

14.10.2024

# Структуры – продолжение. Связанные СПИСКИ

*Филиппов Михаил Витальевич*

[m.filippov@g.nsu.ru](mailto:m.filippov@g.nsu.ru)

89232283872

Императивное программирование, 2024-2025

**N** \* Новосибирский  
государственный  
университет  
**\*НАСТОЯЩАЯ НАУКА**

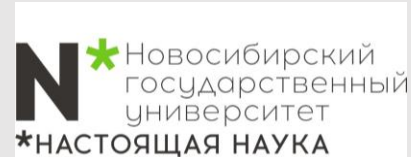


# Давайте познакомимся



## Филиппов Михаил Витальевич

- Окончил магистратуру ФФ НГУ
- Окончил аспирантуру ИТ СО РАН
- Являюсь м.н.с. ИТ СО РАН
- 7+ лет опыт в программировании C/C++



# План лекции

**Структуры –  
продолжение**

**45 минут**

**Связанные  
списки**

**45 минут**

# План лекции

**Структуры –  
продолжение**

**45 минут**

**Связанные  
списки**

**45 минут**

# Массивы структур – Демо ПК

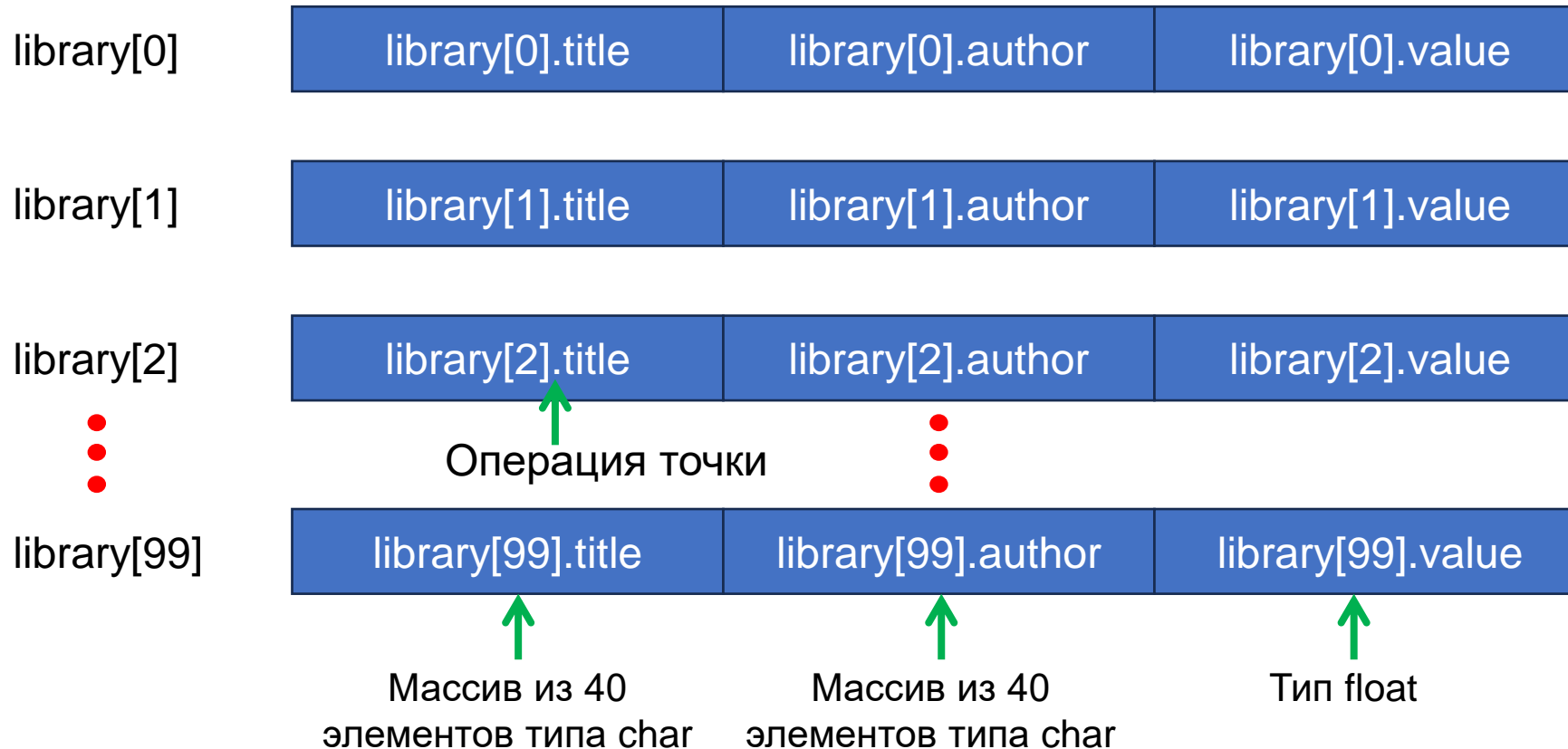
```
// Введите название книги.
// Нажмите [enter] в начале строки, чтобы закончить ввод.
// My Life is a Budgie
// Теперь введите ФИО автора.
// Mack Zackles
// Теперь введите цену.
// 12.95
// Введите название следующей книги.
// Thought and Unthought Rethought
// Теперь введите ФИО автора.
// Kindra Schlagmeyer
// Теперь введите цену.
// 43.50
// Введите название следующей книги.
// Concetro for Financial Instruments
// Теперь введите ФИО автора.
// Tilmore Walletz
// Теперь введите цену.
// 49.99
// Введите название следующей книги.
// The CEO Power Diet
// Теперь введите ФИО автора.
// Buster Downsize
// Теперь введите цену.
// 19.25
// Введите название следующей книги.
// C++ Primer Plus
// Теперь введите ФИО автора.
// Stephen Prata
// Теперь введите цену.
// 59.99
// Введите название следующей книги.

// Каталог ваших книг:
// My Life is a Budgie авторства Mack Zackles: $12.95
// Thought and Unthought Rethought авторства Kindra Schlagmeyer: $43.50
// Concetro for Financial Instruments авторства Tilmore Walletz: $49.99
// The CEO Power Diet авторства Buster Downsize: $19.25
// C++ Primer Plus авторства Stephen Prata: $59.99
```

```
/* manybook.c -- каталог для нескольких книг */
#include <stdio.h>
#include <string.h>
char *s_gets(char *st, int n);
#define MAXTITL 40
#define MAXAUTL 40
#define MAXBKS 100 /* максимальное количество книг */
struct book /* установка шаблона book */
{
    char title[MAXTITL];
    char author[MAXAUTL];
    float value;
};
int main(void)
{
    system("chcp 65001");
    struct book library[MAXBKS]; /* массив структур типа book */
    int count = 0;
    int index;
    printf("Введите название книги.\n");
    printf("Нажмите [enter] в начале строки, чтобы закончить ввод.\n");
    while (count < MAXBKS && s_gets(library[count].title, MAXTITL) != NULL && library[count].title[0]
    != '\0')
    {
        printf("Теперь введите ФИО автора.\n");
        s_gets(library[count].author, MAXAUTL);
        printf("Теперь введите цену.\n");
        scanf("%f", &library[count++].value);
        while (getchar() != '\n')
            continue; /* очистить входную строку */
        if (count < MAXBKS)
            printf("Введите название следующей книги.\n");
    }
    if (count > 0)
    {
        printf("Каталог ваших книг:\n");
        for (index = 0; index < count; index++)
            printf("%s авторства %s: $%.2f\n", library[index].title,
                library[index].author, library[index].value);
    }
    else
        printf("Вообще нет книг? Очень плохо.\n");
    return 0;
}
char *s_gets(char *st, int n)
{
    char *ret_val;
    char *find;
    ret_val = fgets(st, n, stdin);
    if (ret_val)
    {
        find = strchr(st, '\n'); // поиск новой строки
        if (find) // если адрес не равен NULL,
            *find = '\0'; // поместить туда нулевой символ
        else
            while (getchar() != '\n')
                continue; // отбросить остаток строки
    }
    return ret_val;
}
```

# Объявление массива структур

```
struct book library [MAXBKS];
```





# Идентификация членов в массиве структур

Для идентификации членов в массиве структур применяются те же самые правила, что и в случае индивидуальных структур: за именем структуры должна следовать операция точки, а затем имя члена.

