clears the error condition.

For now, we use a simpler mechanism in which an error report is printed and the program terminates.

Assignment of Numbers

When defining a class, you always need to think about "assignment":

```
Number a:
Number b;
…
b = a;
```

Just as in the case of `Bitmaps`, assignment of `Numbers` presents no problems. A `Number` is simply a block of bytes; the compiler will generate "memory copy" instructions to implement the assignment. Later examples will illustrate cases where you may wish to prohibit assignment operations or you may need to change the compiler's default implementation.

Detailed design and implementation

Some of the member functions of class `Number` are simple, and for these it is possible to sketch pseudo-code outlines, or even jump straight to implementation. These member functions include the default constructor:

```
Number::Number()
{
    fPosNeg = 0;
    for(int i=0; i<= kMAXDIGITS; i++) fDigits[i] = 0;
    fLength = 0;
}
```

The next two constructors are slightly more elaborate:

```
Number::Number(long lval)
{
    fPosNeg = 0;
    if(lval<0) { fPosNeg = 1; lval = -lval; }
    for(int i=0; i<= kMAXDIGITS; i++) fDigits[i] = 0;
    fLength = 0;
    while(lval) {
            fDigits[fLength++] = lval % 10;
            lval = lval / 10;
            }
}

Number::Number(char numstr[])
{
    for(int i=0;i<= kMAXDIGITS; i++) fDigits[i] = 0;
    fPosNeg = 0; /* Positive */
    i = 0;
    fLength = strlen(numstr);

    if(numstr[i] =='+') { i++; fLength--; }
    else
    if(numstr[i] =='-') { fPosNeg = 1; i++; fLength--; }
```

Check for ± sign at start of string

```
    int     pos = fLength - 1;
```

Loop to process all characters in string

```
    while(numstr[i] != '\0') {
           if(!isdigit(numstr[i])) {
                  cout << "Bad data in number input\n";
                  exit(1);
                  }
           fDigits[pos] = numstr[i] - '0';
           i++;
           pos--;
           }
```

Fix up any case where given leading zeros

```
    while((fLength>0) && (fDigits[fLength-1] == 0)) fLength--;
}
```

You shouldn't have any difficulty in understanding the constructor that initializes a `Number` from a long integer value. The code just has a loop that fills in digits starting with the units, then the tens and so forth.

The constructor that takes a character string first checks for a + or - sign. Then it loops processes all remaining characters in the string. If any are non-digits, the program terminates with an error message. Digit characters are converted into numeric values in range 0…9 and placed in the correct locations of the array. (Note how the length of the string is used to determine the number of digits in the number, and hence the digit position in the array to be filled in with the first digit taken from the string.) The final loop just corrects for any cases where there were leading zeros in the string.

Copy constructor

Very often you need to say something like "Give me a Number just like this one". This is achieved using a copy constructor. A copy constructor takes an existing class instance as a reference argument and copies the data fields:

```
Number::Number(const Number& other)
{
    fPosNeg = other.fPosNeg;
    fLength = other.fLength;
    for(int i=0; i<= kMAXDIGITS; i++)
           fDigits[i] = other.fDigits[i];
}
```

Other simple functions include the comparison function and the input/output functions:

```
int Number::Compare(const Number& other) const
{
```

