

Программирование на C++

Лекция 9

ПМИ 2 курс

Демяненко Я.М.

ЮФУ 2025

Динамические структуры данных

Динамические структуры данных — это такие структуры, которые в ходе выполнения программы могут менять свой размер.

Удобным средством реализации динамических структур являются списочные структуры (списки).

Списки

Списки — это программно реализуемые структуры данных, элементы которых хранят информацию и связи с другими элементами.

В качестве связи используется указатель на элемент.

По количеству связей и направленности выделяют различные типы списков:

- односвязные, двусвязные и многосвязные;
- линейные, нелинейные, кольцевые и т.д.

Размещение в динамической памяти

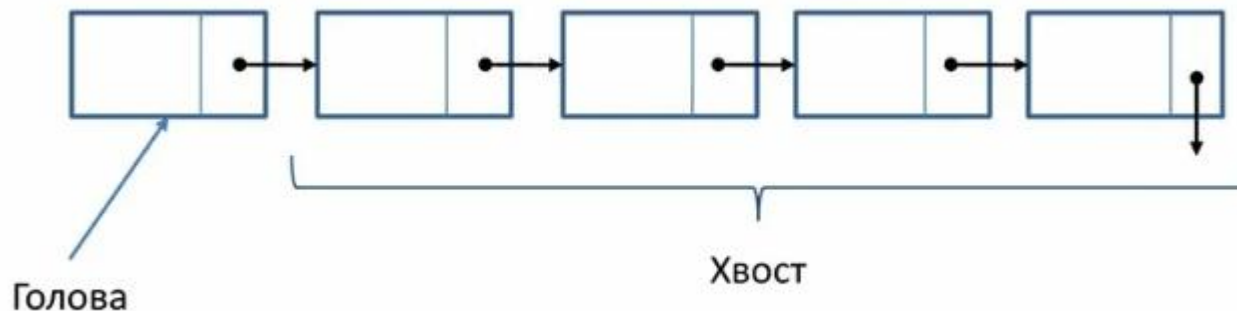
Для размещения элемента списка в динамической памяти используется операция **new**, для удаления — операция **delete**.

Указатели на список

Для линейного односвязного списка указатель должен ссылаться на первый элемент, который называется **головой (началом)** списка.

Иногда при реализации алгоритмов удобно иметь указатель на **последний** элемент списка, который иногда называют **хвостом**.

В общем случае хвостом называется весь список без головы. Такая терминология активно используется в функциональных языках программирования.



Узел для линейного односвязного списка



```
struct node {  
    int data;  
    node* next;  
    // this указатель на себя  
    node (int data, node* next) {  
        this->data=data;  
        this->next=next;  
    }  
};
```

$(*this).data \equiv this->data$

Линейный односвязный список

`node n1(4, nullptr);` При объявлении нового экземпляра структуры `node`, этот объект создается в статической памяти.
То есть объект `n1` будет храниться на стеке:
`node n1(4, nullptr);`

Память под `n1` выделяется на стеке самой программы, а конструктор лишь инициализирует поля.

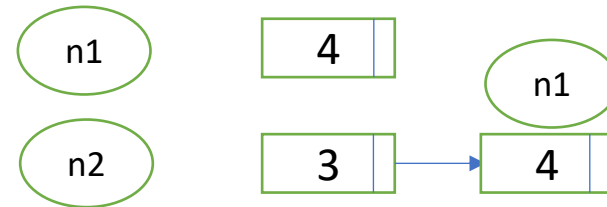


Линейный односвязный список

`node n1(4, nullptr);` При объявлении нового экземпляра структуры `node`, этот объект создается в статической памяти.
То есть объект `n1` будет храниться на стеке:
`node n1(4, nullptr);`

`node n2(3, &n1);`

Память под `n1` выделяется на стеке самой программы, а конструктор лишь инициализирует поля.

