# Отчет №5. Итераторы

Многие алгоритмы обработки списков предполагают, что мы сканируем элементы и попутно совершаем какое-то действие. Производный от List класс предоставляет методы для добавления и удаления данных. В общем случае в нем отсутствуют методы, специально предназначенные для прохождения списка. Подразумевается, что прохождение осуществляет некий внешний процесс, который поддерживает номер текущей записи списка.

В случае массива или объекта L типа SeqList мы можем выполнить прохождение, используя цикл и индекс позиции. Для объекта L типа SeqList доступ к данным класса осуществляется посредством метода GetData.

```
for (pos = 0; pos < ListSize(); pos++)
  cout << L.GetData(pos) << "  ";
```

Для бинарных деревьев, хешированных таблиц и словарей процесс прохождения списка более сложен. Например, прохождение дерева является рекурсивным прямым, обратным или симметричным методами. Эти методы могут быть добавлены в класс обработки бинарных деревьев. Однако рекурсивные функции не позволяют клиенту остановить процесс прохождения, выполнить другую задачу и продолжить итерацию. Итерационное прохождение может быть выполнено путем сохранения указателей на узлы дерева в стеке. Классу деревьев не потребуется содержать итерационную версию для каждого способа прохождения, даже если клиент не может выполнить прохождение дерева или может постоянно использовать один метод прохождения. Предпочтительно отделять абстракцию данных от абстракции управления. Решением проблеммы прохождения списка является создание класса ***итераторов***, задачей которого будет прохождение элементов таких структур данных, как связанные списки или деревья. Итератор инициализируется так, чтобы указывать на начало списка (на голову, корень и тд). У итератора есть методы Next() и EndOfList(), обеспечивающие продвижение по списку. Объект-итератор сохраняет запись состояния итерации между обращениями к Next.

С помощью итератора клиент может приостановить процесс прохождения, проверить содержимое элемента данных, а также выполнить другие задачи. Клиенту дается средство прохождения списка, не требующее сохранения внутренних индексов или указателей. Имея класс, включающий дружественный ему итератор, мы можем связывать с этим классом некоторый подлежащий сканированию объект и обеспечивать доступ к его

элементам через итератор. При реализации методов итератора используется структура внутреннего представления списков.

С помощью виртуальных функций мы объявляем абстрактный базовый класс, используемый в качестве основы для конструирования всех итераторов. Это абстрактный класс предоставляет общий интерфейс для всех операций итератора, несмотря на то что производный итераторы реализуются по-разному.

## Абстрактный базовый класс Iterator

Мы определяем абстрактный класс Iterator как шаблон для итераторов списков общего вида. Каждый из представляемых далее итераторов образован из этого класса, который находится в файле iterator.h.

**Спецификация класса Iterator**

ОБЪЯВЛЕНИЕ

```
template <class T>
class Iterator
{
  protected:
    // флажок, показывающий, достиг ли итератор конца списка.
    // должен поддерживаться производными классами
    int iterationComplete;

  public:
    // конструктор
    Iterator(void);

    // обязательные методы итератора
    virtual void Next(void) = 0;
    virtual void Reset(void) = 0;

    // методы для выборки/модификации данных
    virtual T& Data(void) = 0;

    // проверка конца списка
    virtual int EndOfList(void) const;
};
```

ОБСУЖДЕНИЕ

Итератор является средством прохождения списка. Его основные методы:

- Reset (установка на первый элемент списка)
- Next (установка позиции на следующий элемент)
- EndOfList (обнаружение конца списка).
- Data (доступ к данным текущего элемента списка)

```
// конструктор. устанавливает iterationComplete в 0 (False)
template <class T>
Iterator<T>::Iterator(void): iterationComplete(0)
{}
```

Метод EndOfList просто возвращает значение iteratorComplete. Этот флажок устанавливается в 1 производным методом Reset, если список пуст. Метод Next в производном классе должен устанавливать iterationComplete в 1 при выходе за верхнюю границу списка.

## Образование итераторов списка.

Класс SeqList широко использовался ранее и послужил основой для абстрактного класса List. Ввиду его важности мы начнем с итератора последовательных списков. Этот итератор хранит указатель listPtr, указывающий на сканируемый в данный момент объект типа SeqList. Поскольку SeqListIterator является дружественным по отношению к производному классу SeqList, допускается обращение к закрытым элементам данных класса SeqList.

