由于使用普通整型数组的烦琐，可以使用结构体类型定义高精度数为BigNum，参考程序如下所示。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | struct BigNum  {  int data[maxn];//data[0]为整数位数，如567，共3位，则存为376500...000  int frac[maxn];//frac[0]为小数位数，如.789,共3位，则存为378900...000  int &operator[](int n) //重载运算符[]  {  return data[n]; //返回data[n]的地址  }  int &operator()(int n) //重载运算符()  {  return frac[n]; //返回frac()的地址  }  BigNum() //构造函数  {  data[0]=1; //整数位数为1  data[1]=0; //初始化为0  frac[0]=0; //小数部分为0  }  }; |

可以看出，BigNum内的数据分为整数部分和小数部分，例如有高精度数1234.56789，则保存格式如图8.9所示：

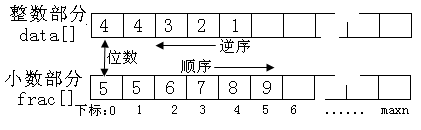


图8.9

例如有语句定义了BigNum x；则调用x中的data[i]或frac[j]的元素进行操作的一般的方法是x.data[i]或x.frac[j]，这种写法不是很方便，C++语言引入了重载运算符的方法，即对已有的运算符重新进行定义，赋予其另一种功能，以适应不同的数据类型。

例如上面的代码第5~8行对符号“[ ]”进行了重新定义，即：

|  |  |
| --- | --- |
| 1  2  3  4 | int &operator[](int n) //重载运算符[]  {  return data[n]; //返回data[n]的地址  } |

这样x[i]即等价于x.data[i]。

代码第9~12行对符号“( )”进行了重新定义，即：

|  |  |
| --- | --- |
| 1  2  3  4 | int &operator()(int n) //重载运算符()  {  return frac[n]; //返回frac()的地址  } |

这样x(j)即等价于x.frac[j]。

此外，结构体中还使用了构造函数，构造函数必须与其结构体同名，构造函数无须调用而自动执行，主要用来为对象成员变量赋初始值。即代码的13~18行：

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | BigNum() //构造函数  {  data[0]=1; //初始化整数位数为1  data[1]=0; //初始化为0  frac[0]=0; //初始化小数部分为0  } |

以下为一个完全的重载运算符和构造函数的简单示例。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49 | //重载运算符和构造函数的简单示例  #include <bits/stdc++.h>  using namespace std;  #define maxn 5  struct BigNum  {  int data[maxn]; //整数部分  int frac[maxn]; //小数部分  int &operator[](int n) //重载运算符[]  {  return data[n]; //返回data[n]的地址  }  int &operator()(int n) //重载运算符()  {  return frac[n]; //返回frac()的地址  }  BigNum() //构造函数  {  data[0]=1; //初始化为0  data[1]=0; //则整数位数为1  frac[0]=0; //小数部分为0  }  };  int main()  {  BigNum x;  for(int i=1; i<maxn; ++i) //使用重载运算符[ ]和（ ）后的输入方式  {  x[i]=i;  x(i)=i\*2;  }  //一般输出方法，将输出11234.02468  for(int i=0; i<maxn; ++i)  cout<<x.data[i]; //data[0]已被构造函数初始化为1  cout<<'.';  for(int i=0; i<maxn; ++i)  cout<<x.frac[i];  cout<<endl;  //重载运算符的输出方法，结果等价于一般输出方法  for(int i=0; i<maxn; ++i)  cout<<x[i];  cout<<'.';  for(int i=0; i<maxn; ++i)  cout<<x(i);  return 0;  } |

