Объектно-ориентированное программирование с использованием языка C++

ШАГ компьютерная АКАДЕМИЯ

Урок №8

Объектно-ориентированное программирование с использованием языка C++

Содержание

Односвязный список

Сегодня мы с вами познакомимся с динамической структурой данных, которая представляет собой нечто похожее на безразмерный массив. Называется эта структура — **список**. Существует несколько разновидностей списков. Для начала мы рассмотрим — **односвязный или однонаправленный список.**

Односвязный список — это совокупность нескольких объектов, каждый из которых представляет собой элемент списка, состоящий из двух частей. Первая часть элемента — значение, которое он хранит, вторая — информация о следующем элементе списка.

Характер информации о следующем элементе зависит от того, где конкретно хранится список. Например, если это оперативная память, то информация будет представлять собой адрес следующего элемента, если файл — позицию следующего элемента в файле. Мы с Вами будем рассматривать реализацию списка хранящегося в оперативной памяти, однако, при желании, вы сможете создать собственный код для хранения списка в файле. Итак, приступим:

Каждый элемент списка мы представим программно с помощью структуры, которая состоит из двух составляющих:

1. Одно или несколько полей, в которых будет содержаться основная информация, предназначенная для хранения.
2. Поле, содержащее указатель на следующий элемент списка.

Отдельные объекты подобной структуры мы далее будем называть узлами, связывая их между собой с помощью полей, содержащих указатели на следующий элемент.

```
//узел списка
struct node
{
    //Информационная часть узла
    int value;

    //Указатель на следующий узел списка
    node *next;
};
```

После создания структуры, нам необходимо инкапсулировать её объекты в некий класс, что позволит управлять списком, как цельной конструкцией. В классе будет содержаться два указателя (на хвост, или начало списка и голову, или конец списка), а так же набор функции для работы со списком.

```
    //голова
    node *phead;

    //хвост
    node *ptail;
```

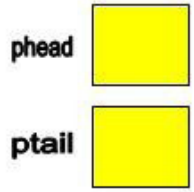
В целом, полученный список можно представить так:

Итак, основные моменты создания списка, мы рассмотрели, переходим непосредственно к его формированию.

Формирование списка

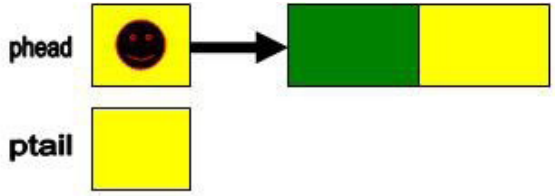
1. Отведем место для указателей в статической памяти.

```
node *phead;
node *ptail;
```



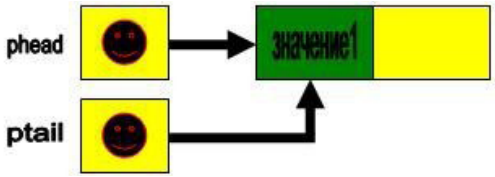
2. Зарезервируем место для динамического объекта.

```
phead=new node;
```



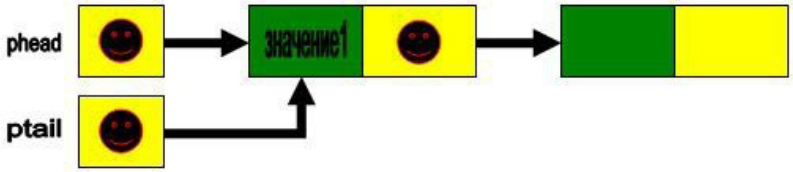
3. Присвоим значение переменной ptail, и поместим в информационное поле значение элемента.

```
ptail = phead;
ptail->value = "значение1";
```



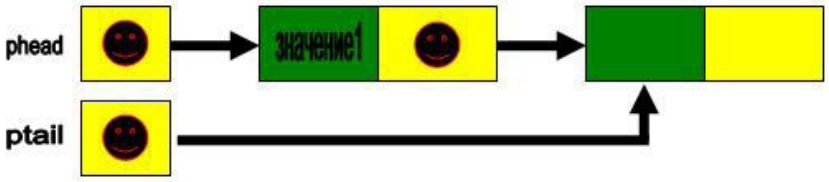
4. Поместим в поле узла адрес еще одного — нового динамического объекта.

```
ptail->next = new node;
```



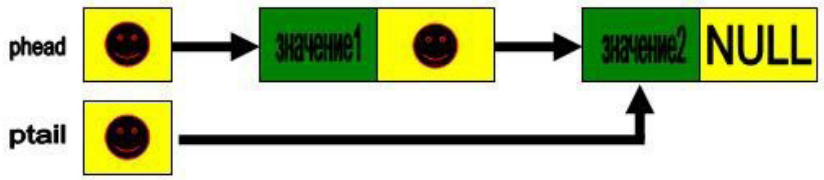
5. Переменная ptail должна содержать адрес последнего добавленного элемента, т. к. он добавлен в конец.

```
ptail = ptail->next;
```



6. Если требуется завершить построение списка, то в поле указателя последнего элемента нужно поместить NULL.

```
ptail->next = NULL;
ptail->value = "значение2";
```

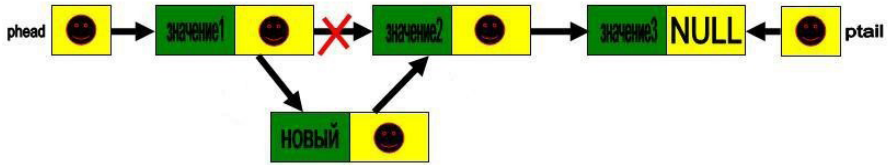



В результате построен линейный односвязный список, содержащий два узла.

