Тема 3. Указатели. Функции. Массивы (часть 2)

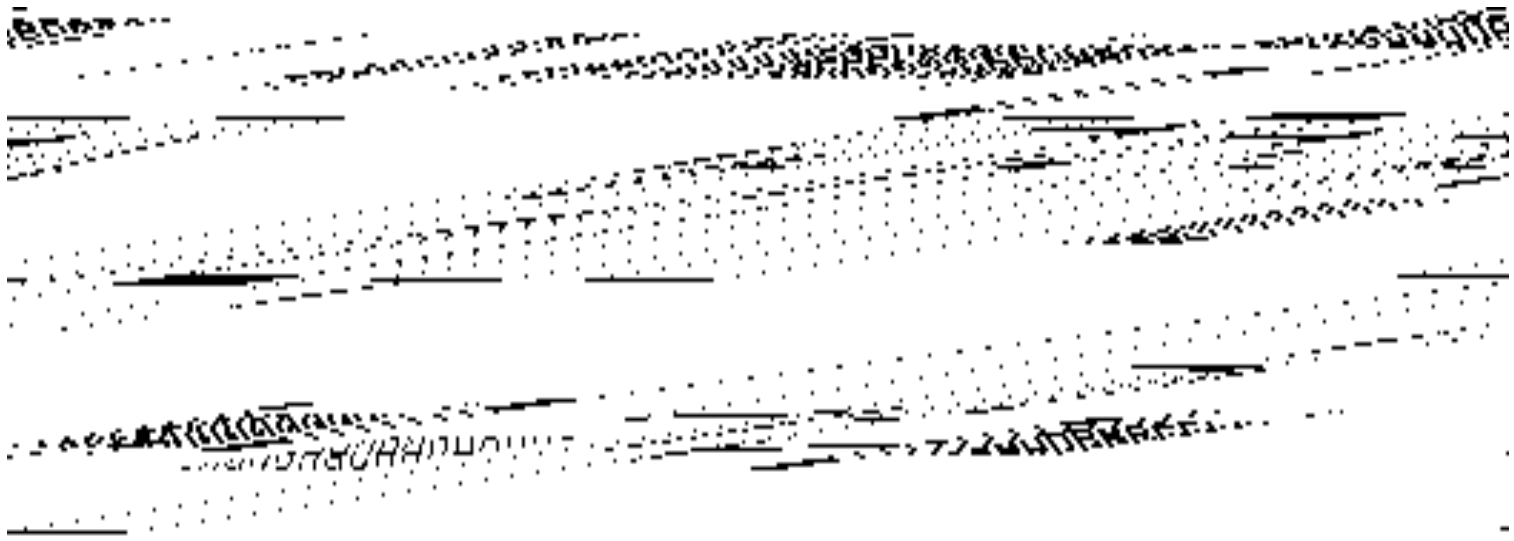
- Указатели и const
- Функция и двумерные массивы
- Реккурсия с множественными рекурсивными вызовами
- Указатели на функции
- Typedef, auto
- Альтернатива массивам
- Функции и объекты array

СПИСКОВАЯ СТРУКТУРА

Реальное многообразие структур данных базируется всего на двух основных способах получения адреса хранимого элемента: вычисление (массив) и хранение (указатель).

При построении структуры данных из указателей получается цепочка (последовательность) элементов, содержащих указатели друг на друга. В простейшем случае она может быть линейной (список), в более сложных случаях – ветвящейся (деревья, графы).

Список – линейная последовательность элементов, каждый из которых содержит указатели (ссылается) на своих соседей.



Физическое размещение в памяти элементов списка не имеет никакого значения, все определяется наличием ссылок на него в других элементах и извне. У массива всегда есть «начало». У списка по определению отсутствует фиксированная привязка к памяти.

ОСНОВНЫЕ РЕЛЯТИВИСТСКИЕ СВОЙСТВА СПИСКА

• **элемент списка доступен в программе через указатель.** «Смысл» этого указателя отражает функциональное назначение элемента списка в программе: **первый, последний, текущий, предыдущий, новый и т.п..** Между указателем и элементом списка имеется такая же взаимосвязь, как между индексом в массиве и элементом массива;

• **в программе список задается посредством заголовка – указателя на первый элемент списка;**

• **порядок следования элементов определяется последовательностью связей между элементами.** Изменение порядка следования элементов (вставка, удаление) осуществляются изменением переустановкой указателей на соседние элементы.