核心函数div()的流程如图8.10所示。

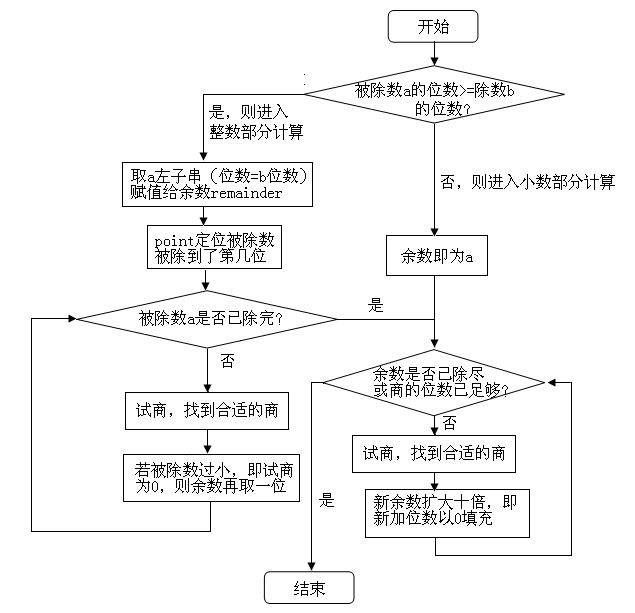


图8.10

完整的代码如下所示。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99  100  101  102  103  104  105  106  107  108  109  110  111  112  113  114  115  116  117  118  119  120  121  122  123  124  125  126  127  128  129  130  131  132  133  134  135  136  137  138  139  140  141  142  143  144  145  146  147  148  149  150  151  152  153  154  155  156  157  158  159  160  161  162  163  164  165  166  167  168  169  170  171  172  173  174  175  176 | //高精度数除高精度数改进算法  #include <bits/stdc++.h>  using namespace std;  #define maxn 5100  #define limit 5000 //限制输出5000个数字  struct BigNum  {  int data[maxn];//整数部分，data[0]为位数，如567，共3位，则存为376500...00  int frac[maxn];//小数部分，frac[0]为位数，如.789,共3位，则存为378900...00  int &operator[](int n) //重载运算符[]  {  return data[n]; //返回data[n]  }  int &operator()(int n) //重载运算符()  {  return frac[n]; //返回frac()  }  BigNum() //构造函数,用于初始化成员变量  {  data[0]=1; //初始化为0  data[1]=0; //则整数位数为1  frac[0]=0; //小数部分为0  }  };  void Output(BigNum temp) //输出高精度数  {  for(int i=temp[0]; i>0; i--) //输出整数部分  cout<<temp[i];  if(temp(0)!=0) //如果有小数部分，则输出小数部分  {  cout<<".";  for(int i=1; i<=limit-temp[0] && i<=temp(0); i++) //控制小数输出位数  cout<<temp(i);  }  }  BigNum Init(string str) //将字符串转化为高精度数  {  BigNum temp;  temp[0]=str.size();  for(int i=1; i<=temp[0]; ++i)  temp[i]=str[temp[0]-i]-'0';  return temp;  }  BigNum Sub(BigNum &a,BigNum &b) //高精度数a－高精度数b,a永远大于等于b  {  BigNum t;  for(int i=1; i<=a[0]; ++i)  {  t[i]+=a[i]-b[i];  t[i+1]=0;  if(t[i]<0)  {  t[i]+=10;  t[i+1]--;  }  if(t[i]!=0)  t[0]=i;  }  return t;  }  BigNum Mul(BigNum &a,int &k) //计算高精度数a乘低精度数k的值  {  BigNum temp;  for(int i=1; i<=a[0]; ++i)  {  temp[i]+=a[i]\*k;  temp[i+1]=temp[i]/10;  temp[i]%=10;  }  temp[0]=a[0]+(temp[a[0]+1]!=0); //更新位数  return temp;  }  BigNum result[10]; //\_b[i]存除数b[]乘k的值  bool saved[10]; //判断是否已存b\*i的值  BigNum Dichotomy(BigNum &a,BigNum &b,int &k)//二分试除,返回余数及k值(0<=k<10)  {  int L=1,R=9; //左边最小值为1，右边最大值为9  while(L<=R)  {  int m=(L+R)>>1; //取中间值m  if(saved[m]==0) //若b\*m的值未保存，则保存  {  saved[m]=1;  result[m]=Mul(b,m); //计算b\*m的值  }  int t=0;  if(result[m][0]>a[0]) //如b\*m的位数>a，则t=1  t=1;  else if(result[m][0]<a[0]) //如b\*m的位数<a,则t=－1  t=-1;  else //如位数相等，则t=0,则逐数进行比较  {  for(int i=result[m][0]; t==0 && i>0; i--)  if(result[m][i]>a[i])  t=1;  else if(result[m][i]<a[i])  t=-1;  }  t>0?R=m-1:L=m+1; //范围缩小一半  }  k=L-1; //确定了K的值  return Sub(a,result[k]); //返回余数为a－b\*k  }  BigNum Div(BigNum &a,BigNum &b)  {  BigNum c,t; //c为a/b的结果,t为临时变量,用于存储余数  int point,j;  //整数部分的运算  if(a[0]>=b[0]) //当a的位数大于b的位数  {  t[0]=b[0]; //保存位数  for(point=a[0]-b[0]+1; point<=a[0]; ++point) //取a前b[0]位给t  t[point-a[0]+b[0]]=a[point]; //倒序排  point=a[0]-b[0]+1; //预测商的位数  c[0]=point;  do  {  t=Dichotomy(t,b,c[point]); //t=余数，除出来的商写在c[point]里  if(point==c[0] && c[point]==0)//避免出现a取前b[0]位的数小于b的情况  c[0]--;  point--;  if(point<1)  break;  for(j=++t[0]; j>1; j--)  t[j]=t[j-1];  t[1]=a[point];  if(t[t[0]]==0)  t[0]--;  }  while(1);  if(c[0]==0)  c[0]=1;//防a取前b[0]位的数小于b时c[0]减少到0,而忽略小数点前0输出  }  else  t=a;  //小数部分的运算  for(c(0)=1; c(0)<maxn && !(t[0]==1 && t[1]==0)/\*余数为0则结束除法\*/; ++c(0))  {  for(j=++t[0]; j>1; j--) //此处默认整数除以整数,因此除完后直接用0来填充  t[j]=t[j-1];  t[1]=0;  t=Dichotomy(t,b,c(c(0)));  }  c(0)--;  return c;  }  int main()  {  string a,b;  cin>>a>>b;  BigNum m,n;  m=Init(a);  n=Init(b);  if(n[0]==1 && n[1]==0)  cout<<"Divisor is 0";  else  Output(Div(m,n));  cout<<endl;  return 0;  } |