Вставка узла в определенное заданное место списка

Примечание: Здесь и далее, мы не будем приводить фрагменты кода, так как они являются крупными. Позже мы с вами просто рассмотрим пример реализации односвязного списка целиком. А, сейчас, разберем операции над списком с помощью иллюстраций.

1. Выделить память под новый узел.
2. Записать в новый узел значение.
3. Записать в указатель на следующий узел адрес узла, который должен располагаться после нового узла.
4. Заменить в узле, который будет располагаться перед новым узлом, записанный адрес на адрес нового узла.



Удаление узла из списка

1. Записать адрес узла, следующего за удаляемым узлом, в указатель на следующий узел в узле, предшествующем удаляемому.
2. Удалить узел, предназначенный для удаления.

Ну что ж, пора переходить к практике — смотрим пример в следующем разделе урока.

Реализация — односвязный список

```
#include <iostream>
using namespace std;

struct Element
{
    //Данные
    char data;
    //Адрес следующего элемента списка
    Element * Next;
};

//Односвязный список
class List
{
    //Адрес головного элемента списка
    Element * Head;
    //Адрес головного элемента списка
    Element * Tail;
```



```
    //Количество элементов списка
    int Count;

public:
    //Конструктор
    List();
    //Деструктор
    ~List();

    //Добавление элемента в список
    //(Новый элемент становится последним)
    void Add(char data);

    //Удаление элемента списка
    //(Удаляется головной элемент)
    void Del();
    //Удаление всего списка
    void DelAll();

    //Распечатка содержимого списка
    //(Распечатка начинается с головного элемента)
    void Print();

    //Получение количества элементов,
    //находящихся в списке
    int GetCount();
};

List::List()
{
    //Изначально список пуст
    Head = Tail = NULL;
    Count = 0;
}

List::~List()
{
    //Вызов функции удаления
    DelAll();
}
```

```
int List::GetCount()
{
    //Возвращаем количество элементов
    return Count;
}

void List::Add(char data)
{
    //создание нового элемента
    Element * temp = new Element;

    //заполнение данными
    temp->data = data;
    //следующий элемент отсутствует
    temp->Next = NULL;
    //новый элемент становится последним элементом списка
    //если он не первый добавленный
    if(Head!=NULL){
        Tail->Next=temp;
        Tail = temp;
    }
    //новый элемент становится единственным
    //если он первый добавленный
    else{
        Head=Tail=temp;
    }
}

void List::Del()
{
    //запоминаем адрес головного элемента
    Element * temp = Head;
    //перебрасываем голову на следующий элемент
    Head = Head->Next;
    //удаляем бывший головной элемент
    delete temp;
}
```



```
void List::DelAll()
{
    //Пока еще есть элементы
    while(Head != 0)
        //Удаляем элементы по одному
        Del();
}

void List::Print()
{
   //запоминаем адрес головного элемента
    Element * temp = Head;

    //Пока еще есть элементы
    while(temp != 0)
    {
        //Выводим данные
        cout << temp->data << " ";
        //Переходим на следующий элемент
        temp = temp->Next;
    }

    cout << "\n\n";
}

//Тестовый пример
void main()
{
    //Создаем объект класса
    List List lst;

    //Тестовая строка
    char s[] = "Hello, World !!!\n";
    //Выводим строку
    cout << s << "\n\n";
    //Определяем длину строки
    int len = strlen(s);
```

```
    //Загоняем строку в список
    for(int i = 0; i < len; i++)
        lst.Add(s[i]);
    //Распечатываем содержимое списка
    lst.Print();
    //Удаляем три элемента списка
    lst.Del();
    lst.Del();
    lst.Del();
    //Распечатываем содержимое списка
    lst.Print();
}
```


Двусвязный список

Вы уже знаете, что есть односвязный список и, поэтому вам будет просто понять принцип работы двусвязного списка. Отличие данной структуры от предыдущей состоит в том, что в двусвязном (или двунаправленном списке) узел состоит не из двух, а из трех частей. В третьем компоненте хранится указатель на предыдущий элемент.

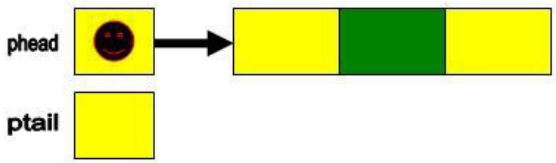
```
//узел (звено) списка
struct node
{
    //Информационный элемент звена списка
    int value;

    //Указатель на предыдущее звено списка
    node *prev;

    //Указатель на следующее звено списка
    node *next;
};
```

Такое построение позволяет производить движение по списку, как в прямом, так и в обратном направлении.