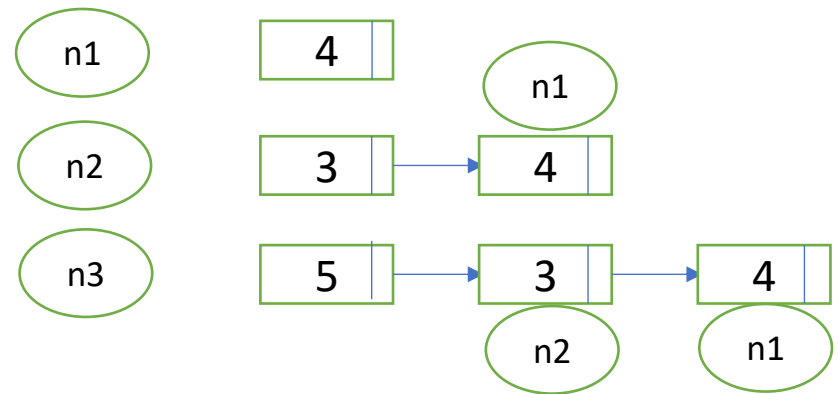


Линейный односвязный список

`node n1(4, nullptr);` При объявлении нового экземпляра структуры `node`, этот объект создается в статической памяти.
То есть объект `n1` будет храниться на стеке:
`node n1(4, nullptr);`

`node n2(3, &n1);`
`node n3(5, &n2);`

Память под `n1` выделяется на стеке самой программы, а конструктор лишь инициализирует поля.



Линейный односвязный список

```
node n1(4, nullptr);
```

При объявлении нового экземпляра структуры `node`, этот объект создается в статической памяти.

То есть объект `n1` будет храниться на стеке:
`node n1(4, nullptr);`

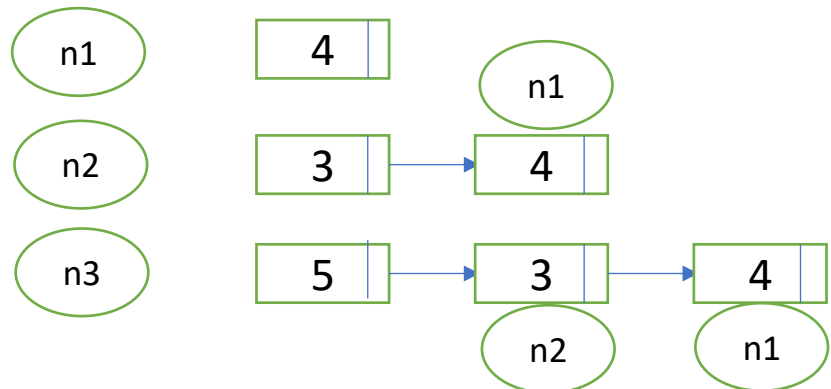
```
node n2(3, &n1);
```

```
node n3(5, &n2);
```

```
node* p = &n3;
```

В реальных программах, ввиду сильной ограниченности размера стека, объекты размещают в динамической памяти.

Память под `n1` выделяется на стеке самой программы, а конструктор лишь инициализирует поля.

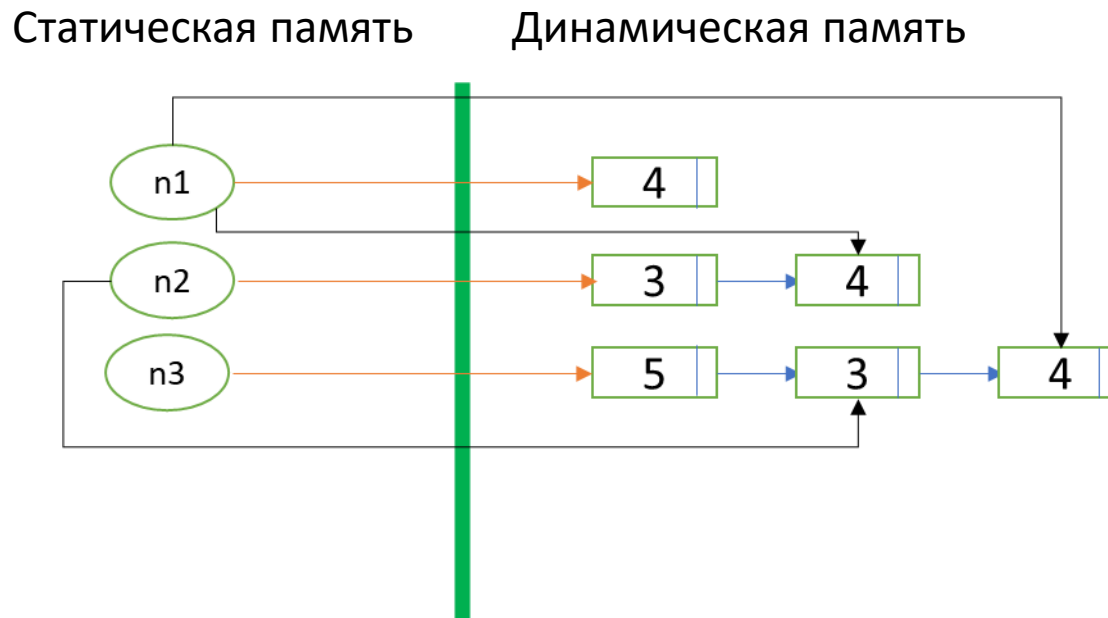


Как создать node в динамической памяти

```
node * pn = new node(5, nullptr);
```

В С++ динамическую память выделяет не конструктор, а оператор new. Конструктор только создаёт объект в выделенной памяти.