Can tell result from signs if these not the same

```
    if(fPosNeg != other.fPosNeg) {
           /* The numbers have opposite signs.
           If this is positive, then return "Greater" else
           return "Less". */
           return (fPosNeg == 0) ? 1 : -1;
```

```
            }

    if(fLength > other.fLength)
                return (fPosNeg == 0) ? 1 : -1;

    if(fLength < other.fLength)
                return (fPosNeg == 0) ? -1 : 1;

    for(int i = fLength-1;i>=0;i--)
            if(fDigits[i] > other.fDigits[i])
                    return (fPosNeg == 0) ? 1 : -1;
            else
            if(fDigits[i] < other.fDigits[i])
                    return (fPosNeg == 0) ? -1 : 1;

    return 0;
}

void Number::WriteTo(ofstream& out) const
{   // Remember, some iostreams+compilers may require (char*)
    out.write(&fDigits, sizeof(fDigits));
    out.write(&fPosNeg, sizeof(fPosNeg));
    out.write((char*)&fLength, sizeof(fLength)); // like this
}

void Number::PrintOn(ostream& printer) const
{
    if(fLength==0) { printer << "0"; return; }

    if(fPosNeg) printer << "-";
    int i = kMAXDIGITS;
    i = fLength - 1;

    while(i>=0) {
            printer << char('0' + fDigits[i]);
            i--;
            }

}
```

Otherwise by number of digits if they differ

But sometimes have to compare digit by digit from top

The `WriteTo()` function uses a separate call to `write()` for each data member. (the `ReadFrom()` function uses equivalent calls to `read()`). Sometimes this gets a bit clumsy; it may be worth inventing a struct that simply groups all the data members so as to facilitate disk transfers.

A few of the member functions are sufficiently simple that they can be included as inlines:

```
inline int Number::Zero_p() const
{
    return (fLength == 0);
```

```
}

inline void Number::ChangeSign()
{
    if(fPosNeg) fPosNeg = 0;
    else fPosNeg = 1;
}
```

As noted previously, such functions have to be defined in the header file with the class declaration.

Designing little programs!

The detailed design of the harder functions, like `Subtract()`, `Multiply()` and `Divide()`, really becomes an exercise in "top down functional decomposition" as illustrated by the many examples in Part III. You have the same situation. You have a "program" to write (e.g. the "Multiply program") and the data used by this program are simple (the data members of two class instances).

The "Add Program" and the "Subtract Program"

The addition and subtraction functions are to be called in code like:

```
Number a("123987115437810017316546l515");
Number b("71757656755466753443546541431765765137654");
Number c;
Number d;
c = a.Add(b);
d = a.Subtract(b);
```

Sign and magnitude representation complicates the process

The `Add()` (and `Subtract()`) functions are to return, via the stack, a value that represents the sum (difference) of the `Numbers a` and `b`. As noted in the introduction, sign and magnitude representations make things a bit more complex. You require lower level "do add" and "do subtract" functions that work using just the magnitude data. The actual `Add()` function will use the lower level "do add" if it is called to combine two values with the same sign, but if the signs are different the "do subtract" function must be called. Similarly, the `Subtract()` function will combine the magnitudes of the two numbers using the lower level "do add" and "do subtract" functions; again, the function used to combine the magnitude values depends upon the signs of the values.

There is a further complication. The actual subtraction mechanism (using "borrows" etc) only works if you subtract the smaller magnitude number from the larger magnitude number. The calls must be set up to get this correct, with the sign of the result being changed if necessary.

The following are sketches for the top level routines:

```
Add
// Returns sum of "this" and other
    initialize result to value of "this"

    if(other is zero)
           return result;

    if("this" and other have same sign) {
           do addition operation combining result with other;
```

```
                return result;
                }
        else
                return DoSubtract(other);
```

The code starts by initializing the result to the value of "this" (so for `a.Add(b)`, the result is initialized to the value of `a`). If the other argument is zero, it is possible to return the result. Otherwise we must chose between using a "do add" function to combine the partial result with the other argument, or calling `DoSubtract()` which will sort out subtractions.

The code for the top-level `Subtract()` function is similar, the conditions for using the lower-level addition and subtraction functions are switched:

```
Subtract
// Returns difference of "this" and other
    initialize result to value of "this"

    if(other is zero)
            return result;

    if("this" and other have same sign)
            return DoSubtract(other);
    else {
            do addition operation combining result with other
            return result;
            }
}
```

New member functions identified

These sketches identify at least one new "private member function" that should be added to the class. It will be useful to have:

```
int SameSign(const Number& other) const;
```

This function checks the `fPosNeg` data members of an object and the second `Number` (`other`) passed as an argument (i.e. `return (this->fPosNeg == other.fPosNeg);`). This can again be an inline function.