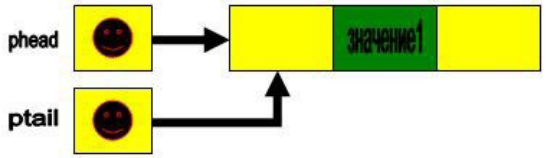
Рассмотрим на иллюстрациях основные действия над двусвязным списком.

Формирование двусвязного списка

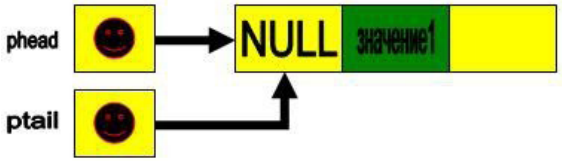
1. Отведем место для указателей в статической памяти и зарезервируем место для динамического объекта.



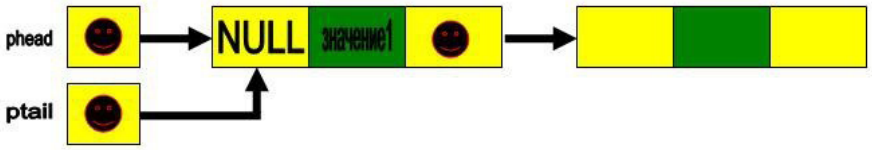
2. Присвоим значение переменной ptail, и поместим в информационное поле значение элемента.



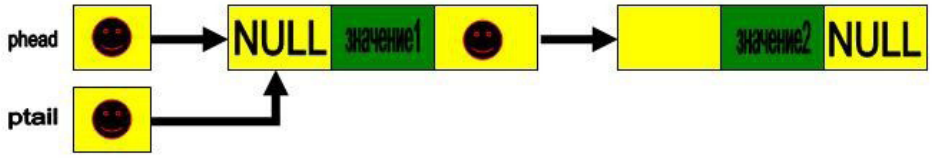
3. Присвоим указателю на предыдущий элемент значение NULL (т. к. элемент первый — предыдущего нет).



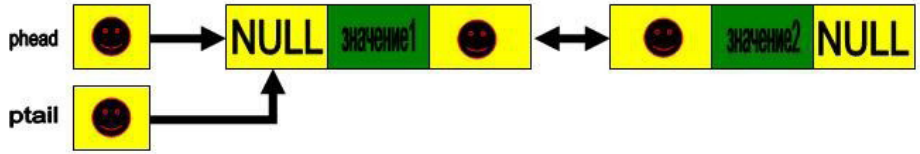
4. Поместим в поле звена адрес еще одного — нового динамического объекта.



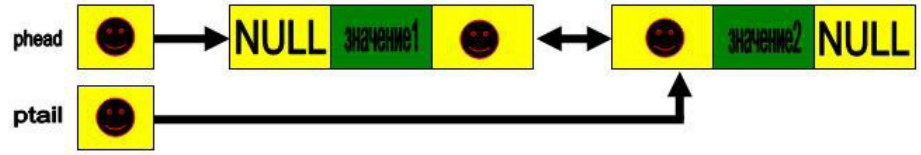
5. В новый добавленный объект записываем значение, в указатель на следующее звено записываем NULL, т.к. объект добавляется в конец.



6. В указатель на предыдущий элемент записываем адрес предыдущего объекта.



7. Переменная ptail должна содержать адрес последнего добавленного элемента, т.к. он добавлен в конец.

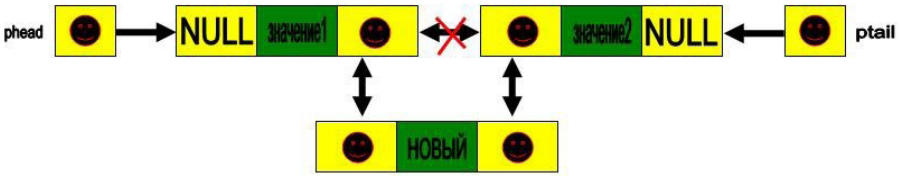


Двусвязный список из двух элементов готов.

Вставка элемента в двусвязный список

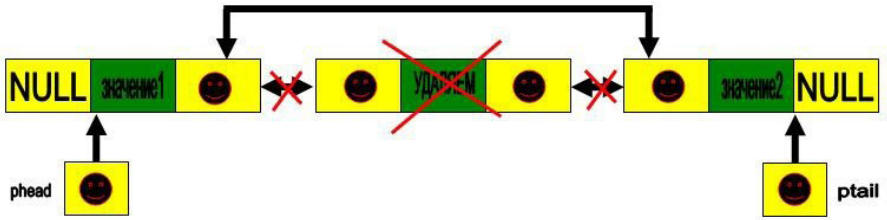
1. Выделить память под новый узел.
2. Записать в новый узел значение.
3. В указатель на предыдущий узел записать адрес узла, который должен располагаться перед новым узлом.

4. Записать в указатель на следующий узел адрес узла, который должен быть расположен после нового узла.
5. В предыдущем узле заменяем значение указателя на следующий узел на адрес нового узла.
6. В следующем узле заменяем значение указателя на предыдущий узел на адрес нового узла.



