Тема 3. Ссылочные схемы хранения данных

Однонаправленные ссылки, стек

В предыдущих непрерывных реализациях стека, дека, очереди последовательные связи между элементами, выражаемые отношением следующий–предыдущий, естественным образом реализовывались путем размещения в соседних ячейках массива (с поправкой на технический прием кольцевого буфера). В этом случае позиция соседнего элемента могла быть вычислена по позиции текущего элемента.

Однако расплатой за непрерывный способ хранения явилось размещение в массиве и как следствие — ограниченный фиксированный объем памяти для реализации.

Можно отказаться от непрерывного размещения и выделять память для каждого отдельного элемента независимо (пока операционная система нам позволяет). Но тогда возникнет проблема как определять следующий или предыдущий, поскольку их позиции, вообще говоря, непредсказуемы, и мы их не можем вычислить. Ответ прост — раз не можем вычислить, то надо запомнить. Это приводит к понятию ссылочной реализации.

Иллюстрация: ссылочная цепочка из нескольких узловых элементов

В данном случае узловой элемент можно описать, например, такой структурой

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

Иллюстрация: однонаправленная ссылочная цепочка

Реализация: StackList.zip — стек на базе однонаправленной ссылочной реализации
List.h — внешний дружественный класс узла
List_1.h — внутренний приватный класс узла

Итак, что было рассмотрено:
— ссылочные однонаправленные реализации линейных цепочек (на примере стека)
— внешний дружественный класс ListNode
— внутренний приватный класс ListNode
— перегрузка оператора вывода
— частные реализации шаблонных функций (например, Print для int стека)
— понятие typename (при операторе :: разрешения контекста)
Все это проиллюстрировано в коде StackList.zip

Однонаправленный список

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.

Иллюстрация: схема однонаправленного списка

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”. В случае одно направленной цепочки эта идеология приводит к понятию однонаправленного списка. 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.

Потенциально можно предложить два варианта добавления и удаоения.

— 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

Еще добавляем операции перемещения по списку.

— встать в начало

— перейти к следующему

— в конце?

В результате мы можем составить примерное описание однонаправленного списка.

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

template <class T>
class SingleList
{
   private:
   ListNode<T> *start, *pos;

   public:
   // конструкторы и деструктор
   SingleList () : start(nullptr), pos(nullptr) {}
   // и т.д.

   // добавление (вставка) и удаление
   bool InsertNext(const T &x);    // позиция переходит на добавленный
```

```
   bool InsertAfter(const T &x);   // позиция  остается на месте
   bool TakeNext(T &x);            // взять следующий и удалить его
   bool TakeCurrent(T &x);         // взять текущий и удалить его
   bool RemoveNext();
   bool RemoveCurrent();

   // доступ
   T & Current();
   T & Next();

   // перемещение
   void  ToStart();
   bool  ToNext();

   // опросы состояния
   bool  AtEnd();
   // и т.п.
};
```

Здесь записано примерно то, что обсуждали. Но в плане перемещения по списку эта идеология не всегда удобна так как текущая позиция неразрывно связана с самим списком. В современных реализациях позиция доступа обычно выражается в терминах так называемого итератора. Но об этом чуть позже.