Тема 2. Непрервыные схемы хранения данных

Лекция 4. Стек, дек, очередь

Массив предоставляет непоследственный 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
```

Мы будем сейчас строить так называемые непрерывные реализации данных схем хранения, когда элементы размещяются 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, на за счет естетсвенного увеличесния трудоемкости при выделении памяти и копировании, присущей операциям с динамическим массивом.

Непрерывная реализация стека. Следуем сформулированным выше 5 пунктам, относящимся к контейнерным объектам.

```
Иллюстрация: Стек. Основы реализации и дисциплина доступа
```

У нас был класс MyException, он нам полностью подходит по функциональности.

```
class MyException
{
   char message[256];
   int code;
 public:
   MyException(const char *msg, int c);
   const char * Message() const;
   int Code();
};

enum ErrorCodes
{
   EC_UNKNOWN = 0,
   EC_MEMORY = -1,
   EC_INDEX = -2,
   EC_ACCESS = -3
};
```

Параметрическая реализация стека.

```
template <class T>
class Stack
{
 private:
   T *mem, *top;
   int  maxsize;
 public:
   Stack(int msize);
   Stack(const Stack<T> & st);
   ~Stack() { delete [] mem; }

   bool Push(const T & x);
   bool Pop(T & x);
   bool Del();

   T & Top();
   const T & Top() const;

   int Size() const { return top-mem+1; }
   int Maxsize() const { return maxsize; }
}

template <class T>
Stack<T>::Stack(int msize)
{
 try {
   maxsize = msize;
   mem = new  T[maxsize];
   top = mem - 1;
 } catch (std::bad_alloc e) {
    throw new MyException("Stack: cannot allocate memory\n", EC_MEMORY);
 }
}

template <class T>
Stack<T>::Stack(const Stack<T> & st)
{
 try {
   maxsize = st.maxsize;
   mem = new  T[maxsize];
   int size = st.Size();
   for (int i=0; i<size; i++) { mem[i] = st.mem[i]; }
   top = mem + size - 1;
 } catch (std::bad_alloc e) {
    throw new MyException("Stack: cannot apply copy constructor\n", EC_MEMORY);
 }
}
```

```
template <class T>
bool Stack<T>::Push(const T & x)
{
   if (Size() == maxsize) return false;
   *(++top) = x;
   return true;
}

template <class T>
bool Stack<T>::Pop(T & x)
{
   if (Size() == 0) return false;
   x = *top--;
   return true;
}

template <class T>
T & Stack<T>::Top()
{
   if (Size() == 0) throw new MyException("Stack::Top: empty stack\n", EC_ACCESS);
   return *top;
}
```

Реализация очереди на базе массива

```
Иллюстрация:  схема хранения, голова и хвост
              кольцевой буфер, пустое и полное состояние
```

```
template <class T>
class Queue
{
 private:
   T *mem, *head, *tail;
   int  size, maxsize;
   T * Next(T * p);
   T * Prev(T * p);
 public:
   Queue(int msize);
   Queue(const Stack<T> & st);
   ~Queue() { delete [] mem; }

   bool PushTail(const T & x);
   bool PopHead(T & x);
   bool DelHead();

   T & Head();
   const T & Head() const;
```

```
   int Size() const { return size; }
   int Maxsize() const { return maxsize; }
}

template <class T>
T * Queue<T>::Next(T * p)
{
   return (p - mem < maxsize - 1) ? p+1 : mem;
}

template <class T>
T * Queue<T>::Prev(T * p)
{
   return (p - mem > 0) ? p-1 : mem + maxsize - 1;
}

template <class T>
Queue<T>::Queue(int msize)
{
 try {
     maxsize = msize;
     mem = new  T[maxsize];
     head = mem;
     tail = Next(head);
     size = 0;
 } catch (std::bad_alloc e) {
    throw new MyException("Queue:: constructor: memory allocation error\n", EC_MEMORY
 }
}

template <class T>
bool Queue<T>::PushTail(const T & x)
{
   if (size == maxsize) return false;
   tail = Prev(tail);
   *tail = x;
   size++;
   return true;
}

template <class T>
bool Queue<T>::PopHead(T & x)
{
   if (size == 0) return false;
   x = *head;
   head = Prev(head);
```

```
   size--;
   return true;
}

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

template <class T>
T & Queue<T>::Head()
{
    if (size==0) throw new MyException("Queue::Head  empty queue\n", EC_ACCESS);
    return *head;
}

template <class T>
const T & Queue<T>::Head() const
{
    if (size==0) throw new MyException("Queue::Head  empty queue\n", EC_ACCESS);
    return *head;
}
```

Дек — это симметричная “двусторонняя” очередь.

```
template <class T>
class Deq
{
 private:
   T *mem, *head, *tail;
   int  size, maxsize;
   T * Next(T * p);
   T * Prev(T * p);
 public:
   Deq(int msize);
   Deq(const Stack<T> & st);
   ~Deq() { delete [] mem; }

   bool PushTail(const T & x);
   bool PushHead(const T & x);
   bool PopHead(T & x);
   bool PopTail(T & x);
   bool DelHead();
   bool DelTail();
```

```
   T & Head();
   const T & Head() const;
   T & Taild();
   const T & Tail() const;

   int Size() const { return top-mem+1; }
   int Maxsize() const { return maxsize; }
}
```

Использование очевидным образом

```
  Stack<int> a(100);
  Stack<double> b(200);
  Queue<const char*> s(1024);
   a.Push(123);
   b.Push(3.14);
   const char *str = "some string";
   s.Push(str);

  Queue<Stack<int> > abc(10);  // не сработает !!!
                                  нужно конструктор без параметров
                                  и оператор присваивания

   Stack();
   Stack(msize = 0);

   const Stack<T> & operator= (const Stack<T> & st);
```

Реализация оператора присваивания для стека — нужно уничтожить старое содержимое и создать копию другого объекта.

```
template <class T>
const Stack<T> & Stack<T>::operator= (const Stack<T> & st)
{
 try {
   delete [] mem;
   maxsize = st.maxsize;
   mem = new  T[maxsize];
   int size = st.Size();
   for (int i=0; i<size; i++) { mem[i] = st.mem[i]; }
   top = mem + size - 1;
   return *this;
 } catch (std::bad_alloc e) {
    throw new MyException("Stack: cannot apply operator=\n", EC_MEMORY);
 }
}
```

Использование:

```
Queue<Stack<int> > abc(10);
Stack<int> a1(10);
Stack<int> a2(10);
Stack<int> a3(10);
Stack<int> a4(10);

a1.Push(123);
a1.Push(234);
a2.Push(1123);
a2.Push(2234);
                      a3.Push(555);

abc.PushTail(a1);
abc.PushTail(a2);
abc.PushTail(a3);

abc.Head().Push(555);
```