• **логический (порядковый) номер элемента списка также задается его естественной нумерацией в цепочке элементов;**

• **список является структурой данных с последовательным доступом.** Для получения n-го по счету элемента необходимо последовательно пройти по цепочке от элемента, на который имеется указатель (например, от заголовка);

• **список удобен для использования именно как динамическая структура данных**: элементы списка обычно создаются как динамические переменные, а связи между ними устанавливаются программно (динамически);

• **список обладает свойством локальности изменений: при вставке/удалении элемента изменения касаются только текущего и его соседей.** Вспомним массив: при

ОПРЕДЕЛЕНИЕ СПИСКА И РАБОТА С НИМ

Преимущества списков проявляются в таких структурах данных, где операции изменения порядка превалируют над операциями доступа и поиска.

Элемент списка является составной (структурированной) переменной, содержащей собственно хранимые данные и указатели на соседей:

```
struct   elem                    // определение структурированного типа
{        int    value;           // значение элемента (хранимые данные)
         elem *next;             // единственный указатель или
         elem *next,*prev;       // два указателя или
         elem *links[10];        // ограниченное количество указателей (не больше 10) или
         elem **plinks;          // произвольное количество указателей (внешний МУ)
};
```

Это еще не список, а описание его составляющих (элементов) как типа данных. Из него следует только, что каждый из них ссылается на аналогичные элементы. Никак нельзя определить ни количества таких переменных в структуре данных, ни характера связей между ними (последовательный, циклический, произвольный). Следовательно, конкретный тип структуры данных (линейный список, дерево, граф) зависит от функций, которые с ней работают. Значит, структура данных «зашита» не столько в этом определении, сколько в алгоритмах, работающих с этим типом.

ОПРЕДЕЛЕНИЕ СПИСКА И РАБОТА С НИМ

Списковые структуры данных обычно являются динамическими по двум причинам:

- сами переменные таких структур создаются как динамические переменные, то есть количество их может быть произвольным;
- количество связей между переменными и их характер также определяются динамически в процессе работы программы.

В зависимости от связей списки бывают следующих видов:

- односвязные - каждый элемент списка имеет указатель на следующий;
- двусвязные - каждый элемент списка имеет указатель на следующий и на предыдущий элементы;
- циклические - первый и последний элементы списка ссылаются друг на друга и цепочка представляет собой кольцо.

«Обычные» (разомкнутые) списки имеют в качестве ограничителя последовательности NULL-указатель. Соответственно, возможет случай пустого списка, в котором заголовок - указатель на первый элемент также содержит значение NULL.

БАЗОВЫЕ ДЕЙСТВИЯ СО СПИСКОМ

Работа со списками осуществляется исключительно через указатели. Каждый из них может перемещаться по списку (переустанавливаться с элемента на элемент), приобретая одну из смысловых интерпретаций – **указатель на первый, последний, текущий, предыдущий, новый и т.п. элементы списка.** Здесь уместной является аналогия с массивом и индексом в нем, но при условии, что индекс меняется линейно, а не произвольно, а текущее количество заполненных элементов в массиве задано отдельной переменной.

	Список	Массив
Определение	struct list {int val; list *next, *prev; };	int A[100]; int n;
Пустой список	list *ph=NULL;	n=0;
Первый	list *p; p=ph;	Int i=0;
Следующий	p->next	i+1
Предыдущий	p->prev	i-1
К следующему	p=p->next	i++
К предыдущему	p=p->prev	i--

БАЗОВЫЕ ДЕЙСТВИЯ СО СПИСКОМ

	Список	Массив
Просмотр	for (p=ph; p!=NULL; p=p->next) …p->val…	for (i=0; i<n; i++) …A[i]…
Проверка на последний	p->next ==NULL	i==n-1
К последнему	for (p=ph; p->next!=NULL; p=p->next);	i=n-1
Новый	list *q = new list; q->val = v;	int v;

Включить последним	for (p=ph; p->next!=NULL;
	p=p->next);
	q->next=NULL; p->next=q;
		A[n++]=v;

Включить первым	q->next=ph; ph=q;	for (i=n; i>0; i--) A[i]=A[i-1]; A[0]=v;

СТАТИЧЕСКИЙ СПИСОК

Основная операция при работе с элементами массива – «стрелочка» выполняется в контексте: указатель на элемент->поле элемента списка.

Собственно сам список представляет собой множество связанных указателями переменных. В простейшем случае, например, для создания тестовых данных, можно использовать статический список: его элементы представляет собой обычные структурированные переменные, связи между ними инициализируются транслятором, и вся структура данных «зашивается» в программный код.

struct list { int val; list *next; } a={0,NULL}, b={1,&a}, c={2,&b}, *ph = &c;

По условиям определения переменных, **список создается «хвостом вперед»**.

