

Объектно-ориентированное
программирование
с использованием языка

C++



Урок №7

Динамические структуры данных

Содержание

Понятие динамической структуры данных	3
Стек	5
Основные операции над стеком и его элементами	5
Реализация — стек	6
Очередь.....	10
Реализация — очередь	11
Кольцевая очередь.....	15
Реализация — кольцевая очередь	15
Очередь с приоритетами	19
Реализация — очередь с приоритетами.....	20
Домашнее задание	26

Понятие динамической структуры данных

Сегодня мы знакомимся с понятием динамической структуры данных. До этого момента мы работали только с данными, имеющими статическую, неизменяемую во время исполнения программы, структуру. Парадокс, но даже динамический массив, по сути своей не очень динамичен, так как для того, что бы изменить его размер, необходимо пересоздать его. Другими словами во время работы программы довольно просто изменять только значения элементов, в то время как изменение количества этих элементов приводит к ряду монотонных процедур. А это, конечно, не всегда удобно.

Предположим, что в программе, предназначеннной для ввода и обработки данных об учениках класса, для хранения данных используются статические массивы. При определении размера массива программисту приходится ориентироваться на некоторое среднее или предельное количество учеников в классе. При этом, если реально учеников в классе меньше предполагаемого количества, то неэффективно используется память компьютера, а если это число больше, то программу использовать уже нельзя (т.к. надо внести изменения в исходный текст и выполнить компиляцию). Создание динамического массива несомненно явится решением, но приведет к усложнению написания кода.

Задачи, обрабатывающие данные, которые по своей природе являются динамическими, удобно решать с помощью динамических структур. Именно с этим интересным средством мы будем учиться работать.

Учитывая всё вышесказанное, можно предположить, что динамические структуры — это некие конструкции, способные при необходимости выделять память под новые элементы или удалять выделенную память для ненужных элементов во время работы программы. Для решения проблемы адресации динамических структур данных используется метод, называемый *динамическим распределением памяти*, то есть память под отдельные элементы выделяется в момент, когда они «начинают существовать» в процессе выполнения программы, а не во время компиляции. Компилятор в этом случае только выделяет фиксированный объем памяти для хранения адреса динамически размещаемого элемента, а не самого элемента.

Существует несколько видов динамических структур данных. При этом каждый из них имеет как достоинства, так и недостатки. Поэтому, какой именно вид использовать, зависит исключительно от конкретной решаемой задачи.

Стек

Стек — динамическая структура данных, представляющая из себя упорядоченный набор элементов, в которой добавление новых элементов и удаление существующих производится с одного конца, называемого вершиной стека.

Кроме того, стек обладает так называемым базовым адресом. Под этим понятием скрывается начальный адрес памяти, в которой размещается стек.

По определению, элементы извлекаются из стека в порядке, обратном их добавлению в эту структуру, то есть действует принцип LIFO (*Last In First Out*) или «последний пришёл первый ушёл».

Наиболее наглядным примером организации стека служит детская пирамидка, где добавление и снятие колец осуществляется как раз согласно определению стека.

ВАЖНО!!! Для хранения стека в памяти отводится сплошная область, граничные адреса которой являются параметрами физической структуры стека. Если в процессе заполнения стека указатель, перемещаясь «вверх», выходит за границу первоначально отведенной области, то происходит переполнение стека (*stack overflow*). Переполнение стека рассматривается как исключительная ситуация, требующая выполнения действий по ее ликвидации.

Основные операции над стеком и его элементами

1. Добавление элемента в стек.
2. Удаление элемента из стека.

3. Проверка, пуст ли стек. (Стек считается «пустым», если указатель вершины совпадает с указателем нижней границы.)
4. Просмотр элемента в вершине стека без удаления.



Применение. Динамическая структура Стек чаще всего используется при синтаксическом анализе всевозможных выражений.

Примечание. Кстати, каждое приложение имеет собственный стек. В нём компилятор создает локальные переменные. А, также, через стек происходит передача параметров в функцию.

