# ЛИНЕЙНЫЕ СПИСКИ

Некоторые задачи требуют введения структур, способных увеличивать или уменьшать свой размер в процессе работы программы.

Основу таких структур составляют динамические переменные, которые хранятся в некоторой области памяти.

Обращение к ним производится с помощью указателя.

Как правило, такие переменные организуются в виде списков, элементы которых являются структурами (***struct***).

Если для связи элементов в структуре задан указатель (адресное поле) на следующий элемент, то такой список называют ***однонаправленным*** (*односвязным*).

Если добавить в структуру еще и указатель на предыдущий элемент, то получится ***двунаправленный*** *список* (*двусвязный*).

Если последний элемент связать указателем с первым, получится ***кольцевой*** *список.*

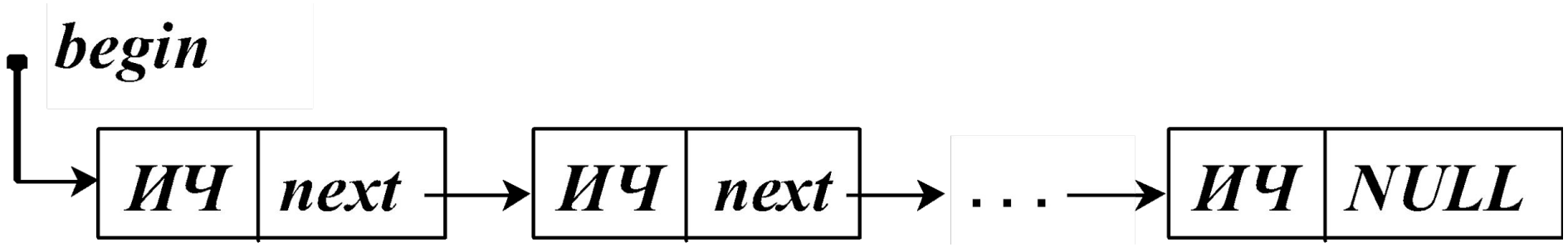
Для работы с ***однонаправленными*** списками шаблон структуры (структурный тип) будет иметь следующий вид:

```
struct TList1 {
    Информационная часть (ИЧ)
    TList1 *next; – Адресная часть
};
```

*Информационная часть* – описание полей (членов структуры), определяющих обрабатываемую в списке информацию;

***next*** – указатель на следующий элемент.

Схема такого списка может иметь вид:




***begin*** – адрес первого элемента в списке;

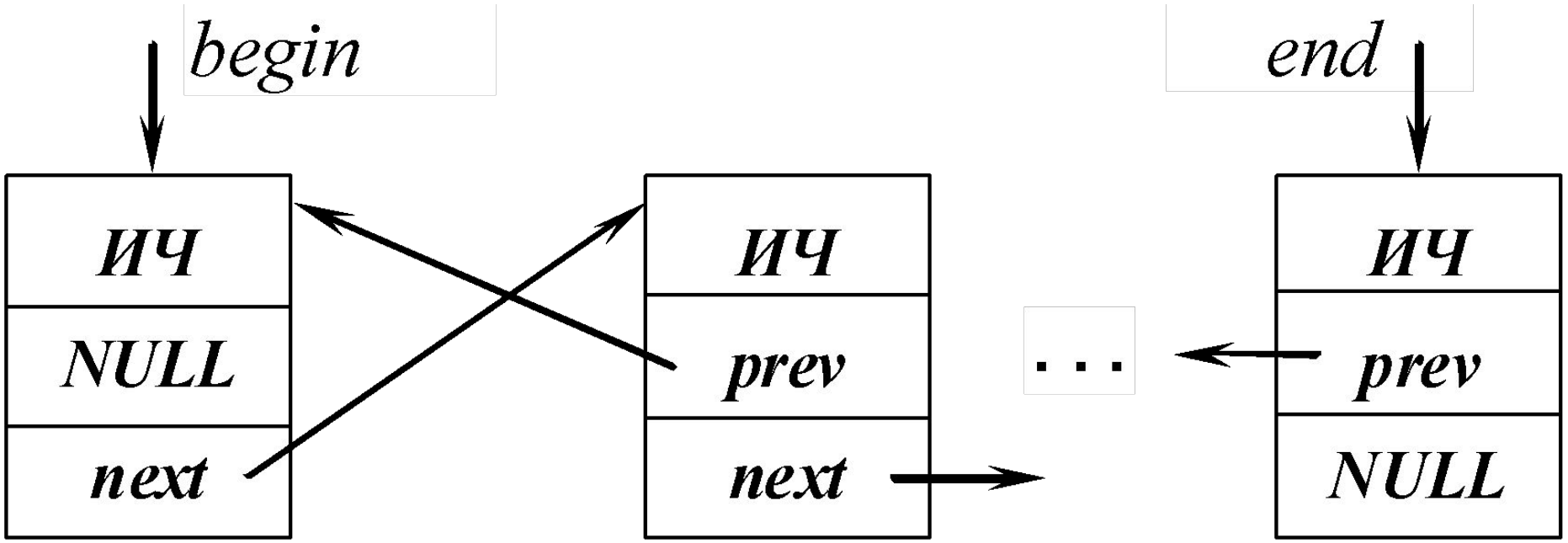
Адресная часть последнего элемента равна ***NULL*** – признак того, что следующего за ним НЕТ!