В двусвязном списке проблема «ссылок вперед» на еще неопределенные элементы решается так: **сначала переменные объявляются, как внешние, а потом определяются и инициализируются.**

struct list2 { int val; list *next,*prev;};

extern list2 a,b,c; // предварительное объявление элементов списка

list2 a={0,&b,NULL}, b={1,&c,&a}, c={2,NULL&b}, *ph = &c; // выделены «ссылки вперед»

СОЗДАНИЕ ДИНАМИЧЕСКОГО СПИСОКА ИЗ СТАТИЧЕСКОГО

Список может содержать ограниченное количество элементов, взятых из массива. Связи устанавливаются динамически, то есть программой. Такой вариант используется, когда заданное количество элементов образуют несколько различных динамических структур (например, очередей), переходя из одной в другую.

```
list       A[100],*ph;                  // Создать список элементов,
for (i=0; i<99; i++) {                   // размещенных в статическом массиве
        A[i].next = A+i+1;              // Адрес следующего вычисляется
        A[i].val = i;
        }
A[99].next = NULL;
ph = &A[0];
```

ДИНАМИЧЕСКИЙ СПИСОК

В динамическом списке элементы являются динамическими переменными, связи между ними устанавливаются программно.

```
list *ph=NULL;                  // Список пустой
for (int i=0; i<n; i++){        // Создать список из n элементов,
        list *q=new list;       // включая очередной в начало
        q->val=i;               // списка:
        q->next=ph;             // следующий за новым – бывший первый
        ph=q; }                 // новый становится первым
```

При модульном проектировании программы функции, работающие со списком, получают его через формальный параметр – заголовок списка.

```
//------ формальный параметр - заголовок списка
 void F1(list *p) {
 for (; p!=NULL; p=p->next) puts(p->val);
 }
```

«Подводный камень». Этот вариант полезен только в том случае, когда первый (по счету) элемент списка остается таковым в процессе работы со списком. Поскольку заголовок передается по значению (как копия), то **его изменение никак не сказывается на оригинале – истинном заголовке списка в main.**

СПОСОБЫ ВКЛЮЧЕНИЯ В НАЧАЛО СПИСКА С ИЗМЕНЕНИЕМ ЗАГОЛОВКА

- **путем возврата измененного значения заголовка в виде результата функции;**

// Вариант 1. Измененный указатель возвращается

```
list *Ins1(list *ph, int v)
{ list *q=new list;
q->val=v; q->next=ph; ph=q;
return ph; }
```

- **передачей указателя на заголовок списка (указателя на указатель);**

// Вариант 2. Используется указатель на заголовок

```
void Ins2(list **pp, int v)
{ list *q=new list;
q->val=v; q->next=*pp; *pp=q; }
```

СПОСОБЫ ВКЛЮЧЕНИЯ В НАЧАЛО СПИСКА С ИЗМЕНЕНИЕМ ЗАГОЛОВКА

- **передачей ссылки на заголовок.**

Ссылка - неявный указатель, использующий при работе синтаксис объекта, который «отображается» на соответствующий ему фактический параметр.

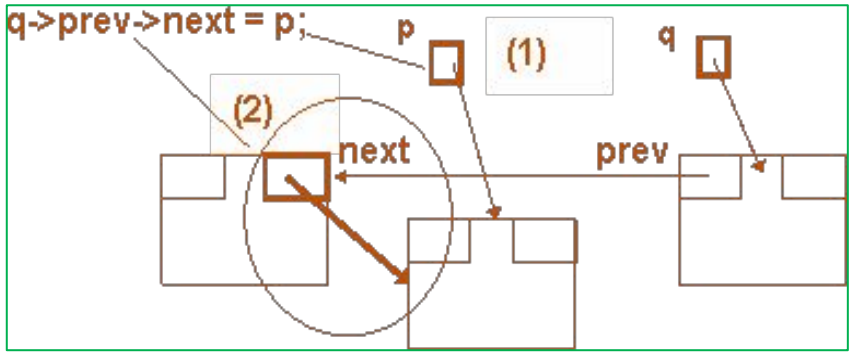
```
// Вариант 3. Используется ссылка на указатель
 void Ins3(list *&pp, int v)
 { list *q=new list;
 q->val=v; q->next=pp; pp=q; }

//---------- Пример вызова--------------------------------------
 void main(){
 list *ph=NULL;                    // пустой список
 ph=Ins1(ph,5);                          // сохранить новый заголовок
 Ins2(&ph,66);                   // передается адрес заголовка
 Ins3(ph,7); }                   // передается ссылка на заголовок
```

ИЗМЕНЕНИЕ ПОРЯДКА СЛЕДОВАНИЯ

Изменение порядка следования элементов осуществляется путем переустановки указателей в элементах списка и производится операциями присваивания указателей. Для понимания их сущности, прежде всего, надо четко разделять понятия «указатель» и «указуемый объект», а также учитывать асимметричный характер операции присваивания. **Содержательная интерпретация этого процесса.**

1. Графическая интерпретация: изменение порядка следования состоит в «соединении стрелкой» связываемых элементов:

