dlink.h

```
001:  #ifndef DLINK
002:  #define DLINK
003:  class Dlink
004:  {
005:     protected:
006:        Dlink* next;                    // адрес следующей записи
007:        Dlink* prev;                    // адрес предыдущей записи
008:     public:
009:        Dlink();                        // функция инициализации ссылок начала и конца
010:        Dlink* append(Dlink*);          // функция добавления новой записи
011:        Dlink* incr();                  // функция получения адреса следующей записи
012:        Dlink* decr();                  // функция получения адреса предыдущей записи
013:        void excluse();                 // функция исключения текущей записи
014:        Dlink* after(Dlink*);           // функция вставки записи после текущей
015:        Dlink* before(Dlink*);          // функция вставки записи перед текущей
016:        Dlink* tohead(void);            // функция получения адреса начала списка
017:        Dlink* tohead(int);             // функция сдвига в направлении начала списка
018:        Dlink* totail(void);            // функция получения адреса конца списка
019:        Dlink* totail(int);             // функция сдвига в направлении конца списка
020:  };
021:  #endif
```

dlink.cc

```
001:   #include <stdio.h>
002:   #include "dlink.h"
003:
004:   Dlink::Dlink()
005:   {
006:      next=prev=NULL;
007:   }
008:
009:   Dlink* Dlink::incr()
010:   {
011:      return(next);
012:   }
013:
014:   Dlink* Dlink::decr()
015:   {
016:      return(prev);
017:   }
018:
019:   Dlink* Dlink::append(Dlink* p)
020:   {
021:      p->next = this;
022:      prev = p;
023:      return(p);
024:   }
025:
026:   void Dlink::excluse()
027:   {
028:      if(next != NULL)
029:         next->prev = prev;
030:      if(prev != NULL)
031:         prev->next = next;
```

```
032:       return;
033:    }
034:
035:    Dlink* Dlink::after(Dlink* p)
036:    {
037:       p->next = next;
038:       p->prev = this;
039:       if(next != NULL)
040:          next->prev = p;
041:       next = p;
042:       return(p->next);
043:    }
044:
045:    Dlink* Dlink::before(Dlink* p)
046:    {
047:       p->next = this;
048:       p->prev = prev;
049:       if(prev != NULL)
050:          prev->next = p;
051:       prev = p;
052:       return(p->prev);
053:    }
054:
055:    Dlink* Dlink::tohead()
056:    {
057:       Dlink* p = this;
058:       Dlink* q = NULL;
059:       while(p != NULL)
060:       {
061:          q = p;
062:          p = p->prev;
063:       }
064:       return(q);
065:    }
066:
067:    Dlink* Dlink::tohead(int n)
068:    {
069:       Dlink* p = this;
070:       Dlink* q = this;
071:       int i = 0;
072:       while(p != NULL)
073:       {
074:          q = p;
075:          if(i == n)
076:             break;
077:          p = p->prev;
078:          i++;
079:       }
080:       return(q);
081:    }
082:
083:   Dlink* Dlink::totail()
085:    {
086:       Dlink* p = this;
087:       Dlink* q = NULL;
088:       while(p != NULL)
```

```
089:      {
090:         q = p;
091:         p = p->next;
092:      }
093:      return(q);
094:   }
095:
096:   Dlink* Dlink::totail(int n)
097:   {
098:      Dlink* p = this;
099:      Dlink* q = this;
100:      int i = 0;
101:      while(p != NULL)
102:      {
103:         q = p;
104:         if(i == n)
105:            break;
106:         p = p->next;
107:         i++;
108:      }
109:      return(q);
110:   }
```

