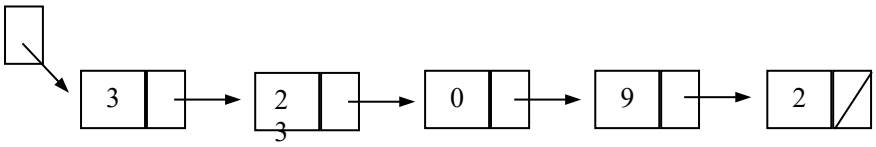
计算机中整数都有一定的范围。对一些需要用到任意大的整数的应用，需要自己想办法解决此问题。 一种解决的方案是用单链表。例如整数 29023 可表示为



注意，数字是用逆序存储的。试设计一个类处理任意大整数。必须实现输入输出操作、加法操作和赋值操作。

【解】如题所述，大整数的存储采用一个单链表，因此这个类只有一个数据成员，即指向结点的一个指针，该指针指向存储个位数的结点。

按照题意，这个类至少实现四个功能：输入重载、输出重载、加法重载和赋值运算符重载函数。前三个函数重载成友元函数，赋值运算符被重载成成员函数。除此之外，因为存储正整数采用的是一个单链表，所以这个类还必须有构造和析构函数。根据上述分析得到的类定义如代码清单 2-25 所示。

代码清单 2-25 大正整数处理类

```
class BigData {
    friend ostream &operator<<( ostream & os, const BigData & x );
    friend istream &operator>>( istream & is, BigData & x );
    friend BigData operator+(BigData a, BigData b);

private:
    struct node {                          // 单链表的结点类型
        int data;
        node *next;

        node(const short &x, node *n = NULL){ data = x; next = n;}
        node():next(NULL){}
    };

    node *num;                             //表头指针
  void clear();                            //   清空函数

public:
    BigData(node *p = NULL)        //构造函数，构造一个值为 0 的正整数
   { if (p == NULL) num = new node(0); else num = p;  }
    BigData(const BigData &);                    //拷贝构造函数
  ~BigData() { clear(); }                        // 析构函数
    BigData &operator=(const BigData &);          // 赋值运算符重载
};
```

由于任意大的整型数使用单链表存放的，很显然不能用缺省的拷贝构造函数，于是在BigData 类中定义了一个拷贝构造函数。其实现如代码清单 2-26 所示。

代码清单 2-26 BigData 类的拷贝构造函数

```
1.  BigData::BigData(const BigData &x) {
2.      num = new node(x.num -> data);  //生成第一个结点
3.      node *p = num, *q = x.num;
4.      while ( q->next != NULL ) {  // 生成其余结点
5.          q = q->next;
6.          p->next = new node(q->data);
7.          p = p->next;
8.      }
9.  }
```

由于保存整型数的单链表的空间都是动态空间，于是需要一个析构函数释放所有的空间。释放链表空间的工作有几个地方都需要用，我们定义了一个私有的成员函数 clear 完成这项工作。析构函数调用 clear 函数完成析构任务。clear 函数的定义如代码清单 2-27 所示。

代码清单 2-27 clear 函数的定义

```
1.  void BigData::clear() {
2.      node *p = num, *q;
3.      while ( p != NULL ) {
4.          q = p;
5.          p = p->next;
6.          delete q;
7.      }
8.      num = NULL;
9.  }
```

下面我们来讨论四个主要函数的实现，首先讨论加法操作。由于数字是逆序存储，因此执行加法操作时，数字是对齐的。我们只需要从两个链表的第一个结点（即个位数）开始，将两个结点的 data 值相加，取它的个位数作为结果的个位数，十位数作为进位。接着依次将对应位置的两个结点值相加，再加上进位，取它的个位数作为结果的这一位数的值，十位数作为下一轮的进位。当某一个数结束时，对另外一个数的剩余位进行处理。处理方法是依次处理每一位，将这一位的值加上进位，个位数作为结果的这一位的值，十位数作为下一次的进位。当两个数都处理完后，检查进位。如进位非 0，表示这两个数相加后位数增加，将进位作为结果的最高位添加到表示结果的链表的最后。这个过程如代码清单 2-28 所示。