Например:

```
library[0].value /* значение value, ассоциированное с первым  
элементом массива */
```

```
library[4].title /* значение title, ассоциированное с пятым элементом  
массива */
```

```
library.value[2] // НЕПРАВИЛЬНО
```

```
library[2].value // ПРАВИЛЬНО
```

```
library // массив структур типа book
```

```
library[2] // элемент массива, т.е. структура book
```

```
library[2].title // символьный массив (член title элемента library [2])
```

```
library[2].title[4] // символ в массиве члена title
```



# Вложенные структур – Демо ПК

```
// Дорогой Билли,  
  
// Благодарю вас за чудесно проведенный вечер, Билли.  
// Вы однозначно продемонстрировали, что персональн  
// являет собою особый тип мужчины. Мы обязательно должны встретиться  
// за восхитительным уж:ином с запеченнымперсональн и весело провести время..  
  
//                                     До скорой встречи,  
//                                     Шейла
```

```
// friend.c -- пример вложенной структуры  
#include <stdio.h>  
#define LEN 20  
const char *msgs[5] =  
{  
    " Благодарю вас за чудесно проведенный вечер, ",  
    "Вы однозначно продемонстрировали, что ",  
    "являет собою особый тип мужчины. Мы обязательно должны  
встретиться",  
    "за восхитительным уж:ином с ",  
    " и весело провести время."};  
struct names // первая структура  
{  
    char first[LEN];  
    char last[LEN];  
};  
struct guy // вторая структура  
{  
    struct names handle; // вложенная структура  
    char favfood[LEN];  
    char job[LEN];  
    float income;  
};  
int main(void)  
{  
    system("chcp 65001");  
    struct guy fellow = { // инициализация переменной  
        {"Билли", "Бонс"},  
        "запеченными омарами",  
        "персональный тренер",  
        68112.00};  
    printf("Дорогой %s, \n\n", fellow.handle.first);  
    printf("%s%s.\n", msgs[0], fellow.handle.first);  
    printf("%s%s\n", msgs[1], fellow.job);  
    printf("%s\n", msgs[2]);  
    printf("%s%s", msgs[3], fellow.favfood, msgs[4]);  
    if (fellow.income > 150000.0)  
        puts("!!");  
    else if (fellow.income > 75000.0)  
        puts("!");  
    else  
        puts(".");  
    printf("\n%40s\n", " ", "До скорой встречи,");  
    printf("%40s\n", " ", "Шейла");  
    return 0;  
}
```



# Указатели на структуры

```
#include <stdio.h>
#define LEN 20
struct names {
    char first[LEN];
    char last[LEN];
};
struct guy {
    struct names handle;
    char favfood[LEN];
    char job[LEN];
    float income;
};
int main(void)
{
    struct guy fellow[2] = {{{"Билли", "Бонс"}, "запеченными омарами", "персональный тренер", 68112.00},
                             {"Джим", "Хокинс"}, "рыбным фрикасе", "редактор таблоида", 232400.00}};

    struct guy *him; /* указатель на структуру */
    printf("адрес #1: %p #2: %p \n", &fellow[0], &fellow[1]);
    him = &fellow[0]; /* сообщает указателю, на что указывать */
    printf("указатель #1: %p #2: %p\n", him, him + 1);
    printf("him->income равно $%.2f: (*him).income равно $%.2f\n", him->income, (*him).income);
    him++; /* указатель на следующую структуру */
    printf("him->favfood равно %s: him->handle.last равно %s\n", him->favfood, him->handle.last);
    return 0;
}
```

```
// адрес #1: 0x7ffec4442bb0 #2: 0x7ffec4442c04
// указатель #1: 0x7ffec4442bb0 #2: 0x7ffec4442c04
// him->income равно $68112.00: (*him).income равно $68112.00
// him->favfood равно : him->handle.last равно Хокинс
```

# Указатели на структуры

Объявить указатель на структуру очень просто:

```
struct guy * him;
```

Инициализация указателя:

```
him = &fellow[0];
```

Получение значение члена структуры:

```
*him равно fellow[0]
```

```
him->income равно fellow[0].income
```

```
fellow[0].income равно (*him).income
```

```
barney.income равно (*him).income равно him->income
```

```
//предполагая, что him == &barney
```



# Передача членов структуры

```
#include <stdio.h>
#define FUNDLLEN 50
struct funds
{
    char bank[FUNDLLEN];
    double bankfund;
    char save[FUNDLLEN];
    double savefund;
};
double sum(double, double);
int main(void)
{
    struct funds stan = {"Garlic-Melon Bank", 4032.27, "Lucky's Savings and Loan", 8543.94};
    printf("Общая сумма на счетах у Стэна составляет $%.2f.\n", sum(stan.bankfund, stan.savefund));
    return 0;
}
double sum(double x, double y)
{ /* суммирование двух чисел типа double */
    return (x + y);
}
// Общая сумма на счетах у Стэна составляет $12576.21.
```

# Использование адреса структуры

```
/* funds2.c -- передача указателя на структуру */
#include <stdio.h>
#define FUNDLLEN 50
struct funds
{
    char bank[FUNDLLEN];
    double bankfund;
    char save[FUNDLLEN];
    double savefund;
};
double sum(const struct funds *); /* аргумент является указателем */
int main(void)
{
    struct funds stan = {"Garlic-Melon Bank", 4032.27, "Lucky's Savings and Loan", 8543.94};
    printf("Общая сумма на счетах у Стэна составляет $%.2f.\n", sum(&stan));
    return 0;
}
double sum(const struct funds *money)
{
    return (money->bankfund + money->savefund);
}
// Общая сумма на счетах у Стэна составляет $12576.21.
```

# Передача в качестве аргумента

```
#include <stdio.h>
#define FUNDLLEN 50
struct funds
{
    char bank[FUNDLLEN];
    double bankfund;
    char save[FUNDLLEN];
    double savefund;
};
double sum(struct funds moolah); /* аргумент является структурой */
int main(void)
{
    struct funds stan = {"Garlic-Melon Bank", 4032.27, "Lucky's Savings and Loan", 8543.94};
    printf("Общая сумма на счетах у Стэна составляет $%.2f.\n", sum(&stan));
    return 0;
}
double sum(struct funds moolah)
{
    return (moolah.bankfund + moolah.savefund);
}
// Общая сумма на счетах у Стэна составляет $12576.21.
```

# Доп. Возможности — демо ПК

```
// Введите свое имя.  
// Mikhail  
// Введите свою фамилию.  
// Philiprov  
// Mikhail Philiprov, ваше имя и фамилия содержат 16 букв.
```

```
/* names1.c -- использует указатели на структуры */  
#include <stdio.h>  
#include <string.h>  
#define NLEN 30  
struct namect  
{  
    char fname[NLEN];  
    char lname[NLEN];  
    int letters;  
};  
void getinfo(struct namect *);  
void makeinfo(struct namect *);  
void showinfo(const struct namect *);  
char *s_gets(char *st, int n);  
int main(void)  
{  
    struct namect person;  
    getinfo(&person);  
    makeinfo(&person);  
    showinfo(&person);  
    return 0;  
}  
void getinfo(struct namect *pst)  
{  
    printf("Введите свое имя.\n");  
    s_gets(pst->fname, NLEN);  
    printf("Введите свою фамилию.\n");  
    s_gets(pst->lname, NLEN);  
}  
void makeinfo(struct namect *pst)  
{  
    pst->letters = strlen(pst->fname) +  
                  strlen(pst->lname);  
}  
void showinfo(const struct namect *pst)  
{  
    printf("%s %s, ваше имя и фамилия содержат %d букв.\n",  
          pst->fname, pst->lname, pst->letters);  
}  
char *s_gets(char *st, int n)  
{  
    char *ret_val;  
    char *find;  
    ret_val = fgets(st, n, stdin);  
    if (ret_val)  
    {  
        find = strchr(st, '\n'); // поиск новой строки  
        if (find)                // если адрес не равен NULL,  
            *find = '\0';        // поместить туда нулевой символ  
        else  
            while (getchar() != '\n')  
                continue; // отбросить остаток строки  
    }  
    return ret_val;  
}
```



# Доп. Возможности – демо ПК 2

```
// Введите свое имя.  
// Mikhail  
// Введите свою фамилию.  
// Philiprov  
// Mikhail Philiprov, ваше имя и фамилия содержат 16 букв.
```

```
#include <stdio.h>  
#include <string.h>  
#define NLEN 30  
struct namect  
{  
    char fname[NLEN];  
    char lname[NLEN];  
    int letters;  
};  
struct namect getinfo(void);  
struct namect makeinfo(struct namect);  
void showinfo(struct namect);  
char *s_gets(char *st, int n);  
int main(void)  
{  
    struct namect person;  
    person = getinfo();  
    person = makeinfo(person);  
    showinfo(person);  
    return 0;  
}  
struct namect getinfo(void)  
{  
    struct namect temp;  
    printf("Введите свое имя.\n");  
    s_gets(temp.fname, NLEN);  
    printf("Введите свою фамилию.\n");  
    s_gets(temp.lname, NLEN);  
    return temp;  
}  
struct namect makeinfo(struct namect info)  
{  
    info.letters = strlen(info.fname) + strlen(info.lname);  
    return info;  
}  
void showinfo(struct namect info)  
{  
    printf("%s %s, ваше имя и фамилия содержат %d букв.\n",  
        info.fname, info.lname, info.letters);  
}  
char *s_gets(char *st, int n)  
{  
    char *ret_val;  
    char *find;  
    ret_val = fgets(st, n, stdin);  
    if (ret_val)  
    {  
        find = strchr(st, '\n');  
        if (find)  
            *find = '\0';  
        else  
            while (getchar() != '\n')  
                continue;  
    }  
    return ret_val;  
}
```

# Структуры или указатели на структуры?

Метод с применением указателей в аргументах:

он работает как в старых, так и в новых реализациях C

- ✓ является быстрым
- ✓ передается всего лишь один адрес.
- ✗ данные менее защищены
- ✗ пользователь должен помнить, каким аргументом должен быть представлен адрес суммы — первым или последним

Метод передачи структур в качестве аргументов:

- ✓ функция имеет дело с копией исходных данных, что безопаснее
- ✓ стиль программирования становится более ясным.
- ✗ старые реализации C могут не воспринимать такой код,
- ✗ неэкономно расходуется время и память

Обычно программисты применяют указатели на структуры в качестве аргументов функции из соображений эффективности, используя `const`, когда необходимо защитить данные от нежелательных изменений. Передача структур по значению чаще всего делается для структур небольших размеров.



# Символьные массивы или указатели на `char` в структурах

Например, в примере имеется следующее объявление:

```
#define LEN 20  
struct names {  
    char first[LEN];  
    char last[LEN];  
};
```

Можно ли вместо этого поступить так?

```
struct pnames {  
    char * first;  
    char * last;  
};
```

Ответ — да, это возможно, но могут возникнуть проблемы, если вы не обдумаете все последствия. Взгляните на показанный ниже код:

```
struct names veep = {"Talia", "Summers"};  
struct pnames treas = ("Brad", "Fallingjaw");  
printf("%s и %s\n", veep.first, treas.first);
```

Этот код допустим, и он работает.