В отличие от .NET в С++ нет сборщика мусора, и ответственным за удаление объекта из динамической памяти, является программист.



Размерная vs ссылочная модель объектов

В C++ **размерная** модель объектов,
а **ссылочную** можно **моделировать** с помощью **указателей**.

Размерная модель (Value Semantics / Модель по значению)

При копировании или передаче объекта создается его полная, независимая копия. Вы работаете с двумя разными «физическими» объектами в памяти.

Как работает:

- Когда вы присваиваете одну переменную другой ($a = b$), все данные из b копируются в a .
- Последующие изменения a не влияют на b , и наоборот.
- Каждая переменная хранит свою собственную копию данных.

Ссылочная модель (Reference Semantics / Модель по ссылке)

При копировании или передаче объекта копируется не сам объект, а ссылка (адрес в памяти) на него. Несколько переменных могут ссылаться на один и тот же объект.

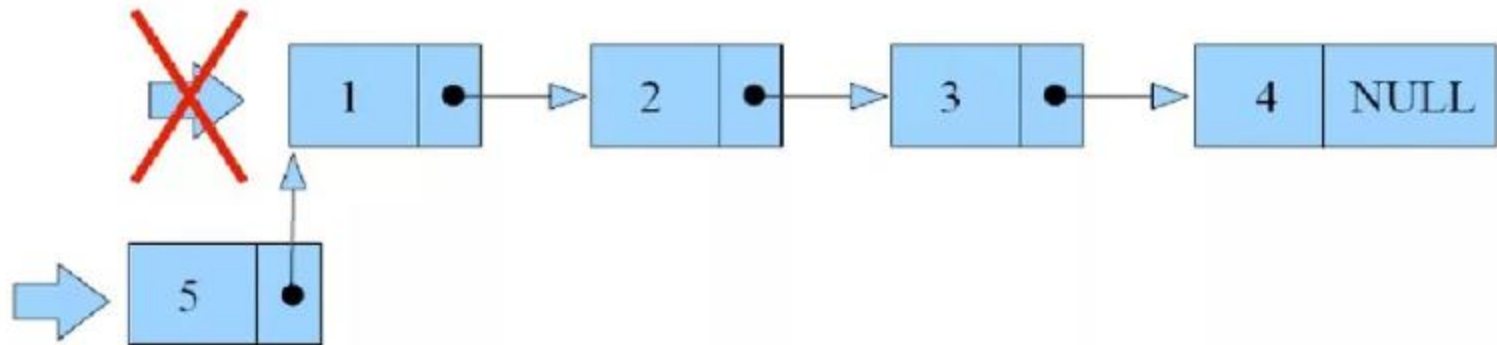
Как работает:

- Когда вы присваиваете одну переменную другой ($a = b$), вы копируете из b в a лишь ссылку на объект.
- Теперь обе переменные (a и b) указывают на один и тот же объект в памяти.
- Изменения объекта через одну переменную немедленно видны через все другие переменные, которые на него ссылаются.

Основные операции со списком

- Добавить элемент в начало, в конец, в середину (на i -ю позицию)
- Удалить элемент из начала, из конца, из середины (i -ый элемент)
- Обработать элементы всего списка
- Проверить на пустоту
- Изменить порядок элементов (без new и delete)

Добавление в начало односвязного списка



Добавление элемента в начало односвязного списка

```
node* pn = nullptr;          // создаём пустой список  
pn = new node(5, pn);        // создать новый элемент и добавили в список
```

Операцию **добавления** первого элемента в односвязный список мы оформим в виде отдельной функции.

```
void add_first(node* &pn, int x) { // pn-указатель на начало списка  
    pn = new node(x, pn);  
}
```