**Спецификация класса SeqListIterator**

**ОБЪЯВЛЕНИЕ**

```
// SeqListIterator образован от абстрактного класса Iterator
template <class T>
class SeqListIterator: public Iterator<T>
{
  private:
    // локальный указатель на объект SeqList
    SeqList<T> *listPtr;
    // по мере прохода по списку необходимо хранить предыдущую и текущую позицию
    Node<T> *prevPtr, *currPtr;

  public:
    // конструктор
    SeqListIterator (SeqList<T>& lst);

    // обязательные методы прохождения
    virtual void Next(void);
    virtual void Reset(void);

    // методы для выборки/модификации данных
    virtual T& Data(void);

    // установить итератор для прохождения нового списка
    void SetList(SeqList<T>& lst);
}
```

**ОБСУЖДЕНИЕ**

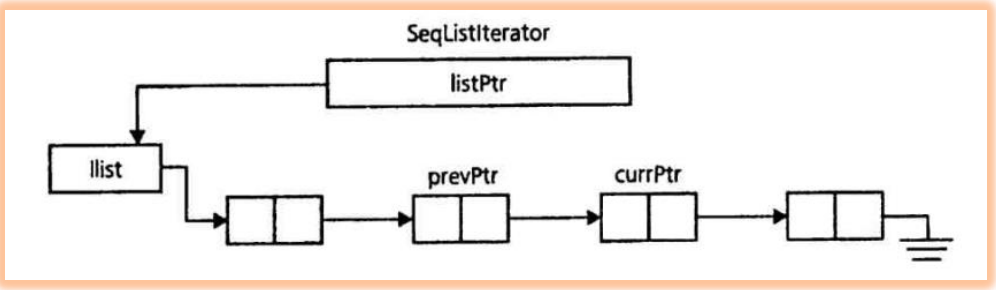
Этот итератор реализует виртуальные функции Next, Reset и Data, которые были объявлены как чистые виртуальные функции в базовом классе Iterator. Метод SeqList является специфичным для класса SeqListIterator и позволяет клиенту присваивать итератор другому объекту типа SeqList. Класс SeqList вместе с итератором находятся в файле seqlist2.h

```
ПРИМЕР
SeqList<int> L;                    // создать список

SeqListIterator<int> iter(L);      // создать итератор и присоединить к списку L
cout << iter.Data();               // распечатать текущее значение данных
iter.Next();                       // перейти на следующую позицию в списке

// цикл, выполняющий проход по списку и распечатывающий его элементы
for (iter.Reset(); !iter.EndOfList(); iter.Next() )
  cout << iter.Data() << "   ";
```

## Построение итератора SeqList

Итератор, создаваемый конструктором, ограничен определенным классом SeqList, и все его операции применимы к последовательному списку. Итератор хранит указатель на объект типа SeqList.

После присоединения итератора к списку мы инициализируем iterationComplete и устанавливаем текущую позицию на первый элемент списка.




```
// конструктор. инициализировать базовый класс и локальный указатель SeqList
template <class T>
SeqListIterator<T>::SeqListIterator(SeqList<T>& lst):
      Iterator<T>(), listPtr(&lst)
{
  // выяснить, пуст ли список
  iterationComplete = listPtr->llist.ListEmpty();
  // позиционировать итератор на начало списка
  Reset();
}
```

**Reset** устанавливает итератор в начальное состояние, инициализируя iterationComplete и устанавливая указатели prevPtr и currPtr на свои позиции в начале списка. Класс SeqListIterator является также дружественным по отношению к классу LinkedList и, следовательно, имеет доступ к члену класса front.

```
// перейти к началу списка
template <class T>
void SeqListIterator<T>::Reset(void)
{
  // переприсвоить состояние итерации
  iterationComplete = listPtr->llist.ListEmpty();

  // вернуться, если список пуст
  if (listPtr->llist.front == NULL)
    return;

  // установить механизм прохождения списка с первого узла
  prevPtr = NULL;
  currPtr = listPtr->llist.front;
}
```

Метод **SetList** является эквивалентом конструктора времени исполнения. Новый объект lst типа SeqList передается в качестве параметра, и теперь итератор идет по списку lst. Переназначьте listPtr и вызовите Reset.

```
// сейчас итератор должен проходить список lst.
// переназначьте listPtr и вызовите Reset.
template <class T>
void SeqListIterator<T>::SetList(SeqList<T>& lst)
{
  listPtr = &lst;

  // инициализировать механизм прохождения для списка lst
  Reset();
}
```

Итератор получает доступ к данным текущего элемента списка с помощью метода **Data**(). Эта функция возвращает значение данных текущего элемента списка, используя currPtr для доступа к данным узла LinkedList. Если список пуст или итератор находится в конце списка, выполнение программы прекращается после выдачи сообщения об ошибке.

```
// возвратить данные, расположенные в текущем элементе списка
template <class T>
void SeqListIterator<T>::Data(void)
{
  // ошибка, если список пуст или прохождение уже завершено
  if (listPtr->llist.ListEmpty() || currPtr == NULL)
  {
    cerr << "Data: недопустимая ссылка!" << endl;
    exit(1);
  }
  return currPtr->data;
}
```

