Тема 3. Ссылочные схемы хранения данных

Двунаправленный список

Если мы дополнительно в каждом узле будем хранить указатель на предыдущий элемент, то получится схема двунаправленного списка. Соответственно мы можем рассматривать такую узловую структуру

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

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

Формализация дисциплины работы со списком может привести к весьма разным программным решениям. Одним из примеров является реализация к класса list в библиотеке STL C++ (Standard Template Library). Мы с ней познакомимся чуть позже, а пока посмотрим на проблему с позиций “внутренней логики” ссылочной организации хранения данных.

Рассмотрим двунаправленный список.

текущая позиция — некоторый узел списка

можно читать или изменять значение в текущем узле

можно вставить новый узел до или после текущего

можно удалить узел до или после текущего (если они есть)

можно удалить текущий узел с переносом текущей позции на следующий или предыдущий элемент

есть специальные состояния, когда текущая позиция находится “перед первым” или “после последнего” узла списка

Возможна и другая интерпретация текущей позиции как положения между двумя последовательными узлами. При этом у нас в общем случае есть следующий и предыдущий элементы, а позиции в начале или конце характеризуются отсутствием соответствующего соседа. В этом случае дисциплина работы со списком модет выражаться в таких пунктах

Иллюстрация: список с текущей позицией между элементами

Текущая позиция есть положение между двумя соседними элементами

можно читать или изменять значение в следующем и предыдущем узлах текущей позиции

можно вставить новый узел между двумя текущими и сделать его новым следующим либо новым предыдущим

можно удалить следующий или предыдущий узел (если они есть)

состояния "в начале" или "в конце" определяются по отсутствию соответствующего узла в паре узлов текущей позиции.

Попробуем формализовать эти решения в рамках отдельного класса. Узловой элемент сделаем внутренним объектом класса DList.

```
template <class T>
class DList
{
private:
   struct DListNode
   {
      T value;
      DListNode *next, *prev;
      DListNode() { next = prev = 0; }
      void LinkNext(DListNode *p) { next = p; if (p) p->prev = this; }
      void LinkPrev(DListNode *p) { prev = p; if (p) p->next = this; }
   };

   DListNode *currentNode;
   DListNode *first, *last;
   int border;   // признак выхода за края списка 2-начало, 1-конец
   int size;

public:
   // создание, уничтожение
   DList();
   ~DList();

   // добавление
   void InsAfter(const T &x);
   void InsBefore(const T &x);

   void PushFirst(const T &x);
   void PushLast(const T &x);

   // удаление
   void DelAfter();
   void DelBefore();
   void DelCurForAfter();
   void DelCurForBefore();

   // доступ
```

```
   T & Value();
   const T & Value() const;

   // перемещение позиции доступа
   void ToNext();
   void ToPrev();
   void ToFirst();
   void ToLast();

   // проверка состояния
   bool AtBegin() const;   // уже вышли "за границы"
   bool AtEnd() const;
   int  Size();
};
```

Как пример, реализация вставки и удаления элемента (с иллюстрацией)

```
template <class T>
void DList<T>::InsAfter(const T &x)
{
 try {
    if (border == 1) return;
    DListNode *p = new DListNode;
    p->value = x;
    switch(border)
    {
     case 0:
        p->LinkNext(currentNode->next);
        p->LinkPrev(currentNode);
        if (currentNode == last) { last = p; }
        break;
     case 2:
        p->LinkNext(first);
        first = p;
        break;
     case 3:
        first = last = p;
        border = 2;
        break;
    }
    ++size;
 } catch (std::bad_alloc) {
    throw new MyException ("DList<T>::InsAfter: memory allocation error\n", EC_MEMOR
 }
}

template <class T>
void DList<T>::DelAfter()
```

```
{
    DListNode *p;
    switch(border) {
     case 0:
        p = currentNode->next;
        if (p) {
            currentNode->LinkNext(p->next);
            delete p;
            --size;
            if (currentNode->next == 0) {
               last = currentNode;
            }
        }
     case 2:
        p = first;
        first = first->next;
        delete p;
        --size;
        if (first == 0) {
           last = 0;
           border = 3;
        }
        break;
     case 1:
     case 3:
        return;
    }
}
```

Проход по списку:

```
DList<int> a;
....
for (a.ToFisrt(); !a.AtEnd(); a.ToNext())
{
   a.Value()++;
}
```

Пример: как это работает DList_0.h Test-DList_0.cpp

Упражнение: попробуйте реализовать остальные функции.

Недостатки: Текущая позиция непосредственно связана с объектом. Т.е. нельзя на один и тот же список смотреть в нескольких точках.

Пробема !!! Невозможно перемещаться по константному списку.

Например, просто посмотреть и напечатать содержимое.

```
const DList<int> & b = a;
for (b.ToFisrt(), !b.AtEnd(); b.ToNext())
```

```
{
   std::cout << b.Value() << std::endl;
}
```

Можно было бы дать наружу указатель текущей позиции,

```
DList<int>::DListNode * p = b.GetCurrentPositionPtr();  { return curentNode; }
```

но это противоречит идеологии сокрытия данных.

Естественным образом вознкает понятие итератора и const итератора!

Итераторы

Итератор — отдельный класс, который символизирует собой некоторую позицию с множестве элементов (списка) и позволяет последовательно перебирать эти элементы для получения или изменения их значений. При этом для констентных контейнеров допускается использование итератора только на чтение. Обычно перемещение итератора реализуют в виде операций `++` или `--`, а доступ к значению как перегрузку оператора разименования `*` .

Итератор обычно реализуется как внутренний класс, хотя это совсем не обязательно, и хранит в себе указатель на позицию текущего элемента. Для линейно организованных контейнеров типа списков перемещение итераторы выполняется естественным образом. Для более сложных структур (например, деревьев) перемещение итератора задается некоторой логикой обхода всего множества элементов.

```
DList<int> a;
DList<int>::Iterator i;
....
int k;
for (i = a.begin(); i != a.end(); ++i)
{
   k = *i + 10;
   cout << *i << " " << k << endl;
}
```

Т.е. нам надо реализовать операции `++` `--` `*` `!=` `=` для итератора и методы `begin()` и `end()` для класса `DList<T>`.

Внутренность итератора фактически будет повторять работу с текущей позицией currentNode в предыдущей реализации. Ну и еще для удобства (поскольку итератор у нас является отдельным классом) будем в нем сохранять указатель на список, с которым он работает.

```
class Iterator
{
    DList<T> *pList;
    DListNode *currentNode;
    int border;
 public:
```

```
    Iterator ();
    Iterator (const Iterator & i);
    Iterator (DList<T> *p, DListNode *pos, int brdr);

    bool operator!=(const Iterator &i);
    Iterator & operator++();        // это префиксный вариант
    Iterator & operator--();
    T & operator*();
};
```

Только немного изменим понимание border
border == 0 — существующий узел списка
border == 1 — позиция после конца списка
border == 2 — позиция до начала списка
Других значений нет.

Интерпретируется как обозначение конечной позиции при проходе по списку. Т.е. даже для пустого списка значени 1 и 2 имеют различный смысл.

По идее, можно получить несколько итераторов для одного и того же списка и пользоваться ими независимо. При этом, если они изменяют список, но не мешают друг другу, то все должно работать.

Как это реализуется и работает.

Пример: DList_1.h Test-DList_1.cpp