# Тема 3. Ссылочные схемы хранения данных

## Лекция 8. Итераторы, продолжение

Мы уже отмечали некоторые проблемы, которые могут возникнуть при итерировании по константному списку. В первую очередь — это запрет изменять значения элементов списка, а также запрет добавлять или удалять элементы из конктантного списка.

Понятно, что от константного объекта мы можем вызывать только константные методы, поэтому добавление и удаление элементов заведомо не являются константными (как методы списка) и тем самым не могут быть вызваны от констрантого объекта. А вот с доступом к значениям и вообще с инициализацией итератора нам надо разобраться.

Нужно просто ввести класс ConstIterator и все сопутствующие указатели объявить const.

**Пример:** `Dlist_2.h` и тест

Мы рассмотрели итератор, реализованный как внутренний класс. С тем же успехом можно рассмотреть реализацию итератора как внешнего класса. По сути разницы нет никакой, разве что тип такого итератора записывается короче.

**Пример:** ExternalIterator.cpp

В некоторых реализациях итераторы также различаются по направлению первичного прохода, т.е. в том, что считается начальным элементом итератор и что есть конечный элемент. В наших реализациях мы введи понятие признака позиции за концом и перед началом. Но в некоторых реализациях это просто включается в само понятие итератора, т.е. для прямого итератора началом (begin()) является первый элемент списка, а end() соответствует положению после последнего элемента. Для обратного итератора началоной позиции становится последний элемент (rbegin()) а концом является позиция перед первым элементом (rend()).

```
Iterator              begin()    end()
ReverseIterator       rbegin()   rend()
ConstantIterator      cbegin()   cend()
ConstantReverseIterator  crbegin()  crend()
```

Можно и дальше развивать понятие итератора, если это служит каким-то практическим целям. Но в нашем варианте мы на этом остановимся, полагая, что общие принципы построения итераторов и примеры, приведенные в курсе, позволят вам понять что собой представляют итераторы в существующих библиотеках контейнерных классов.

В качестве одной из таких библиотек можно привести пример библиотеки STL (Standard Template Library). В ней реализован класс двуззсвязного списка List с определенным набором методов и классы итераторов, которые, в частности, включают следующие

```
// std::list

 #include <list>
```

```
list<int> a;
list<int>::iterator i;
```

— итератор
— добавление в начало и конец
— вставка по позиции итератора
— удаление по позиции итератора
— доступ по позиции итератора
— и многое другое идейно близкое нашему списку

Эти списки и итераторы очень походи на то, что мы разбирали здесь, и поэтому мы не будем на них останавливаться, отсылая слушателей к документации по соответствующим конструкциям библиотеки STL.

## Деревья.

Следующая структура, естественно возникающая в рамках ссылочной реализации, — это дерево.

Определение дерева всем известно* (по крайней мере, на интуитивном уровне). Поэтому условимся о следующих терминах и обозначениях

* Ориентированный связный граф, причем только одна вершина не имеет входящих ребер (корень), а остальные вершины имеют ровно одно входящее ребро.

** Неориентированный связный граф без циклов с одной выделенной вершиной, называемой корнем.

— корень
— потомок и родитель
— концевая вершина (лист) — узел, не имеющий потомков
— ветвь — цепочка "родителей – потомков"от корня до листа

можно также рассматривать восходящую и нисходящую ветви (коренб вверху, лист внизу).

— растояние от корня до верщины — количество вершин в ветви от корня до данного узла
— длина ветви — количество вершин от корня до листа
— уровень — множество вершин, лежащих на одном расстоянии от корня

Произвольное (сильно ветвящееся дерево) — количество потомков конкретного узла может быть любым.

Бинарное дерево — каждая вершина имеет не более двух потомков.

**Иллюстрации:** деревья, термины, ссылки.

Схема ссылочной реализации бинарного дерева. Структура узла

```
template <class T>
struct TreeNode {
    T value;
    TreeNode *left, *right, *parent;
};
```

Схема ссылочной реализации произвольного дерева. Структура узла

```
template <class T>
struct TreeNode {
    T value;
    TreeNode *child, *brother, *parent;
};
```

Лучше их реализовывать внутренним образом так как много разных типов деревьев, и пусть специфические узлы определяются внутри с одинаковыми именами, чтобы не плозить разные имена типов узлов под каждое специфическое дерево.

**Обходы.**

Иллюстрация обхода бинарного и произвольного дерева

Рекурсивная процедура обхода.

Бинарное дерево

```
template <class T>
void Walk ( typename BinTree<T>::TreeNode *root, void (*Process)(T &x) )
{  // top-bottom, left-right
    if (root == 0) return;
    Process(root->value);
    Walk(root->left, Process);
    Walk(root->right, Process);
}
```

Например, поиск узла полным проходом по дереву

```
template <class T>
typename BinTree<T>::TreeNode * TotalSearch
( typename BinTree<T>::TreeNode *root, const T &x )
{
    typename BinTree<T>::TreeNode * p;
    if (root == 0) return 0;
    if (root->value == x) return root;
    p = TotalSearch (root->left, x);
    if (p) return p;
    p = TotalSearch (root->left, x);
    if (p) return p;
    return 0;
}
```

Произвольное дерево

```
template <class T>
void Walk ( typename GenTree<T>::TreeNode *root, void (*Process)(T &x) )
{
    typename GenTree<T>::TreeNode * p;
    if (root == 0) return;
    Process(root->value);
    for (p = root->child; p; p = p->next) {
        Walk(p, Process);
    }
}
```

но если иерархия не важна, то можно по аналогии с бинарным деревом

```
template <class T>
void Walk ( typename Gen<T>::TreeNode *root, void (*Process)(T &x) )
{
    if (root == 0) return;
    Process(root->value);
    Walk(root->child, Process);
    Walk(root->next, Process);
}
```

Обратите внимание, ссылка parent не используется !

Процедуры обходов не являются итератором, поскольку основаны на стеке вложенных вызов и не могут прервать обход, а потом продолжить с того же места.

Итератор по дереву.

Рассмотрим бинарное дерево. Пусть находимся в некоторой вершине. Какая вершина должна стать следующей (++ для итератора) ?

```
p - указатель на текущую вершину
следующая:
   p->left
   если p->left == 0, то
      следующая p->right
          если p->right == 0, то
          надо подняться вверх по ветви,
          пока не обнаружим указатель направо
          и идти на первый же узел справа.
          (различаем подъем слева и справа)
            если и этого нет, то конец

  TreeNode * NextPosition (TreeNode *p)
  {
      TreeNode *q;
      if (p->left) return p->left;
      if (p->right) return p->right;
      if (p->parent) {
          for (q = p->parent; q; p = q, q = q->parent) {
              if (q->right && q->right != p) return q->right;
          }
          return 0;
      } else {
          return 0;
      }
  }

  TreeNode * NextPosition (TreeNode *p)
  {
      TreeNode *q;
```

```
    if (p == 0) return 0;
    if (p->left) return p->left;
    if (p->right) return p->right;
    for (q = p->parent; q; p = q, q = q->parent) {
        if (q->right && q->right != p) return q->right;
    }
    return 0;
}
```