# Символьные массивы или указатели на `char` в структурах

Давайте посмотрим, когда это ограничение превращается в проблему. Взгляните на следующий код:

```
struct names accountant;  
struct pnames attorney;  
puts ("Введите фамилию вашего бухгалтера:");  
scanf("%s", accountant.last);  
puts("Введите фамилию вашего адвоката:");  
scanf("%s", attorney.last) ; /* здесь скрыта опасность */
```

С точки зрения синтаксиса этот код допустим. Но куда сохраняются входные данные? Фамилия бухгалтера записывается в последний член переменной `accountant`; эта структура содержит массив для хранения строки. В случае фамилии адвоката функция `scanf()` получает указание поместить строку фамилии по адресу, заданному как `attorney.last`. Из-за того, что эта переменная не инициализирована, адрес может иметь произвольное значение, и программа может попытаться поместить фамилию куда угодно. Если повезет, то программа будет работать, по крайней мере, некоторое время, либо сразу же аварийно завершится. Однако если программа работает, то вам на самом деле не повезло, т.к. в ней присутствует катастрофическая ошибка, о которой вы не знаете.



# Структура, указатели и malloc()

malloc() позволяет выделить ровно столько памяти, сколько необходимо для строки. Новое определение структуры будет выглядеть следующим образом:

```
struct namect {  
    char * fname; // использование указателей вместо массивов  
    char * lname; int letters;  
};  
  
void getinfo (struct namect * pst) {  
    char temp[SLEN];  
    printf("Введите свое имя. \n");  
    s_gets(temp, SLEN); /  
    / выделение памяти для хранения имени  
    pst->fname = (char *) malloc(strlen(temp) + 1); // копирование имени в  
    выделенную память  
    strcpy(pst->fname, temp);  
    printf("Введите свою фамилию.\n");  
    s_gets(temp, SLEN);  
    pst->lname = (char *)malloc(strlen(temp) + 1);  
    strcpy(pst->lname, temp);  
}
```





# Структура, указатели и malloc() – демо ПК

```
// Введите свое имя.  
// Mikhail  
// Введите свою фамилию.  
// Philiprov  
// Mikhail Philiprov, ваше имя и фамилия содержат 16 букв.
```

```
#include <stdio.h>  
#include <string.h> // для strcpy (), strlen()  
#include <stdlib.h> // для malloc (), free()  
#define SLEN 81  
struct namect  
{  
    char *fname; // использование указателей  
    char *lname;  
    int letters;  
};  
void getinfo(struct namect *); // выделение памяти  
void makeinfo(struct namect *);  
void showinfo(const struct namect *);  
void cleanup(struct namect *); // освобождение памяти, когда она больше не нужна  
char *s_gets(char *st, int n);  
int main(void)  
{  
    struct namect person;  
    getinfo(&person);  
    makeinfo(&person);  
    showinfo(&person);  
    cleanup(&person);  
    return 0;  
}  
void getinfo(struct namect *pst)  
{  
    char temp[SLEN];  
    printf("Введите свое имя.\n");  
    s_gets(temp, SLEN);  
    pst->fname = (char *)malloc(strlen(temp) + 1); // выделение памяти для хранения имени  
    strcpy(pst->fname, temp); // копирование имени в выделенную память  
    printf("Введите свою фамилию.\n");  
    s_gets(temp, SLEN);  
    pst->lname = (char *)malloc(strlen(temp) + 1);  
    strcpy(pst->lname, temp);  
}  
void makeinfo(struct namect *pst)  
{  
    pst->letters = strlen(pst->fname) + strlen(pst->lname);  
}  
void showinfo(const struct namect *pst)  
{  
    printf("%s %s, ваше имя и фамилия содержат %d букв\n", pst->fname, pst->lname, pst->letters);  
}  
void cleanup(struct namect *pst)  
{  
    free(pst->fname);  
    free(pst->lname);  
}  
char *s_gets(char *st, int n)  
{  
    char *ret_val;  
    char *find;  
    ret_val = fgets(st, n, stdin);  
    if (ret_val)  
    {  
        find = strchr(st, '\n'); // поиск новой строки  
        if (find) // если адрес не равен NULL,  
            *find = '\0'; // поместить туда нулевой символ  
        else  
            while (getchar() != '\n')  
                continue; // отбросить остаток строки  
    }  
    return ret_val;  
}
```



# Составные литералы и структуры (C99)

```
struct book
{ // шаблон структуры: book - дескриптор
    char title[MAXITL];
    char author[MAXUTL];
    float value;
};

int main(void)
{
    struct book readfirst;
    int score;
    printf("Введите рейтинг: ");
    scanf("%d", &score);
    if (score >= 84)
        readfirst = (struct book){ "Преступление и наказание", "Федор Достоевский", 11.25 };
    else
        readfirst = (struct book){ "Красивая шляпа мистера Баунси", "Фред Уинсом", 5.99 };
    printf("Назначенные вами рейтинги:\n");
    printf("%s by %s: %.2f\n", readfirst.title, readfirst.author, readfirst.value);
    return 0;
}
```

```
// Введите рейтинг: 1
// Назначенные вами рейтинги:
// Красивая шляпа мистера Баунси by Фред Уинсом: $5.99
```

# Члены с типами гибких массивов (C99)

В стандарте C99 предлагается новое средство, которое называется **членом типа гибкого массива**. Оно позволяет объявлять структуру, последний член в которой является массивом со специальными свойствами.

- такой массив не существует — во всяком случае, не появляется немедленно.
- при наличии корректного кода член типа гибкого массива можно использовать, как если бы он существовал и имел нужное количество элементов.

представлены правила, регламентирующие создание члена типа гибкого массива.

1. Член типа гибкого массива должен быть последним в структуре.
2. В структуре должен присутствовать, по крайней мере, еще один член другого типа.
3. Гибкий массив объявляется с пустыми квадратными скобками.

**пример:**

```
struct flex {  
    int count;  
    double average;  
    double scores []; // член типа гибкого массива  
};  
struct flex *pf; // объявление указателя  
pf = malloc(sizeof(struct flex) + 5 * sizeof(double));
```

# Члены с типами гибких массивов (C99) – демо ПК

```
// Рейтинги: 20 19 18 17 16
// Среднее значение: 18
// Рейтинги: 20 19.5 19 18.5 18 17.5 17 16.5 16
// Среднее значение: 17
```

```
#include <stdio.h>
#include <stdlib.h>
struct flex
{
    size_t count;
    double average;
    double scores[]; // член с типом гибкого массива
};
void showFlex(const struct flex *p);
int main(void)
{
    struct flex *pf1, *pf2;
    int n = 5;
    int i;
    int tot = 0;
    // выделение памяти для структуры и массива
    pf1 = malloc(sizeof(struct flex) + n * sizeof(double));
    pf1->count = n;
    for (i = 0; i < n; i++)
    {
        pf1->scores[i] = 20.0 - i;
        tot += pf1->scores[i];
    }
    pf1->average = tot / n;
    showFlex(pf1);
    n = 9;
    tot = 0;
    pf2 = malloc(sizeof(struct flex) + n * sizeof(double));
    pf2->count = n;
    for (i = 0; i < n; i++)
    {
        pf2->scores[i] = 20.0 - i / 2.0;
        tot += pf2->scores[i];
    }
    pf2->average = tot / n;
    showFlex(pf2);
    free(pf1);
    free(pf2);
    return 0;
}
void showFlex(const struct flex *p)
{
    int i;
    printf("Рейтинги: ");
    for (i = 0; i < p->count; i++)
        printf("%g ", p->scores[i]);
    printf("\nСреднее значение: %g\n", p->average);
}
```

# Анонимные структуры (C11)

Анонимная структура — это член структуры, который является неименованной структурой.

Стандарт C11 позволяет определять структуру, используя в качестве члена вложенную неименованную структуру:

```
struct person {  
    int id;  
    struct {char first[20]; char last[20];}; // анонимная структура  
};
```

Эту структуру можно было бы инициализировать в той же манере:

```
struct person ted = {8483, {"Ted", "Grass"}};
```

Но доступ к членам упрощается, поскольку для этого применяются имена членов вроде `first`, как если бы они были членами `person`:

```
puts(ted.first);
```

Средство анонимности более полезно с вложенными объединениями.



# Функции с массивом структур

```
#define FUNDLLEN 50
#define N 2
struct funds {
    char bank[FUNDLLEN];
    double bankfund;
    char save[FUNDLLEN];
    double savefund;
};
double sum(const struct funds money[], int n)
{
    double total;
    int i;
    for (i = 0, total = 0; i < n; i++)
        total += money[i].bankfund + money[i].savefund;
    return (total);
}
int main(void)
{
    struct funds jones[N] = {"Garlic-Melon Bank", 4032.27, "Lucky's Savings and Loan", 8543.94},
                           {"Honest Jack's Bank", 3620.88, "Party Time Savings", 3802.91}};
    printf("Общая сумма на счетах у Джонсов составляет $%.2f.\n", sum(jones, N));
    return 0;
}
```

Общая сумма на счетах у Джонсов составляет \$20000.00.



# Объединения: краткое знакомство

Объединение — это тип, который позволяет хранить данные разных типов в одном и том же месте памяти (но не одновременно).