Для работы с ***двунаправленными*** списками шаблон структуры будет иметь следующий вид:

```
struct TList2 {
    Информационная часть (ИЧ)
    TList2  *prev, *next;
};
```

***prev*** – указатель на предыдущий элемент;
***next*** – указатель на следующий элемент.

Схема такого списка будет иметь вид:




***begin*** и ***end*** – адреса первого и последнего элементов в списке;

Адресные части ***prev*** первого элемента и ***next*** последнего элемента равны ***NULL***.

Над списками обычно выполняются следующие *операции:*

– начальное формирование списка (создание первого элемента);

– добавление нового элемента в список;

– обработка (просмотр, поиск, удаление и т.п.);

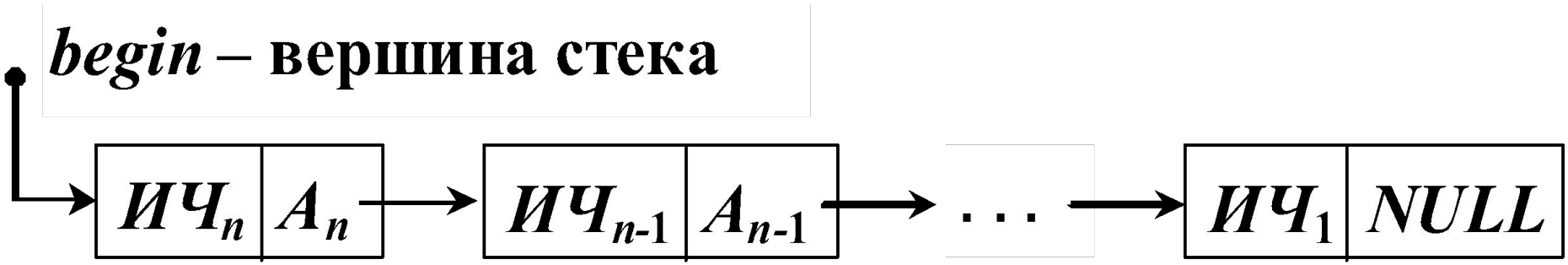
– освобождение памяти, занятой всем списком.

# *Структура данных СТЕК*

***Стек*** – упорядоченный набор данных, в котором добавление и удаление элементов производится только с конца, который на-зывают ***вершиной*** стека, т.е. стек – список с одной точкой доступа к его элементам.

Стек это структура данных типа ***LIFO*** (*Last In, First Out*) – последним вошел, первым выйдет.

Графически Стек можно изобразить так:




Стек получил свое название из-за схожести с обоймой патронов:

когда добавляется новый элемент, прежний проталкивается вниз и становится недоступным;

когда верхний элемент удаляется, следующий за ним поднимается вверх и становится опять доступным.

Число элементов стека не ограничивается. При добавлении элементов в стек память должна динамически выделяться и освобождаться при удалении. Таким образом, стек – динамическая структура данных, состоящая из переменного числа элементов одинакового типа.

Операции, выполняемые над стеком, имеют специальные названия:

***push*** – добавление элемента (вталкивание);

***pop*** – удаление (извлечение) элемента, верхний элемент стека удаляется (не может применяться к пустому стеку).

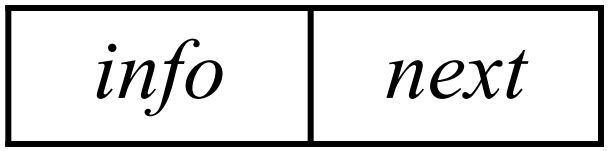
Кроме этих обязательных операций используется операция ***top*** (*peek*) для чтения информации в вершине стека без извлечения.

