# Тема 3. Ссылочные схемы хранения данных

## Списки

Следующее понятие, связанное с линейно-упорядоченными цепочками элементов - это список. Здесь помимо взаимной упорядоченности “следующий-предыдущий” также вводится понятие текущей позиции и возможность добавлять и удалять элементы, а также доступ к значениям элементов разрешается проводить только в окрестности текущей позиции. Самый наглядный пример — строка текста и курсор в текстовом редакторе.

На прошлой лекции на примере стека фактически рассматривался список с возможностью доступа к элементам только с одной стороны (в соответствие со стековой дисциплиной). В понятие же списка как структуры данных также входит понятие текущей позиции, как места, в окрестности которого также можно выполнять добавление и удаление элементов вместе с доступом к значениям некоторых элементов. При этом сама текущая позиция также может перемещаться по списку от текущего элемента к соседнему.

Различают однонаправленный и двунаправленный списки. Это различие состоит в том, к какому соседу разрешено перемещение текущей позиции — только к следующему, или к следующему и к предыдущему.

Символически схему однонаправленного и двунаправленного списков можно изобразить так.

Иллюстрация: схема одно- и двунаправленного списков

Теперь можно попытаться определиться с тем, что разрешается делать с элементами и текущей позицией, и что такое вообще эта текущая позиция.

Конкретные правила изменения списка и доступа к элементам могут слегка различаться, но в целом они подчиняются правилам “разрешаем делать то, что не требует много работы”.

Мы можем принять, что текущая позиция определяется указателем на некоторый элемент. Тогда доступ к значению этого элемента осуществляется непосредственно по этому указателю. И еще мы имеем быстрый доступ к следующему элементу через указатель, хранящийся в текущем элементе. Таким образом, мы можем быстро перенести текущую позицию на следующий элемент и тем самым последовательно пройти по всей оставшейся цепочке списка. Остается только запомнить указатель на первый элемент и договориться как контролировать попадание текущей позиции в конец списка. Добавление и удаление элементов также реализуется после текущей позиции.

Для двунаправленного списка эта идея просто “размножается” на два направления. Соответственно далее будем рассматривать двунаправленный список, делая необходимые замечания, если они потребуются для однонаправленного случая.

Таким образом, в качестве элемента списка можно взять структуру

```
template <class T>
struct ListNode
{
   T value;
   ListNode *next, *prev;
};
```

А вот с указателем текущей позиции возникает множество вопросов.

В качестве текущей позиции мы можем взять конкретный элемент списка и хранить эту позицию в виде указателя на элемент. В этом случае нам быдет непосредственно доступен этот элемент и два его ближайших соседа next и prev.

В таком случае естественно реализуется множество функций для модификации списка в окрестности данной позиции:

— доступ к значению текущего элемента и его соседей;

— добавление нового элемена до или после текущего;
— удаление текущего элемента или какого-то из его соседей;

Дополнительно надо договориться меняется ли текущая позиция после выполнения этих операций. Например, при удалении текущего элемента она обязательно должна измениться.

Следующий вопрос — как должны выполняться эти действия в окрестности начала и конца списка.

Другим вариантом является указание текущей позиции как бы между элементами, т.е. в виде согласованной пары указателей на два соседних элемента. Тогда у нас получается два текущих элемента, и мы можем симметрично (по направлению) задать операции для работы с ними.

Наконец, есть вопрос о том как формировать ссылки на (отсутствующих) соседей по краям списка. Можно задавать их через nullptr, а можно ввести два или даже один фиктивный элемент, который и будет символизировать собой край списка.

Конкретизация каждого из описанных выше подходов приводит к разным реализациям и, ввобще говоря, к разным интерфейсам списка.

Теперь об использовании текущей позиции, которую нам надо как-то предоставить пользователю.

Предположим, что указатель текущей позиции является членом класса список. Такое решение не дает нам возможности перемещать этот указатель для константного списка так как мы не можем менять состояние класса в этом случае.