We also need to initialize a variable to the same value as "this" (the object executing the function). This can actually be done by an assignment; but the syntax of that kind of assignment involves a deeper knowledge of pointers and so is deferred until after the next chapter. Instead we can use a small `CopyTo()` function:

```
void Number::CopyTo(Number& dest) const
{
    // This function is not needed.
    // Can simply have dest = *this; at point of call.
    // Function CopyTo introduced to delay discussion of *this
    // until pointers covered more fully.
```

```
    dest.fLength = this->fLength;
    dest.fPosNeg = this->fPosNeg;
    for(int i=0; i<= kMAXDIGITS; i++)
           dest.fDigits[i] = this->fDigits[i];
}
```

Both `Add()` and `Subtract()` invoked the auxiliary function `DoSubtract()`. This function has to sort out the call to the actual subtraction routine so that the smaller magnitude number is subtracted from the larger (and fix up the sign of the result):

```
DoSubtract
// Set up call to subtract smaller from larger magnitude

    if(this is larger in magnitude than other)
           Initialize result to "this"
           call actual subtraction routine to subtract other
                  from result
    else
           Initialize a temporary with value of this
           set result to value of other;

           call actual subtraction routine to subtract temp.
                  from result
           if(subtracting) change sign of result
    return result;
```

Yet another auxiliary function shows up. We have to compare the magnitude of numbers. The existing `Compare()` routine considers sign and magnitude but here we just want to know which has the larger digit string. The extra function, `LargerThan()`, becomes an additional private member function. Its code will be somewhat similar to that of the `Compare()` function involving checks on the lengths of the numbers, and they have the same number of digits then a loop checking the digits starting with the most significant digit.

An outline for the low level "do add" routine was given earlier when the "carry" mechanism was explained (see Figure 19.4). The lowest level subtraction routine has a rather similar structure with a loop that uses "borrows" between units and tens, tens and hundreds, etc.

Class handling additions and subtractions

With the auxiliary functions needed for addition and subtraction the class declaration becomes:

```
class Number {
public:
    Number();
    …
    // Public Interface unchanged
    …
    int     Compare(const Number& other) const;
private:
```

```
    int     SameSign(const Number& other) const;
    void    DoAdd(const Number& other);
    Number  DoSubtract(const Number& other, int subop) const;
    void    SubtractSub(const Number& other);

    int     LargerThan(const Number& other) const;
    void    CopyTo(Number& dest) const;

    unsigned char  fDigits[kMAXDIGITS+1];
    unsigned char  fPosNeg;
    short          fLength;
};
```

The implementation for representative functions is as follows:

```
Number Number::Subtract(const Number& other) const
{
    Number result;
    CopyTo(result);

    if(other.Zero_p()) return result;

    if(this->SameSign(other))
            return DoSubtract(other, 1);
    else {
            result.DoAdd(other);
            return result;
            }
}
```

The `Add()` routine is very similar. It should be possible to code it, and the auxiliary `DoSubtract()` from the outlines given.

Both the `DoAdd()` and the `SubtractSub()` routines work by modifying a `Number`, combining its digits with the digits of another number. The implementation of `DoAdd()`, "add with carry", is as follows:

Loops limited by magnitude of larger number

```
void Number::DoAdd(const Number& other)
{
    int     lim;
    int     Carry = 0;
    lim = (fLength >= other.fLength) ? fLength : other.fLength;
    for(int i=0;i<lim; i++) {
            int temp;
            temp = fDigits[i] + other.fDigits[i] + Carry;
            if(temp>=10) { Carry = 1; temp -= 10; }
            else Carry = 0;
            fDigits[i] = temp;
            }
    fLength = lim;
```

Deal with any final carry, check for Overflow

```
        if(Carry) {
                if(fLength == kMAXDIGITS) Overflow();
                fDigits[fLength] = 1;
                fLength++;
                }
}
```

Handling overflow

A routine has to exist to deal with overflow (indicated by carry out of the end of the number). This would not need to be a member function. The implementation used a static filescope function defined in the same file as the Numbers code; the function printed a warning message and terminated.