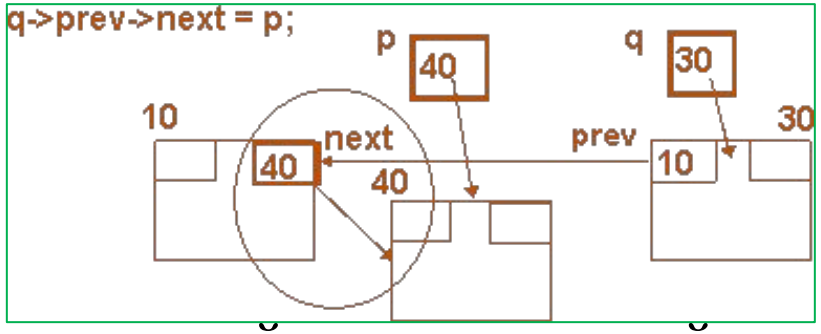


• в левой части операции присваивания должно находиться обозначение ячейки, в которую заносится новое значение указателя – ссылка на нее (2). При этом она должна быть достижима через имеющиеся рабочие указатели. На рисунке этому соответствует цепочка операций q->prev->next;

• в правой части операции присваивания должно находиться обозначение ячейки, из которой берется значение указателя элемента, на который ссылаются (1).

ИЗМЕНЕНИЕ ПОРЯДКА СЛЕДОВАНИЯ

2. Адресная интерпретация. Содержимым указателя является адрес указуемой переменной. Тогда элементам списка можно присвоить условные адреса – целые значения и в этом свете рассматривать операции копирования значений указателей.



3. Смысловая интерпретация. При работе со списками каждый указатель имеет определенный смысл – ссылка на первый, текущий, следующий, предыдущий и т.п. элементы списка. Поля prev, next также интерпретируются как указатели на следующий и предыдущий в элементе списка, доступном через указатель. Присваивание указателей однозначно можно однозначно выразить в словесной формулировке. Например, последовательность действий по включению нового элемента (указатель q) в двусвязный список перед текущим (указатель p) словесно формулируется так:

```
q->next=p;                    // следующий для нового = текущий
q->prev=p->prev;              // предыдущий для нового = предыдущий
if (p->prev == NULL) ph = q;  // для текущего включение в начало списка
else p->prev->next = q;       // включение в середину следующий для предыдущего = новый
p->prev=q;                    // предыдущий для текущего = новый
```

ОДНОСВЯЗНЫЙ СПИСОК

Простейший случай: элемент списка содержит единственный указатель на следующий (next), что создает «улицу с односторонним движением» и позволяет двигаться по списку только в одном направлении.

Операций включения и исключения элемента в начало списка вообще реализуются парой присваиваний, для включения в конец списка требуется дополнительный цикл просмотра списка до последнего элемента. Поэтому с помощью односвязного списка удобно представлять такие структуры данных, как стек и очередь: операции извлечения из очереди и стека реализуются через исключение из списка первого элемента, операция включения в стек соответствует включению элемента в начло списка, а включение в очередь – добавлению в конец.

ПРЕДСТАВЛЕНИЕ ОЧЕРЕДИ ОДНОСВЯЗНЫМ СПИСКОМ

```
// Включение в конец списка - помещение в очередь
void In(list *&ph,int v){
        list *q=new list;
        q->next=NULL;
        q->val=v;
        if (ph==NULL) ph=q;                  // список пуст - единственный
        else {  for (list *p=ph; p->next!=NULL; p=p->next); // цикл поиска конца списка
                p->next=q; }}                // следующий за последним - новый
// Включение в конец списка - помещение в очередь
void In(list *&ph,int v){
        list *q=new list;
        q->next=NULL;
        q->val=v;
        if (ph==NULL) ph=q;                  // список пуст - единственный
        else {   for (list *p=ph; p->next!=NULL; p=p->next); // цикл поиска конца списка
                p->next=q; }}                // следующий за последним - новый
```

ПРЕДСТАВЛЕНИЕ СТЕКА ОДНОСВЯЗНЫМ СПИСКОМ

```
// Включение в начало списка - помещение в стек
void PUSH(list *&ph,int v){
        list *q=new list;
        q->next=NULL;
        q->val=v;
        q->next=ph;                    // следующий за новым - бывший первый
        ph=q; }                        // новый теперь первый
// Исключение из очереди и стека – удалени первого элемента списка
int Out(list *&ph){
        if (ph==NULL) return -1;
        list *q=ph;                    // запомнить текущий
        ph=ph->next;                   // сдвинуться к следующему
        int v=q->val;
        delete q;                      // удалить текущий
        return v;}
```