Продвижение от элемента к элементу обеспечивается методом Next. Процесс сканирования продолжается до тех пор, пока текущая позиция не достигнет конца списка. Это событие отражается значением члена

iterationComplete класса Iterarot, который должен поддерживаться функцией Next.

```
// продвинуться к следующему элементу списка
template <class T>
void SeqListIterator<T>::Next(void)
{
  // если currPtr == NULL, мы в конце списка
  if (currPtr == NULL)
    return;

  // передвинуть указатели prevPtr/currPtr на один узел вперед
  prevPtr = currPtr;
  currPtr = currPtr->NextNode();

  // если обнаружен конец связанного списка,
  // установить флажок "итерация завершена"
  if (currPtr == NULL)
    iterationComplete = 1;
```

## Программа 12.4. Использование класса SeqListIterator

Некая компания ежемесячно создает записи SalesPerson, состоящие из личного номера продавца и количества проданного товара. Список salesList содержит накопленные за некоторый отрезок времени записи SalesPerson. Во втором списке, idList, хранятся только личные номера служащих. Из файла sales.dat вводится информация о продажах за несколько месяцев, и каждая запись включается в salesList. Поскольку записи охватывают несколько месяцев, одному продавцу может соответствовать несколько записей. Однако в список idList каждый сотрудник включается только единожды.

После ввода данных соответствующим спискам назначаются итераторы idIter и salesIter. Сканируя список idList, мы идентифицируем каждого служащего по его личному номеру и передаем этот номер в качестве параметра функции PrintTotalSales. Эта функция сканирует список salesList и подсчитывает суммарное количество товара, проданного сотрудником с данным личным номером. В конце распечатывается личный номер служащего и суммарное количество проданного им товара.

```
#include <iostream.h>
#include <fstream.h>

#include "seqlist2.h"



// использовать класс SeqList, наследующий класс List, и SeqListIterator
// запись, содержащая личный номер продавца и количество проданного товара
struct SalesPerson
{
  int idno;
  int units;
};

// оператор == сравнивает служащих по личному номеру
int operator == (const SalesPerson &a, const SalesPerson &b)
{
  return a.idno == b.idno;
}
```

```
// взять id в качестве ключа и пройти список.
// суммировать количество товара, проданное сотрудником с личным номером id
// печатать результат
void PrintTotalSales(SeqList<SalesPerson> & L, int id)
{
  // объявить переменную типа SalesPerson и инициализировать поля записи
  SalesPerson salesP = {id, 0};

  // объявить итератор последовательного списка
  // и использовать его для прохождения списка
  SeqListIterator<SalesPerson> iter(L);

  for(iter.Reset(); !iter.EndOfList(); iter.Next() )
    // если происходит совпадение с id, прибавить количество товара
    if (iter.Data() == salesP)
      sales.P += (iter.Data()).units;

  // печатать личный номер и суммарное количество продаж
  cout << "Служащий " << salesP.idno
       << "   Количество проданного товара " << salesP.units
       << endl;
}

void main(void)
{
  // список, содержащий записи типа SalesPerson,
  // и список личных номеров сотрудников
  SeqList<SalesPerson> SalesList;
  SeqList<int> idList;

  ifstream salesFile;   // Входной файл
  SalesPerson salesP;   // Переменная для ввода
  int i;

  // открыть входной файл
  salesFile.open("sales.dat", ios::in | ios::nocreate);
  if (!salesFile)
  {
    cerr << "Файл sales.dat не найден!";
    exit(1);
  }
  // читать данные в форме "личный номер    количество товара"
  // до конца файла
  while (!salesFile.eof())
  {
    // ввести поля данных и вставить в список salesList
    salesFile >> salesP.idno >> salesP.units;
    salesList.Insert(salesP);
    // если id отсутствует в idList, включить этот id


  if (!idList.Find(salesP.idno))
    idList.Insert(salesP.idno);
}

// создать итераторы для этих двух списков
SeqListIterator<int> idIter(idList);
SeqListIterator<SalesPerson> salesIter(salesList);

// сканировать список личных номеров и передавать каждый номер
// в функцию PrintTotalSales для добавления количества  // проданного това-
ра к общему числу его продаж
for(idIter.Reset(); !idIter.EndOfList(); idIter.Next() )
  PrintTotalSales(salesList, idIter.Data());
}
/*
```

```
<Файл sales.dat>
300     40
100     45
200     20
200     60
100     50
300     10
400     40
200     30
300     10

<Прогон программы 12.4>

Служащий 300    Количество проданного товара 70
Служащий 100    Количество проданного товара 95
Служащий 200    Количество проданного товара 110
Служащий 400    Количество проданного товара 40
*/
```