The function `SubtractSub()` implements the "subtract with borrow" algorithm:

```
void Number::SubtractSub(const Number& other)
{
    int     Borrow = 0;
    int     newlen = 0;
    for(int i=0;i<fLength; i++) {
            int    temp;
            temp = fDigits[i] - other.fDigits[i] - Borrow;
            if(temp < 0) { temp += 10; Borrow = 1; }
            else Borrow = 0;
            fDigits[i] = temp;
            if(temp) newlen = i+1;
            }
    fLength = newlen;
    if(fLength==0) fPosNeg = 0;
}
```

(If the result is ±0, it is made +0. It is confusing to have positive and negative version of zero.)

In a phased implementation of the class, this version with just addition and subtraction would be tested.

The "Multiply Program"

The usage of the multiplication routine is similar to that of the addition and subtraction routines:

```
Number a("123987115437810017316546l515");
Number b("71757656755466753443546541431765765137654");
Number c;
c = a.Multiply(b);
```

Function `Multiply()` again returns a `Number` on the stack that represents the product of the `Number` executing the routine ('`a`') and a second `Number` ('`b`').

Sign and magnitude representation doesn't result in any problems. Getting the sign of the product correct is easy; it the two values combined have the same sign the product is positive otherwise it is negative. The basics of the algorithm was outlined in the discussion related to Figure 19.5. The code implementing the algorithm is:

```
Number Number::Multiply(const Number& other) const
{
    Number Result;  // Number defaults to zero

    if(other.Zero_p()) { return Result; }

    for(int i=0;i<fLength; i++)
           if(fDigits[i]) {
                  Number temp(other);

                  temp.Times10(i);
                  temp.Product(fDigits[i]);
                  Result = Result.Add(temp);
                  }

    if(SameSign(other))
           Result.fPosNeg = 0;
    else Result.fPosNeg = 1;

    return Result;
}
```

Note use of "Copy constructor"

The routine forms a product like:

```
19704 * 6381
```

by calculating

4	*	6381	25524	$(4*6381*10^0)$
0	*	63810	0	$(0*6381*10^1)$
7	*	638100	4466700	$(7*6381*10^2)$
9	*	6381000	57429000	$(9*6381*10^3)$
1	*	63810000	63810000	$(1*6381*10^4)$
Total =			125731224	

Further private member functions

Two more auxiliary functions are needed. One, `Times10()`, implements the shifts to multiply by a specified power of ten. The other, `Product()`, multiplies a `Number` by a value in the range 0…9. Both change the `Number` for which they are invoked. Partial results from the individual steps accumulate in `Result`.

```
void Number::Times10(int power)
{
    if((fLength+power)>kMAXDIGITS) Overflow();
    for(int i = fLength-1;i>=0;i--) fDigits[i+power] = fDigits[i];
    for(i = power-1;i>=0;i--) fDigits[i] = 0;
    fLength += power;
}
```

Function `Times10()` needs to check that it isn't going to cause overflow by shifting a value too far over. If the operation is valid, the digits are just moved leftwards and the number filled in at the right with zeros.

The `Product()` routine is a bit like "add with carry":

```
void Number::Product(int k)
{
     int lim;
     int     Carry = 0;
     if(k==0) {
            MakeZero();
            return;
            }

     lim = fLength;
     for(int i=0;i<lim; i++) {
            int temp;
            temp = fDigits[i]*k + Carry;
            Carry = temp / 10;
            temp = temp % 10;
            fDigits[i] = temp;
            }
     if(Carry) {
            if(fLength == kMAXDIGITS) Overflow();
            fDigits[fLength] = Carry;
            fLength++;
            }
}
```

The "Divide" Program

You must remember this:

```
             0000029------------------
      49853  |1467889023451770986543175
               99706
               470829
               448677
```

(unless your primary school teachers let you use a calculator). You found long division hard then; it still is.

