Тема 2. Непрерывные схемы хранения данных

Динамический массив

Данные, с которыми приходится работать, часто бывают определенным образом организованы и структурированы. Это означает, что существует определенная дисциплина использования и доступа к этим данным. К настоящему времени сложилось несколько основных схем представления данных, которые отражают эту структуру и правила использования.

Подобные схемы решают задачи хранения данных в условях появления или удаления этих данных и поддерживают определенные правила доступа к этим данным.

Каждая схема в итоге реализуется как некоторый объект, который хранит данные и поддерживает требуемую дисциплину доступа к этим данным. Иногда такие объекты называют контейнерами. Далее мы предполагаем, что элементы занных, с которыми нам надо работать, являются однородными по типу, в частности, не меняют свой размер в процессе работы. Например, это числа определенного типа (целые, вещественные).

Таким образом, контейнер должен реализовать 4 категории действий в соответствии с требуемой дисциплиной:

1. создание и уничтожение контейнера
2. добавление и удаление данных в контейнер
3. доступ к элементам, хранящимся в контейнере
4. опрос состояний контейнера

Кроме этого он должен обеспечить

5. реакцию на попытки некорректного использования

Простейший вариант контейнера - это массив.
С обычным массивом фиксированной длины все просто, и легко:

1. malloc/free или new/delete
2. не реализуется
3. прямой доступ по индексу
4. не требуется
5. контроль выхода индекса за границы массива

Более интересен пример, когда массив может изменять свой размер, т.е. мы имеем право добавлять и удалять элементы.

Такой массив в каждый момент времени имеет определенный размер, поэтому пп. 1, 3, 5 остаются те же, а вот пп. 2 и 4 требуют разработки. Т.е. нам надо определиться как трактовать возможность добавления и удаления элементов.

1. массив создается на некоторый заданный начальный размер
2. можно добавлять в конец или вставлять в середину можно удалять по индексу или отрезать хвост
3. прямой доступ по индексу
4. узнать текущую длину (или проверить индекс на корректность)
5. вызывать исключение при выходе индекса за границу

Теперь это надо перевести в формальное определение класса. Но надо определиться с типом данных элемента массива. Пусть для простоты это вначале будет int.

```
------ файл IntArray.h

class IntArray
{
 private:
    int *value;
    int size;
 public:
    IntArray (int _size) { value = new int[_size]; size = _size; }
    IntArray (const IntArray & other);
    ~IntArray() { delete [] value; }

    void Add(const int & x);       // push_back
    void Remove(int i);
    void Insert(int i, const int & x);

    int & operator[](int i) { return value[i]; }
    const int & operator[] (int i) const  { return value[i]; }


    int Size() const {return size;}
};
------- файл с использованием этого массива

    IntArray a(10);
    a[1] = a[2] = 123;
    a.Add(456);
    a.Add(789);
    a[3] = a[10] + a[11];
    a[15] = 0;  // !!! исключение !!!
    a.Remove(1);
    a[11] = 0;  // !!! исключение !!!
    int n = a.Size();
```