symlink.cpp

```
001:   #include  < stdio.h>
002:   #include  <stdlib.h>
003:   #include  "dlink.h"
004:   using namespace std;
005:
006:   class SymLink : public Dlink
007:   {
008:      private:
009:         unsigned char sym;
010:      public:
011:         SymLink(unsigned char c) : Dlink(), sym(c) {};
012:         SymLink* incr() {return((SymLink*) Dlink::incr());};
013:         SymLink* decr() {return((SymLink*) Dlink::decr());};
014:         SymLink* seek(int);
015:         int print();
016:   }
017:
018:   SymLink* SymLink::seek(int n)
019:   {
020:      if(n > 0)
021:         return((SymLink* ) Dlink::totail(n));
022:      if(n < 0)
023:         return((SymLink* ) Dlink::tohead(abs(n)));
024:      return(this);
025:   }
026:
027:   int SymLink::print()
028:   {
029:      SymLink* p = this;
030:      SymLink* q;
031:      int n = 0;
032:      while(p != NULL)
```

```
033:      {
034:         putchar(p->sym);
035:         q = p->incr();
036:         p = q;
037:         n++;
038:      }
039:      return(n-2);
040:   }
041:
042:   int main(int argc, char* argv[])
043:   {
044:      unsigned seed = 0;
045:      int count = 0;
046:      int length;
047:      int ch;
048:      unsigned pos;
049:      int side;
050:      SymLink* watch[2];
051:      SymLink* head;
052:      SymLink* tail;
053:      SymLink* p, *q;
054:      Dlink* (Dlink::* insert[])(Dlink*) = {&Dlink::after, &Dlink::before};
055:      if(argc >1)
056:        seed = atoi(argv[1]);
057:      watch[0] = head = new  SymLink('\n');
058:      watch[1] = tail = new  SymLink('\n');
059:      tail ->before(head);
060:      while((ch = getchar()) != '\n')
061:      {
062:         q = new SymLink(ch);
063:         tail->before(q);
064:      }
065:      if((length = head->print ( ) - 1) < 2)
066:         count = length;
067:      srand(seed);
068:      while(count < length)
069:      {
070:         side = rand() % 2;
071:         while((pos = rand() % length) == 0);
072:             printf("%*c\n", pos, '^');
073:          q = head->seek(pos);
074:          q->excluse();
075:          (watch[side]->*insert[side])(q);
076:          head->print();
077:          count++;
078:      }
079:      p = tail;
080:      while(p != NULL)
081:      {
082:         q = p->decr();
083:         p->excluse();
084:         delete p;
085:         p = q;
086:      }
087:      return(length+1);
088:   }
```

Разработать ООП для перестановки символов заданной строки в случайном порядке. Зерно случайной последовательности генератора псевдослучайных чисел должно передаваться через аргумент командной строки. Исходная строка должна передаваться программе через поток стандартного ввода. Результирующую строку образует последовательное перемещение символов из случайно выбранных позиций строки в ее начало или конец. После каждой такой перестановки полученная строка должна отображаться через поток стандартного вывода, а очередной случайно выбранный символ в ней указывается знаком ^. Число случайных перестановок равно длине строки. Разработка программы должна быть основана на использовании абстрактной структуры двунаправленного связанного списка с реализацией операций просмотра, удаления и вставки его элементов. Публичные методы должны определить базовый класс элемента абстрактного списка с защищенными адресными полями данных. Ему должен наследовать производный класс элемента списка символов с приватным полем кода символов и публичной перегрузкой базовых методов с адресным возвратом. Кроме того в нем должны быть определенны собственные публичные методы. Они должны обеспечивать стандартный вывод списка символов и адресацию их позиций по смещению от концов списка. В основной функции программы должна быть реализована необходимая функциональная обработка списка символов с использованием динамического распределения памяти для его элементов.

Наследование

Наследование представляет собой способность производного класса наследовать характеристики существующего базового класса.

Наследование бывает множественным и одиночным.

При наследовании наследуются не только информационные члены, но и методы класса. Не наследуются конструктор, деструктор, операция присваивания и дружественность.

Синтаксис наследования:

class <имя производного класса> **:** <тип наследования> <имя базового класса1>, <тип наследования> <имя базового класса2>…

Если тип наследования не указан, по умолчанию используется тип private для классов и public для структур.

Для инициализации элементов производного класса программа должна вызвать конструкторы базового и производного классов.

Последовательность вызова конструкторов – сперва вызываются конструкторы базовых классов, затем производного. Деструкторы вызываются в обратном порядке – сперва деструктор производного класса, затем базового.

Единственный способ вызова конструктора базового класса – использование списка инициализации.

Если в конструкторе производного класса в явном виде не указан конструктор базового класса, то вызывается конструктор по умолчанию или конструктор без аргументов. Таким образом, если при создании объекта производного класса требуется вызов конструктора по умолчанию базового класса, то этот конструктор по умолчанию можно не указывать в явном виде в списке инициализации производного класса.

Синтаксис конструктора производного класса:

<имя производного класса> (формальные параметры) **:** <имя базового класса> (фактические параметры) **{}**

Конструктор производного класса может быть пустым – это возможно в случае, если конструктор производного класса лишь вызывает конструктор базового класса.

Важно особенностью работы с объектами производных и базовых классов, является то, что объекты базового типа можно присвоить объект производного типа. При этом будут проинициализированы только совпадающие поля классов.

Так же вместо объекта базового класса возможно подстановка объектов производного класса. Аналогичный принцип распространяется и на указатели на объекты. Обратная подстановка запрещена. Это обусловлено тем, что помимо полей, наследованных от базового класса, производный класс содержит собственные поля, значения которых может быть некорректно при работе с объектом базового класса.

Например, указатель объекта производного класса может быть преобразован в

указатель базового класса неявно:
Proizclass p;
Bazclass* b = &p;

Для обратного преобразование требуется явное преобразование типа:
Bazclass b;
Proizclass* p = (Proizclass*) &b;