Тема 2. Непрерывные схемы хранения данных

Стек, дек, очередь

Массив предоставляет непосредственный доступ к своим элементам в любой момент времени. Бывают ситуации, когда целесообразно ограничить доступ к элементам в зависимости от момента их появления. Исторически было сформулировано две дисциплины подобных ограничений доступа, которые получили названия

LIFO — Last In First Out — стек stack

FIFO — First In First Out — очередь queue

И далее к ним было добавлено понятие дек — double ended queue, очередь с двумя концами как обобщение понятий стека или очереди.

```
Иллюстраци: стек, очередь, дек
вершина стека, голова и хвост очереди
дисциплина доступа
```

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

Мы будем сейчас строить так называемые непрерывные реализации данных схем хранения, когда элементы размещаются в некотором ограниченном массиве. Таким образом, эти контейнеры будут иметь ограничение по максимальному количесту элементов, хотя это ограничение можно легко снять за счет использования динамического массива, но с естественным увеличением трудоемкости при выделении памяти и копировании, присущей операциям с динамическим массивом.

Непрерывная реализация стека.

Следуем сформулированным выше 5 пунктам, относящимся к контейнерным объектам.

Параметрическая реализация стека.

```
template <class T>
class Stack
{
 private:
   T *mem, *top;      // указатели на массив и текущую вершину стека
```

```
   size_t  capacity;  // максимально доступный размер
 public:
   Stack(size_t msize);
   Stack(const Stack<T> & st);
   ~Stack() { delete [] mem; }

   void Push(const T & x);
   void Pop();
   T & Top();
   const T & Top() const;

   size_t Size() const { return top-mem+1; }
   size_t Capacity() const { return capacity; }
};

template <class T>
Stack<T>::Stack(size_t msize) : capacity(msize)
{
   mem = new  T[capacity];
   top = mem - 1;
}

template <class T>
Stack<T>::Stack(const Stack<T> & st) : capacity(st.capacity)
{
   mem = new  T[capacity];
   size_t size = st.Size();
   for (size_t i=0; i<size; i++) { mem[i] = st.mem[i]; }
   top = mem + size - 1;
}

template <class T>
void Stack<T>::Push(const T & x)
{
   if (Size() < capacity) { *(++top) = x; }
}

template <class T>
void Stack<T>::Pop()
{
   if (Size() > 0) { top--; }
}

template <class T>
T & Stack<T>::Top()
{
   if (Size() == 0) throw std::range_error("Stack::Top: empty stack");
   return *top;
}
```

В этих реализациях есть ряд упрощений, хотя они полностью обеспечивают дисциплину работы со стеком. Можно было бы возвращать bool из функций добавления и удаления элементов стека. Тогда успешность этих операций можно было бы проверить сразу.

Реализация очереди и дека на базе массива.

Здесь используется кольцевой буфер, т.е. при проходе по массиву и при достижении конца текузая позиция переносится в начало массива. Массив как бы становится закольцованным для перемещения текущей позиции как вперед так и назад.

Так как дек является обобщением очереди (или очередь является частным случаем дека), то реализацию проведем для дека. Для очереди можно просто удалить половину "симметричных"функций.

```
template <class T>
class Deque
{
 private:
   T *mem, *endmem, *front, *back;  // указатели на массив и концы дека
   size_t  size;
   T * Next(T * p) { return (p == endmem) ? 0 : p+1; }
   T * Prev(T * p) { return (p == 0) ? endmem : p-1; }
 public:
   Deque(size_t msize);
   Deque(const Deque<T> & q);
   ~Deque() { delete [] mem; }

   void Push_Front(const T & x);
   void Push_Back(const T & x);
   void Pop_Front();
   void Pop_Back();
   T & Front();
   T & Back();
   const T & Front() const;
   const T & Back() const;

   size_t Size() const { return size; }
   size_t Maxsize() const { return endmem - mem + 1; }
}

template <class T>
Deque<T>::Deque(size_t msize)
{
     mem = new  T[msize];
     endmem = mem + msize - 1;
     back = mem;          // back   "растет направо"
     front = Next(back);  // front  "растет налево"
     size = 0;
}

template <class T>
void Deque<T>::Push_Back(const T & x)
{
   if (size == Capacity()) return;
   back = Next(back);
   *back = x;
   size++;
}

template <class T>
void Deque<T>::Push_Front(const T & x)
{
```

```
   if (size == Capacity()) return;
   front = Prev(front);
   *front = x;
   size++;
}

template <class T>
bool Deque<T>::Pop_Back()
{
   if (size == 0) return false;
   back = Prev(back);
   size--;
}

template <class T>
bool Deque<T>::Pop_Front()
{
   if (size == 0) return false;
   front = Next(front);
   size--;
}

template <class T>
bool Deque<T>::DelHead()
{
   if (size == 0) return false;
   head = Prev(head);
   size--;
   return true;
}

template <class T>
T & Deque<T>::Front()
{
    if (size==0) throw std::range_error("Deque::Front() : empty queue\n");
    return *front;
}
```

и т.д.

Можно ли стек передать параметром в функцию или вернуть как значение?

Можно, но нужен констуктор копирования!

Но при такой передаче класса как параметра или возвращаемого значения происходит копирование. Для простых объектов это легко, а для сложных может быть трудоемко. Поэтому всегда лучше в качестве параметра и возвращаемого значения использовать (константную) ссылку на объект (если эти ничему не противоречит).

В более поздних версиях C++ появилось понятие перемещения (move), что позволяет в некоторых случаях существенно ускорить работу, исключив излишние копирования при реализации операций с объектом. Но об этом будет отдельный разговор.