## Итератор массива

Стремясь привязать итераторы к классам списков, мы, возможно, упустили из виду класс Array. Между тем итератор массивов является весьма полезной абстракцией. Настроив итератор так, чтобы тот начинался и заканчивался на конкретных элементах, можно исключить работу с индексами. Кроме того, один и тот же массив может обрабатываться несколькими итераторами одновременно. Здесь приводится пример использования нескольких итераторов при слиянии двух отсортированных последовательностей, находящихся в одном массиве.

### Спецификация класса ArrayIterator

**ОБЪЯВЛЕНИЕ**

```
#include "iterator.h"

template <class T>
class ArrayIterator: public Iterator<T>
{
  private:
    // начальная, текущая и конечная точки
    int startIndex;

    int currentIndex;
    int finishIndex;

    // адрес объекта типа Array, подлежащего сканированию
    Array<T> *arr;

  public:
    // конструктор
    ArrayIterator(Array<T>& A, int start=0, int finish=-1);

    // стандартные операции итератора, обусловленные базовым классом
    virtual void Next(void);
    virtual void Reset(void);
    virtual T& Data(void);
};
```

**ОБСУЖДЕНИЕ**

Конструктор связывает объект типа Array с итератором и инициализирует начальный и конечный индексы массива. По умолчанию начальный индекс равен 0 (итератор находится на первом элементе массива), а конечный индекс равен -1 (верхней границей массива является индекс последнего элемента). На любом шаге итерации currIndex является индексом текущего элемента массива. Его начальное значение равно startIndex. Класс ArrayIterator находится в файле arriter.h.

Класс ArrayIterator имеет минимальный набор общедоступных функций-членов, подменяющих чистые виртуальные методы базового класса.

**ПРИМЕР**

```
// массив 50 чисел с плавающей точкой от 0 до 49
Array<double> A(50);

// итератор массива сканирует A от 3-го до 10-го индекса
ArrayIterator<double> arriter(Arr, 3, 10);

// печатать массива с 3-го по 10-й элемент
for (arriter.Reset(); !arriter.EndOfList(); arriter.Next() )
  cout << arriter.Data() << " ";
```

## Приложение: слияние сортированных последовательностей

В гл.14 формально изучаются алгоритмы сортировки, включая и внешнюю сортировку слиянием, которая упорядочивает файл данных на диске. Этот алгоритм разделяет список элементов на сортированные подсписки, называемые последовательностями.

**ОПРЕДЕЛЕНИЕ**

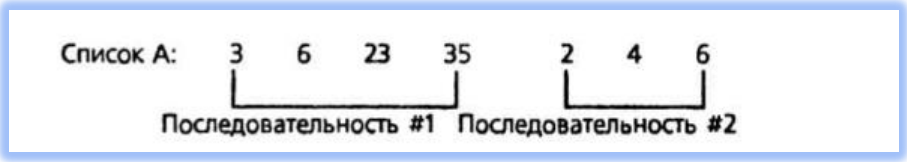
В списке $X_0, X_1, \ldots, X_{n-1}$ последовательностью является подсписок $X_a, X_{a+1}, \ldots, X_b$, где

$X_i \le X_{i+1}$ при $a \le i < b$
$X_{a-1} > X_a$ при $a > 0$
$X_{b+1} < X_b$ при $b < n-1$

Например, подсписок $X_2 \ldots X_5$ есть последовательность в массиве X

X: 20 35 15 25 30 65 50 70 10

В процессе слияния последовательности вкладываются друг в друга, создавая тем самым более длинные упорядоченные подписки до тех пор, пока в результате не получится отсортированный массив.



Это приложение реализует лишь очень ограниченную часть полного алгоритма. Предполагается, что данные хранятся в виде двух последовательностей в N-элементном массиве. Первая последовательность заключена в диапазоне от 0 до R-1, вторая — от R до N-1. Например, в семиэлементном массиве A последовательности разделяются на индексе R = 4.

Поэлементное слияние порождает сортированный список. Текущая точка прохождения устанавливается на начало каждой последовательности. Значения в текущей точке сравниваются, и наименьшее из них копируется в массив. Когда значение в последовательности обработано, выполняется шаг вперед к следующему числу и сравнение продолжается. Поскольку подсписки изначально упорядочены, элементы копируются в выходной массив в сортированном порядке. Когда одна из последовательностей заканчивается, оставшиеся члены другой последовательности копируются в выходной массив.

