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, в общем случае 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

Обычный массив

Массив является традиционной и простейшей формой организации хранения набора однотипных объектов. При этом основой доступа к различным элементам этой коллекции является нумерация — индексирование. Все алгоритмические языки реализуют массивы на уровне базового синтаксиса. Относительно C и C++ есть два основных подхода — массив заранее известной фиксированной (константной) длины и массив, который создается по запросу на заданную длину. Во втором случае нам приходится заботиться об освобождении захваченного для массива ресурса — памяти. Ошибки при освобождении памяти могут привести к разнообразным тяжелым и трудно обнаруживаемым ошибкам в программе. Второй традиционный дефект — это выход индекса за границы массива, что также может привести к аварийному завершению или к неверному ответу из-за использования значений памяти, не относящейся к области размещения массива. Попробуем отчасти решить эту проблему с использованием идеологии RAII.

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

Заметим, что стандартная библиотека C++ включает класс std::array, который представляет собой обертку над обычным C-массивом константной длины. Однако она не позволяет создавать такой массив по запросу на неизвестный во время компиляции размер.

Постановка задачи: реализовать работу с массивом элементов произвольного типа с возможностью задавать размер массива в момент его создания и автоматическим освобождением памяти, когда управление выходит из области видимости этого массива. Из некоторых соображений (см. далее) назовем такой класс Wvector_base.

```
template <class T>
class Wvector_base
{
    T * value;
    size_t size;
public:
    Wvector_base(size_t n) : size(n), value(nullptr) { value = new T[size]; }
    Wvector_base(const Wvector_base<T> &v) : size(v.size), value(nullptr) {
       value = new T[size];
       for (size_t i = 0; i < size; ++i) { value[i] = v.value[i]; }
    }
    ~Wvector_base() { delete [] value; }
    size_t Size() const { return size; }
    T & operator[] (size_t i) { return value[i]; }
    const T & operator[] (size_t i) const { return value[i]; }
};
```

Эта реализация работает, но не защищает от плохих индексов. Исправим.

```
#include <exception>

    T & operator[] (size_t i) {
        if (i >= size ) throw std::range_error("Bad lvalue vector index\n");
        return value[i];
    }
    const T & operator[] (size_t i) const {
        if (i >= size ) throw std::range_error("Bad rvalue vector index\n");
        return value[i];
    }
```

Проверим как работает

```
int main()
{
    Wvector_base<int> a(10);
    a[0] = 1;
    a[1] = 7;
    a[2] = a[1] + a[0];
    std::cout << a[2] << std::endl;

    try {
        a[12] = 123;
    } catch (std::exception &e) {
        std::cout << e.what() << std::endl;
    }

    const Wvector_base<int> b(a);
    std::cout << b[1] << std::endl;

    try {
        int x;
        x = b[12];
    } catch (std::exception &e) {
        std::cout << e.what() << std::endl;
    }
    return 0;
}
```

Динамический массив

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. Для начала можем принять такую дисциплину:

1. добавить элемент в конец (push_back);
2. вставить элемент по индексу (insert);
3. удалить элемен по индексу (erase).

Можно вести еще много других операций, например, вставку или удаление сразу нескольких элементов, укорачивание массива, очистка массива целиком и т.д. Но это все можно реализовать по образу и подобию перечисленных выше операций.

Другой вопрос — логика изменения размера массива в процессе работы, т.е. при добавлении или удалении элементов. Поскольку захват и освобождение памяти — это

сравнительно трудоемкие операции, то обычно в таких случаях массив создается с некоторым "запасом" и его размер меняется при исчерпании этого запаса. Логика изменения размера обсуждается на лекции.