ПРЕДСТАВЛЕНИЕ ОЧЕРЕДИ ОДНОСВЯЗНЫМ СПИСКОМ

```
// Указатели на первый и последний элементы;
// list    *PH[2];  -   заголовок очереди, [0]-первый, [1]-последний
void      In(list *ph[], int v){
        list *p= new list;                    // создать элемент списка;
        p->val=v;                             // и заполнить его
        p->next=NULL;                         // новый элемент - последний
        if (ph[0]==NULL) ph[0]=ph[1]=p;  // включение в пустую очередь
        else {                                // включение за последним элементом
                ph[1]->next = p;              // следующий за последним = новый
                ph[1] = p; }}                 // последний = новый

int       Out(list *ph[]){                    // извлечение из очереди
        if (ph[0]==NULL) return -1;           // очередь пуста
        list *q=ph[0];                        // исключение первого элемента
        ph[0]=q->next;
        if (ph[0]==NULL) ph[1]=NULL;     // элемент единственный
        int v = q->val;
        delete q; return v; }
```

Чтобы не «мотаться все время» к концу односвязного списка, можно добавить еще один указатель на последний элемент, и тогда операция помещения в очередь аналогично реализуется парой присваиваний.

ВКЛЮЧЕНИЕ В ОДНОСВЯЗНЫЙ С СОХРАНЕНИЕМ ПОРЯДКА

В операциях, связанных с элементами в середине списка возникает проблема – недоступность предыдущего элемента.

Если необходимо изменить его содержимое, то потребуются некоторые ухищрения, например:

- дополнительный цикл, пробегающий от начала списка до предыдущего элемента для найденного, например for(q=ph; q->next!=p; q=q->next);
- явное использование цикла с указателем на элемент, предыдущий по отношению к проверяемому, например for(p=ph->next; p->next!=NULL && p->next->val!=x; p=p->next). Обратите внимание, что такой цикл начинается со второго элемента, а значит, требуется отдельная проверка для первого элемента списка;
- использование дополнительного указателя на предыдущий элемент, запоминающего свое значение при переходе к следующему.

Последний вариант используется при вставке с сохранением порядка. Вставка происходит перед очередным, большим заданного, что требует коррекции предыдущего элемента списка.

ВКЛЮЧЕНИЕ В ОДНОСВЯЗНЫЙ С СОХРАНЕНИЕМ ПОРЯДКА

```
// pr - указатель на предыдущий элемент списка
 void InsSort(list *&ph, int v)
 { list *q ,*pr,*p;                    // перед переходом к следующему указатель на текущий
 q=new list; q->val=v;                 // запоминается как указатель на предыдущий
 for ( p=ph,pr=NULL; p!=NULL && v>p->val; pr=p,p=p->next);
 if (pr==NULL)                         // включение перед первым
          { q->next=ph; ph=q; }
 else                                  // иначе после предыдущего
          { q->next=p;                 // следующий для нового = текущий
          pr->next=q; }}               // следующий для предыдущего = новый
```

Дополнительная проверка «крайних ситуаций» показывает, что фрагмент, производящий поиск места включения, корректно работает и в случае пустого списка (производит включение перед первым).

СОРТИРОВКА ОДНОСВЯЗНОГО СПИСКА ВСТАВКАМИ

При сортировке массивов происходит физическое перемещение упорядочиваемых элементов (например, обмен значений). В списках принята другая технология, основанная на том, что **вместо перемещения элементов происходит переустановка их связей, что создает «иллюзию» изменения порядка.** Поэтому создание и уничтожение динамических переменных – элементов списка не требуется. Вместо этого в программе имеются два списка – входной и выходной, и в процессе сортировки элементы перемещаются из одного в другой путем «переброски» указателей. В случае вставок внешний цикл поочередно выбирает элементы входного списка, а внутренний – включает их в выходной список с сохранением порядка. Заметим, что программа составлена достаточно формально из перечисленных операций со списками.

```
list *sort(list *ph)                          // функция возвращает заголовок нового списка
 { list *q, *out, *p , *pr;
 out = NULL;                                  // выходной список пуст
 while (ph !=NULL)                            // пока не пуст входной список
        { q = ph; ph = ph->next;              // исключить очередной
for ( p=out,pr=NULL; p!=NULL && q->val>p->val; pr=p,p=p->next);  // поиск места включения
        if (pr==NULL) { q->next=out; out=q; }    // включение перед первым
        else   { q->next=p; pr->next=q; }        // иначе после предыдущего
        } return out; }
```

ДВУСВЯЗНЫЙ СПИСОК

Двусвязный список **позволяет двигаться по цепочке элементов в обоих направлениях**, поскольку доступны следующий и предыдущий элементы. Расплачиваться за это приходится увеличением количества операций над указателями.