Удаление элемента из двусвязного списка

1. Записать адрес узла, следующего за удаляемым узлом, в указатель на следующий узел узла, являющегося предыдущим для удаляемого узла.
2. Записать адрес узла, являющегося предыдущим для удаляемого, в указатель на предыдущий узел узла, следующего за удаляемым узлом.
3. Удалить узел, предназначенный для удаления.



Вот и всё — а теперь переходим к следующему разделу урока и рассматриваем пример.

Реализация — двусвязный список

```
#include <iostream>
using namespace std;

struct Elem
{
    int data; //данные
    Elem * next, * prev;
};

class List
{
    //Голова, хвост
    Elem * Head, * Tail;
    //Количество элементов
    int Count;

public:
    //Конструктор
    List();
    //Конструктор копирования
    List(const List&);
    //Деструктор
    ~List();

    //Получить количество
    int GetCount();
    //Получить элемент списка
    Elem* GetElem(int);

    //Удалить весь список
    void DelAll();
    //Удаление элемента, если параметр
    //не указывается,
    //то функция его запрашивает
    void Del(int pos = 0);
```

```
    //Вставка элемента, если параметр не указывается,
    //то функция его запрашивает
    void Insert(int pos = 0);

    //Добавление в конец списка
    void AddTail(int n);

    //Добавление в начало списка
    void AddHead(int n);

    //Печать списка
    void Print();
    //Печать определенного элемента
    void Print(int pos);

    List& operator = (const List&);
    //сложение двух списков (дописывание)
    List operator + (const List&);

    //сравнение по элементам
    bool operator == (const List&);
    bool operator != (const List&);
    bool operator <= (const List&);
    bool operator >= (const List&);
    bool operator < (const List&);
    bool operator > (const List&);

    //переворачивание списка
    List operator - ();
};

List::List()
{
    //Изначально список пуст
    Head = Tail = NULL;
    Count = 0;
}
```



```
List::List(const List & L)
{
    Head = Tail = NULL;
    Count = 0;

    //Голова списка, из которого копируем
    Elem * temp = L.Head;
    //Пока не конец списка
    while(temp != 0)
    {
        //Передираем данные
        AddTail(temp->data);
        temp = temp->next;
    }
}

List::~List()
{
    //Удаляем все элементы
    DelAll();
}

void List::AddHead(int n)
{
    //новый элемент
    Elem * temp = new Elem;

    //Предыдущего нет
    temp->prev = 0;
    //Заполняем данные
    temp->data = n;
    //Следующий - бывшая голова
    temp->next = Head;

    //Если элементы есть?
    if(Head != 0)
        Head->prev = temp;
```

```
    //Если элемент первый, то он одновременно
    //и голова и хвост
    if(Count == 0)
        Head = Tail = temp;
    else
        //иначе новый элемент - головной
        Head = temp;

    Count++;
}

void List::AddTail(int n)
{
    //Создаем новый элемент
    Elem * temp = new Elem;
    //Следующего нет
    temp->next = 0;
    //Заполняем данные
    temp->data = n;
    //Предыдущий - бывший хвост
    temp->prev = Tail;

    //Если элементы есть?
    if(Tail != 0)
        Tail->next = temp;

    //Если элемент первый, то он одновременно
    //и голова и хвост
    if(Count == 0)
        Head = Tail = temp;
    else
        //иначе новый элемент - хвостовой
    Tail = temp;

    Count++;
}

void List::Insert(int pos)
{
```



```
    //если параметр отсутствует или равен 0,
    //то запрашиваем его
    if(pos == 0)
    {
        cout << "Input position: ";
        cin >> pos;
    }

    //Позиция от 1 до Count?
    if(pos < 1 || pos > Count + 1)
    {
        //Неверная позиция
        cout << "Incorrect position !!!\n";
        return;
    }

    //Если вставка в конец списка
    if(pos == Count + 1)
    {
        //Вставляемые данные
        int data;
        cout << "Input new number: ";
        cin >> data;
        //Добавление в конец списка
        AddTail(data);
        return;
    }
    else if(pos == 1)
    {
        //Вставляемые данные
        int data;
        cout << "Input new number: ";
        cin >> data;
        //Добавление в начало списка
        AddHead(data);
        return;
    }
```

```
    //Счетчик
    int i = 1;
    //Отсчитываем от головы n - 1 элементов
    Elem * Ins = Head;

    while(i < pos)
    {
        //Доходим до элемента,
        //перед которым вставляемся
        Ins = Ins->next;
        i++;
    }

    //Доходим до элемента,
    //который предшествует
    Elem * PrevIns = Ins->prev;

    //Создаем новый элемент
    Elem * temp = new Elem;

    //Вводим данные
    cout << "Input new number: ";
    cin >> temp->data;

    //настройка связей
    if(PrevIns != 0 && Count != 1)
        PrevIns->next = temp;

    temp->next = Ins;
    temp->prev = PrevIns;
    Ins->prev = temp;

    Count++;
}

void List::Del(int pos)
{
```



```
    //если параметр отсутствует или равен 0,
    //то запрашиваем его
    if(pos == 0)
    {
        cout << "Input position: ";
        cin >> pos;
    }
    //Позиция от 1 до Count?
    if(pos < 1 || pos > Count)
    {
        //Неверная позиция
        cout << "Incorrect position !!!\n";
        return;
    }

    //Счетчик
    int i = 1;

    Elem * Del = Head;

    while(i < pos)
    {
        //Доходим до элемента,
        //который удаляется
        Del = Del->next;
        i++;
    }

    //Доходим до элемента,
    //который предшествует удаляемому
    Elem * PrevDel = Del->prev;

    //Доходим до элемента, который следует за удаляемым
    Elem * AfterDel = Del->next;