Реализация — стек

```
#include <iostream>
#include <string.h>
#include <time.h>
using namespace std;
```

```
class Stack
{
    //Нижняя и верхняя границы стека
    enum {EMPTY = -1, FULL = 20};

    //Массив для хранения данных
    char st[FULL + 1];

    //Указатель на вершину стека
    int top;

public:
    //Конструктор
    Stack();

    //Добавление элемента
    void Push(char c);

    //Извлечение элемента
    char Pop();

    //Очистка стека
    void Clear();

    //Проверка существования элементов в стеке
    bool IsEmpty();

    //Проверка на переполнение стека
    bool IsFull();

    //Количество элементов в стеке
    int GetCount();
};

Stack::Stack()
{
    //Изначально стек пуст
    top = EMPTY;
}
```

```
void Stack::Clear()
{
    //Эффективная "очистка" стека
    //(данные в массиве все еще существуют,
    //но функции класса, ориентированные
    //на работу с вершиной стека,
    //будут их игнорировать)
    top = EMPTY;

bool Stack::IsEmpty()
{
    //Пуст?
    return top == EMPTY;
}

bool Stack::IsFull()
{
    //Полон?
    return top == FULL;
}

int Stack::GetCount()
{
    //Количество присутствующих в стеке элементов
    return top + 1;
}

void Stack::Push(char c)
{
    //Если в стеке есть место, то увеличиваем указатель
    //на вершину стека и вставляем новый элемент
    if(!IsFull())
        st[++top] = c;
}

char Stack::Pop()
{
    //Если в стеке есть элементы, то возвращаем
```

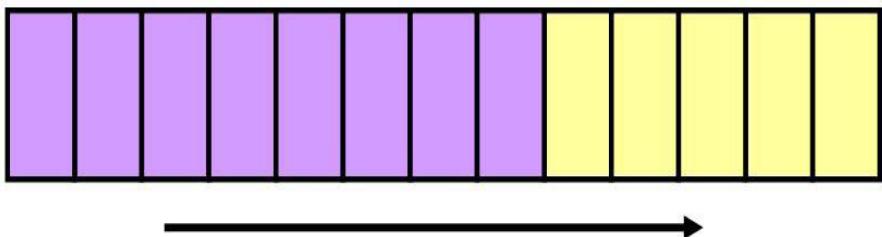
```
//верхний и уменьшаем указатель на вершину стека
if(!IsEmpty())
    return st[top--];
else //Если в стеке элементов нет
    return 0;
}

void main()
{
    srand(time(0));
    Stack ST;
    char c;
    //пока стек не заполнится
    while(!ST.IsFull()) {
        c=rand()%4+2;
        ST.Push(c);
    }
    //пока стек не освободится
    while(c=ST.Pop()) {
        cout<<c<<" ";
    }
    cout<<"\n\n";
}
```

Очередь

Следующая динамическая структура, которую мы рассмотрим — простая очередь. Очередь, так же как и стек можно реализовывать на практике с помощью массива.

Очередь — это последовательный набор элементов с переменной длиной. При этом, добавление элементов в очередь происходит с одной стороны, а удаление — с другой стороны. Данная конструкция функционирует по идеологии FIFO (*First In — First Out*), то есть «первым пришел — первым вышел». Для очереди принято выделять конечную последовательность элементов, из которых в каждый текущий момент времени элементами очереди заняты лишь часть последовательных элементов.



В принципе, с простой очередью вы сталкиваетесь постоянно. Вот лишь некоторые из примеров: *очередь в мавзолей, очередь печати принтера, даже действия линейного алгоритма выполняются по очереди*.