As with multiplication, getting the sign right is easy. If the dividend and divisor have the same sign the result is positive, otherwise it is negative. The `Divide()` routine itself can deal with the sign, and also with some special cases. If the divisor is zero, you are going to get overflow. If the divisor is larger than the dividend the result is zero. A divisor that is a one digit number is another special case handled by an auxiliary routine. The `Divide()` function is:

```
Number Number::Divide(const Number& other) const
{
```

```
    if(other.Zero_p()) Overflow();
    Number Result; // Initialized to zero

    if(!LargerThan(other))
           return Result; // return zero

    CopyTo(Result);

    if(other.fLength == 1)
           Result.Quotient(other.fDigits[0]);
    else
           Result.LongDiv(other);

    if(other.fPosNeg == fPosNeg) Result.fPosNeg = 0;
    else Result.fPosNeg = 1;

    return Result;
}
```

The auxiliary (private member) function `Quotient()` deals with simple divisions (e.g. 77982451 ÷ 3 = 2599…). You start with the most significant digit, do the division to get the digit, transfer any remainder to the next less significant digit:

```
void Number::Quotient(int k)
{
    int    Carry = 0;
    int newlen = 0;
    for(int i= fLength-1;i>=0;i--) {
           int    temp;
           temp = 10*Carry + fDigits[i];
           fDigits[i] = temp / k;
           if((newlen==0)  && (fDigits[i] !=0)) newlen = i+1;
           Carry = temp % k;
           }
    fLength = newlen;
}
```

The main routine is `LongDiv()`. Brinch Hansen's algorithm is moderately complex and involves several auxiliary functions that once again become private member functions of the class. The code is given here without the explanation and analysis (available in Brinch Hansen's paper in Software Practice and Experience):

```
void Number::LongDiv(const Number& other)
{
    int    f;
    Number d(other);
    Number r;
    CopyTo(r);
```

Using lengths of numbers to get idea of size of quotient

```
    int     m = other.fLength;
    int     n = fLength;

    f = 10 / (other.fDigits[m-1] + 1);
    r.Product(f);
    d.Product(f);

    int newlen = 0;
```

loop filling in successive digits of quotient

```
    for(int k = n - m; k>=0; k--) {
            int     qt;
            qt = r.Trial(d,k,m);
            if(qt==0) {
                    fDigits[k] = 0;
                    continue;
                    }

            Number dq(d);
            dq.Product(qt);
            if(r.Smaller(dq,k,m)) { qt--; dq = dq.Subtract(d); }
            if((newlen==0) && (qt !=0)) newlen = k+1;
            fDigits[k] = qt;
            r.Difference(dq,k,m);
            }
    fLength = newlen;
}
```

Taking a guess at next digit of quotient

```
int Number::Trial(const Number& other, int k, int m)
{
    int     km = k + m;
    int     r3;
    int d2;
    km = k + m;
    r3 = (fDigits[km]*10 + fDigits[km-1])*10 + fDigits[km-2];
    d2 = other.fDigits[m-1]*10 + other.fDigits[m-2];
    int temp = r3 / d2;
    return (temp<9) ? temp : 9;
}

int Number::Smaller(const Number& other, int k ,int m)
{
    int     i;
    i = m;
    for(;i>0;i--)
            if(fDigits[i+k] != other.fDigits[i]) break;
    return fDigits[i+k] < other.fDigits[i];
}

void Number::Difference(const Number& other, int k, int m)
{
    int     borrow = 0;
    for(int i = 0; i <= m; i++) {
            int diff = fDigits[i+k] -
```

```
                    other.fDigits[i] - borrow + 10;
            fDigits[i+k] =  diff % 10;
            borrow = 1 - (diff / 10);
            }
    if(borrow) Overflow();
}
```