代码清单 2-28 加法函数的实现

```
1.  BigData operator + ( BigData a, BigData b )
2.  {
3.      BigData tmp;                        //存放加的结果
4.      BigData::node *p, *q , *end;        //p,q 分别指向被加数当前处理的位
5.      int carry;                          // 存放每一位相加时的进位
```

```
6.
7.      // 处理个位数
8.    tmp.num = end = new BigData::node(a.num->data + b.num->data);
9.      carry = tmp.num->data / 10;
10.    tmp.num->data %= 10;
11.
12.  p = a.num->next;
13.  q = b.num->next
14.    end = tmp.num;
15.    while ( p != NULL && q != NULL ) {       //从十位数开始将对应为相加
16.          end->next = new BigData::node(p->data + q->data + carry);
17.          end = end->next;
18.       carry = end->data / 10;
19.       end->data %= 10;
20.       p = p->next;
21.          q = q->next;
22.    }
23.
24.    if (p == NULL) p = q;          // p 指向尚未处理完的数的未处理的最低位
25.    while (p != NULL) {
26.          end->next = new BigData::node(p->data + carry);
27.          end = end->next;
28.       carry = end->data / 10;
29.       end->data %= 10;
30.       p = p->next;
31.    }
32.    if (carry != 0)  end->next = new BigData::node(carry);   //最高位有进位
33.
34.    return tmp;
35.}
```

赋值运算符重载的实现类似于拷贝构造函数。在检查了是否自我赋值后，清空当前对象，然后根据 x 的值设置当前对象的值。其定义见代码清单 2-29。

代码清单 2-29 赋值运算符重载函数的实现：

```
1.  BigData &BigData::operator=(const BigData &x)
2.  {
3.      if (&x == this) return *this;              // 检查是否自我赋值
4.
5.      clear();                                   //将当前对象清空
6.      num = new node(x.num -> data);
7.      node *p = num, *q = x.num;
8.      while ( q->next != NULL ) {
9.          q = q->next;
```

```
            p->next = new node(q->data);
            p = p->next;
        }
        return *this;
}
```

无限大的整型数的输出有点麻烦。由于存放时是按逆序存放，个位数存放在最前面，最高位存放在最后面。但输出时，最高位应该最先输出。于是我们用了一个字符串变量 s 作为过渡。开始时，字符串 s 为空。然后遍历单链表。每遍历到一个数字，将它添加到 s 的前面。最终，s 包含了这个数字的字符串的表示。输出这个字符串就是输出了这个数字。具体过程见代码清单 2-30。

代码清单 2-30 输出运算符重载函数的实现

```
ostream & operator<<( ostream & os, const BigData & x )
{
    std::string s;
    BigData:: node *p = x.num;
    while (p != NULL) { // 遍历单链表
        s = char(p->data +'0') + s;
        p = p->next;
    }
    for (int i = 0; i< s.size(); ++i) os << s[i];  //输出 s
    return os;
}
```

输入和输出有同样的问题。输入时是从高位输入到低位，但存储时低位在前高位在后。这个问题比较容易解决。在读入一位时，将存储这一位的结点插入到单链表的表头。输入运算符重载函数定义如代码清单 2-31 所示。

代码清单 2-31 输入运算符重载函数的实现

```
istream & operator>>( istream & is, BigData & x ) {
    char ch;  //存储当前输入位的值

    x.clear(); //清空当前链表

    while ((ch = is.get()) !='\n') { //输入每一位，直到遇到回车
        if (ch < '0' || ch > '9') throw WrongInput(); //错误输入，抛出异常
        x.num = new BigData::node(ch - '0', x.num);
    }
    return is;
}
```

在输入运算符重载函数中，当输入不是 0 到 9 之间的字符时，意味着输入有错，则抛出一个 WrongInput 的异常。否则将输入的字符转换成数字插入到单链表的表头。异常类定义见代码清单 2-32。抛出异常时，会显示一个出错信息。

代码清单 2-32 大正整数类中异常类的定义

```
12. class WrongInput {
13. public:
14.     WrongInput()
15.     {     cout << "输入有误，只能输入 0-9。" << endl;     }
16. };
```