    //Если удаляем не голову
    if(PrevDel != 0 && Count != 1)
```

```
        PrevDel->next = AfterDel;
    //Если удаляем не хвост
    if(AfterDel != 0 && Count != 1)
        AfterDel->prev = PrevDel;

    //Удаляются крайние?
    if(pos == 1)
        Head = AfterDel;
    if(pos == Count)
        Tail = PrevDel;

    //Удаление элемента
    delete Del;

    Count--;
}

void List::Print(int pos)
{
    //Позиция от 1 до Count?
    if(pos < 1 || pos > Count)
    {
        //Неверная позиция
        cout << "Incorrect position !!!\n";
        return;
    }

    Elem * temp;

    //Определяем с какой стороны
    //быстрее двигаться
    if(pos <= Count / 2)
    {
        //Отсчет с головы
        temp = Head;
        int i = 1;

        while(i < pos)
        {
```



```
            //Двигаемся до нужного элемента
            temp = temp->next;
            i++;
        }
    }
    else
    {
        //Отсчет с хвоста
        temp = Tail;
        int i = 1;

        while(i <= Count - pos)
        {
            //Двигаемся до нужного элемента
            temp = temp->prev;
            i++;
        }
    }
    //Вывод элемента
    cout << pos << " element: ";
    cout << temp->data << endl;
}

void List::Print()
{
    //Если в списке присутствуют элементы,
    //то пробегаем по нему
    //и печатаем элементы, начиная с головного
    if(Count != 0)
    {
        Elem * temp = Head;
        cout << "( ";
        while(temp->next != 0)
        {
            cout << temp->data << ", ";
            temp = temp->next;
        }
        cout << temp->data << " )\n";
    }
}
```

```
void List::DelAll()
{
    //Пока остаются элементы, удаляем по одному с головы
    while(Count != 0)
        Del(1);
}

int List::GetCount()
{
    return Count;
}

Elem * List::GetElem(int pos)
{
    Elem *temp = Head;

    //Позиция от 1 до Count?
    if(pos < 1 || pos > Count)
    {
        //Неверная позиция
        cout << "Incorrect position !!!\n";
        return 0;
    }

    int i = 1;
    //Ищем нужный нам элемент
    while(i < pos && temp != 0)
    {
        temp = temp->next;
        i++;
    }

    if(temp == 0)
        return 0;
    else
        return temp;
}
```



```
List & List::operator = (const List & L)
{
    //Проверка присваивания элемента "самому себе"
    if(this == &L)
        return *this;

    //удаление старого списка
    this->~List(); // DelAll();

    Elem * temp = L.Head;

    //Копируем элементы
    while(temp != 0)
    {
        AddTail(temp->data);
        temp = temp->next;
    }

    return *this;
}

//сложение двух списков
List List::operator + (const List& L)
{
    //Заносим во временный список элементы первого
    //списка
    List Result(*this);
    //List Result = *this;
    Elem * temp = L.Head;

    //Добавляем во временный список элементы
    //второго списка
    while(temp != 0)
    {
        Result.AddTail(temp->data);
        temp = temp->next;
    }
```

```
    return Result;
}

bool List::operator == (const List& L)
{
    //Сравнение по количеству
    if(Count != L.Count)
        return false;

    Elem *t1, *t2;

    t1 = Head;
    t2 = L.Head;

    //Сравнение по содержанию
    while(t1 != 0)
    {
        //Сверяем данные, которые
        //находятся на одинаковых позициях
        if(t1->data != t2->data)
            return false;

        t1 = t1->next;
        t2 = t2->next;
    }

    return true;
}

bool List::operator != (const List& L)
{
    //Используем предыдущую функцию сравнения
    return !(*this == L);
}

bool List::operator >= (const List& L)
{
    //Сравнение по количеству
    if(Count > L.Count)
        return true;
```



```
    //Сравнение по содержанию
    if(*this == L)
        return true;

    return false;
}

bool List::operator <= (const List& L)
{
    //Сравнение по количеству
    if(Count < L.Count)
        return true;
    //Сравнение по содержанию
    if(*this == L)
        return true;

    return false;
}

bool List::operator > (const List& L)
{
    if(Count > L.Count)
        return true;

    return false;
}

bool List::operator < (const List& L)
{
    if(Count < L.Count)
        return true;

    return false;
}

//переворот
List List::operator - ()
{
```

```
    List Result;

    Elem * temp = Head;
    //Копируем элементы списка, начиная с головного,
    //в свой путем добавления элементов в голову, таким
    //образом временный список Result будет содержать
    //элементы в обратном порядке
    while(temp != 0)
    {
        Result.AddHead(temp->data);
        temp = temp->next;
    }

    return Result;
}

//Тестовый пример
void main()
{
    List L;

    const int n = 10;
    int a[n] = {0,1,2,3,4,5,6,7,8,9};
    //Добавляем элементы, стоящие на четных
    //индексах, в голову,
    //на нечетных - в хвост
    for(int i = 0; i < n; i++)
        if(i % 2 == 0)
            L.AddHead(a[i]);
        else
            L.AddTail(a[i]);

    //Распечатка списка
    cout << "List L:\n";
    L.Print();

    cout << endl;

    //Вставка элемента в список
    L.Insert();
```



```
    //Распечатка списка
    cout << "List L:\n"; L.Print();

    //Распечатка 2-го и 8-го элементов списка
    L.Print(2);
    L.Print(8);

    List T;

    //Копируем список
    T = L;
    //Распечатка копии
    cout << "List T:\n";
    T.Print();

    //Складываем два списка
    //(первый в перевернутом состоянии)
    cout << "List Sum:\n";
    List Sum = -L + T;
    //Распечатка списка
    Sum.Print();
}
```