Рассмотрим основные алгоритмы работы со стеком, взяв для простоты в качестве информационной части целые числа (***int info;***), хотя информационная часть может состоять из любого количества объектов допустимого типа, за исключением файлов.

## *Алгоритм формирования стека*

Рассмотрим данный алгоритм для первых двух элементов.

1. Описание типа для структуры, содержащей информационное и адресное поля:

struct Stack 


Структурный тип (шаблон) рекомендуется объявлять глобально:

```
struct   Stack {
    int  info;
    Stack   *next;
} ;
```

2. Объявление указателей на структуру (можно объявить в шаблоне глобально):

Stack *begin, – Вершина стека

*t; – Текущий указатель

3. Так как первоначально стек пуст:

begin = NULL;

4. Захват памяти под первый элемент:

t = new Stack;

в памяти формируется конкретный адрес (обозначим его ***A1***) для первого элемента, т.е. адрес текущего элемента ***t*** равен ***A1***.

5. Ввод информации (обозначим ***i*1**);

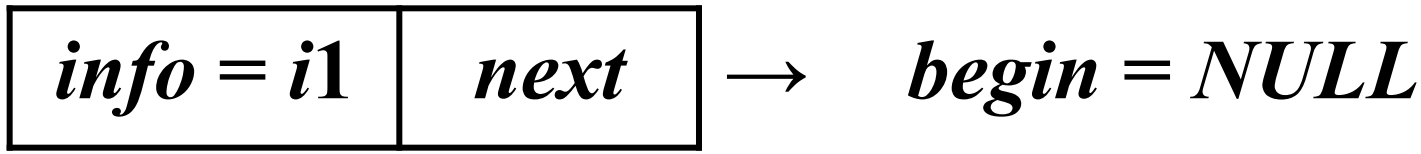
а) формирование информационной части:

```
t -> info = i1;
```

б) формирование адресной части: значение адреса вершины стека записываем в адресную часть текущего элемента (там был *NULL*)

```
t -> next = begin;
```

На этом этапе получили:



6. Вершина стека переносится на созданный первый элемент:

begin = t;

В результате получается следующее:

7. Захват памяти под второй элемент:

t = new Stack;

формируется конкретный адрес (***A2***) для второго элемента.

8. Ввод информации для второго элемента (***i2***);

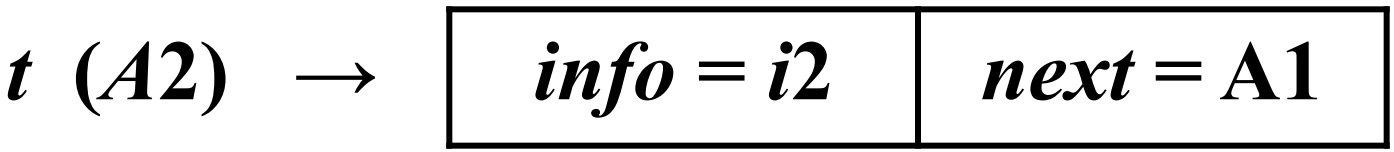
а) формирование информационной части:

t -> info = i2;

б) в адресную часть записываем адрес вершины, т.е. адрес первого (предыдущего) элемента (***A1***):

t -> next = begin;

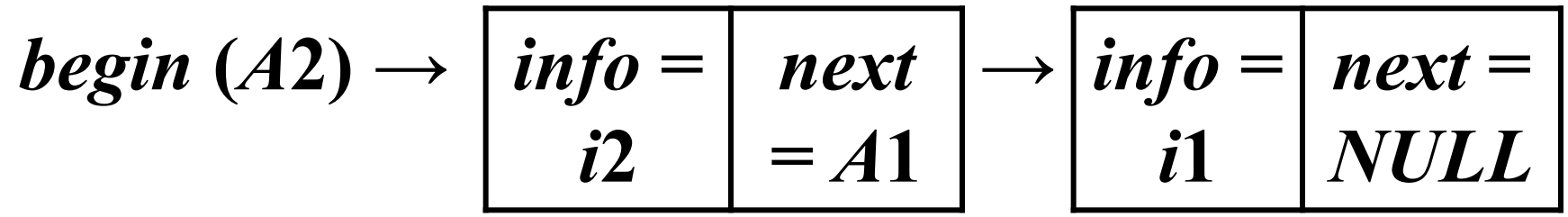
Получаем:



9. Вершина стека снимается с первого и устанавливается на новый элемент (***A2***):

begin = t;

Получается следующая цепочка:




***Обратите внимание***, что действия 7, 8 и 9 идентичны действиям 4, 5 и 6, т.е. добавление новых элементов в стек можно выполнять в цикле, до тех пор, пока это необходимо.