Запись **node* &pn** означает, что pn это **ссылка на указатель** типа node, и изменения, происходящие с указателем внутри функции, повлияют и на изменение фактического параметра.

```
node* pn = nullptr;
```

```
add_first(pn, 5);
```

```
add_first(pn, 3);
```

```
...
```

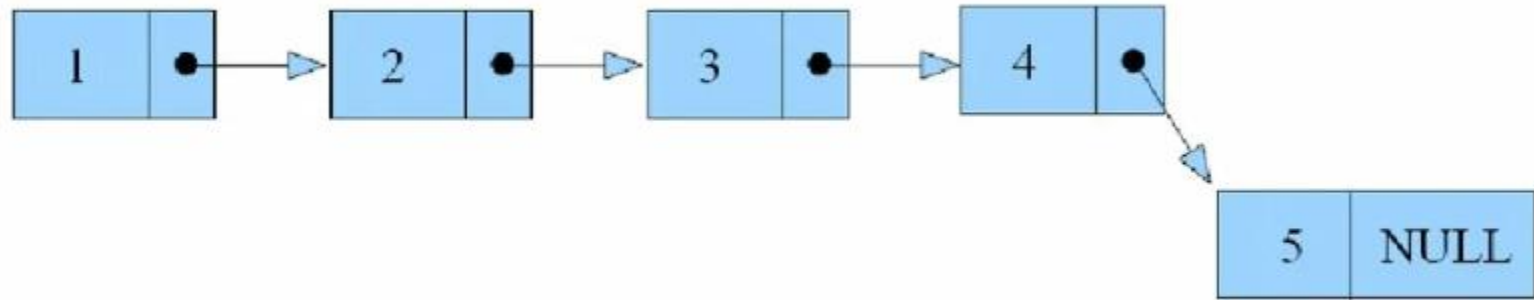
Цикл по односвязному списку

```
void print(node* p) {  
    while(p) { // p!= nullptr  
        cout << p -> data << ' ';  
        p = p -> next;  
    }  
}
```

```
void print(node* p) {  
    for ( ; p; p=p->next)  
        cout << p -> data << ' ';  
}
```

Пример. Создать линейный односвязный список целых чисел добавлением элементов в конец списка.

Операции создания списка, удаления и вывода на экран оформить в виде функций.



```
struct list {  
    int inf;  
    list* next;  
};  
  
list* sp_create_to_tail();  
void print_sp(list* L);  
void erase(list*&L);
```

```
int main() {  
    list* F=nullptr;  
    F = sp_create_to_tail();  
    print_sp(F);  
    erase(F);  
    return 0;  
}
```

```

list* sp_create_to_tail() {
    list* head, *p, *tail;
    int n;  cout << "size list->";  cin >> n;
    head = tail = nullptr;
    if (n > 0) {
        tail=head = new list;
        cout << "list item ->";
        cin >> head->inf;
        head->next = nullptr;
    }
    for (int i = 1; i<n; ++i) {
        p = new list;
        cout << "list item ->";
        cin >> p->inf;
        p->next = nullptr;
        tail->next = p;
        tail = p;
    }
    return head;
}

```

Какой верный? Почему?

```
void print_sp(list* L) {  
    list * p = L;  
    while (p != nullptr) {  
        cout << p->inf << " ";  
        p = p->next;  
    }  
    cout << "\n";  
}
```

```
void print(list * p) {  
    while(p) {  
        cout << p -> inf << ' ';  
        p = p -> next;  
    }  
}
```

```
void erase(list*&L) {  
    list* t = L;  
    while (t) {//!= nullptr // L != nullptr // L  
        L = t->next;  
        delete t;  
        t = L;  
    }  
}
```

«Особые» элементы списка

Список пуст или указатель вне списка

if (F== nullptr) ... или if (!F)...

if (p== nullptr) ... или if (!p)...

Список не пуст или указатель в списке

if (F!= nullptr) ... или if (F)...

if (p!= nullptr) ... или if (p)...

Первый (голова)

if (p==F)...

И ещё «особые» элементы списка

Последний (хвост)

`if (p->next == nullptr)...` или `if (!p->next)...`

Для общего случая

`if (p && !p->next) ...`

Предпоследний

`if (p->next->next == nullptr) ...` или `if (! p->next->next)`

...