Шаблоны классов

Как говориться, и снова — здравствуйте. Мы с вами изучали шаблоны и раньше. Однако, это были шаблоны функций. Что ж, никогда не стоит останавливаться на достигнутом. в C++ существует возможность определить, так называемый, обобщенный класс. Это значит, что Вы можете создать класс, который описывает все используемые в нем функции, но тип данных членов класса задается при создании объектов этого класса. Другими словами, когда Вам необходимо разработать класс, обрабатывающий значения с разными типами данных, то средства лучше шаблонов, вы не найдете.

Общий синтаксис создания шаблона для класса таков:

```
template <class тип_данных> class имя_класса {
    //....описание класса.......
};
```

Примечание: Важно!!! Не стоит забывать, что вместо ключевого слова class для определения типа шаблона, мы, как и ранее, можем использовать typename!!!

Комментарии

- тип_данных — имя типа шаблона, которое в зависимости от ситуации будет замещаться реальным типом данных. Здесь можно определять несколько параметризированных типа данных, разделяя их запятой.
- Внутри определения класса имя шаблона можно использовать в любом месте.

Для создания реализации класса-шаблона используется следующий синтаксис:

Комментарии

- **тип_данных** — имя реального типа данных, который встанет на место типа-шаблона.

Примечание: Важно! Функции-члены шаблонного класса автоматически являются шаблонными. Кроме того, если Вы пишите реализацию этих функций внутри класса, их ненужно объявлять как шаблонные с помощью ключевого слова template.

Пример

```
#include <iostream>
using namespace std;

//параметризованный класс
template <class T> class TestClass {
private:
    //объявим поле tempo
    //какого оно будет типа,
    //это можно будет выяснить ТОЛЬКО во
    //время создания конкретного экземпляра класса
    T tempo;
public:
    TestClass(){tempo=0;}
    //тестируемая функция
    T testFunc();
};

//функция-член класса TestClass
//Так как метод реализован вне класса,
//используем явное упоминание template
template <class T>
```

```
T TestClass<T>::testFunc() {
    //программа выводит на экран количество байт
    //занимаемое переменной tempo, типа T
    cout<<"Type's size is: "<<sizeof(tempo)<<"\n\n";
    return tempo;
}

void main()
{
    //создадим конкретные экземпляры класса TestClass
    //char
    TestClass<char> ClassChar;
    ClassChar.testFunc();
    //int
    TestClass<int> ClassInt;
    ClassInt.testFunc();
    //double
    TestClass<double> ClassDouble;
    ClassDouble.testFunc();
}
```