Здесь уместно напомнить о такой проблеме проектирования как крайние ситуации. Необходимо рассматривать все возможные качественные комбинации входных данных и все варианты структур данных, с которыми может столкнуться программа. Применительно к двусвязному списку **при удалении выбранного элемента возможны следующие варианты:**

- **пустой список;**
- **единственный элемент;**
- **удаление первого элемента;**
- **удаление последнего элемента;**
- **удаление из середины.**

«Пропуск» одного из вариантов при проектировании приводит к появлению «мерцающих» ошибок, проявляющихся не постоянно, а периодически в зависимости от входных данных и порядка выполнения операций.

ДВУСВЯЗНЫЙ СПИСОК

Чтобы учесть эти варианты при уже написанном программном коде, нужно рассмотреть «историческое» поведение программы на каждом из этих вариантов:

• написанный код может корректно отрабатывать крайнюю ситуацию, она «вписывается» в существующий программный код;

• крайняя ситуация не отрабатывается программным кодом, в него вносится дополнительное проверочное условие с «заплаткой», обеспечивающей правильное поведение программы.

```
//-------- Удаление элемента списка по заданному логическому номеру
 list *Remove(list *&pp, int n)
 { list *q;                                          // Указатель на текущий элемент
 for (q = pp; q!=NULL && n!=0; q = q->next, n--);   // Отсчитать n -ый
 if (q==NULL) return NULL;                            // нет элемента с таким номером
 if (q->prev==NULL)                                  // удаление первого -
          pp=q->next;                                // коррекция заголовка
 else     q->prev->next = q->next;                    // следующий для предыдущего  =
                                                    // следующий за текущим
 if (q->next!=NULL)                                   // удаление не последнего -
          q->next->prev = q->prev;                 // предыдущий для следующего  =
 return q; }                                          // предыдущий текущего
```

ДВУСВЯЗНЫЙ СПИСОК

Витиеватое выражение **q->prev->next=q->next** производит «выкусывание» текущего элемента из списка путем перекидывания указателей с предыдущего на следующий, минуя текущий, что в словесной интерпретации звучит так: «указатель на следующий элемент в предыдущем получает значение указателя на следующий, взятый из текущего». Не обладающие достаточным образным воображением могут прибегнуть к соответствующей графической интерпретации.

В разомкнутом списке проверяется наличие следующего и предыдущего элементов для удаляемого. При отсутствии предыдущего корректируется заголовок (удаление первого). Все крайние ситуации корректно «вписываются» в программный код.

Сравнение операций включения с сохранением порядка в односвязный и двусвязный список дает ожидаемые выводы: сохранять указатель на предыдущий элемент нет необходимости, но программный код разрастается за счет увеличения количества переназначаемых указателей и крайних ситуаций.

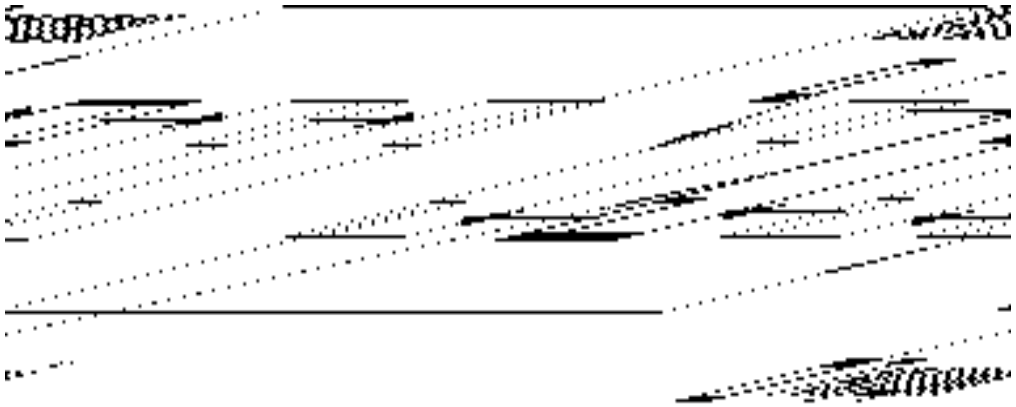
ВКЛЮЧЕНИЕ В ДВУСВЯЗНЫЙ СПИСОК С СОХРАНЕНИЕМ ПОРЯДКА

```
void InsSort(list *&ph, int v)                  // ссылка на заголовок
 { list *q , *p=new list;                       // новый элемент списка
 p->val = v;
 p->prev = p->next = NULL;
          if (ph == NULL) {                     // включение в пустой список
          ph=p; return ;
          }                                  // поиск места включения - q
 for (q=ph; q !=NULL && v > q->val; q=q->next);
 if (q == NULL){                                // включение в конец списка
          for (q=ph; q->next!=NULL; q=q->next);
          p->prev = q;                        // восстановить указатель на последний
          q->next = p;                        // включить перед текущим
          return; }
 p->next=q;                                     // следующий за новым = текущий
 p->prev=q->prev;                               // предыдущий нового = предыдущий текущего
 if (q->prev == NULL)                           // включение в начало списка
          ph = p;
 else                                                 // включение в середину
          q->prev->next = p;                    // следующий за предыдущим = новый
 q->prev=p;                                     // предыдущий текущего = новый
 }
```