Для общего случая

`if (p && p->next && !p->next->next) ...`

Добавление в «голову» (начало списка)

```
typedef list* sp_ptr;
```

```
void inHead (sp_ptr &L , int a){  
    sp_ptr p = new list;          // list * p;  
    p->inf = a;  
    p->next = L;  
    L = p;  
}
```

вызов в main

```
sp_ptr F = nullptr;           // или list * F = nullptr;  
inHead(F, 5);
```

Добавление в «голову» (начало списка)

Другой подход

```
sp_ptr inHeadAdd (sp_ptr L , int a){  
    sp_ptr p=new list;  
    p->inf = a;  
    p->next = L;  
    return p;  
}
```

вызов в main

```
F = inHeadAdd (F, 5);
```

Добавление в начало списка (с использованием конструктора)

```
struct node {  
    int data;  
    node* next;  
    node (int data, node* next) {  
        // this указатель на себя  
        this->data=data;  
        this->next=next;  
    }  
};
```

Добавление в начало списка (с использованием конструктора)

```
void add_first(node* &pn, int x) {  
    pn = new node(x, pn);  
}
```

```
node* pn = nullptr;  
add_first(5, pn);
```

```
node* add_first(node* pn, int x) {  
    pn = new node(x, pn);  
    return pn;  
}
```

```
node* pn = nullptr;  
pn = new node(5, pn);
```

Добавление в «хвост» (в конец списка)

Можно каждый раз искать место добавления

```
void inTail (sp_ptr &L , int a){  
    if (!L) inHead(L,a);      // или L=inHeadAdd(L,a);  
    else{  
        sp_ptr p =L;  
        while (p->next)      // ищем последний элемент  
            p=p->next;  
        p->next = new list;   // добавили  
        p=p->next;           // перешли к добавленному  
        p->inf=a;             // инициализируем  
        p->next=nullptr;  
    }  
}
```

вызов в main

```
inTail(F, -5);
```

Добавление в «хвост» (в конец списка)

А можно запоминать «хвост» и указывать, куда добавлять
(для пустого списка не применять!!!)

Частный случай

```
sp_ptr inTail (sp_ptr T , int a){  
    assert(T!=0);          // нужен #include <cassert>  
    T->next = new list;  
    T->next->inf  = a;  
    T->next->next = nullptr;  
    return T->next;  
}
```

Добавление в конец списка (с использованием конструктора)

```
struct node {  
    int data;  
    node* next;  
    node (int data, node* next) {  
        // this указатель на себя  
        this->data=data;  
        this->next=next;  
    }  
};
```


Добавление в конец списка (с использованием конструктора)

```
void add_last(node* &pn, int x) {  
    if (!pn)  
        pn = new node(x, pn);  
    else{  
        node* p = pn;  
        while (p->next)  
            p = p->next;  
        p->next = new node(x, nullptr);  
    }  
}
```

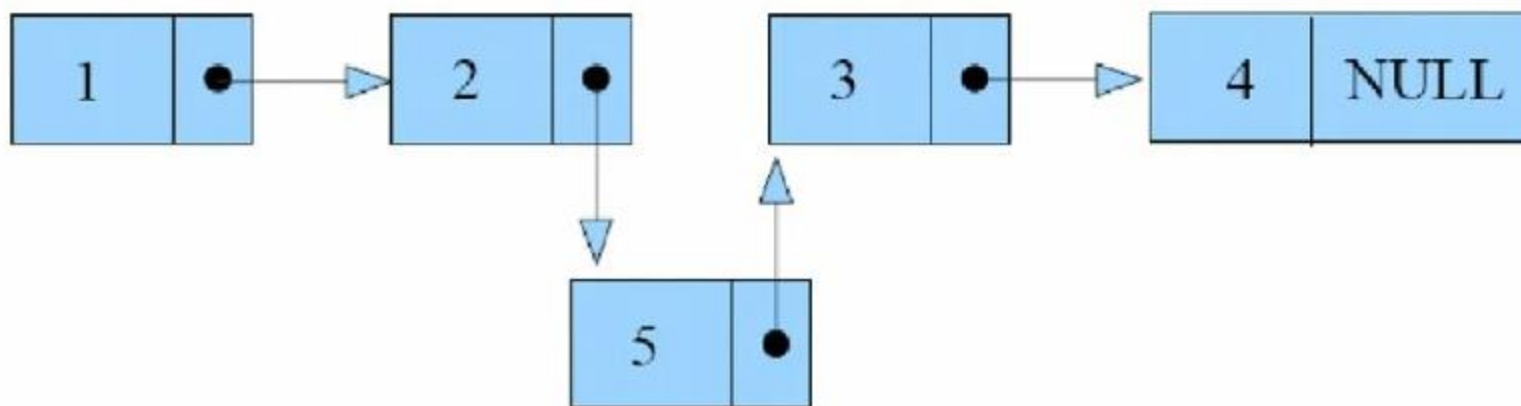
```
void inTail (node* &pn, node* & T , int a){  
    if (!pn)  
        T = pn = new node(x, pn);  
    else{  
        T->next = new node(x, nullptr);  
        T = T->next;  
    }  
}
```