Этот алгоритм изящно реализуется с помощью трех итераторов: left, right и output. Итератор left проходит первую последовательность, right — вторую, а output используется для записи данных в выходной массив. Пример работы алгоритма показан на рис. 12.2.

## Программа 12.5. Слияние сортированных последовательностей

Функция Merge получает две последовательности, расположенные в массиве A, и сливает их в выходной массив Out. Этот используют итераторы left и right, которые инициализируются параметрами lowIndex, endOfRunIndex и highIndex. Итератор output записывает отсортированные данные в Out. Процесс прекращается по достижении конца одной из последовательностей. Функция Copy дописывает данные, оставшиеся в другой последовательности, в массив Out. После сбрасывания итератора output в начальное состояние отсортированный список копируется обратно в A.

Эта программа вводит 20 целых чисел из файла rundata. В процессе ввода мы сохраняем данные в массиве A и распознаем индекс конца последовательности, который потребуется функции Merge. Функция Merge сортирует массив, который затем распечатывается.

```
#include <iostream.h>
#include <fstream.h>

#include "array.h"
#include "arriter.h"

// копирование одного массива в другой с помощью их итераторов
void Copy(ArrayIterator<int>& Source, ArrayIterator<int>& Dest)
```

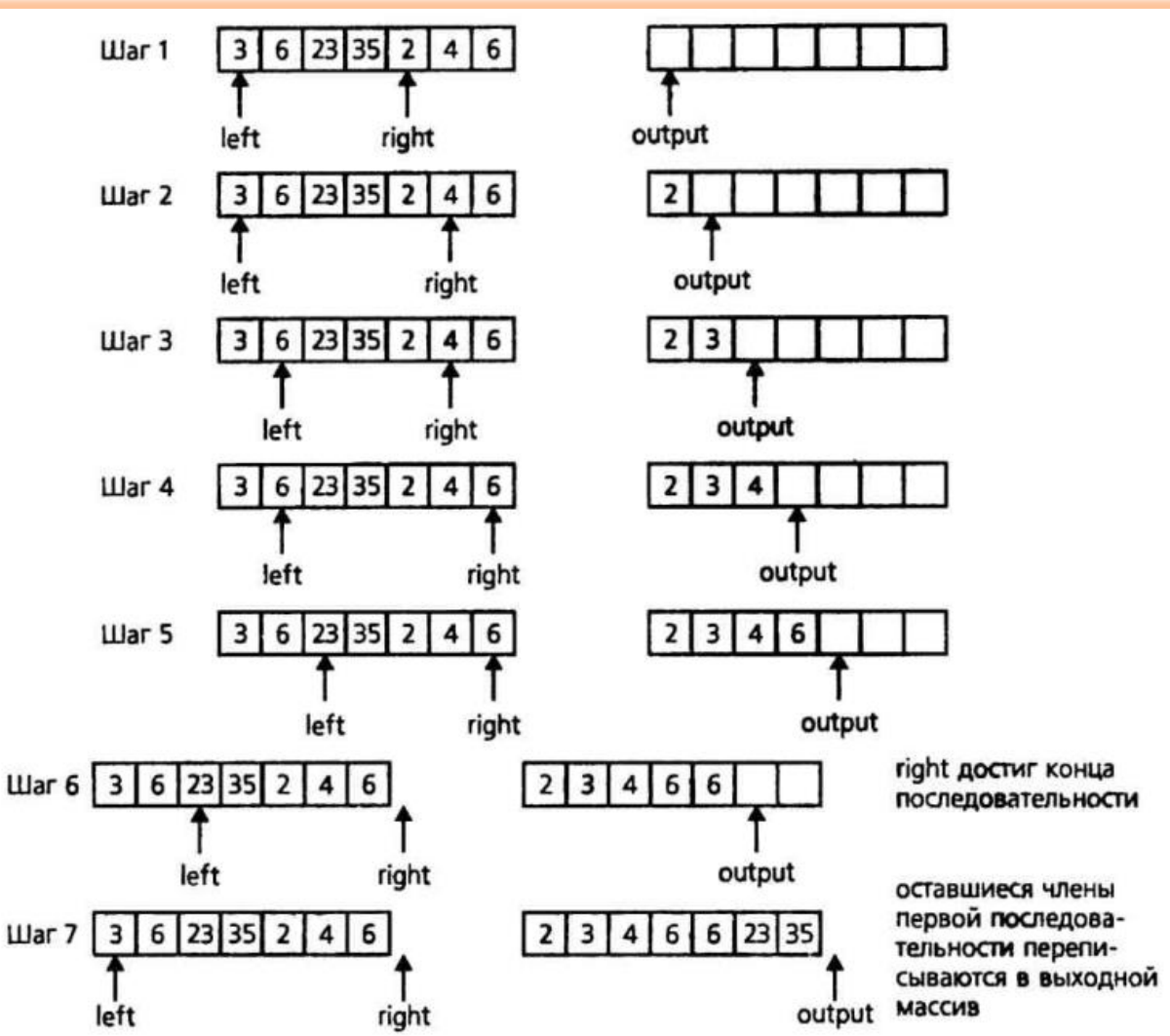



**Рис. 12.2.** Слияние сортированных последовательностей

```
{
  while ( !Source.EndOfList() )
  {
    Dest.Data() = Source.Data();
    Source.Next();
    Dest.Next();
  }
}

// слияние сортированных последовательностей в массиве A.
// первая последовательность заключена в диапазоне индексов
// lowIndex..endOfRunIndex-1, // вторая — в диапазоне endOfRunIndex..highIn-
dex
void Merge(Array<int>& A, int lowIndex, int endOfRunIndex,
           int highIndex)
{
  // массив, в котором объединяются сортированные последовательности
  Array<int> Out(A.ListSize());

  // итератор left сканирует 1-ю последовательность;
  // итератор right сканирует 2-ю последовательность;
  ArrayIterator<int> left(A, lowIndex, endOfRunIndex-1);
  ArrayIterator<int> right(A, endOfRunIndex, highIndex);
```

```
  // итератор output записывает отсортированные данные в Out
  ArrayIterator<int> output(Out);

  // копировать, пока не кончится одна или обе последовательности
  while (!left.EndOfList() && !right.EndOfList())
  {
    // если элемент "левой" последовательности  с итератором left меньше или
    // равен элемент "правой" последовательности, то записать его в массив Out.
    // перейти к следующему элементу "левой" последовательности
    if (left.Data() <= right.Data())