Наконец, динамический массив должен поддерживать те же правила работы, которые обеспечивал и просто массив. Поэтому здесь естественно употребить наследование, для чего немного подправим описание класса vector_base. В итоге мы можем получить примерно такое описание.

```
template <class T>
class Wvector_base
{
protected:
    T * value;
    size_t size;
public:
    Wvector_base(size_t n);
    Wvector_base(const Wvector_base<T> &v);
    ~Wvector_base();
    size_t Size() const;
    T & operator[] (size_t i);
    const T & operator[] (size_t i) const;
    T* Data() { return value; }
    const T* Data() const { return value; }
};

template <class T>
class Wvector : public Wvector_base<T>
{
protected:
    size_t maxsize;
public:
    Wvector(size_t n) : Wvector_base(n) { maxsize = size; }
    Wvector(size_t n, size_t m) : Wvector_base(m) { size = min(n, m); }
    Wvector(const Wvector<T> &v);
    ~Wvector() {}
    size_t Capacity() const { return maxsize; }
    void  Resize(size_t n);
    void  Reserve(size_t m);
    void  Clear();
    size_t Push_back(const T& x);
    size_t Insert(size_t ind, const T& x);
    size_t Erase(int ind);
};
```

Теперь надо решить проблемы с реализацией методов, затрагивающих память (указатель value).

— конструкторы — они вызывают базовый конструктор.

— деструктор — он отрабатывает сам (ничего не делает), а потом обращается к базовому деструктору, который все и освобождает.

Остальные функции работают с существующим массивом в памяти, и в некоторые моменты, возможно, должны этот массив перенести в другое место с сохранением или изменением значений элементов. Это происходит по очевидной схеме:

— если запасного места хватает, то оно используется для новых значений (возможно, старые значения также перемещаются при вставке или удалении), и потом корркtируется переменная размера массива.

— если запасного места нет, то захватывается новый массив достаточного размера, необходимые значения записываются туда, а старый массив освобождается.

При увеличении массива можно применять разные стратегии, например, при небольших размерах каждый раз увеличивать размер в два раза, а потом по достижении некоторого порога увеличивать постоянными приращениями.

Примеры реализаций будут приведены в отдельных файлах.

В данном примере наследование служит очень простой цели — расширению функциональности базового класса. При этом полиморфизм здесь отсутствует, и виртуальные функции не нужны.

Экзотические массивы

Рассмотрим пример, когда нам хочется работать с массивом объектов, но сами объекты не придставляют собой независимые сущности или поддержка такой “независимости” дорого обходится. Типичный пример — массив bool переменных (т.е. по сути битов). Конечно, можно вместо отдельного бита брать целое число со значениями 0 и 1, но это приводит к значительному объему пропадающей зря памяти. Т.е. вопрос — что получится, если объявить `Wvector<bool> a(1024);` — сколько места в памяти это займет?

Мы можем построить отдельную реализацию такого массива для bool, но с тем же самым интерфейсом, который был разработан для произвольных типов.

Начнем с базового класса.

```
template <>
class Wvector_base<bool>
{
protected:
    int * value;  // в каждом элементе по 32 бита
    size_t size;  // битовый размер
    size_t act_size; // актуальное количество целых в массиве
public:
    Wvector_base(size_t n) : size(n), value(nullptr) {
        act_size = (size + 31) / 32;
        value = new int[act_size];
    }

    Wvector_base(const Wvector_base<T> &v)
    : size(v.size), act_size(v.act_size), value(nullptr) {
        value = new int[act_size];
```

```
        for (int i=0; i<act_size; ++i) value[i] = v.value[i];
    }

    ~Wvector_base() { delete [] value; }
    size_t Size() const { return size; }
    bool operator[] (size_t i) const { return value[i>>5] & (1<<(i & 0x1f)); }

//    T* Data() { return value; }
//    const T* Data() const { return value; }
};
```

На самом деле этот код не совсем корректен, поскольку наследование шаблонных классов имеет свою специфику, в частности, для доступа к элементам базового класса из порожденного класса необходимо явно указывать контекст. В нашем случае вместо обращения к переменной базового класса `size`, нужно записывать ее в виде `Wvector_base<T>::size` и аналогично с другими объектами (например, `value`).

Аккуратная частичная реализация этих классов вместе с простейшими тестами приведена в архиве vector.zip.