Использование

```
int main() {  
    list* F= nullptr, *T= nullptr;  
  
    F=inHead(F,1);  
    T = F;  
  
    for (int i = 0; i<5; i++) {  
        F = inHead(F,i);  
        T = inTail(T,i);  
    }  
    print_sp(F);  
    erase(F);  
    T=nullptr; //T=F;  
    return 0;  
}
```

Результат вывода 43210101234

Вставка на произвольное место



Обобщение – вставка на произвольное место

```
sp_ptr insert (sp_ptr L1 , sp_ptr L2, int a) {
```

```
// поместить элемент между L1 и L2
```

```
// L1 и L2 должны быть расположены подряд
```

```
// если L2 – голова, то L1 должно быть nullptr – в начало
```

```
// если L1 – хвост, то L2 должно быть nullptr – в конец
```

```
    sp_ptr p = new list;
```

```
    p->inf = a;
```

```
    p->next = L2;
```

```
    if (L1)
```

```
        L1->next=p;
```

```
    return p;
```

```
}
```

Обобщение – вставка на произвольное место с использованием конструктора

```
node* insert (node* & L1 , node* & L2, int a) {  
  
    // поместить элемент между L1 и L2  
    // L1 и L2 должны быть расположены подряд  
    // если L2 – голова, то L1 должно быть nullptr  
    // если L1 – хвост, то L2 должно быть nullptr  
  
    node* p = new node(a, L2);  
    if (L1)  
        L1->next=p;  
    return p;  
}
```

Пример использования

```
int main() {  
    sp_ptr F= nullptr, T= nullptr;  
    T=F = insert (nullptr, nullptr, 1); // в пустой список  
  
    for (int i = 0; i<5; i++) {  
        F = insert(nullptr, F, i);    //в начало  
        T = insert (T, nullptr, i);    // в конец  
    }  
    sp_ptr p=F;  
    p = p->next;  
    p= insert(p, p->next, -1);  
    print_sp(F);  
    erase(F); T=nullptr;  
    return 0;  
}
```

Обработка всех элементов

```
int sum ( sp_ptr L){
    int s=0;
    for (sp_ptr p=L; p; p=p->next)
        s+=p->inf;
    return s;
}
```

```
// можно и так
int sum ( sp_ptr L){
    int s=0;
    for (; L; L=L->next)
        s+=L->inf;
    return s;
}
```

Рекурсивный вариант

```
int sum_r ( sp_ptr L){
    if (L)
        return L->inf+sum_r(L->next);
    else
        return 0;
}
```

Обработка с предикатом

```
int kol (sp_ptr L, bool (*f) (int)) {  
    int k=0;  
    for ( ; L; L=L->next)  
        if (f(L->inf) )  
            k++;  
    return k;  
}
```

```
bool isNull(int x) {  
    return (x==0);  
}
```

```
int N = kol (F, isNull);
```

```
bool twoTerm (int x) {  
    return ( x%10 == 2);  
}
```

```
int k2 = kol (F, twoTerm);
```


Обработка соседних элементов

```
int kolPair (sp_ptr L){  
    int k=0;  
    if (L)  
        for (sp_ptr p=L; p->next; p=p->next)  
            if (p->inf == p->next->inf)  
                k++;  
    return k;  
}
```

Проверка свойств

```
bool allZerro (sp_ptr p) {  
    while (p && (p->inf==0))  
        p=p->next;  
    return (p==nullptr);  
}
```

Проверка свойств с предикатом

```
bool allPred (sp_ptr p, bool (*f) (int)) {  
    while (p && f(p->inf) )  
        p=p->next;  
    return (p==nullptr);  
}
```

Удаление

Удаление первого элемента («головы»)

```
void delHead( sp_ptr &L) {  
    if (L) {  
        sp_ptr p= L;  
        L=L->next;  
        delete p;  
    }  
}
```

