Этапы развития программирования

- «Блюдо спагетти»
- Процедурное программирование
- Структурное программирование
- Объектное программирование
- Объектно-ориентированное программирование

Предпосылки для появления структурного программирования

- Необходимость группировки переменных в составные типы
- Необходимость оперирования высокоуровневыми понятиями предметной области

Объявление структуры

Структура – составной тип, который включает в себя несколько полей других типов

```
struct Circle                    //объявляем новый тип
{
        double x, y;             //поля структуры
        double radius;
        int color;
};      //точка с запятой должна присутствовать!

int main (void)
{
        Circle c;                //заводим переменную типа Circle
        c.x = 10; c.y = 20.5; /*обращение к полям структуры,
                                  c.x – это переменная!*/
        return 0;
}
```

Использование структур

```
struct Circle
{
        double x, y;
        double radius;      //double radius = 1; – ошибка
        int color;
};

int main (void)
{
        Circle c;
        c.x = 10; c.y = 20.5;
        c.radius = 2; c.color = 0;
        Circle c1 = c;      //одну структуру можно инициализировать другой
        c = c1;             //и присваивать
        //Circle c2 = c + c1; - ошибка  - сложение не определено
        return 0;
}
```

Инициализация структур

```
struct Circle
{
        double x, y;
        double radius;
        int color;
};

Circle initCircle (void) //Можно возвращать структуру из функции
{
        Circle circle;
        circle.x = 0; circle.y = 0;
        circle.radius = 1; circle.color = 0;
        return circle;
}

int main (void)
{
        Circle c = initCircle ();
        return 0;
}
```

Указатели на структуры

```
struct Circle
{
        double x, y;
        double radius;
        int color;
};

void initCircle (Circle * circle) //получает указатель на структуру
{
        (*circle).x = 0; (*circle).y = 0;
        (*circle).radius = 1; (*circle).color = 0;
}

int main (void)
{
        Circle c;
        initCircle (&c);
        return 0;
}
```

Указатели на структуры

```
struct Circle
{
        double x, y;
        double radius;
        int color;
};

void initCircle (Circle * circle)
{
        circle -> x = 0; circle -> y = 0;
        circle -> radius = 1; circle -> color = 0;
}

int main (void)
{
        Circle c;
        initCircle (&c);
        return 0;
}
```

Использование структур

- Объявление структуры – это объявление составного типа данных
- Поля структуры не являются переменными и не могут быть инициализированы при ее объявлении
- Размер (sizeof) структуры не менее, чем сумма размеров полей

Операции над структурами

```
lvalue = expr          //определено присваивание и
                       //инициализация

expr . field           //обращение к полю структуры

ptr -> field           //обращение к полю структуры
                       //через указатель на нее
```

Контейнеры данных

Контейнер данных – сущность которая содержит в себе набор однотипных данных, и позволяет совершать над ними следующие операции:

1. Вставка элемента
2. Удаление элемента
3. Обращение к элементу (получение и изменение значения)

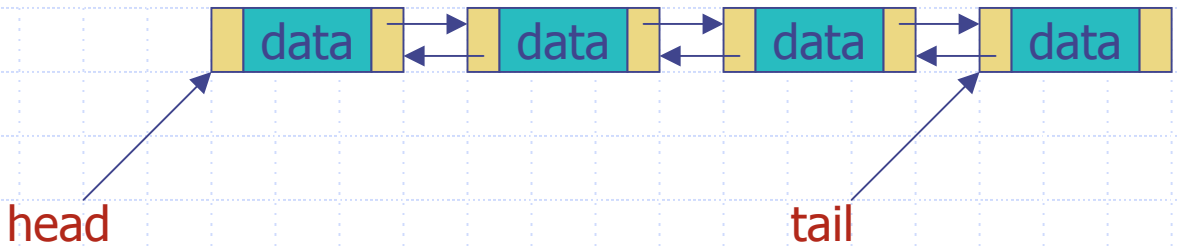
Контейнеры различаются способами доступа к элементам, и время выполнения операции (измеряется количеством операций присваивания и сравнения элементов)

Контейнеры данных

Массив: доступ к элементам осуществляется через индексы и время доступа не зависит от размера массива. Вставка и удаление элементов эквивалентна пересозданию массива (время пропорционально размеру)

Стек: доступ осуществляется только к последнему вставленному элементу. Время вставки, удаления и доступа не зависит от размера массива.