    {
      output.Data() = left.Data();
      left.Next();
    }
    // иначе записать в Out элемент "правой" последовательности
    // и перейти к следующему элементу "правой" последовательности
    else
    {
      output.Data() = right.Data();
      right.Next();
    }
    output.Next();      // продвинуть итератор выходного массива
  }

  // если одна из последовательностей не обработана до конца,
  // скопировать этот остаток в массив Out
  if (!left.EndOfList())
    Copy(left, output);
  else (!right.EndOfList())
    Copy(right, output);

  // сбросить итератор выходного массива и скопировать Out в A
  output.Reset();
  ArrayIterator<int> final(A); // массив для копирования обратно в A
  Copy(output, final);
}
void main(void)
{
  // массив для сортированных последовательностей, введенных из потока fin
  Array<int> A(20);
  ifstream fin;
  int i;
  int endOfRun = 0;

  // открыть файл rundata
  fin.open("rundata", ios::in | ios::nocreate);
  if (!fin)
  {
    cerr << "Нельзя открыть файл rundata", << endl;
    exit(1);
  }

  // читать 20 чисел, представленных в виде двух
  // сортированных последовательностей
  fin >> A[0];

  for (i=1; i<20; i++)
  {
    fin >> A[i];
    if (A[i] < A[i-1])
      endOfRun = i;
  }
```

```
  // слияние последовательностей
  Merge(A, 0, endOfRun, 19);

  // распечатать отсортированный массив по 10 чисел в строке
  for (i=0; i<20; i++)
  {
    cout << A[i] << " ";
    if (i == 9)
      cout << endl;
  }
}
/*
<Файл rundata>

1 3 6 9 12 23 33 45 55 68 88 95
2 8 12 25 33 48 55 75

<Выполнение программы 12.5>

1 2 3 6 8 9 12 12 23 25
33 33 45 48 55 55 68 75 88 95
*/
```

## Реализация класса ArrayIterator

**Конструктор задает начальное состояние итератора. Он привязывает итератор к массиву и инициализирует три индекса. Если для индексов startIndex и finishIndex используются значения по умолчанию (0 и -1), то итератор проходит через весь массив.**

```
// конструктор. инициализирует базовый класс и данные-члены
template <class T>
ArrayIterator<T>::ArrayIterator(Array<T>& A, int start,
                                int finish): arr(&A)
{
  // последний доступный индекс массива
  int ilast = A.ListSize() - 1;

  // инициализировать индексы. если finish == -1,
  // то сканируется весь массив
  currentIndex = startIndex = start;
  finishIndex = finish != -1 ? finish : ilast;

  // индексы должны быть в границах массива
  if (!((startIndex>=0 && startIndex<=ilast) &&
        (finishIndex>=0 && finishIndex<=ilast) &&
        (startIndex <= finishIndex)))
  {
    cerr << "ArrayIterator: Неверные параметры индекса!"
         << endl;
    exit(1);
  }
}
```