Реализация шаблонного двусвязного списка

```
#include <iostream>
using namespace std;

template <typename T>
struct Elem
{
    // Любые данные
    T data;
    Elem * next, * prev;
};

template <typename T>
class List
{
```



```
    // Голова хвост
    Elem<T> * Head, * Tail;
    int Count;

public:
    List();
    List(const List&);
    ~List();

    int GetCount();
    Elem<T>* GetElem(int);

    void DelAll();
    void Del(int);
    void Del();

    void AddTail();
    void AddTail(T);

    void AddHead(T);
    void AddHead();

    void Print();
    void Print(int pos);

    List& operator = (const List&);
    List operator + (const List&);

    bool operator == (const List&);
    bool operator != (const List&);
    bool operator <= (const List&);
    bool operator >= (const List&);
    bool operator < (const List&);
    bool operator > (const List&);

    List operator - ();
};
```

```
template <typename T>
List<T>::List()
{
    Head = Tail = 0;
    Count = 0;
}

template <typename T>
List<T>::List(const List & L)
{
    Head = Tail = 0;
    Count = 0;

    Elem<T> * temp = L.Head;
    while(temp != 0)
    {
        AddTail(temp->data);
        temp = temp->next;
    }
}

template <typename T>
List<T>::~List()
{
    DelAll();
}

template <typename T>
Elem<T>* List<T>::GetElem(int pos)
{
    Elem<T> *temp = Head;

    //Позиция от 1 до Count?
    if(pos < 1 || pos > Count)
    {
        //Неверная позиция
        cout << "Incorrect position !!!\n";
```



```
        return;
    }

    int i = 1;
    while(i < pos && temp != 0)
    {
        temp = temp->next;
        i++;
    }

    if(temp == 0)
        return 0;
    else
        return temp;
}

template <typename T>
void List<T>::AddHead()
{
    Elem<T> * temp = new Elem<T>;

    temp->prev = 0;

    int n;
    cout << "Input new number: ";
    cin >> n;

    temp->data = n;
    temp->next = Head;

    if(Head != 0)
        Head->prev = temp;

    if(Count == 0)
        Head = Tail = temp;
    else
        Head = temp;
```

```
    Count++;
}

template <typename T>
void List<T>::AddHead(T n)
{
    Elem<T> * temp = new Elem<T>;

    temp->prev = 0;
    temp->data = n;
    temp->next = Head;

    if(Head != 0)
        Head->prev = temp;

    if(Count == 0)
        Head = Tail = temp;
    else
        Head = temp;
    Count++;
}

template <typename T>
void List<T>::AddTail()
{
    Elem<T> * temp = new Elem<T>;

    temp->next = 0;
    int n;
    cout << "Input new number: ";
    cin >> n;

    temp->data = n;
    temp->prev = Tail;

    if(Tail != 0)
        Tail->next = temp;

    if(Count == 0)
```



```
        Head = Tail = temp;
    else
        Tail = temp;

    Count++;
}

template <typename T>
void List<T>::AddTail(T n)
{
    Elem<T> * temp = new Elem<T>;

    temp->next = 0;
    temp->data = n;
    temp->prev = Tail;

    if(Tail != 0)
        Tail->next = temp;

    if(Count == 0)
        Head = Tail = temp;
    else
        Tail = temp;

    Count++;
}

template <typename T>
void List<T>::Del()
{
    int n;
    cout << "Input position: ";
    cin >> n;

    if(n < 1 || n > Count)
    {
        cout << "Incorrect position !!!\n";
        return;
    }
```

```
    int i = 1;
    Elem<T> * Del = Head;

    while(i <= n)
    {
        //Доходим до элемента, который удаляется
        Del = Del->next;
        i++;
    }

    //Доходим до элемента, который предшествует
    //удаляемому
    Elem<T> * PrevDel = Del->prev;
    //Доходим до элемента, который следует за удаляемым
    Elem<T> * AfterDel = Del->next;

    if(PrevDel != 0 && Count != 1)
        PrevDel->next = AfterDel;

    if(AfterDel != 0 && Count != 1)
        AfterDel->prev = PrevDel;

    if(n == 1)
        Head = AfterDel;
    if(n == Count)
        Tail = PrevDel;
    delete Del;

    Count--;
}

template <typename T>
void List<T>::Del(int n)
{
    if(n < 1 || n > Count)
    {
        cout << "Incorrect position !!!\n";
        return;
}
```



```
    int i = 1;
    Elem<T> * Del = Head;

    while(i < n)
    {
       //Доходим до элемента, который удаляется
       Del = Del->next;
       i++;
    }


    //Доходим до элемента, который предшествует
    //удаляемому
    Elem<T> * PrevDel = Del->prev;
    //Доходим до элемента, который следует за
    //удаляемым
    Elem<T> * AfterDel = Del->next;

    if(PrevDel != 0 && Count != 1)
        PrevDel->next = AfterDel;

    if(AfterDel != 0 && Count != 1)
        AfterDel->prev = PrevDel;

    if(n == 1)
        Head = AfterDel;
    if(n == Count)
        Tail = PrevDel;

    delete Del;

    Count--;
}

template <typename T>
void List<T>::Print(int pos)
{
```

```
    //Позиция от 1 до Count?
    if(pos < 1 || pos > Count)
    {
        //Неверная позиция
        cout << "Incorrect position !!!\n";
        return;
    }

    Elem<T> * temp;


    //Определяем с какой стороны
    //быстрее двигаться
    if(pos <= Count / 2)
    {
        //Отсчет с головы
        temp = Head;
        int i = 1;

        while(i < pos)
        {
            // Двигаемся до нужного элемента
            temp = temp->next;
            i++;
        }
    }
    else
    {
        //Отсчет с хвоста
        temp = Tail;
        int i = 1;

        while(i <= Count - pos)
        {
            //Двигаемся до нужного элемента
            temp = temp->prev;
            i++;
        }
    }
```



```
    //Вывод элемента
    cout << pos << " element: ";
    cout << temp->data << "\n";
}

template <typename T>
void List<T>::Print()
{
    if(Count != 0)
    {
        Elem<T> * temp = Head;
        while(temp != 0)
        {
            cout << temp->data << "\n";
            temp = temp->next;
        }
    }
}

template <typename T>
void List<T>::DelAll()
{
    while(Count != 0)
        Del(1);
}

template <typename T>
int List<T>::GetCount()
{
    return Count;
}

template <typename T>
List<T>& List<T>::operator = (const List<T> & L)
{
    if(this == &L)
        return *this;
```

```
    this->~List();
    Elem<T> * temp = L.Head;
    while(temp != 0)
    {
       AddTail(temp->data);
       temp = temp->next;
    }

    return *this;
}

template <typename T>
List<T> List<T>::operator + (const List<T>& L)
{
    List Result(*this);

    Elem<T> * temp = L.Head;

    while(temp != 0)
    {
       Result.AddTail(temp->data);
       temp = temp->next;
    }

    return Result;
}

template <typename T>
bool List<T>::operator == (const List<T>& L)
{
    if(Count != L.Count)
       return false;

    Elem<T> *t1, *t2;

    t1 = Head;
    t2 = L.Head;
```



```
    while(t1 != 0)
    {
        if(t1->data != t2->data)
            return false;

        t1 = t1->next;
        t2 = t2->next;
    }

    return true;
}

template <typename T>
bool List<T>::operator != (const List& L)
{
    if(Count != L.Count)
        return true;
    Elem<T> *t1, *t2;
    t1 = Head;
    t2 = L.Head;

    while(t1 != 0)
    {
        if(t1->data != t2->data)
            return true;

        t1 = t1->next;
        t2 = t2->next;
    }

    return false;
}

template <typename T>
bool List<T>::operator >= (const List& L)
{
    if(Count > L.Count)
        return true;
```

```
    if(*this == L)
        return true;

    return false;
}

template <typename T>
bool List<T>::operator <= (const List& L)
{
    if(Count < L.Count)
        return true;

    if(*this == L)
        return true;

    return false;
}

template <typename T>
bool List<T>::operator > (const List& L)
{
    if(Count > L.Count)
        return true;

    return false;
}

template <typename T>
bool List<T>::operator < (const List& L)
{
    if(Count < L.Count)
        return true;

    return false;
}
```



```
template <typename T>
List<T> List<T>::operator - ()
{
    List Result;
    Elem<T> * temp = Head;

    while(temp != 0)
    {
        Result.AddHead(temp->data);
        temp = temp->next;
    }

    return Result;
}

//Тестовый пример
void main()
{
    List <int> L;

    const int n = 10;
    int a[n] = {0,1,2,3,4,5,6,7,8,9};

    //Добавляем элементы, стоящие на четных
    //индексах, в голову, на нечетных - в хвост
    for(int i = 0; i < n; i++)
        if(i % 2 == 0)
            L.AddHead(a[i]);
        else
            L.AddTail(a[i]);

    //Распечатка списка
    cout << "List L:\n";
    L.Print();

    cout << "\n\n";
```

```
    //Распечатка списка
    cout << "List L:\n";
    L.Print();

    //Распечатка 2-го и 8-го элементов списка
    L.Print(2);
    L.Print(8);

    List <int> T;

    //Копируем список
    T = L;
    //Распечатка копии
    cout << "List T:\n";
    T.Print();

    //Складываем два списка
    //(первый в перевернутом состоянии)
    cout << "List Sum:\n";
    List <int> Sum = -L + T;
    //Распечатка списка
    Sum.Print();
}
```