Ниже показан пример шаблона объединения с дескриптором:

```
union hold {  
    int digit;  
    double bigfl;  
    char letter;  
}; // объединение может хранить значение типа int или double или char.
```

Вот пример определения трех переменных объединения типа hold:

```
union hold fit; // переменная объединения типа hold  
union hold save[10]; // массив из 10 переменных объединения  
union hold * ru; // указатель на переменную типа hold
```

Три варианта инициализации:

- инициализировать объединение другим объединением того же типа, инициализировать первый элемент объединения
- в случае C99 применить назначенный инициализатор

```
union hold valA;  
valA.letter = 'R';  
union hold valB = valA; // инициализация одного объединения другим  
union hold valC = {88}; // инициализация члена digit объединения  
union hold valD = {.bigfl = 118.2}; // назначенный инициализатор
```





# Использование объединений

Ниже показано, как можно использовать объединение:

```
fit.digit = 23; // в переменной fit хранится 23; используются 2 байта  
fit.bigfl = 2.0; // 23 очищено, 2.0 сохранено; используются 8 байтов  
fit.letter = 'h'; // 2.0 очищено, h сохранено; используется 1 байт
```

Операция точки показывает, какой тип данных применяется в текущий момент.

Вы можете использовать операцию `->` с указателями на объединения в той же манере, как применяли ее с указателями на структуры:

```
pu = &fit;  
x = pu->digit; // то же, что и x = fit.digit
```

Ниже показано, как не следует поступать:

```
fit.letter = 'A';  
flnum = 3.02*fit.bigfl; // ОШИБКА!
```

Эта последовательность ошибочна, т.к. сохранено значение типа `char`, но в следующей строке предполагается, что содержимое `fit` имеет тип `double`.



# Использование объединений

Другой ситуацией применения объединений является структура, в которой сохраняемая информация зависит от значения одного из ее членов. Предположим, что у вас есть структура, представляющая автомобиль. Если автомобиль принадлежит пользователю, вы хотите, чтобы член структуры описывал владельца. Если автомобиль взят напрокат, необходимо, чтобы член описывал компанию по прокату. Тогда можно записать так:

Пусть flits — это структура car\_data. Тогда если значение flits. status равно 0, программа может использовать flits.ownerinfo.owncar.socsecurity, а если значение flits. status равно 1 - то flits.ownerinfo.leasecar.name.

```
struct owner
{
    char socsecurity[12];
    ...
};
struct leasecompany
{
    char name[40];
    char headquarters[40];
    ...
};
union data
{
    struct owner owncar;
    struct leasecompany leasecar;
};
struct car_data
{
    char make[15];
    int status; /* 0 = принадлежит,
1 = взят напрокат */
    union data ownerinfo;
    ...
};
```



# Анонимные объединения (C11)

Анонимное объединение — это неименованное объединение, являющееся членом структуры или объединения.

```
struct owner
{
    char socsecurity[12];
    ...
};
struct leasecompany
{
    char name[40];
    char headquarters[40];
    ...
};
struct car_data
{
    char make[15];
    int status; /* 0 = принадлежит, 1 = взят напрокат */
    union
    {
        struct owner owncar;
        struct leasecompany leasecar;
    };
    ...
};
```

Теперь, если flits — это структура car\_data, мы можем применять flits.owncar.socsecurity вместо flits.ownerinfo.owncar.socsecurity.

# Перечислимые типы

Перечислимый тип можно использовать для объявления символических имен, представляющих целочисленные константы. Ключевое слово ***enum*** позволяет создать новый “тип” и указать значения, которые для него допускаются. (На самом деле константы `enum` имеют тип `int`, поэтому их можно применять везде, где разрешено использовать тип `int`.) Целью перечислимых типов является улучшение читабельности программы. Их синтаксис похож на синтаксис, применяемый для структур. Например, можно записать следующие объявления:

```
enum spectrum {red, orange, yellow, green, blue, violet};  
enum spectrum color;
```

Первое объявление устанавливает `spectrum` как имя дескриптора, который позволяет использовать `enum spectrum` в качестве имени типа. Второе объявление делает `color` переменной этого типа. Идентификаторы внутри фигурных скобок перечисляют возможные значения, которые может иметь переменная `spectrum`. Символические константы (`red`, `orange`, `yellow` и т.д.) называются перечислителями.





# Перечислимые типы

## **Стандартные значения**

По умолчанию константам в списке перечислений присваиваются целочисленные значения 0, 1, 2 и т.д. Следовательно, объявление `enum kids {nippy, slats, skippy, nina, liz};` приводит к тому, что `nina` имеет значение 3.

## **Присвоенные значения**

При желании вы можете выбрать целочисленные значения, которые должны иметь константы. Для этого просто включите нужные значения в объявление:

```
enum levels {low = 100, medium = 500, high = 2000};
```

Если значение присваивается одной константе, но не следующим за ней, то дальнейшие константы получат последовательно возрастающие значения. Например, взгляните на следующее объявление:

```
enum feline {cat, lynx = 10, puma, tiger};
```

В этом случае `cat` получает стандартное значение 0, а `lynx`, `puma` и `tiger` — соответственно, 10, 11 и 12.



# enum – демо ПК

```
// Введите цвет (или пустую строку для выхода):
// blue
// Колокольчики синие.
// Введите следующий цвет (или пустую строку для выхода):
// green
// Трава зеленая.
// Введите следующий цвет (или пустую строку для выхода):
// purple
// Цвет purple не известен.
// Введите следующий цвет (или пустую строку для выхода):

// Программа завершена.
```

```
/* enum.c -- использование перечислимых значений */
#include <stdio.h>
#include <string.h> // для strcmp(), strchr()
#include <stdbool.h> // средство C99
char *s_gets(char *st, int n);
enum spectrum
{
    red,
    orange,
    yellow,
    green,
    blue,
    violet
};
const char *colors[] = {"red", "orange", "yellow", "green", "blue", "violet"};
#define LEN 30
int main(void)
{
    char choice[LEN];
    enum spectrum color;
    bool color_is_found = false;
    puts("Введите цвет (или пустую строку для выхода):");
    while (s_gets(choice, LEN) != NULL && choice[0] != '\0')
    {
        for (color = red; color <= violet; color++)
        {
            if (strcmp(choice, colors[color]) == 0)
            {
                color_is_found = true;
                break;
            }
        }
        if (color_is_found)
        {
            switch (color)
            {
                case red:
                    puts("Розы красные.");
                    break;
                case orange:
                    puts("Маки оранжевые.");
                    break;
                case yellow:
                    puts("Подсолнухи желтые.");
                    break;
                case green:
                    puts("Трава зеленая.");
                    break;
                case blue:
                    puts("Колокольчики синие.");
                    break;
                case violet:
                    puts("Фиалки фиолетовые.");
                    break;
            }
        }
        else
            printf("Цвет %s не известен.\n", choice);
        color_is_found = false;
        puts("Введите следующий цвет (или пустую строку для выхода):");
    }
    puts("Программа завершена.");
    return 0;
}

char *s_gets(char *st, int n)
{
    char *ret_val;
    char *find;
    ret_val = fgets(st, n, stdin);
    if (ret_val)
    {
        find = strchr(st, '\n'); // поиск новой строки
        if (find)                // если адрес не равен NULL,
            *find = '\0';        // поместить туда нулевой символ
        else
            while (getchar() != '\n')
                continue; // отбросить остаток строки
    }
    return ret_val;
}
```



# Средство typedef

Возможность typedef представляет собой усовершенствованное средство манипулирования данными, которое позволяет создавать собственное имя для типа. В этом отношении оно подобно директиве #define, но с тремя отличиями.

- В отличие от #define, средство typedef ограничено назначением символических имен только типам, но не значениям.
- Интерпретация typedef выполняется компилятором, а не препроцессором.
- В рамках своих ограничений средство typedef является более гибким, чем #define.

## Пример:

```
typedef unsigned char BYTE;
```

После этого BYTE можно применять для определения переменных:

```
BYTE x, y [10], *z;
```

Область видимости этого определения зависит от местоположения оператора typedef. Если определение находится внутри функции, область видимости будет локальной в пределах этой функции. Если определение находится вне функции, область видимости будет глобальной.



# Средство typedef

Для определений typedef часто используются прописные буквы, чтобы напоминать пользователю о том, что имя типа в действительности является символическим сокращением, но имена в typedef подчиняются тем же правилам, которые регламентируют создание допустимых имен переменных:

```
typedef unsigned char byte;
```

Некоторые возможности typedef можно продублировать с помощью #define. **Например, указание**

```
#define BYTE unsigned char
```

заставляет препроцессор заменять BYTE типом unsigned char. Ниже приведен пример typedef, который невозможно воспроизвести посредством #define:

```
typedef char *STRING;
```

Средство typedef можно также использовать со структурами:

```
typedef struct complex {  
    float real;  
    float imag;  
} COMPLEX;
```



# Средство typedef

При использовании typedef для именования типа структуры дескриптор можно не указывать:

```
typedef struct {double x; double y;} rect;
```

Предположим, что определенный посредством typedef идентификатор применяется так, как показано ниже:

```
rect r1 = {3.0, 6.0}; rect r2;
```

Этот код транслируется в следующие операторы:

```
struct {double x; double y;} r1 = {3.0, 6.0};
```

```
struct {double x; double y;} r2;
```

```
r2 = r1;
```

Если две структуры объявлены без дескриптора, но с идентичными членами (совпадают имена членов и типов), то в С эти две структуры считаются имеющими один и тот же тип, поэтому присваивание r1 переменной r2 является допустимой операцией.

Вторая причина использования typedef связана с тем, что имена typedef часто применяются для сложных типов. Например, объявление

```
typedef char (* FRPTC ()) [5];
```

делает FRPTC идентификатором типа, который является функцией, возвращающей указатель на массив из 5 элементов char.