Удаление

Удаление последнего («хвост»)

```
void delTail( sp_ptr &L) {  
    if (L) {  
        if (L->next) {  
            sp_ptr p = L;  
            while (p->next->next) //предпоследний элемент  
                p=p->next;  
            delete p->next;  
            p->next = nullptr;  
        }  
        else {  
            delete L;  
            L =nullptr;  
        }  
    }  
}
```

Удаление

Если есть указатель на «хвост»

```
void delTail( sp_ptr &H, sp_ptr &T) {  
    if (H) {  
        if (H->next) { // H!=T  
            sp_ptr p = H;  
            while (p->next != T)  
                p=p->next;  
            delete p->next;  
            p->next = nullptr;  
            T=p;  
        }  
        else {  
            delete H;  
            T=H= nullptr;  
        }  
    }  
}
```

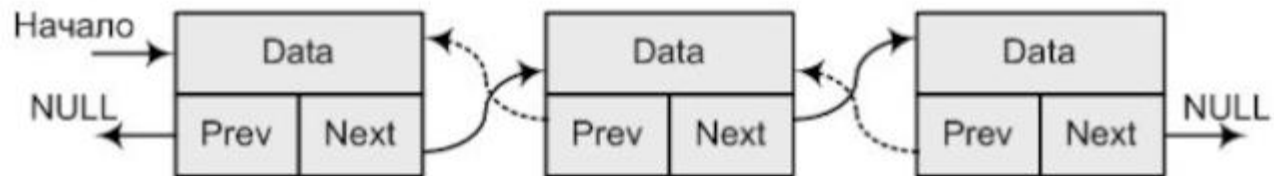
Классификация списочных структур

- По количеству полей связи
 - односвязные
 - двусвязные
 - многосвязные
- По возможному порядку просмотра элементов
 - линейные
 - нелинейные
 - циклические