Список (list)



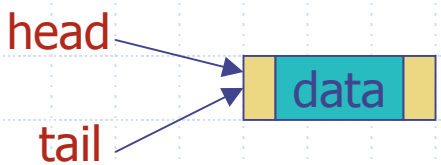
Каждый элемент содержит в себе данные, а также указатели на предыдущий и последующий элементы

Объявление типа «список»

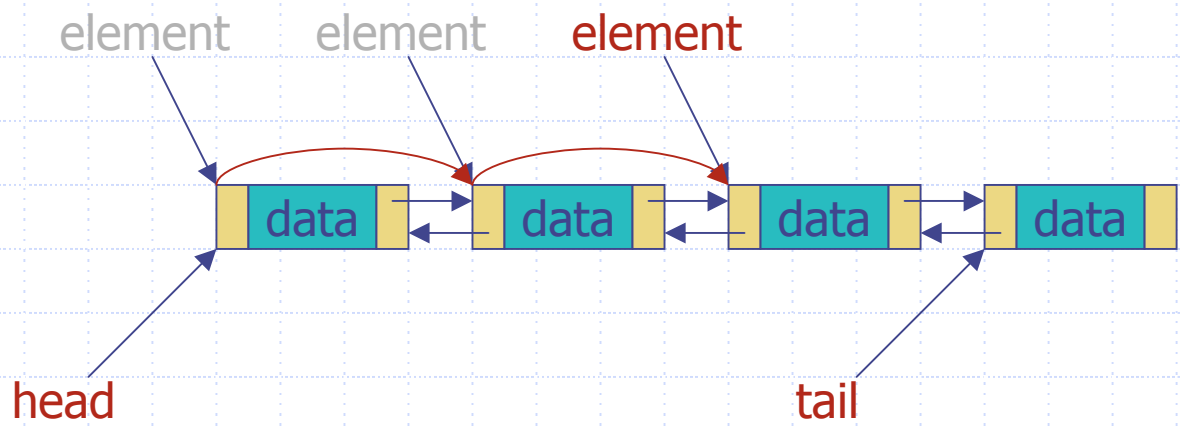
```
typedef double ListData;    //тип хранимых данных - double

struct List
{
        ListData data;          //хранимые данные
        List * prev;    //указатель на предыдущий элемент
        List * next;    //указатель на следующий элемент
};

int main (void)
{
        List * head = 0, * tail = 0;
        return 0;
}
```

Создание списка

```
...
int main (void)
{
        List * head = 0, * tail = 0;
        head = new List;
        head.prev = 0;          //нет предыдущего элемента
        head.next = 0;          //нет следующего элемента
        head.data = 0;
        tail = head;            //первый элемент является
                                //последним

        return 0;
}
```



Доступ к элементу



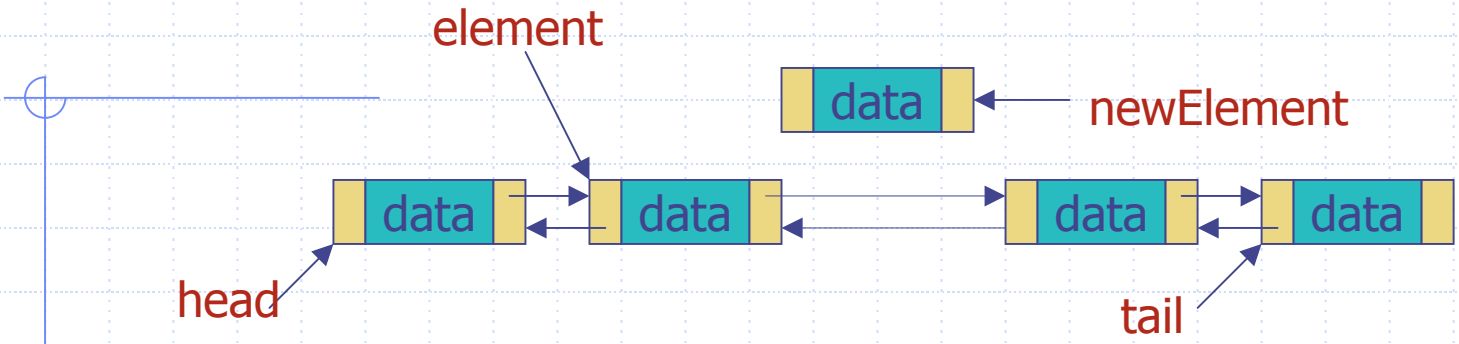
Доступ к элементу

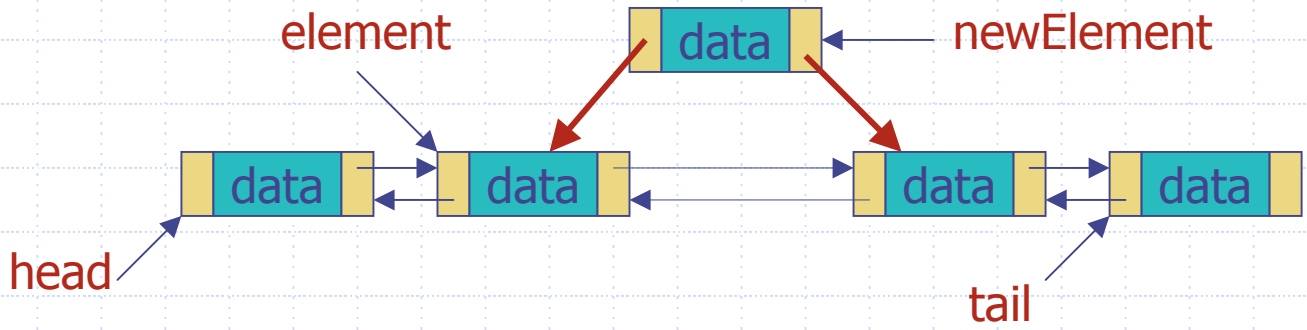
```
List * next (List * elem)
{
        if (elem == 0) return 0;
        return elem -> next;
}

List * prev (List * elem)
{
        if (elem == 0) return 0;
        return elem -> prev;
}
```