Домашнее задание

Добавить в класс "Односвязный список" следующие функции: вставка элемента в заданную позицию, удаление элемента по заданной позиции, поиск заданного элемента (функция возвращает позицию найденного элемента в случае успеха или NULL в случае неудачи).

1. Реализовать шаблонный класс "Очередь" на основе двусвязного списка.
2. Создать шаблонный класс-контейнер Array, который представляет собой массив, позволяющий хранить объекты заданного типа. Класс должен реализовывать следующие функции:
   A) **GetSize** — получение размера массива (количество элементов, под которые выделена память);
   B) **SetSize(int size, int grow = 1)** — установка размера массива (если параметр size больше предыдущего размера массива, то выделяется дополнительный блок памяти, если нет, то "лишние" элементы теряются и память освобождается); параметр grow определяет для какого количества элементов необходимо выделить память, если количество элементов превосходит текущий размер массива. Например, SetSize(5, 5); означает, что при добавлении 6-го элемента размер массива становится равным 10, при добавлении 11-го — 15 и т. д.;

C) **GetUpperBound** — получение последнего допустимого индекса в массиве. Например, если при размере массива 10, вы добавляете в него 4 элемента, то функция вернет 3;

D) **IsEmpty** — массив пуст?

E) **FreeExtra** — удалить "лишнюю" память (выше последнего допустимого индекса);

F) **RemoveAll** — удалить все;

G) **GetAt** — получение определенного элемента (по индексу);

H) **SetAt** — установка нового значения для определенного элемента (индекс элемента должен быть меньше текущего размера массива);

I) **operator []** — для реализации двух предыдущих функций;

J) **Add** — добавление элемента в массив (при необходимости массив увеличивается на значение grow функции SetSize);

K) **Append** — "сложение" двух массивов;

L) **operator** =;

M) **GetData** — получения адреса массива с данными;

N) **InsertAt** — вставка элемента(-ов) в заданную позицию O.RemoveAt — удаление элемента(-ов) с заданной позиции.