# План лекции

**Структуры –  
продолжение**

**45 минут**

**Связанные  
списки**

**45 минут**



# Введение

**Связанный список** — это набор элементов данных, называемых узлами, в которых линейное представление задается ссылками от одного узла к следующему узлу.

**Массив** — это линейный набор элементов данных, в котором элементы хранятся в последовательных ячейках памяти. При объявлении массивов мы должны указать размер массива, что ограничит количество элементов, которые может хранить массив. Например, если мы объявим массив как `int marks[10]`, то массив может хранить максимум 10 элементов данных, но не больше.

Проблемы с массивами:

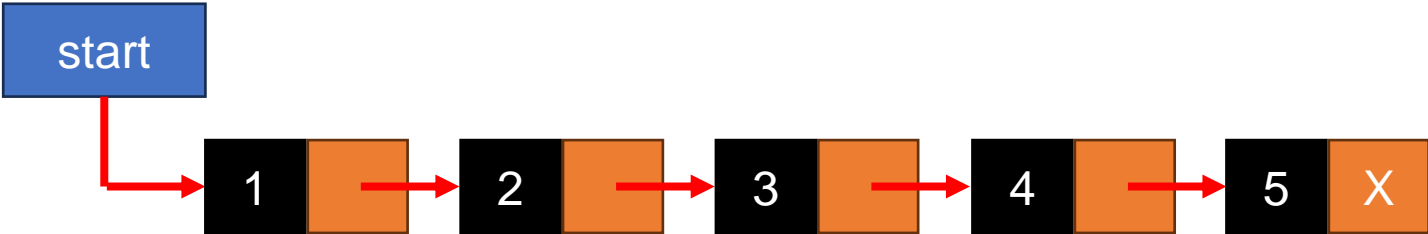
- заранее не уверены в количестве элементов
- чтобы эффективно использовать память, элементы должны храниться случайным образом в любом месте, а не в последовательных ячейках.

Таким образом, должна быть структура данных, которая снимает ограничения на максимальное количество элементов и условие хранения для написания эффективных программ. Связанный список — это структура данных, которая свободна от вышеупомянутых ограничений.



# Основные термины

Связанный список, говоря простыми словами, представляет собой линейный набор элементов данных. Эти элементы данных называются узлами. Связанный список — это структура данных, которая, в свою очередь, может использоваться для реализации других структур данных. Таким образом, он действует как строительный блок для реализации структур данных, таких как стеки, очереди и их вариации. Связанный список можно воспринимать как поезд или последовательность узлов, в которых каждый узел содержит одно или несколько полей данных и указатель на следующий узел.



```
struct node {
    int data;
    struct node *next;
};
```

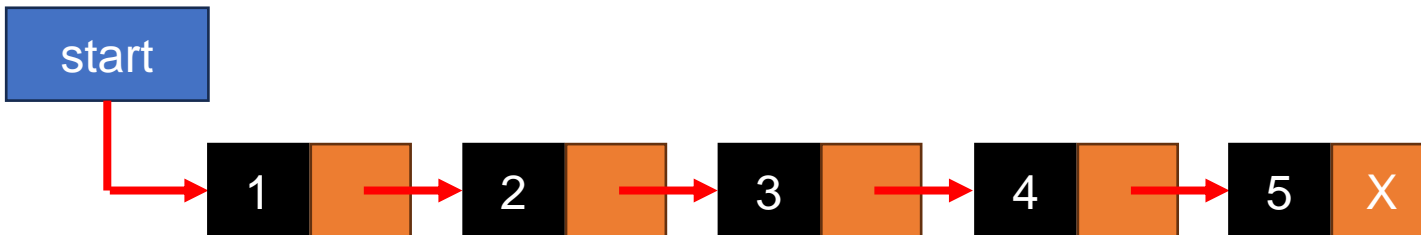
	Data	Next
1	H	4
2		
3		
4	E	7
5		
6		
7	L	8
8	L	10
9		
10	O	-1





# Основные термины

Связанный список, говоря простыми словами, представляет собой линейный набор элементов данных. Эти элементы данных называются узлами. Связанный список — это структура данных, которая, в свою очередь, может использоваться для реализации других структур данных. Таким образом, он действует как строительный блок для реализации структур данных, таких как стеки, очереди и их вариации. Связанный список можно воспринимать как поезд или последовательность узлов, в которых каждый узел содержит одно или несколько полей данных и указатель на следующий узел.



```
struct node {  
    int data;  
    struct node *next;  
};
```

start			
1		Data	Next
1		H	4
2			
3			
4		E	7
5			
6			
7		L	8
8		L	10
9			
10		O	-1

# Основные термины

Связанные списки обеспечивают эффективный способ хранения связанных данных и выполнения основных операций, таких как вставка, удаление и обновление информации, за счет дополнительного пространства, необходимого для хранения адреса следующих узлов.

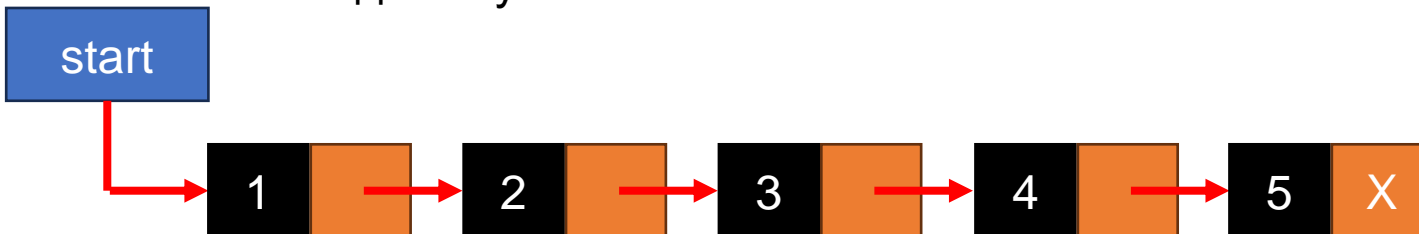
Когда мы удаляем определенный узел из существующего связанного списка или удаляем весь связанный список, занимаемое им пространство должно быть возвращено в свободный пул, чтобы память могла быть повторно использована какой-либо другой программой, которой требуется пространство памяти. Операционная система выполняет эту задачу по добавлению освобожденной памяти в свободный пул. Операционная система будет выполнять эту операцию всякий раз, когда она обнаруживает, что ЦП простаивает или когда программам не хватает места в памяти. Операционная система просматривает все ячейки памяти и отмечает те ячейки, которые используются какой-либо программой. Затем она собирает все ячейки, которые не используются, и добавляет их адрес в свободный пул, чтобы эти ячейки могли повторно использоваться другими программами. Этот процесс называется сборкой мусора.



# Односвязный список

Односвязный список — это простейший тип связанного списка, в котором каждый узел содержит некоторые данные и указатель на следующий узел того же типа данных. Говоря, что узел содержит указатель на следующий узел, мы подразумеваем, что узел хранит адрес следующего узла в последовательности. Односвязный список позволяет перемещаться по данным только одним способом. Связанный список содержит:

- переменную-указатель `START`, которая хранит адрес первого узла списка.
- конец списка отмечается сохранением `NULL` или `-1` в поле `NEXT` последнего узла.



```
struct node {  
    int data;  
    struct node *next;  
};
```



# Обход односвязного списка

Для обхода связанного списка мы используем другую переменную-указатель PTR, которая указывает на узел, к которому в данный момент осуществляется доступ.


1. инициализируем PTR с адресом START.
2. выполняется цикл while, который повторяется до тех пор, пока PTR не обработает последний узел, то есть пока не встретит NULL.
3. процесс (например, print) к текущему узлу, то есть узлу, на который указывает PTR.
4. переход к следующему узлу, заставляя переменную PTR указывать на узел, адрес которого хранится в поле NEXT.

```
struct node *display(struct node *start)
{
    struct node *ptr;
    ptr = start;
    while (ptr != NULL)
    {
        printf("\t %d", ptr->data);
        ptr = ptr->next;
    }
    return start;
}
```

```
struct node {
    int data;
    struct node *next;
};
```







Как посчитать количество  
узлов в связанном списке?

# Поиск значения в связанном списке

Например ищем 37



Здесь PTR -> DATA = 11. Так как PTR -> DATA != 37, мы переходим к следующему узлу.



Здесь PTR -> DATA = 2. Так как PTR -> DATA != 37, мы переходим к следующему узлу.



Здесь PTR -> DATA = 37. Так как PTR -> DATA = 37, POS = PTR. POS теперь хранит адрес узла, который содержит VAL

```
struct node {  
    int data;  
    struct node *next;  
};
```

Однако, если поиск не удался, POS устанавливается в NULL, что указывает на то, что VAL отсутствует в связанном списке.



# Вставка узла в начало связанного списка

```

struct node *insert_beg(struct node *start)
{
    struct node *new_node;
    int num;
    printf("\n Enter the data : ");
    scanf("%d", &num);
    new_node = (struct node*)
malloc(sizeof(struct node));
    new_node->data = num;
    new_node->next = start;
    start = new_node;
    return start;
}

```

```

struct node {
    int data;
    struct node *next;
};

```

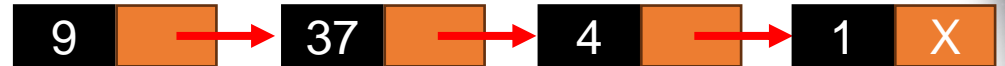


START

Выделите память для нового узла и инициализируйте его часть DATA значением 9.