Реализация — очередь

```
#include <iostream>
#include <string.h>
#include <time.h>
using namespace std;

class Queue
{
    //Очередь
    int * Wait;
    //Максимальный размер очереди
    int MaxQueueLength;
    //Текущий размер очереди
    int QueueLength;

public:
    //Конструктор
    Queue (int m);

    //Деструктор
    ~Queue () ;

    //Добавление элемента
    void Add(int c);

    //Извлечение элемента
    int Extract();

    //Очистка очереди
    void Clear();

    //Проверка существования элементов в очереди
    bool IsEmpty();

    //Проверка на переполнение очереди
    bool IsFull();
```

```
//Количество элементов в очереди
int GetCount();

//демонстрация очереди
void Show();
};

void Queue::Show() {
    cout<<"\n-----\n";
    //демонстрация очереди
    for(int i=0;i<QueueLength;i++) {
        cout<<Wait[i]<<" ";
    }
    cout<<"\n-----\n";
}

Queue::~Queue()
{
    //удаление очереди
    delete[]Wait;
}

Queue::Queue(int m)
{
    //получаем размер
    MaxQueueLength=m;
    //создаем очередь
    Wait=new int[MaxQueueLength];
    //Изначально очередь пуста
    QueueLength = 0;
}

void Queue::Clear()
{
    //Эффективная "очистка" очереди
    QueueLength = 0;
}
```

```
bool Queue::IsEmpty()
{
    //Пуст?
    return QueueLength == 0;
}

bool Queue::IsFull()
{
    //Полон?
    return QueueLength == MaxQueueLength;
}

int Queue::GetCount()
{
    //Количество присутствующих в стеке элементов
    return QueueLength;
}

void Queue::Add(int c)
{
    //Если в очереди есть свободное место,
    //то увеличиваем количество
    //значений и вставляем новый элемент
    if(!IsFull())
        Wait[QueueLength++] = c;
}

int Queue::Extract()
{
    //Если в очереди есть элементы, то возвращаем тот,
    //который вошел первым и сдвигаем очередь
    if(!IsEmpty()){
        //запомнить первый
        int temp=Wait[0];

        //сдвинуть все элементы
        for(int i=1;i<QueueLength;i++)
            Wait[i-1]=Wait[i];
    }
}
```

```
//уменьшить количество
QueueLength--;

//вернуть первый (нулевой)
return temp;
}

else //Если в стеке элементов нет
    return -1;
}

void main()
{
    srand(time(0));

    //создание очереди
    Queue QU(25);

    //заполнение части элементов
    for(int i=0;i<5;i++){
        QU.Add(rand()%50);
    }
    //показ очереди
    QU.Show();

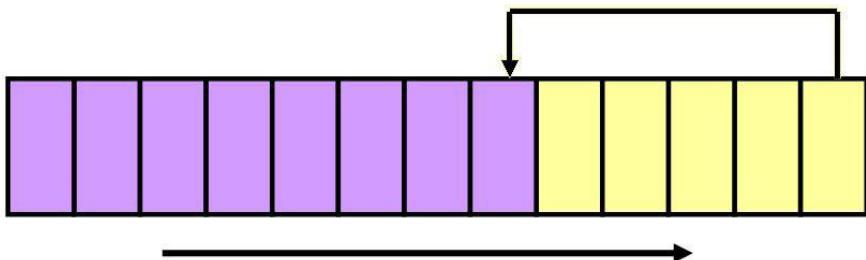
    //извлечение элемента
    QU.Extract();

    //показ очереди
    QU.Show();
}
```

Кольцевая очередь

Кольцевая очередь очень похожа на простую очередь. Она тоже построена на идеологии FIFO, напоминаем — *элемент, который добавили в очередь первым, первым её и покинет.*

Разница лишь в том, что элемент, который выходит из начала очереди, будет перемещён в её конец.



В качестве самого простого примера, можно привести известный вам с детства *круговорот воды в природе*, или трамваи, курсирующие по круговому маршруту.

Реализация — кольцевая очередь

```
#include <iostream>
#include <string.h>
#include <time.h>
using namespace std;

class QueueRing
{
    //Очередь
    int * Wait;
    //Максимальный размер очереди
    int MaxQueueLength;
```

```
//Текущий размер очереди
int QueueLength;

public:
    //Конструктор
    QueueRing(int m);

    //Деструктор
    ~QueueRing();

    //Добавление элемента
    void Add(int c);

    //Извлечение элемента
    bool Extract();

    //Очистка очереди
    void Clear();

    //Проверка существования элементов в очереди
    bool IsEmpty();

    //Проверка на переполнение очереди
    bool IsFull();

    //Количество элементов в очереди
    int GetCount();

    //демонстрация очереди
    void Show();
};

void QueueRing::Show() {
    cout<<"\n-----\n";
    //демонстрация очереди
    for(int i=0;i<QueueLength;i++) {
        cout<<Wait[i]<<" ";
    }
}
```

```
cout<<"\n-----\n";  
}  
  
QueueRing::~QueueRing()  
{  
    //удаление очереди  
    delete []Wait;  
}  
  
QueueRing::QueueRing(int m)  
{  
    //получаем размер  
    MaxQueueLength=m;  
    //создаем очередь  
    Wait=new int [MaxQueueLength];  
    //Изначально очередь пуста  
    QueueLength = 0;  
}  
  
void QueueRing::Clear()  
{  
    //Эффективная "очистка" очереди  
    QueueLength = 0;  
}  
  
bool QueueRing::IsEmpty()  
{  
    //Пуст?  
    return QueueLength == 0;  
}  
  
bool QueueRing::IsFull()  
{  
    // Полон?  
    return QueueLength == MaxQueueLength;  
}
```

```
int QueueRing::GetCount()
{
    //Количество присутствующих в стеке элементов
    return QueueLength;
}

void QueueRing::Add(int c)
{
    //Если в очереди есть свободное место,
    //то увеличиваем количество
    //значений и вставляем новый элемент
    if(!IsFull())
        Wait[QueueLength++] = c;
}

bool QueueRing::Extract()
{
    //Если в очереди есть элементы, то возвращаем тот,
    //который вошел первым и сдвигаем очередь
    if(!IsEmpty()) {
        //запомнить первый
        int temp=Wait[0];

        //сдвинуть все элементы
        for(int i=1;i<QueueLength;i++)
            Wait[i-1]=Wait[i];

        //забрасываем первый "вытолкнутый элемент
        //в конец"
        Wait[QueueLength-1]=temp;
        return 1;
    }
    else return 0;
}

void main()
{
    srand(time(0));
}
```

```
//создание очереди  
QueueRing QUR(25);  
  
//заполнение части элементов  
for(int i=0;i<5;i++){  
    QUR.Add(rand()%50);  
}  
//показ очереди  
QUR.Show();  
  
//извлечение элемента  
QUR.Extract();  
  
//показ очереди  
QUR.Show();  
}
```