为了确保程序的正确性，可以写个随机生成测试数据的程序，再写个批处理文件自动进行“对拍测试”。

随机生成测试数据程序参考代码如下。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | //随机生成测试数据，文件名为Rand.cpp  #include <bits/stdc++.h>  using namespace std;  int main()  {  srand((unsigned)time(NULL));  freopen("div.in","w",stdout);  int a\_len=rand()%5000;  int b\_len=rand()%5000;  printf("%d",rand()%9+1);  for(int i=1; i<a\_len; i++)  printf("%d",rand()%10);  printf(" %d",rand()%9+1);  for(int i=1; i<b\_len; i++)  printf("%d",rand()%10);  return 0;  } |

“对拍”的批处理文件（在记事本软件中写，不要在Dev-CPP中写）如下所示。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | ::使用说明：  ::本文件名保存为fcc.bat，注意不要保存为fcc.bat.txt之类的文件名了  ::将第一个高精度除法的代码改名为div1.cpp,输出文件改为div1.out,编译生成div1.exe  ::将第二个高精度除法的代码改名为div2.cpp,输出文件改为div2.out,编译生成div2.exe  ::将随机生成测试数据的代码以rand.cpp文件名保存，并编译生成可执行文件rand.exe  ::将fcc.bat,div1.exe,div2.exe,rand.exe放在同一文件夹下，执行本文件即可  @echo off  :loop  rand ::调用rand.exe,则生成测试数据文件div.in  div1 ::调用div1.exe,则生成div1.out  div2 ::调用div2.exe,则生成div2.out  fc div1.out div2.out && goto loop ::比较文件差异，有差异则结束，否则继续循环 |

双击运行该批处理文件，比较界面如图8.11所示：



图8.11

|  |  |
| --- | --- |
|  | 需要注意的是，不要过分迷信“对拍”程序，首先“对拍”程序可能需要运行较长的一段时间例如20分钟以上，才可能有较大的概率发现代码隐藏的bug，除此之外，还需要手工制作一批特殊的小数据等进行测试才可基本确保程序的正确性。  当然，这一切都是建立在“对拍”程序是正确的前提下。 |