Добавьте новый узел в качестве первого узла списка, сделав часть NEXT нового узла содержащей адрес START.



START

Теперь сделайте так, чтобы START указывал на первый узел списка.



START

# Вставка узла в конец связанного списка

```
struct node *insert_end(struct node *start)
{
    struct node *ptr, *new_node;
    int num;
    printf("\n Enter the data : ");
    scanf("%d", &num);
    new_node = (struct node*)
    malloc(sizeof(struct node));
    new_node->data = num;
    new_node->next = NULL;
    ptr = start;
    while (ptr->next != NULL)
        ptr = ptr->next;
    ptr->next = new_node;
    return start;
}
```

```
struct node {
    int data;
    struct node *next;
};
```



START

Выделите память для нового узла и инициализируйте его часть DATA значением 9, а часть NEXT — значением NULL. **9 X**

Возьмите переменную-указатель PTR, которая указывает на START. Переместите PTR так, чтобы она указывала на последний узел списка.



START

PTR

Добавьте новый узел после узла, на который указывает PTR. Это делается путем сохранения адреса нового узла в части NEXT.



START

PTR

# Вставка узла перед заданным узлом в связанном списке

```
struct node {
    int data;
    struct node *next;
};
```



START

Выделите память для нового узла и инициализируйте его часть DATA значением 9.



Возьмите две переменные указателя PTR и PREPTR и инициализируйте их с помощью START так, чтобы START, PTR и PREPTR указывали на первый узел списка.



START

PTR

PREPTR

Перемещайте PTR и PREPTR до тех пор, пока часть DATA PREPTR не станет равной значению узла, после которого необходимо выполнить вставку. PREPTR всегда будет указывать на узел непосредственно перед PTR.



Добавьте новый узел между узлами, на которые указывают PREPTR и PTR.



START

PREPTR

PTR



NEW\_NODE

# Вставка узла перед заданным узлом в связанном списке

```
struct node *insert_before(struct node *start)
{
    struct node *new_node, *ptr, *preptr;
    int num, val;
    printf("\n Enter the data : ");
    scanf("%d", &num);
    printf("\n Enter the value before which the data has to be inserted : ");
    scanf("%d", &val);
    new_node = (struct node *)malloc(sizeof(struct node));
    new_node->data = num;
    ptr = start;
    while (ptr->data != val)
    {
        preptr = ptr;
        ptr = ptr->next;
    }
    preptr->next = new_node;
    new_node->next = ptr;
    return start;
}
```

```
struct node {
    int data;
    struct node *next;
};
```

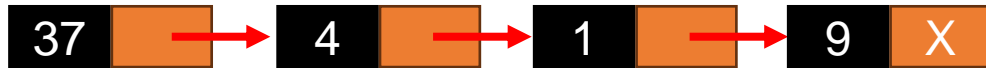
# Вставка узла после заданного узла в связанном списке

```
struct node *insert_after(struct node *start)
{
    struct node *new_node, *ptr, *preptr;
    int num, val;
    printf("\n Enter the data : ");
    scanf("%d", &num);
    printf("\n Enter the value after which the data has to be inserted : ");
    scanf("%d", &val);
    new_node = (struct node *)malloc(sizeof(struct node));
    new_node->data = num;
    ptr = start;
    preptr = ptr;
    while (preptr->data != val)
    {
        preptr = ptr;
        ptr = ptr->next;
    }
    preptr->next = new_node;
    new_node->next = ptr;
    return start;
}
```

```
struct node {
    int data;
    struct node *next;
};
```

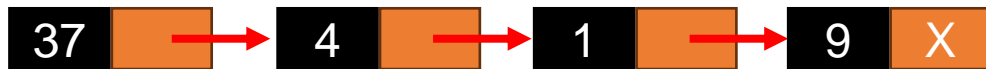


# Удаление первого узла из связанного списка



START

Сделайте START указателем на следующий узел в последовательности.



START

Очистите



START

```
struct node {  
    int data;  
    struct node *next;  
};
```

```
struct node *delete_beg(struct node *start)  
{  
    struct node *ptr;  
    ptr = start;  
    start = start->next;  
    free(ptr);  
    return start;  
}
```

# Удаление последнего узла из связанного списка

```
struct node *delete_after(struct node *start)
{
    struct node *ptr, *preptr;
    int val;
    printf("\n Enter the value after which the node has to deleted : ");
    scanf("%d", &val);
    ptr = start;
    preptr = ptr;
    while (preptr->data != val)
    {
        preptr = ptr;
        ptr = ptr->next;
    }
    preptr->next = ptr->next;
    free(ptr);
    return start;
}
```

```
struct node {
    int data;
    struct node *next;
};
```

# Вставка узла после заданного узла в связанном списке

```
struct node *delete_end(struct node *start)
{
    struct node *ptr, *preptr;
    ptr = start;
    while (ptr->next != NULL)
    {
        preptr = ptr;
        ptr = ptr->next;
    }
    preptr->next = NULL;
    free(ptr);
    return start;
}
```

```
struct node {
    int data;
    struct node *next;
};
```

# Сортировка в связанном списке

```
struct node *sort_list(struct node *start)
{
    struct node *ptr1, *ptr2;
    int temp;
    ptr1 = start;
    while (ptr1->next != NULL)
    {
        ptr2 = ptr1->next;
        while (ptr2 != NULL)
        {
            if (ptr1->data > ptr2->data)
            {
                temp = ptr1->data;
                ptr1->data = ptr2->data;
                ptr2->data = temp;
            }
            ptr2 = ptr2->next;
        }
        ptr1 = ptr1->next;
    }
    return start; // Had to be added
}
```

```
struct node {
    int data;
    struct node *next;
};
```

# Forward list – демо ПК

\*\*\*\*\*MAIN MENU \*\*\*\*\*

- 1: Create a list
- 2: Display the list
- 3: Add a node at the beginning
- 4: Add a node at the end
- 5: Add a node before a given node
- 6: Add a node after a given node
- 7: Delete a node from the beginning
- 8: Delete a node from the end
- 9: Delete a given node
- 10: Delete a node after a given node
- 11: Delete the entire list
- 12: Sort the list
- 13: EXIT

```

#include <iostream>
using namespace std;

// Node structure
struct Node {
    int data;
    Node* next;
};

// Global variables
Node* head = NULL;
Node* tail = NULL;
int count = 0;

// Function to create a new node
Node* createNode(int data) {
    Node* newNode = new Node;
    newNode->data = data;
    newNode->next = NULL;
    return newNode;
}

// Function to add a node at the beginning
void addBeginning(int data) {
    Node* newNode = createNode(data);
    newNode->next = head;
    head = newNode;
    count++;
}

// Function to add a node at the end
void addEnd(int data) {
    Node* newNode = createNode(data);
    if (tail == NULL) {
        head = newNode;
        tail = newNode;
    } else {
        tail->next = newNode;
        tail = newNode;
    }
    count++;
}

// Function to add a node before a given node
void addBefore(int data, int key) {
    if (key == 0) {
        addBeginning(data);
        return;
    }
    Node* temp = head;
    for (int i = 0; i < key; i++) {
        temp = temp->next;
    }
    Node* newNode = createNode(data);
    newNode->next = temp->next;
    temp->next = newNode;
    count++;
}

// Function to add a node after a given node
void addAfter(int data, int key) {
    if (key == 0) {
        addBeginning(data);
        return;
    }
    Node* temp = head;
    for (int i = 0; i < key; i++) {
        temp = temp->next;
    }
    Node* newNode = createNode(data);
    newNode->next = temp->next;
    temp->next = newNode;
    count++;
}

// Function to delete a node from the beginning
void deleteBeginning() {
    if (head == NULL) {
        cout << "List is empty" << endl;
        return;
    }
    head = head->next;
    count--;
}

// Function to delete a node from the end
void deleteEnd() {
    if (tail == NULL) {
        cout << "List is empty" << endl;
        return;
    }
    Node* temp = head;
    while (temp->next != NULL) {
        temp = temp->next;
    }
    temp->next = NULL;
    tail = temp;
    count--;
}

// Function to delete a given node
void deleteNode(int key) {
    if (key == 0) {
        deleteBeginning();
        return;
    }
    Node* temp = head;
    for (int i = 0; i < key; i++) {
        temp = temp->next;
    }
    temp->next = temp->next->next;
    count--;
}

// Function to delete a node after a given node
void deleteAfter(int key) {
    if (key == 0) {
        deleteBeginning();
        return;
    }
    Node* temp = head;
    for (int i = 0; i < key; i++) {
        temp = temp->next;
    }
    temp->next = temp->next->next;
    count--;
}

// Function to delete the entire list
void deleteList() {
    head = NULL;
    tail = NULL;
    count = 0;
}

// Function to sort the list
void sortList() {
    // Bubble sort implementation
    Node* temp = head;
    while (temp != NULL) {
        Node* curr = temp;
        while (curr->next != NULL) {
            if (curr->data > curr->next->data) {
                // Swap data
                int tempData = curr->data;
                curr->data = curr->next->data;
                curr->next->data = tempData;
            }
            curr = curr->next;
        }
        temp = temp->next;
    }
}

// Function to display the list
void displayList() {
    Node* temp = head;
    while (temp != NULL) {
        cout << temp->data << " ";
        temp = temp->next;
    }
    cout << endl;
}

// Main function
int main() {
    int choice;
    do {
        cout << "*****MAIN MENU *****" << endl;
        cout << "1: Create a list" << endl;
        cout << "2: Display the list" << endl;
        cout << "3: Add a node at the beginning" << endl;
        cout << "4: Add a node at the end" << endl;
        cout << "5: Add a node before a given node" << endl;
        cout << "6: Add a node after a given node" << endl;
        cout << "7: Delete a node from the beginning" << endl;
        cout << "8: Delete a node from the end" << endl;
        cout << "9: Delete a given node" << endl;
        cout << "10: Delete a node after a given node" << endl;
        cout << "11: Delete the entire list" << endl;
        cout << "12: Sort the list" << endl;
        cout << "13: EXIT" << endl;
        cout << "Enter your choice: ";
        int choice;
        cin >> choice;

        switch (choice) {
            case 1: {
                int data;
                cout << "Enter data: ";
                cin >> data;
                addEnd(data);
                break;
            }
            case 2: {
                displayList();
                break;
            }
            case 3: {
                int data;
                cout << "Enter data: ";
                cin >> data;
                addBeginning(data);
                break;
            }
            case 4: {
                int data;
                cout << "Enter data: ";
                cin >> data;
                addEnd(data);
                break;
            }
            case 5: {
                int data;
                cout << "Enter data: ";
                cin >> data;
                int key;
                cout << "Enter key: ";
                cin >> key;
                addBefore(data, key);
                break;
            }
            case 6: {
                int data;
                cout << "Enter data: ";
                cin >> data;
                int key;
                cout << "Enter key: ";
                cin >> key;
                addAfter(data, key);
                break;
            }
            case 7: {
                deleteBeginning();
                break;
            }
            case 8: {
                deleteEnd();
                break;
            }
            case 9: {
                int key;
                cout << "Enter key: ";
                cin >> key;
                deleteNode(key);
                break;
            }
            case 10: {
                int key;
                cout << "Enter key: ";
                cin >> key;
                deleteAfter(key);
                break;
            }
            case 11: {
                deleteList();
                break;
            }
            case 12: {
                sortList();
                break;
            }
            case 13: {
                cout << "Exiting program" << endl;
                return 0;
            }
            default: {
                cout << "Invalid choice" << endl;
            }
        }
    } while (choice != 13);

    return 0;
}