Очередь с приоритетами

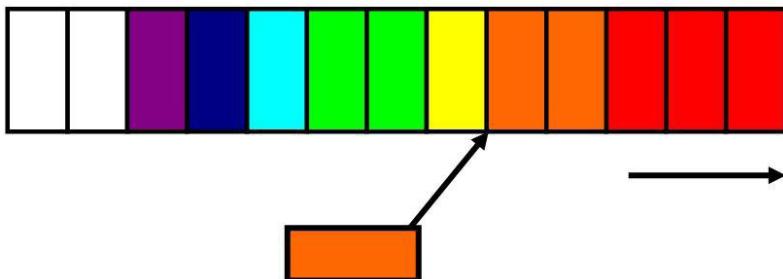
Мы уже познакомились с двумя типами очередей и они оказались достаточно простыми. Однако существует еще одна очередь — **очередь с приоритетом**.

Дело в том, что часто необходимо создать очередь, где (извините за тавтологию) очередность выхода зависит от приоритетов элементов. В качестве приоритета может выступать какое-либо число, временная константа и т. п. В момент извлечения элемента будет выбран тот элемент, который обладает большим приоритетом.

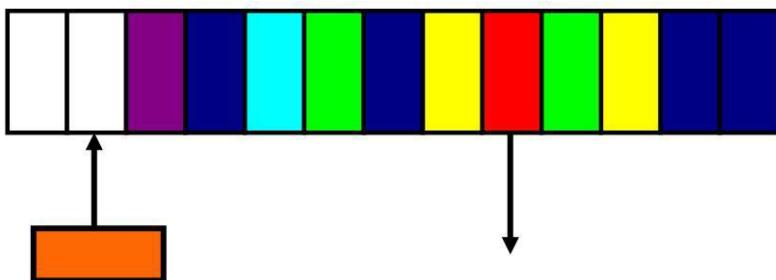
Существует несколько видов приоритетных очередей:

1. **Очередь с приоритетным включением** — последовательность элементов очереди является строго упорядоченной. Другими словами, каждый элемент при попадании

в очередь сразу же располагается согласно своего приоритета. А в момент исключения элемента просто извлекается элемент из начала.



2. **Очереди с приоритетным исключением** — элемент добавляется в конец очереди, а при извлечении осуществляется самого приоритетного элемента, который впоследствии удаляется из очереди.