Что плохо?
— это надо теперь реализовывать и по возможности жффективно
— это только для int, а как быть с другими типами
— как аккуратно внедрить исключения?
Как надо делать:
— аккуратное управление памятью для хранения массива (запас, перемещения)
— template реализация
— сделать отдельный класс исключения для массива

```
----- файл MyException.h

#pragma once
#inclde <string.h>

enum ErrorCode
{
   EC_RANGE = -3,
   EC_MEMORY = -2,
   EC_UNKNOWN = -1,
   EC_OK = 0
};

class MyException
{
 private:
    char message[256];
    int code;
 public:
    MyException(const char * msg, int _code = EC_UNKNOWN) {
      code = _code;
      strncpy(message, msg, 255);
    }
    int Code() const { return code; }
    const char * Message() const { return message; }
};
\енв{verbatim}

\begin{verbatim}
----- файл Array.h
#pragma once
#include <stdio.h>
#include <new>

template <class T>
class Array
{
 private:
    T * value;
```

```
    int size;
    int maxsize;
 public:
    Array(int _size);
    Array(int _sise, int _maxs);
    Array(const Array<T> & other);
    ~Array() { delete [] value; }

void SetSize(int _size, int _maxs);

    void Add(const T &x);
    void Insert(int i, const T &x);
    void Remove(int i);
                                                //     Array<int> a(10);
    T & operator[](int i);                      //     a[i]=1;     x = a[i];
    const T & operator[](int i) const;          //     const Array<int> & b = a;
                                                //     x = b[i];  // b[i] = 1;
    int Size()  const {return size;}
    bool TestInd(int i) const { return (0<=i && i<size) ? true : false; }
};

template <class T>
void Array<T>::SetSize(int _size, int _maxs)
{
    if (_size > _maxs) _maxs = _size;
 try {
       maxsize = _maxs;
       size = _size;
       value = new T[maxsize];
    } catch (std::bad_alloc e) {
       char msg[64];
       sprinf(msg, "Array::SetSize: cannot allocate size %d\n", maxsize*sizeof(T));
       throw new MyException(msg, EC_MEMORY);
    }
}

template <class T>
Array<T>::Array(int _size)
{
   SetSize(_size, _size);
}

template <class T>
Array<T>::Array(int _size, int _maxs)
{
   SetSize(_size, _maxs);
}
```

```
template <class T>
void Array<T>::Add(const T &x)
{
 try {
   if (size == maxsize) {
       maxsize = 2*size; //  совсем примитивный вариант !!!
       T * new_value = new T[maxsize];
       for (int i=0; i<size; i++) { new_value[i] = value[i]; }
       delete [] value;
       value = new_value;
   }
   value[size++] = x;
 } catch (std::bad_alloc e) {
       char msg[64];
       sprinf(msg, "Array::Add: cannot allocate size %d\n", maxsize*sizeof(T));
       throw new MyException(msg, EC_MEMORY);
 }
}

//#define __DEBUG
template <class T>
void Array<T>::Remove(int i)
{
#ifdef __DEBUG
    if (!TestInd(i)) {
        char msg[64];
        sprintf(msg,"Array::Remove: bad index i=%d size=%d\n", i, size);
        throw new MyException(msg, EC_INDEX)
    }
#endif
    for ( ; i<size - 1; i++) value[i] = value[i+1];  // не всегда лучший вариант
    --size;
}                                   // memcpy   memmove

template <class T>
void Array<T>::Insert(int i, const T & x)
{
  проверка индекса на попадание в границы массива
  try блок на выделение памяти
  проверка достаточно ли размера maxsize для расширения массива
     если памяти недостаточно, то выделить новый массив
     и переместить в него элементы с индесами от 0 до i-1
  сдвинуть хвост массива от i до size-1 на одну позицию право
  value[i] = x;
}
```

```
template <class T>
T & Array<T>::operator[](int i)
{
   проверка индекса
   return value[i];
}

template <class T>
const T & Array<T>::operator[](int i) const
{
   проверка индекса
   return value[i];
}
```

Использование:

```
  try{
    Array<int> a(20), b(10,100);
    Array<double> r(200,220);
    a[2] = b[9];
    a.Add(25);
    b.Remove(3);
    b.Insert(5, a[20]);
  } catch (MyException *e) {
      printf("My exception: %s  code %d\n", e->Message(), e->Code());
      if (e->Code() == EC_MEMORY) ....
      if (e->Code() == EC_INDEX) ....
      delete e;
      exit(-1); // или какое другое действие ...
  }
```

Заключительные комментарии.

Хитрая процедура изменения размера — (геометрическая, потом арифметическая прогрессии).

Отключение контроля индекса — на этапе компиляции, разные методы, переключение режима.

Дополнительные методы - обрезка хвоста, поиск, сортировки и т.д.

Специальные реализации сдвига (swap).