## *Функция формирования элемента стека*

Простейший вид функции (типа *push*), в которую передаются указатель на вершину (***p***) и введенная информация (***in***), а измененное значение вершины (***t***) возвращается в точку вызова оператором *return*:

```
Stack* InStack (Stack  *p, int  in) {
    Stack *t = new Stack; // Захват памяти
    t -> info = in;      // Формируем ИЧ
    t -> next = p;       // Формируем АЧ
    return   t;
}
```

Участок программы с обращением к функции ***InStack*** для добавления ***n*** элементов (используем случайные числа) в стек может иметь следующий вид:

```
    for (i = 1; i <= n; i++)
{
        in = random (20);
        begin = InStack (begin, in);
    }
```

Если в функцию ***InStack*** указатель на вершину передавать по адресу, то она может иметь следующий вид:

```
void InStack (Stack  **p, int  in)
 {
     Stack *t = new Stack;
     t -> info = in;
     t -> next = *p;
     *p = t;
 }
```

Обращение к ней в данном случае будет:

***InStack*** (&begin, in);

## ***Просмотр стека (без извлечения)***

1. Устанавливаем текущий указатель на вершину

   ***t = begin;***

2. Проверяем, если стек пуст (***begin = NULL)***, то выводим сообщение и либо завершаем работу, либо переходим на формирование стека.

3. Если стек не пуст, выполняем цикл до тех пор, пока текущий указатель ***t*** не равен ***NULL***, т.е. пока не обработаем последний элемент, адресная часть которого равна *NULL*.

4. ИЧ текущего элемента ***t -> info*** выводим на экран.
5. Переставляем текущий указатель ***t*** на следующий элемент:

***t = t -> next;***

6. Конец цикла.

Функция, реализующая этот алгоритм:

```
void View (Stack *p) {
    Stack *t = p;
    if ( p == NULL ) {      // Или   if (!p)
        cout << " Стек пуст! " << endl;
        return;
    }
    while( t != NULL) {
        cout << t->info << endl;
        t = t -> next;
    }
}
```

Обращение к этой функции: ***View*** (***begin***);

## *Алгоритм освобождения памяти*

1. Начинаем цикл, выполняющийся пока ***begin*** не станет равным ***NULL***.
2. Устанавливаем текущий указатель на вершину:

   ***t = begin***;
3. Вершину переставляем на следующий элемент:

   ***begin = begin -> next***;
4. Освобождаем память, занятую бывшей вершиной

   ***delete t***;
5. Конец цикла.

Функция, реализующая этот алгоритм, может иметь следующий вид:

```
void  Del_All ( Stack  **p ) {
    Stack *t;
    while ( *p != NULL) {
        t = *p;
        *p = (*p) -> next;
        delete t;
    }
}
```

Параметром данной функции является указатель на указатель, т.е. в функцию передаем адрес указателя на вершину стека для того, чтобы его измененное значение было возвращено из функции в точку вызова.

Обращение к этой функции:

***Del_All*** (&begin);

После ее выполнения указатель на вершину ***begin*** будет равен ***NULL***.

Эту функцию можно реализовать и иначе:

```
Stack*   Del_All ( Stack  *p ) {
      Stack  *t;
      while ( p != NULL) {
          t = p;
          p = p -> next;
          delete  t;
      }
      return   p;
}
```

В этом случае обращение к ней будет:

```
begin = Del_All (begin);
```

В данном случае в функцию передаем указатель на вершину стека, а его измененное значение, равное ***NULL***, возвращаем из функции в точку вызова с помощью оператора ***return***.

## *Алгоритм получения информации из вершины стека с извлечением*

1. Устанавливаем текущий указатель ***t*** на вершину

   t = begin;

2. Сохраняем значение ИЧ ***out*** (выводим на экран)

   out = begin -> info;

3. Переставляем вершину на следующий элемент

   begin = begin -> next;

4. Освобождаем память бывшей вершины ***t***

   delete t;

После этого в переменной ***out*** находится нужное нам значение, а стек стал короче на один элемент.