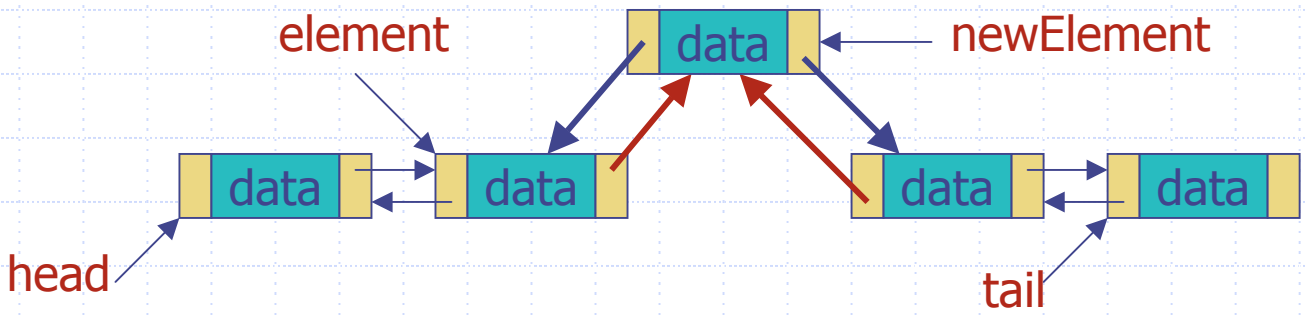
Доступ к элементу по индексу

```
List * element (List * head, unsigned index)
{
        List * result = head;          //текущий элемент
        /*в цикле index изменяется и показывает
        расстояние от текущего элемента до нужного*/
        for (;index > 0; --index)
                //переход к следующему элементу
                result = next (result);

        return result;
}
```

Добавление элемента







Добавление элемента

```
List * insertAfter (List * element, ListData value)
{
        List * newElement = new List;        //создаем элемент
        newElement -> data = value;          //инициализируем хранимые данные
        newElement->prev = element;          //связываем с предыдущим
        if (element != 0)                    //есть ли предыдущий?
        {
                //связываем со следующим
                newElement -> next = element -> next;
                //встраиваем в список
                element -> next = newElement;
                if (newElement -> next != 0)
                        newElement -> next -> prev = newElement;
        }
        else
        {
                 newElement->next = 0;
        }
        return newElement;
}
```

Удаление элемента

```
List * removeElement (List * element){
        if (!element) return 0;
        List * next = element -> next;
        //исключаем элемент из списка
        if (element -> prev)
                element -> prev -> next = element -> next;
        if (element -> next)
                element -> next -> prev = element -> prev;
        //освобождаем память
        delete element;
        return next;
}
```

```
void removeList (List * head){
        while (head != 0)
                head = removeElement (head);
}
```

Использование списка

```
void printList (List * head)
{
        List *elem;
        printf ("list\n");
        for (elem = head; elem; elem = next(elem))
                printf ("\tdata: %lg\n", elem->data);
        printf ("end_list\n");
}
int main (void)
{
    List * head = 0, * tail = 0;
    head = insertAfter (0, 1);          //создаем элемент
    insertAfter (head, 2);              //добавляем элемент в конец
    insertAfter (head, 3);              //добавляем в середину
    printList (head);
    element (head, 1) -> data = 10; //изменяем средний элемент
    printList (head);
    removeElement (element (head, 1);//удаляем средний элемент
    printList (head);
    removeList (head);                  //удаление списка
    return 0;
}
```

```
list
        data: 1
        data: 3
        data: 2
end_list
list
        data: 1
        data: 10
        data: 2
end_list
list
        data: 1
        data: 2
end_list
```

Характеристики списка

Время доступа к элементу пропорционально размеру

Время вставки и удаления элемента не зависит от размера

Если количество элементов практически не меняется, но необходимо часто обращаться к произвольным элементам, то целесообразно использовать массив (вектор)

Если необходимо делать много вставок и удалений, то целесообразно использовать список

Очередь

Очередь (queue, FIFO – First In First Out) – контейнер данных, для которого определяются понятия начала и конца а также операции добавления элемента в конец и удаления элемента из начала

Работа с очередью

```
List * pushElement (List * tail, ListElement value)
{
        return insertAfter (tail, value);
}

List * popElement (List * head)
{
        return removeElement (head);
}

int main (void)
{
        List * head = 0, * tail = 0;
        tail = pushElement (tail, 1);          //вставляем элемент
        head = tail;
        tail = pushElement (tail, 2);          //вставляем еще один элемент
        double val = head -> data;             //обращаемся к первому элементу
        head = popElement (head);              //удаляем первый элемент
        return 0;
}
```

Двусторонняя очередь

Двусторонняя очередь (deque, double-ended queue) – контейнер данных, для которого определяются понятия начала и конца, а также операции добавления элемента в начало и конец и удаления элемента из начала и конца