Тема 2. Структуры данных. Непрерывные схемы хранения.

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

Динамический массив

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, в начало 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. Если мы хотим делать универсальную "библиотечную" реализацию, то не можем ориентироваться на какой-либо конкретный тип, поскольку нам могут потребоваться массивы разных типов. Поэтому сразу будем рассматривать реализацию с помощбю шаблонов template.

Еще один аспект — реакция на некорректные ситуации. Один из вариантов — сделать некоторый общий класс исключения, который будет в себе сохранять и передавать дополнительную информацию о произошедшем отказе. Тогда, выбрасывая и потом перехватывая это исключение, мы получаем возможность выяснить в какой ситуации возник отказ и более эффективно исправлять сопутсвующие ошибки.

Для определенности начнем сразу с этого класса-исключения. По сути этот класс просто хранит условный код возникшей исключительной ситуации и наше информационное сообщение по этому поводу

```
//  файл MyException.h

#pragma once              // для однократного включения этого файла
#include <cstring>        // для функций формирования C-строк

enum ErrorCode            // список условных кодов отказов
{
   EC_RANGE = -3,
   EC_MEMORY = -2,
   EC_UNKNOWN = -1,
   EC_OK = 0
};

class MyException
{
   char message[256];
   ErrorCode  code;
 public:
   MyException(ErrorCode c, const char *msg) {
      code = c;
      strncpy(message, msg, 255);
   }
   const char * Message() const { return message; }
   ErrorCode   Code() const { return code; }
};
```

В реальной практике подобные классы могут получать и сохранять указание на место в коде (файл, номер строки), где возник отказ и также иметь способность более развернутого формирования строки сообщения с включением разных значений переменных, имеющих отношение к текущей ситуации.

Задачки для знатоков. Как добиться следующей функциональности от класса MyException?

1. Нужно иметь возможность при вызове конструктора сразу формировать информационное сообщение с включением значений некоторого (переменного) количества параметров. Например, по аналогии с функцией printf или выводом в поток std::cout :

```
MyException(EC_MEMORY, "ошибка памяти size=%u maxsize=%u\n", size, maxsize);
или
MyException(EC_MEMORY, "ошибка памяти size=", size, " maxsize=", maxsize, std::endl);
```

2. При выполнении п.1 нужно обеспечить “безопасность”, т.е. не допустить переполнения строк msg и любых других.

Для последующей релизации введем отдельную упрощенную функцию формирования такой строки — ArrangeMessage.

Теперь перейдем к объявлению и реализации массива, назвем этот класс Array.

```
// файл Array.h

#pragma once
#include <cstdio>
#include <new>

#define  CHECK_IND  // это чтобы включать и выключать проверку индексов

template <class T>
class Array
{
 private:
    T * value;       // обычный массив значений
    size_t size;     // текуший размер
    size_t maxsize;  // максимально доступный размер
    const char * ArrangeMessage(const char * text, size_t s, size_t ms);
    bool TestInd(size_t i) const { return i < size; }
 public:
    Array(size_t s = 0) : value(nullptr) { SetSize(s, s); }
    Array(size_t s, size_t ms) : value(nullptr) { SetSize(s, ms); };
    Array(const Array<T> & other);
    ~Array() { delete [] value; }

    void SetSize(size_t s, size_t ms);  // изменить размер массива

    void Add(const T &x);               // push_back
    void Insert(size_t i, const T &x);
```

```
    void Remove(size_t i);

    T & operator[] (size_t i);              // lvalue
    const T & operator[](size_t i) const;   // rvalue;

    size_t Size()  const {return size;}
    size_t MaxSize()  const {return maxsize;}    // capacity
};
```

Так как реализация тоже является шаблоном, то она должна быть видна везде, где может использоваться. Поэтому мы ее помещаем в тот же файл Array.h

```
// файл Array.h, продолжение

template <class T>
void Array<T>::SetSize(size_t s, size_t ms)
{
    if (s > ms) {
       throw new MyException(EC_RANGE,
          ArrangeMessage("SetSize: size > maxsize ", s, ms));
    }
    if (value) { delete [] value; }  // примитивная логика изменения размера
                                     // все старое уничтожается
try {
       value = new T[ms];
       maxsize = ms;
       size = s;
    } catch (std::bad_alloc e) {
       throw new MyException(EC_MEMORY,
          ArrangeMessage("SetSize: allocation error: size, maxsize", s, ms));
    }
}

// конструктор копирования --- копирует все элементы массива
template <class T>
Array<T>::Array(const Array<T> & other)
{
    value = nullptr;
    SetSize(other.size, other.maxsize);
    for (size_t i = 0; i < size; ++i) {
        value[i] = other[i];
    }
}

// для добавления надо иногда увеличивать массив
// сначала удваиваем длину, а после определеного
// предела ее увеличиваем на заданную константу
```

```
#define  DBL_LENGTH_MAX   4096  // это так, с потолка взято

template <class T>
void Array<T>::Add(const T &x)
{
 try {
   if (size == maxsize) {
       maxsize = (maxsize < DBL_LENGTH_MAX) ? 2*maxsize : maxsize + DBL_LENGTH_MAX;
       T * new_value = new T[maxsize];
       for (size_t i = 0; i < size; ++i) { new_value[i] = value[i]; }
       delete [] value;
       value = new_value;
   }
   value[size++] = x;
 } catch (std::bad_alloc e) {
       throw new MyException(EC_MEMORY,
          ArrangeMessage("Add: allocation error: size, maxsize", size, maxsize));
 }
}

template <class T>
void Array<T>::Remove(size_t i)
{
#ifdef CHECK_IND
    if (!TestInd(i)) {
       throw new MyException(EC_RANGE,
          ArrangeMessage("Remove: bad index: ind, size ", i, size));
    }
#endif
    for ( ; i < size - 1; i++) value[i] = value[i+1];  // не всегда лучший вариант
    --size;
}

template <class T>
void Array<T>::Insert(size_t 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[](size_t i)
```

```
{
#ifdef CHECK_IND
    if (!TestInd(i)) {
       throw new MyException(EC_RANGE,
          ArrangeMessage("[]: bad index: ind, size ", i, size));
    }
#endif
   return value[i];
}

template <class T>
const T & Array<T>::operator[](size_t i) const
{
#ifdef CHECK_IND
    if (!TestInd(i)) {
       throw new MyException(EC_RANGE,
          ArrangeMessage("[] const: bad index: ind, size ", i, size));
    }
#endif
   return value[i];
}
```

Пример использования и некоторые тесты см. в файле array.zip.

Заключительные комментарии.

Включение или отключение контроля индекса здесь идет на этапе компиляции (можно закомментировать define). Можно это сделать внутренним параметром класса и включать/выключать прямо в коде в любых вариантах.

Можно сокращать массив при удалении элемента. Можно сделать изменение размеров с копированием старых значений элементов (по каким правилам?).

Можно навешивать дополнительные методы — поиск, сортировку, разные преобразования и т.п. Но лучше это сделать с помощью отдельных (дружественных) функций, предоставив им непосредственный доступ к массиву value. Например, можно ввести в класс функцию

```
const T * Data() const { return values; }
или (с осторожностью!)
T * Data() { return values; }
или
friend void Sort(Array<T> & a, int (*cmp)(const T&, const T&) = nullptr);
```

где эта функция получает непосредственный доступ к private value для сортировки по заданной функции сравнения элементов, либо по их оператору $<$ (если не задана) (в C++ можно это сделать и по-другому, но об этом не сейчас ...)