***Функция*** (типа *pop*), в которую передаются вершина (***p***) и адрес переменной ***out*** для интересующей нас информации, измененное значение вершины (***p***) возвращается в точку вызова оператором *return*:

```
Stack* OutStack (Stack *p, int *out)   {
    Stack *t = p;
    *out = p -> info;
    p = p -> next;
    delete t;
    return p;
}
```

Обращение к этой функции:

begin = ***OutStack*** ( begin, &***out*** );

Необходимая нам информация ***out*** (в нашем примере тип *int*) известна в точке вызова, т.к. передается по адресу.

Если в функцию ***OutStack*** указатель на вершину передавать по адресу, а нужную информацию ***out*** возвращать из функции оператором *return*, то она может иметь следующий вид:

```
int  OutStack ( Stack **p )   {
        int  out ;
        Stack *t = *p;
        out = (*p) -> info;
        *p = (*p) -> next;
        delete t;
        return  out;
   }
```

Обращение к ней в данном случае будет:

```
out = OutStack (&begin);
```

Рассмотрим примеры удаления из стека элементов, кратных 5.

Текст функции удаления непосредственно из стека может иметь следующий вид:

```
Stack* Del_5(Stack *b)
{
    b = InStack (b, 21);     // Добавляем любое число
    Stack *p = b;
    t = p ->next;            // p предыдущий, t текущий
    while (t != NULL) {
            if ( t->info % 5 == 0 ) {
                    p -> next = t -> next;
                    delete t;
                    t = p -> next;
            }
```

```
        else {
                    p = t;
                    t = t -> next;
                }
        }
        t = b;          // Удаление из вершины 21
        b = b -> next;
        delete t;
        return b;
}
```

Обращение к функции: **begin = Del_5 (begin);**

Указатель на вершину стека передаем в функцию, а его измененное значение, возвращаем из функции в точку вызова с помощью оператора ***return***.

Функция удаления из стека элементов, кратных 5, с использованием динамического массива:

```
Stack*  Del_5_mas (Stack  *b)
{
        int  n = 0,  *a,  i,  m;
        Stack  *t = b;
//---------  Расчет количества элементов  n -------
    while (t != NULL) {
         n++;
                t = t -> next;
        }
    cout <<“n=”<<n;
}
```

```
a = new int[n];   // Создаем динамический массив
// Извлекаем все элементы из стека в массив
        for ( i = 0; i < n; i++ )
                b = OutStack ( b, a + i );
/* Удаляем из массива кратные 5, т.е. переписы-
   ваем в массив только те, что не кратны 5  */
        for ( m = i = 0; i < n; i++ )
                if ( a[i] % 5 != 0 )
                        a[m++] = a[i];
// m – количество оставшихся элементов
```

```
/* Создаем стек снова – переписываем в него
   элементы, оставшиеся в массиве: */
      for ( i = 0; i < m; i++ )
            b = InStack ( b, a[i] );
      delete []a;       // Освобождаем память
/*  Возвращаем в точку вызова вершину вновь
   созданного стека  */
      return b;
}
```

Обращение к функции:

```
begin = Del_5_mas ( begin );
```

И в этом случае в функцию передаем указатель на вершину стека, а его измененное значение, возвращаем из функции в точку вызова оператором ***return***.

Функция создания нового стека из элементов уже созданного стека, не кратных 5:

```
Stack*  New_Stack_5 (Stack  *b)
{
      int  inf;
      Stack  *new_b = NULL;
   while (b != NULL) {
       b = OutStack ( b, &inf  );
       if ( inf  % 5 != 0 )
            new_b = InStack ( new_b, inf  );
   }
      return  new_b;
}
```

Обращение к функции:

**begin = New_Stack_5 ( begin );**

***Линейный поиск*** нужной информации в стеке может быть выполнен на базе функции просмотра ***View***.

Например, найдем в стеке количество элементов кратных 5 :

```
int  Poisk (Stack *p)
{
 int  k = 0;
 Stack *t = p;
 while( t != NULL) {
     if ( t -> info % 5 == 0 )
         k ++ ;
     t = t -> next;
 }
 return  k;
}
```

Часть кода с обращением к этой функции:

```
    . . .
int  kol;
    . . .
kol = Poisk (begin);
if (kol == 0)
    Сообщение, что таких НЕТ!
else
    Вывод результата!
    . . .
```

В функцию передаем вершину стека, а в точку вызова возвращаем количество найденных элементов…