Реализация — очередь с приоритетами

```
#include <iostream>
#include <string.h>
#include <time.h>
using namespace std;

class QueuePriority
{
```

```
//Очередь
int * Wait;
//Приоритет
int * Pri;
//Максимальный размер очереди
int MaxQueueLength;
//Текущий размер очереди
int QueueLength;

public:
    //Конструктор
    QueuePriority(int m);

    //Деструктор
    ~QueuePriority();

    //Добавление элемента
    void Add(int c,int p);

    //Извлечение элемента
    int Extract();

    //Очистка очереди
    void Clear();

    //Проверка существования элементов в очереди
    bool IsEmpty();

    //Проверка на переполнение очереди
    bool IsFull();

    //Количество элементов в очереди
    int GetCount();

    //демонстрация очереди
    void Show();
};
```

```
void QueuePriority::Show() {
    cout<<"\n-----\n";
    //демонстрация очереди
    for(int i=0;i<QueueLength;i++)
        { cout<<Wait[i]<< " - "<<Pri[i]<<"\n\n";
    }
    cout<<"\n-----\n";
}

QueuePriority::~QueuePriority()
{
    //удаление очереди
    delete[] Wait;
    delete[] Pri;
}

QueuePriority::QueuePriority(int m)
{
    //получаем размер
    MaxQueueLength=m;
    //создаем очередь
    Wait=new int[MaxQueueLength];
    Pri=new int[MaxQueueLength];
    //Изначально очередь пуста
    QueueLength = 0;
}

void QueuePriority::Clear()
{
    //Эффективная "очистка" очереди
    QueueLength = 0;
}

bool QueuePriority::IsEmpty()
{
    //Пуст?
    return QueueLength == 0;
}
```

```
bool QueuePriority::IsFull()
{
    //Полон?
    return QueueLength == MaxQueueLength;
}

int QueuePriority::GetCount()
{
    //Количество присутствующих в стеке элементов
    return QueueLength;
}

void QueuePriority::Add(int c,int p)
{
    //Если в очереди есть свободное место,
    //то увеличиваем количество
    //значений и вставляем новый элемент
    if(!IsFull()){
        Wait[QueueLength] = c;
        Pri[QueueLength] = p;
        QueueLength++;
    }
}

int QueuePriority::Extract()
{
    //Если в очереди есть элементы, то возвращаем тот,
    //у которого наивысший приоритет и сдвигаем очередь
    if(!IsEmpty()){

        //пусть приоритетный элемент - нулевой
        int max_pri=Pri[0];
        //а приоритетный индекс = 0
        int pos_max_pri=0;

        //ищем приоритет
        for(int i=1;i<QueueLength;i++)
    }
}
```

```
//если встречен более приоритетный элемент
if(max_pri<Pri[i]){
    max_pri=Priority[i];
    pos_max_pri=i;
}
//вытаскиваем приоритетный элемент
int temp1=Wait[pos_max_pri];
int temp2=Priority[pos_max_pri];

//сдвигнуть все элементы
for(int i=pos_max_pri;i<QueueLength-1;i++) {
    Wait[i]=Wait[i+1];
    Priority[i]=Priority[i+1];
}
//уменьшаем количество
QueueLength--;
//возврат извлеченного элемента
return temp1;

}
else return -1;
}

void main()
{
    srand(time(0));

    //создание очереди
    QueuePriority QUP(25);
    //заполнение части элементов
    for(int i=0;i<5;i++){

        //значения от 0 до 99 (включительно)
        //и приоритет от 0 до 11 (включительно)
        QUP.Add(rand()%100,rand()%12);
    }
}
```

```
//показ очереди  
QUP.Show();  
  
//извлечение элемента  
QUP.Extract();  
  
//показ очереди  
QUP.Show();  
}
```

Домашнее задание

1. Создать имитацию игры «однорукий бандит». Например, при нажатии кнопки «Enter» происходит «вращение» трех барабанов (естественно, количество вращений каждого из них выбирается случайно), на которых изображены разные значки; и если выпадает определенная комбинация, то игрок получает какой-то выигрыш.
2. Создать имитационную модель «остановка маршрутных такси». Необходимо вводить следующую информацию: среднее время между появлением пассажиров на остановке в разное время суток, среднее время между появлениями маршруток на остановке в разное время суток, тип остановки (конечная или нет). Необходимо определить: среднее время пребывания человека на остановке, достаточный интервал времени между приходами маршруток, чтобы на остановке находилось не более N людей одновременно. Количество свободных мест в маршрутке является случайной величиной.
3. Разработать приложение, имитирующее очередь печати принтера. Должны быть клиенты, посылающие запросы на принтер, у каждого из которых есть свой приоритет. Каждый новый клиент попадает в очередь в зависимости от своего приоритета. Необходимо сохранять статистику печати (пользователь, время) в отдельной очереди. Предусмотреть вывод статистики на экран.



Урок №7

Динамические структуры данных

© Компьютерная Академия «Шаг», www.itstep.org

Все права на охраняемые авторским правом фото-, аудио- и видеопроизведения, фрагменты которых использованы в материале, принадлежат их законным владельцам. Фрагменты произведений используются в иллюстративных целях в объёме, оправданном поставленной задачей, в рамках учебного процесса и в учебных целях, в соответствии со ст. 1274 ч. 4 ГК РФ и ст. 21 и 23 Закона Украины «Про авторське право і суміжні права». Объём и способ цитируемых произведений соответствует принятым нормам, не наносит ущерба нормальному использованию объектов авторского права и не ущемляет законные интересы автора и правообладателей. Цитируемые фрагменты произведений на момент использования не могут быть заменены альтернативными, не охраняемыми авторским правом аналогами, и как таковые соответствуют критериям добросовестного использования и честного использования.

Все права защищены. Полное или частичное копирование материалов запрещено. Согласование использования произведений или их фрагментов производится с авторами и правообладателями. Согласованное использование материалов возможно только при указании источника.

Ответственность за несанкционированное копирование и коммерческое использование материалов определяется действующим законодательством Украины.