ДВУСВЯЗНЫЙ ЦИКЛИЧЕСКИЙ СПИСОК

Циклический список связывает крайние элементы списка между собой, образуя кольцевую структуру. На практике «замкнутость» списка является удобным технологическим решением, позволяющим исключить постоянные проверки на «первый» и «последний». Т.е. логически последовательность элементов для внешнего пользователя остается линейной. Особенности такого списка:

- поле next последнего элемента ссылается на первый, а поле prev первого – на последний элемент списка;
- единственный элемент списка ссылается сам на себя (q->next=q и q->prev =q);
- операции включение элемента в начало и конец списка идентичны (перед первым и после последнего) за исключением того, что при включении в начало меняется заголовок.



ДВУСВЯЗНЫЙ ЦИКЛИЧЕСКИЙ СПИСОК

Цикл просмотра списка завершается, когда указатель текущего элемента возвращается на начало списка. Это удобно сделать в цикле с постусловием.

```
list *p=ph;
do      {
        // тело цикла для текущего элемента – p
        p=p->next;
        } while (p!=ph);
```

Доступность первого и последнего элемента циклического списка через заголовок позволяет реализовать на нем стек и очередь, не используя циклов движения по списку.

ОЧЕРЕДЬ НА ЦИКЛИЧЕСКОМ СПИСКЕ

```
void In(list *&ph, int v){
 list *q = new list;                    // Новый элемент как единственный
 q->val = v; q->next = q->prev = q;
 if (ph == NULL) { ph=q; return; } // Список пуст - единственный
 q->next = ph;                          // следующий за новым = первый
 q->prev = ph->prev;                    // предыдущий для нового = последний
 ph->prev->next = q;                    // следующий для последнего = новый
 ph->prev = q; }                        // предыдущий для первого = новый
int Out(list *&ph){
  if (ph==NULL) return -1;
  int v=ph->val;
  list *q=ph;                           // Запомнить текущий
  ph=ph->next;                          // Перейти на следующий
  if (ph->next==ph) ph=NULL;      // Единственный стал пустой
  q->next->prev=q->prev; // Элемент сам себя "выкусывает"
  q->prev->next=q->next;
  delete q;
  return v;
}
```

ВКЛЮЧЕНИЕ В ЦИКЛИЧЕСКИЙ СПИСОК С СОХРАНЕНИЕМ ПОРЯДКА

Все перечисленные особенности можно увидеть в примере включения нового элемента с сохранением упорядоченности. Поиск места включения завершается обнаружением первого элемента, большего заданного, либо возвращением на начало списка. В последней ситуации (вставка в конец списка) место вставки перед текущим элементом также является корректным: вставка перед первым – есть вставка после последнего.

```
list *InsSort(list *ph, int v)                  // Функция возвращает новый заголовок
 { list *q = new list;                          // Новый элемент как единственный
 q->val = v; q->next = q->prev = q;
 if (ph == NULL) return q;                      // Список пуст вернуть новый
 list *p = ph;
          do { if (v < p->val) break;           // Место вставки перед первым,
          p=p->next;                            // большим заданного, иначе -
          } while (p!=ph);                      // перед первым в списке (после последнего)
 q->next = p;                                   // следующий за новым = текущий
 q->prev = p->prev;                             // предыдущий для нового = предыдущий текущего
 p->prev->next = q;                             // следующий для предыдущего = новый
 p->prev = q;                                   // предыдущий для текущего = новый
 if ( ph->val > v) ph=q;                        // включение перед первым -
 return ph; }                                   // коррекция заголовка
```

ВКЛЮЧЕНИЕ В ЦИКЛИЧЕСКИЙ СПИСОК С СОХРАНЕНИЕМ ПОРЯДКА

Еще одна крайняя ситуация – вставка перед первым, почти «вписывается» общую схему. Цикл при этом не делает ни одного шага, и место вставки выбирается корректно, но после вставки заголовок требуется переустановить на новый элемент.

Ошибки при работе со списками

При отладке программ, работающих со списками, могут возникать специфические ошибки, связанные некорректной переустановкой связей между элементами:

- потеря связности – в результате нарушения порядка присваиваний (переустановки связей) некоторые элементы списка могут оказаться недоступными, или «потерянными»;
- топологические ошибки - нарушается линейность списка, «список перестает быть списком», а становится топологически более сложной структурой, на которой поведение программы становится непредсказуемым.