With all these auxiliary private member functions, the final class declaration becomes:

Final class declaration

```
class Number {
public:
    Number();
    Number(long);
    Number(const Number& other);
    Number(char numstr[]);
    Number(istream&);

    void   ReadFrom(ifstream& in);
    void   WriteTo(ofstream& out) const;
    void   PrintOn(ostream& printer) const;

    Number Add(const Number& other) const;
    Number Subtract(const Number& other) const;
    Number Multiply(const Number& other) const;
    Number Divide(const Number& other) const;
    int    Zero_p() const;
    void   MakeZero();
    int    Equal(const Number& other) const;
    void   ChangeSign();
    int    Compare(const Number& other) const;
private:
    int    SameSign(const Number& other) const;
    void   DoAdd(const Number& other);
    Number DoSubtract(const Number& other, int subop) const;
    void   SubtractSub(const Number& other);

    void   Product(int k);
    void   Quotient(int k);

    void   Times10(int power);
    int    LargerThan(const Number& other) const;

    void   LongDiv(const Number& other);
    int    Trial(const Number& other, int, int);
    int    Smaller(const Number& other, int, int);
    void   Difference(const Number& other, int, int);

    void   CopyTo(Number& dest) const;

    unsigned char fDigits[kMAXDIGITS+1];
```

```
    unsigned char  fPosNeg;
    short          fLength;
};
```

Testing

The class has to be tested. Simple programs like:

```
int main()
{
    Number a("123456789");
    Number b("-987654");
    Number c;
    c = a.Add(b); cout << "Sum "; c.PrintOn(cout); cout <<
endl;
    c = a.Subtract(b); cout << "Difference ";
                  c.PrintOn(cout); cout  << endl;
    c = a.Multiply(b); cout << "Product ";
                  c.PrintOn(cout); cout  << endl;
    c = a.Divide(b); cout << "Quotient ";
           c.PrintOn(cout); cout  << endl;
    return 0;
}
```

check the basics. As long the values are less than ten digits you can verify them on a calculator or with a spreadsheet program (or even by hand).

Automatic test programs

You would then proceed to programs such as one calculating the expressions (a-b)*(c+d) and (ac-bc+ad-bd) for different randomly chosen values for the `Numbers` a, b, c, and d (the two expressions should have the same value). A similar test routine for the multiplication and divide operations would check that (a*b*c*d)/(a*c) did equal (b*d); again the expressions would be evaluated for hundreds of randomly selected values. (A small random value can be obtained for a `Number` as follows: `long lv; lv = rand() % 25000; lv -= 12500; Number a(lv);` …. Alternatively, an additional "Randomize()" function can be added to the class that will fill in `Numbers` with randomly chosen 40 digit sequences and a random sign. You limit the size of these random values so that you could multiply them without getting overflows). Such test programs need only generate outputs if they encounter cases where the pair of supposedly equal calculated values are not in fact equal.

Optimising the code

If the class were intended for serious use in an application, you would then use a profiling tool like that described in Chapter 14. This would identify where a test program spent most time. The test program has to use `Numbers` in the same manner as the intended application. If the application uses a lot of division so should the test program.

Such an analysis was done on the code and the `Product()` routine showed up as using a significant percentage of the run time. If you look back at the code of this routine you will see that it treats multiplication by zero as a special case (just zero out the result and return); other cases involve a loop that multiplies each digit in the `Number`, getting the carry etc. Multiplying by 1 could also be made a special case. If the multiplier is 1, the `Number` is unchanged. This special case would be easy to incorporate by an extra test with a return just before the main loop in the `Product()` routine. Retesting showed that this small change reduced the time per call to `Product()` from 0.054 to 0.049µs (i.e. worthwhile). Similar analysis of other member functions could produce a number of other speed ups; however, generally, such improvements would be minor, in the order of 5% to 10% at most.