```

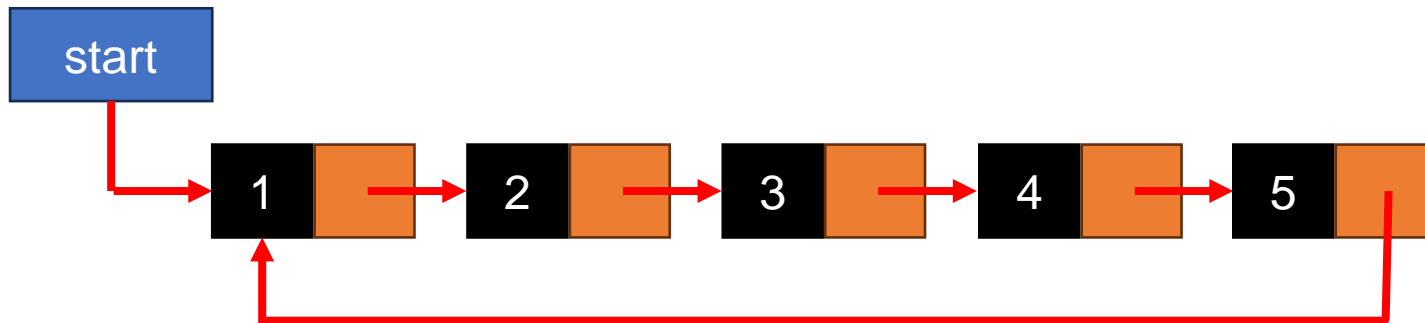


# Кольцевой односвязный список

В кольцевом связанном списке последний узел содержит указатель на первый узел списка. Таким образом, циклический связанный список не имеет начала и конца.

- Единственным недостатком циклического связанного списка является сложность итерации

Циклические связанные списки широко используются в операционных системах для обслуживания задач. **Пример.** Когда мы просматриваем Интернет, мы можем использовать кнопки Назад и Вперед, чтобы перейти на предыдущие страницы, которые мы уже посетили.



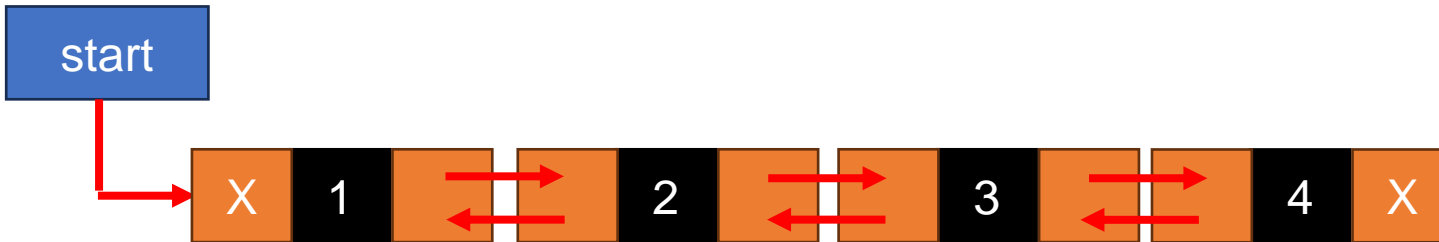
```
struct node {  
    int data;  
    struct node *next;  
};
```

# Двухсвязный список

Двухсвязный список или двусторонний связанный список — это более сложный тип связанного списка, который содержит указатель на следующий, а также на предыдущий узел в последовательности. Таким образом, он состоит из трех частей — данных, указателя на следующий узел и указателя на предыдущий узел

Поле PREV первого узла и поле NEXT последнего узла будут содержать NULL. Поле PREV используется для хранения адреса предыдущего узла, что позволяет нам проходить список в обратном направлении.

Двухсвязный список требует больше места на узел и более дорогих базовых операций. Однако двухсвязный список обеспечивает простоту манипулирования элементами списка, поскольку он поддерживает указатели на узлы в обоих направлениях (вперед и назад). Главное преимущество использования двухсвязного списка заключается в том, что он делает поиск в два раза эффективнее.



```
struct node {  
    struct node *prev;  
    int data;  
    struct node *next;  
};
```

# Вставка узла в начало двухсвязного списка

```
struct node {
    struct node *prev;
    int data;
    struct node *next;
};
```

```
struct node *insert_beg(struct node *start)
{
    struct node *new_node;
    int num;
    printf("\n Enter the data : ");
    scanf("%d", &num);
    new_node = (struct node *)
    malloc(sizeof(struct node));
    new_node->data = num;
    start->prev = new_node;
    new_node->next = start;
    new_node->prev = NULL;
    start = new_node;
    return start;
}
```

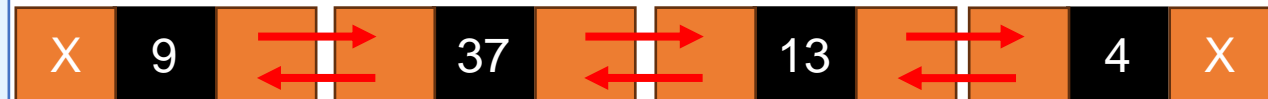


START

Выделите память для нового узла и инициализируйте его часть DATA значением 9, а поле PREV — значением NULL.



Добавьте новый узел перед узлом START. Теперь новый узел станет первым узлом списка.



# Вставка узла в конец двухсвязного списка

```
struct node *insert_end(struct node *start)
{
    struct node *ptr, *new_node;
    int num;
    printf("\n Enter the data : ");
    scanf("%d", &num);
    new_node = (struct node *)
    malloc(sizeof(struct node));
    new_node->data = num;
    ptr = start;
    while (ptr->next != NULL)
        ptr = ptr->next;
    ptr->next = new_node;
    new_node->prev = ptr;
    new_node->next = NULL;
    return start;
}
```

```
struct node {
    struct node *prev;
    int data;
    struct node *next;
};
```

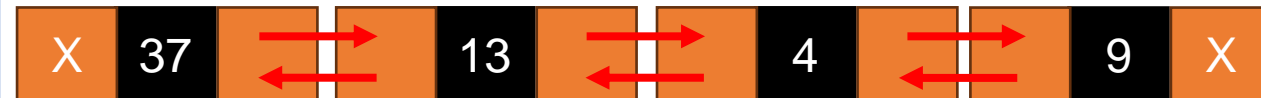


START

Выделите память для нового узла и инициализируйте его часть DATA значением 9, а поле NEXT — значением NULL



Возьмите переменную-указатель PTR и сделайте так, чтобы она указывала на первый узел списка. Переместите PTR так, чтобы он указывал на последний узел списка. Добавьте новый узел после узла, на который указывает PTR.