1. Односвязный линейный список



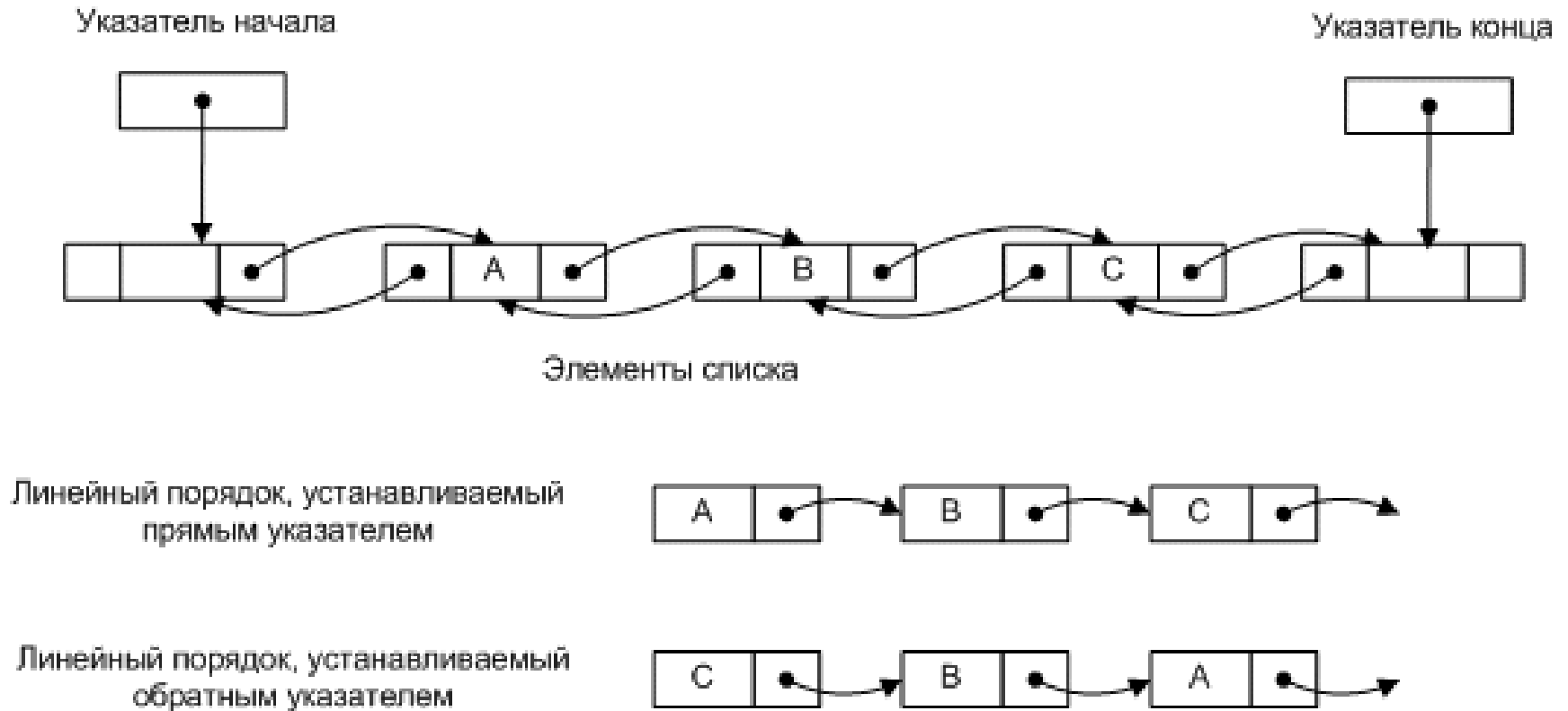
2. Двусвязный линейный список



3. Односвязный линейный кольцевой список



© С



Вторая связь позволяет эффективней реализовывать операции, в которых требуется «возвращаться» к предыдущему элементу списка. Например, найти и удалить элемент с заданным значением

Двусвязный, двунаправленный

```
struct list2 {  
    int inf;  
    list2* next, *prev;  
};
```

```
typedef list2* sp_ptr2;
```

//Все операции должны учитывать вторую связь

```
sp_ptr2 inHead (sp_ptr2 L, int a) {  
    sp_ptr2 p = new list2;  
    p->inf = a; p->next = L; p->prev=nullptr;  
    if (L) {  
        L->prev = p;  
    }  
    return p;  
}
```

Двусвязный, двунаправленный

```
void delete( sp_ptr2 &L, int v) {  
    if (L) {  
        sp_ptr2 p= L;  
        while (p && p->inf !=v)  p=p->next;    // ищем  
        if (p) {                                // если нашли  
            if (p!=L && p->next){                // не голова и не хвост  
                p->prev->next=p->next;  
                p->next->prev=p->prev;  
            }  
            else if (p==L){                      // голова  
                L = p->next;  
                if (L) L->prev = nullptr;        // если был неединственный  
            }  
            else { p->prev->next = p->next; }  
            delete p;  
        }  
    }  
}
```

Поместить элемент между L1 и L2

```
void insert(pList &F, pList &L, int a, pList L1, pList L2 ) {  
    assert(L1 == nullptr || L2 == nullptr || L1->next == L2);  
    pList p = new list;  
    p->inf = a;  
    p->prev = L1;  
    p->next = L2;  
    if (L1)  
        L1->next = p;  
    else  
        F = p;  
    if (L2)  
        L2->prev = p;  
    else  
        L = p;  
    //if (!L1 && !L2)  
        //F = L = p;  
}
```

// L1 и L2 должны указывать на расположенные подряд
// элементы одного списка, причем L1 левее L2
// или L1 должно быть nullptr, если L2 – первый
// или L2 должно быть nullptr, если L1 – последний

Напечатать список в обратном порядке

Двусвязный

```
void reversePrint(pList L){
    pList p;
    p = L;
    while (p != 0) {
        cout << p->inf << " ";
        p = p->prev;
    }
    cout << "\n";
}
```

Односвязный и двусвязный

```
void reversePrint(pList F){
    if (F != nullptr) {
        reversePrint(F->next);
        cout << F->inf << " ";
    }
}
```

Другие варианты двусвязного списка

- Вторая связь может связывать не все элементы списка, например, если список символов содержит текст, то вторая связь может позволять возвращаться на начало слова (связывает последнюю и первую буквы).
- Вторая связь может иметь направление не только противоположное основной. Например, список символов, представляет произвольный текст. Вторая связь позволяет переходить от слова к слову, т.е. связывает первые буквы слов

Другие варианты двусвязного списка

- DOM (*Document Object Model* — «объектная модель документа») — это независимый от платформы и языка программный интерфейс, позволяющий программам и скриптам получить доступ к содержимому HTML-, XHTML- и XML-документов, а также изменять содержимое, структуру и оформление таких документов.
- Любой документ известной структуры с помощью **DOM** может быть представлен в виде **дерева узлов**, каждый узел которого представляет собой элемент, атрибут, текстовый, графический или любой другой объект. **Узлы связаны между собой отношениями «родительский-дочерний»**