**Reset переустанавливает текущий индекс на стартовую точку и обнуляет iterationComplete, показывая тем самым, что начался новый процесс прохождения.**

```
// сброс итератора массива
template <class T>
void ArrayIterator<T>::Reset(void)

{
  // установить текущий индекс на начало массива
  currentIndex = startIndex;

  // итерация еще не завершена
  iterationComplete = 0;
}
```

**Метод Data использует currentIndex для доступа к данным-членам. Если текущая точка прохождения заходит за верхнюю границу списка, генерируется сообщение об ошибке и программа прекращается.**

```
// возвратить значение текущего элемента массива
template <class T>
T& ArrayIterator<T>::Data(void)
{
  // если весь массив пройден, то вызов метода невозможен
  if (iterationComplete)
  {
    cerr << "Итератор прошел весь список до конца!"
         << endl;
    exit(1);
  }

  return (*arr) [currentIndex];
}
```

Если итерация завершается, метод Next просто возвращает управление. В противном случае он увеличивает currentIndex и обновляет логическую переменную базового класса iterationComplete.

```
// перейти к следующему элементу массива
template <class T>
void ArrayIterator<T>::Next (void)
{
  // если итерация не завершена, увеличить currentIndex
  // если пройден finishIndex, то итерация завершена
  if (!iterationComplete)
  {
    currentIndex++;
    if (currentIndex > finishIndex)
      iterationComplete = 1;
  }
}
```

# 12.6. Упорядоченные списки

Класс SeqList создает список, элементы которого добавляются в хвост. В результате получается неупорядоченный список. Однако во многих приложениях требуется списковая структура с таким условием включения, при котором элементы запоминаются в некотором порядке. В этом случае приложение сможет эффективно определять наличие того или иного элемента в списке, а также выводить элементы в виде отсортированных последовательностей.

Чтобы создать упорядоченный список, мы используем класс SeqList в качестве базового и образуем на его основе класс OrderedList, который вставляет элементы в возрастающем порядке с помощью оператора "<". Это пример наследования в действии. Мы переопределяем только метод Insert, поскольку все другие операции не влияют на упорядочение и могут быть унаследованы от базового класса.

## Спецификация класса OrderedList

**ОБЪЯВЛЕНИЕ**

```
#include "seqlist2.h"

template <class T>
class OrderedList: public SeqList<T>
{
  public:
    // конструктор
    OrderedList(void);

    // подменить метод Insert для формирования упорядоченного списка
    virtual void Insert (const T& item);
};
```

**ОПИСАНИЕ**

Все операции, за исключением Insert, взяты из SeqList, так как они не влияют на упорядочение. Поэтому должен быть объявлен только метод Insert, чтобы подменить одноименный метод из SeqList. Эта функция сканирует список и включает в него элементы, сохраняя порядок.

Класс OrderedList находится в файле ordlist.h.

## Реализация класса OrderedList

В классе OrderedList определяется конструктор, который просто вызывает конструктор класса SeqList. Тем самым инициализируется этот базовый класс, а он в свою очередь инициализирует свой базовый класс List. Мы имем пример трех-классовой иерархической цепи.

```
// конструктор. инициализировать базовый класс
template <class T>
OrderedList::OrderedList(void): SeqList<T>()
{}
```

В этом классе определяется новая функция Insert, которая включает элементы в подходящее место списка. Новый метод Insert использует встроенный в класс LinkedList механизм поиска первого элемента, большего, чем включаемый элемент. Метод InsertAt используется для включения в связанный список нового узла в текущем месте. Если новое значение больше, чем все имеющиеся, оно дописывается в хвост списка. Метод Insert отвечает за обновление переменной size, определенной в базовом классе List.

```
// вставить элемент в список в возрастающем порядке
template <class T>
void OrderedList::Insert(const T& item)
{
  // использовать механизм прохождения связанных списков
  // для обнаружения места вставки
  for( llist.Reset(); !llist.EndOfList(); llist.Next() )
    if (item < llist.Data())
      break;

  // вставить item в текущем месте
  llist.InsertAt(item);
  size++;
}
```

**Приложение: длинные последовательности.** В программе 12.5 описана часть алгоритма сортировки слиянием, который включал слияние двух сортированных последовательностей в одну, тоже сортированную. В программе предполагалось, что ваши входные данные уже заранее разбиты на две последовательности. Сейчас мы обсудим методику фильтрации (предварительной обработки) данных для получения более длинных последовательностей.