Аналогичный эффект можно получить, если забыть скорректировать заголовок списка, если операция касается первого элемента. Например, если в сортировке односвязного списка вставками (63-04.cpp) забыть присваивание указателя при вставке перед первым (out=q), то в такой ситуации очередной элемент входного списка просто будет потерян. Этот эффект будет «мерцающим»: теряться будут некоторые элементы со значением меньше всех предыдущих во входном списке (например, 6,7,3,8,4,2,6,5). Но программа все-таки будет работать. В случаях более серьезного нарушения топологии она может зациклиться или вообще «упасть».

ДОМАШНЕЕ ЗАДАНИЕ

Определите вид списка, «смысл» каждого указателя, выполняемое действие над списком, напишите вызов функций для статического списка.

```
struct list { int val; list *next,*prev; };
 //-------------------------------------------------- 1
 int F1(list *p)
 { int n;
 for (n=0; p!=NULL; p=p->next, n++);
 return n; }
 //-------------------------------------------------- 2
 list *F2(list *ph, int v)
 { list *q = new list;
 q->val = v; q->next = ph; ph = q;
 return ph; }
 //-------------------------------------------------- 3
 list *F3(list *p, int n)
 { for (; n!=0 && p!=NULL; n--, p=p->next);
 return p; }
```

ДОМАШНЕЕ ЗАДАНИЕ

```
//---------------------------------------------------- 4
list *F4(list *ph, int v)
{ list *p,*q = new list;
q->val = v; q->next = NULL;
if (ph == NULL) return q;
for ( p=ph ; p ->next !=NULL; p = p->next);
p ->next = q; return ph; }
//---------------------------------------------------- 5
list *F5(list *ph, int n)
{ list *q ,*pr,*p;
for ( p=ph,pr=NULL; n!=0 && p!=NULL; n--, pr=p, p =p->next);
if (p==NULL) return ph;
    if (pr==NULL) { q=ph; ph=ph->next; }
    else { q=p; pr->next=p->next; }
delete q;
return ph; }
```

ДОМАШНЕЕ ЗАДАНИЕ

```
//---------------------------------------------------- 6
 int F6(list *p)
 { int n; list *q;
 if (p==NULL) return 0;
 for (q = p, p = p->next, n=1; p !=q; p=p->next, n++);
 return n; }
 //---------------------------------------------------- 7
 list *F7(list *p, int v)
 { list *q;
 q = new list;
 q->val = v; q->next = q->prev = q;
 if (p == NULL) p = q;
 else
      { q->next = p; q->prev = p->prev;
      p->prev->next = q; p->prev = q; p=q;
      }
 return p; }
```

ДОМАШНЕЕ ЗАДАНИЕ

```
//---------------------------------------------------- 8
 list *F8(list *ph)
 { list *q, *out, *p , *pr;
 out = NULL;
 while (ph !=NULL)
      { q = ph; ph = ph->next;
      for ( p=out,pr=NULL; p!=NULL && q->val>p->val; pr=p,p=p->next);
      if (pr==NULL)
           { q->next=out; out=q; }
      else
           { q->next=p; pr->next=q; }
      } return out; }
```

ДОМАШНЕЕ ЗАДАНИЕ

```
//---------------------------------------------------- 9
list *F9(list *pp, int n)
{ list *q;
for (q = pp; n!=0; q = q->next, n--);
    if (q->next == q) { delete q; return NULL; }
if (q == pp) pp = q->next;
q->prev->next = q->next;
q->next->prev = q->prev;
delete q; return pp; }
//---------------------------------------------------- 10
list *F10(list *ph, int v)
{ list *q ,*pr,*p;
q=new list; q->val=v; q->next=NULL;
if (ph==NULL) return q;
for ( p=ph,pr=NULL; p!=NULL && v>p->val; pr=p,p=p->next);
    if (pr==NULL) { q->next=ph; ph=q; }
    else { q->next=p; pr->next=q; }
return ph; }
```

ДОМАШНЕЕ ЗАДАНИЕ

```
//---------------------------------------------------- 11
 list *F11(list *ph, int v)
 { list *q = new list;
 q->val = v; q->next = q->prev = q;
 if (ph == NULL) return q;
 list *p = ph;
      do {
      if (v < p->val) break;
      p=p->next;
      } while (p!=ph);
 q->next = p; q->prev = p->prev;
 p->prev->next = q; p->prev = q;
 if ( ph->val > v) ph=q;
 return ph; }
```

ДОМАШНЕЕ ЗАДАНИЕ

```
//---------------------------------------------------- 12
 void F12(list *&ph, int v, int n)
 { list *q = new list;
 int n0=n;
 q->val = v; q->next = q->prev = q;
 if (ph == NULL) { ph=q; return; }
 list *p;
 for (p=ph; n--!=0; p=p->next);
 q->next = p; q->prev = p->prev;
 p->prev->next = q; p->prev = q;
 if ( n0==0) ph=q; }
```