Мы можем создать такой указатель отдельно от класса и передать его пользователю. Но это не является хорошей идеей так как мы даем прямой доступ к цепочке элемнетов списка извне класса, что может привести к ошибкам и разрушению этой цепочки. Кроме этого независимое перемещение текущей позиции может также привести к некорректному доступу к памяти.

Решением в данном случае является специальная “обертка” вокруг такого указателя, которая реализует концепцию итератора, т.е. специального класса, обеспечивающего доступ к определенным элементам списка и позволяющего безопасно менять эту текущую позицию.

В стандартной библиотеке C++ понятие итератора доведено до высшей степени общности и детализации, и мы в это пока погружаться не будем, а рассмотрим самые простейшие концепции.

Итак, текущая позиция определяется классом итератором. Доступ, добавление и удаление элементов происходит в окрестности этой текущей позиции. Остается определиться как именно это можно реализовать.

Реализуем итератор как внутренний класс для класса списка. Для начала и для простоты рассмотрим однонаправленный список. Итератор показывает на некоторый элемент этого списка.

Итератор позволят:
— получить доступ к значению текущего элемента;
— установить текущую позицию “в начало списка”;
— передвинуть текущую позицию на шаг вперед;
— проверить нахождение в конце списка.

Список позволяет:
— добавить элемент после текущего, текущая позиция не меняется
— добавить после текущего, текущая позиция сдвигается на добавленный эоемент
— удалить после текущего, текущая позиция не меняется
— удалить после текущего, текущая позиция сдвигается на следующий элемент

На деле не все подобные предписания реализовываются. Достаточно, чтобы они обеспечивали некоторую удобную логику работы со списком.

Как пример, может получиться, скажем, такой интерфейс однонаправленного списка.

```
template <class T>   // внешний класс так как может использоваться в разных вариантах
struct ListNode
```

```
{
   T value;
   ListNode *next;
   ListNode(const T& x, ListNode<T> *n);
};

template <class T>
class ForwardList
{
   private:
   ListNode<T> *front;

   public:

   class Iterator
   {
       const ForwardList<T> *list;
       ListNode<T> *pos;
       Iterator(const ForwardList<T> *l, ListNode<T> * p) : list(l), pos(p) {}
    public:
       Iterafor() : list(nullptr), pos(nullptr) {}
       Iterafor(const Iterator &i);           // коструктор копирования
       Iterator operator=(const Iterator &i); // присваивание
       bool operator==(const Iterator &i);    // сравнения
       bool operator!=(const Iterator &i);
       Iterator operator++();                 // перемещение вперед
       Iterator operator++(int);              // перемещение вперед
       T & operator*();                       // доступ к значению текущего элемента
   };
    // специальные позиции для итератора
   Iterator begin() const { return Iterator(this, front); }
   Iterator end() const   { return Iterator(this, nullptr); }

   // конструкторы и деструктор списка
   ForwardList () : first(nullptr) {}
  ~ForwardList();

   // операции
   Iterator InsertAfter(Iterator &i, const T &x); // return - итератор на добавленный
   Iterator EraseAfter(Iterator &i);              // return - итератор на следующий э
   // и т.п.
   // можно добавить отдельные операции по концам
   // очистку в целом, empty ...
};
```

Внимательный анализ этого интерфейса показывает, что он неудачный. Мы не можем добавить новый элемент перед началом непустого списка. Как решить эту проблему? Варианты:

— ввести фиктивный элемент в начале;

— ввести специальный адрес для индикации такого “положения”

**Задача для размышлений.** Можно одновременно использовать несколько разных итераторов по одному и тому же списку. Однако некоторые операции могут конфликтовать друг с другом. Например, если один итератор указывал на некоторый элемент списка, а потом этот элемент был удален с помощью другого итератора. Тогда первый итератор окажется в некорректном состоянии так как будет указывать на несуществующий элемент. Заметим, что вставка элементов не вызывает таких непри-

ятностей. Как решить эту проблему? Вариант — запретить удалять элемент по итератору, если на этот элемент указывают другие итераторы. Как это реализовать?