Предположим, что большой блок данных хранится в случайном порядке в массиве или на диске. Тогда эти данные можно представить в виде ряда коротких последовательностей. Например, следующее множество из 15-и символов состоит из восьми последовательностей.

```
CharArray: [a k] [g] [c m t] [e n] [l] [c r s] [c b f]
```

Попытка использовать сортировку слиянием для упорядочения этих данных была бы тщетной ввиду значительного числа коротких последовательностей, подлежащих объединению. В нашем примере четыре слияния дают следующие последовательности.

```
[a g k] [c e m t] [c l r s] [b c f]
```

Сортировка слиянием предписывает объединить на следующем проходе эти четыре последовательности в две и затем создать полностью отсортированный список. Алгоритм работал бы лучше, если бы изначально последовательности имели разумную длину. Этого можно достичь путем сканирования элементов и объединения их в сортированные подсписки. Алгоритм внешней сортировки должен противостоять относительно медленному времени доступа к диску и часто включает в себя фильтр для предварительной обработки данных. Мы должны постараться, чтобы время, затраченное на фильтрацию данных, повышало бы общую эффективность алгоритма.

Упорядоченный список является примером простого фильтра. Предположим, что исходный массив или файл содержит N элементов. Мы вставляем каждую группу из k элементов в некоторый упорядоченный список, а затем копируем этот список обратно в массив. Этот фильтр гарантирует, что последовательности будут иметь длину, по крайней мере, k. Например, пусть k=5, и мы обрабатываем данные массива CharArray. Тогда результат будет таким:

```
[a c g k m]  [c e l n t]  [b c f r s]
```

## Программа 12.6. Длинные последовательности

Эта программа фильтрует массив 100 случайных целых чисел в диапазоне от 100 до 999 в последовательности, по крайней мере, из 25 элементов, используя упорядоченный список. Каждой новое случайное число вставляется в объект L типа OrderedList. Для каждых 25 элементов функция Copy удаляет эти элементы из списка L и вставляет их обратно в массив A. Программа заканчивается печатью результирующего массива A.

```
#include <iostream.h>

#include "ordlist.h"
#include "array.h"
#include "arriter.h"
#include "random.h"

// пройти целочисленный массив и распечатать каждый элемент
// по 10 чисел в строке

void PrintList(Array<int>& A)
{
  // использовать итератор массива
  ArrayIterator<int> iter(A);
  int count;

  // прохождение и печать списка
  count = 1;
  for(iter.Reset(); !iter.EndOfList(); iter.Next(), count++)
  {
    cout << iter.Data() << "  ";
    // печатать по 10 чисел в строке
    if (count % 10 == 0)
      cout << endl;
  }
}

// удалять элементы из упорядоченного списка L и вставлять их в массив A.
// обновить loadIndex, указывающий следующий индекс в A
void Copy(OrderedList<int> &L, Array<int> &A, int &loadIndex)
{
  while (!L.ListEmpty())
    A[loadIndex++] = L.DeleteFront();
}
```

```
void main(void)
{
  // создать последовательности в A с помощью упорядоченного списка L
  Array<int> A(100);
  OrderedList<int> L;

  // генератор случайных чисел
  RandomNumber rnd;

  int i, loadIndex = 0;

  // сгенерировать 100 случайных чисел в диапазоне от 100 до 999.
  // отфильтровать их через 25-элементный упорядоченный список.
  // после заполнения списка копировать его в массив A
  for (i=1; i<=100; i++)
  {
    L.Insert(rnd.Random(900) + 100);
    if (i % 25 == 0)
      Copy(L, A, loadIndex);
  }
  // печатать итоговый массив A
  PrintList(A);
}
```

```
/*
<Выполнение программы 12.6>

110  116  149  152  162  240  345  370  422  492
500  532  578  601  715  730  732  754  815  833
850  903  929  947  958  105  132  139  139  190
205  216  221  243  287  348  350  445  466  507
513  524  604  634  641  730  784  940  969  982
296  375  412  437  457  466  507  550  594  652
725  728  771  799  803  815  859  879  909  915
940  990  991  992  994  101  118  123  155  310
343  368  372  434  443  489  515  529  557  574
641  739  774  784  829  875  883  922  967  972
*/
```

Источники:


1. https://metanit.com/cpp/tutorial/7.3.php
2. У. Топп, У. Форд – структуры данных в c++