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. Такм 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.

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.

Выделить такой указатель отдельно от класса и передать его пользователю тоже не является хорошей идеей так как мы даем прямой доступ к цепочке списка извне класса, что может привести к ошибкам и разрушению этой цепочки.

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

Итак, текущая позиция определяется классом итератором. Доступ, добавление и удаление элементов происходит в окрестности этой текущей позиции. Остается определиться как именно осуществляются эти операции.

Потенциально можно предложить несколько вариантов:

— добавить после текущего, текущая позиция не меняется

— добавить после текущего, текущая позиция сдвигается на добавленный эоемент

— удалить после текущего, текущая позиция не меняется

— удалить текущий, текущая позиция сдвигается на следующий элемент

и т.д.

Еще добавляем операции перемещения по списку.

— встать в начало / конец

— перейти к следующему / предыдущему

— проверить в конце или нет

Не обязательно реализовывать все подобные предписания. Достаточно, чтобы они обеспечивали заданную логику работы со списком.

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

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

template <class T>
class SList
{
   private:
   ListNode<T> *first;

   public:
   // конструкторы и деструктор
   SList () : first(nullptr) {}
  ~SList();
```

```
   // и т.д.

   // итератор
   class Iterator
   {
       const SList<T> *list;
       ListNode<T> *pos;
       Iterator(const SList<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
       T & operator*();                       // доступ к значению текущего элемента
   };

   // специальные позиции для итератора
   Iterator begin() const { return Iterator(this, first); }
   Iterator end() const   { return Iterator(this, nullptr); }

   // добавление (вставка) и удаление
   // конкретный смысл этих операций
   // еще требует уточнения и проверки, например,
   bool InsertAt(Iterator &i, const T &x);
          // вставить по позиции итератора
          // итератор указывает на добавленный элемент
   bool RemoveAfter(Iterator &i);
   bool TakeAfter(Iterator &i, T &x);
          // удалить или взять элемент, следующий за
          // позицией итератора, итератор остается на старом месте

   // и т.п.
};
```

Аналогично можно рассмотреть двунаправленный список, в котором текущий элемент основан на классе

```
template <class T>
struct DListNode  // D --- double linked list
{
   T value;
   DListNode * next, * prev;
}
```

И соответтсвенно разрешены перемещения итератора в двух направлениях и вставка/удаление до, после и в текущей позиции итератора.

Пример. DList.zip — файлы с частичным примером двунвправленного списка с итератором.

Кроме собственно цепочки последовательно связанных узлов, в понятие списка также включается концепция текущей позиции как определенного элемента (элементов), к которому разрешен доступ в данный момент времени и в окрестности которого можно выполнять операции добавления и удаления элементов. Кроме этого, также определяются правила перемещения текущей позиции, что дает возможность потенциально получить доступ к каждому элементу списка. Так как перемещения по списку опираются на наличие указателей к следующем или предыдущему узлу, то типичной и естественной операцией является смещение текущей позиции на один шаг в нужную сторону вплоть до достижения крайнего элемента в цепочке. Эти крайние положения текущей позиции обычно описываются словами “начало” или “конец” списка.

Иллюстрация: схема двунаправленного списка, текущая позиция, вставка, удаление элементов

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