START

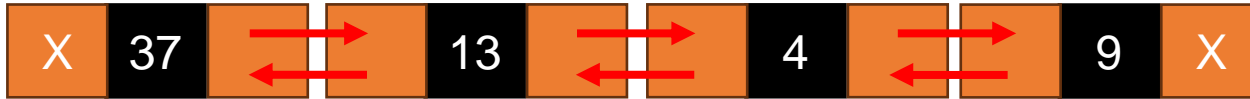
PTR

# Вставка узла после заданного узла в двусвязном списке

```
struct node *insert_before(struct node *start)
{
    struct node *new_node, *ptr;
    int num, val;
    printf("\n Enter the data : ");
    scanf("%d", &num);
    printf("\n Enter the value before which the data has to be inserted : ");
    scanf("%d", &val);
    new_node = (struct node *)malloc(sizeof(struct node));
    new_node->data = num;
    ptr = start;
    while (ptr->data != val)
        ptr = ptr->next;
    new_node->next = ptr;
    new_node->prev = ptr->prev;
    ptr->prev->next = new_node;
    ptr->prev = new_node;
    return start;
}
```

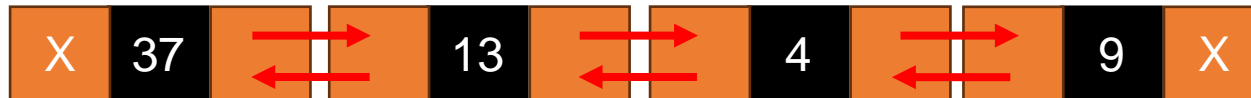


# Удаление узла, следующего за заданным узлом в двусвязном списке



START

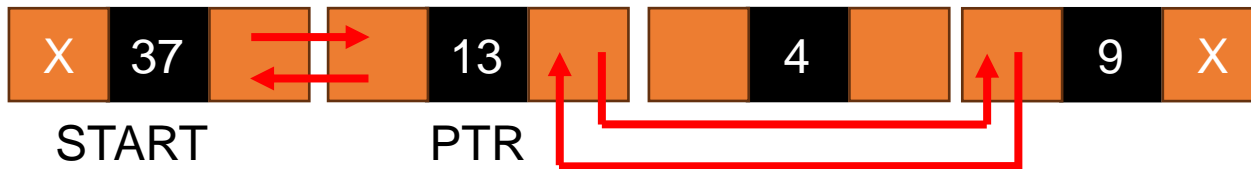
Возьмите переменную-указатель PTR и сделайте так, чтобы она указывала на первый узел списка. Переместите PTR дальше, чтобы его часть данных была равна значению, после которого должен быть удален узел.



START

PTR

Удалите узел, следующий за PTR.



START

PTR



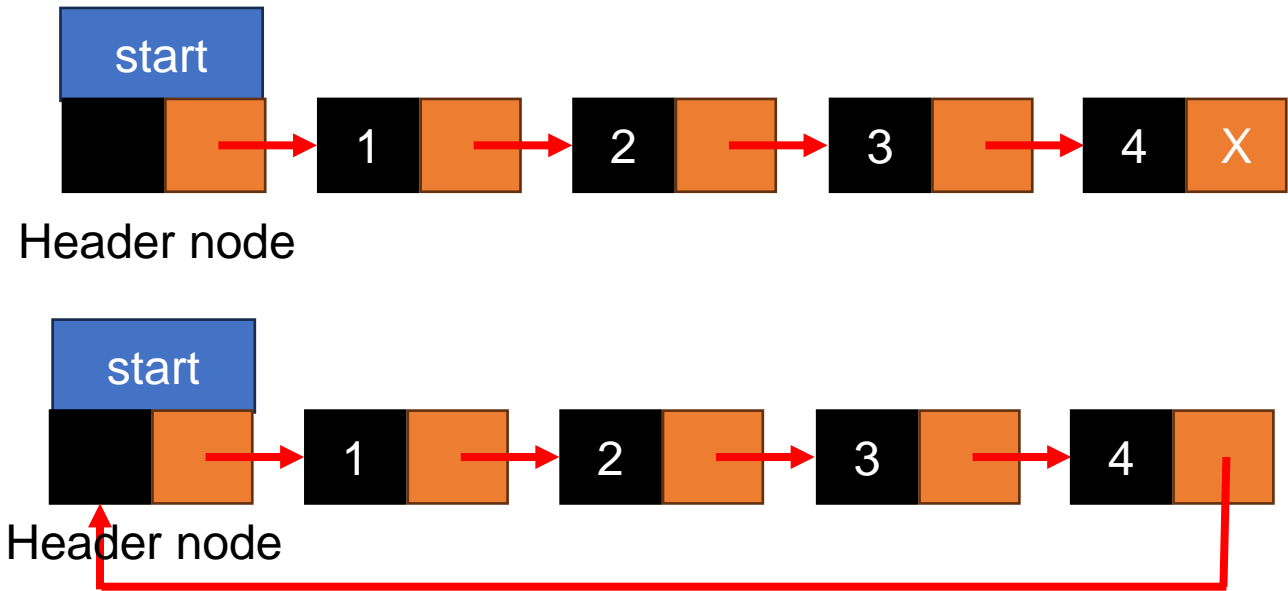
START

# Удаление узла, следующего за заданным узлом в двусвязном списке

```
struct node *delete_after(struct node *start)
{
    struct node *ptr, *temp;
    int val;
    printf("\n Enter the value after which the node has to deleted : ");
    scanf("%d", &val);
    ptr = start;
    while (ptr->data != val)
        ptr = ptr->next;
    temp = ptr->next;
    ptr->next = temp->next;
    temp->next->prev = ptr;
    free(temp);
    return start;
}
```

# Связанные заголовком списки

Связанный заголовком список — это особый тип связанного списка, который содержит узел заголовка в начале списка. Таким образом, в связанном заголовком списке START не будет указывать на первый узел списка, но START будет содержать адрес узла заголовка.



# Многосвязанные списки

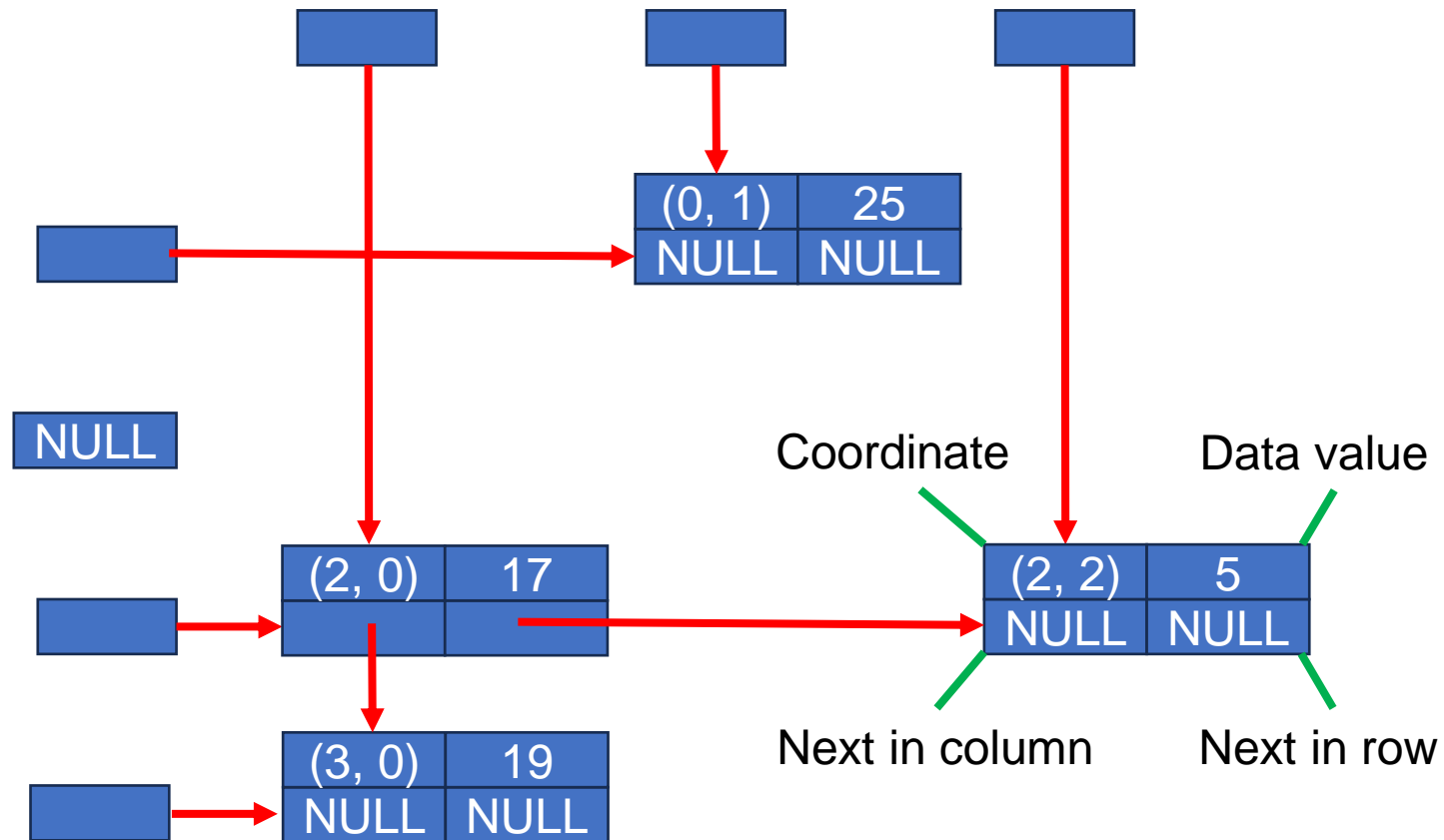
В многосвязном списке каждый узел может иметь  $n$  указателей на другие узлы. Двусвязный список является особым случаем многосвязных списков. Однако, в отличие от двусвязных списков, узлы в многосвязном списке могут иметь или не иметь обратные для каждого указателя. Мы можем отличить двусвязный список от многосвязного списка двумя способами:

- (a) Двусвязный список имеет ровно два указателя. Один указатель указывает на предыдущий узел, а другой указывает на следующий узел. Но узел в многосвязном списке может иметь любое количество указателей.
- (b) В двусвязном списке указатели являются точными обратными друг другу, т. е. для каждого указателя, который указывает на предыдущий узел, есть указатель, который указывает на следующий узел. Это не относится к многосвязному списку. Многосвязные списки обычно используются для организации нескольких порядков одного набора элементов. Например, если у нас есть связанный список, в котором хранятся имена и оценки, полученные учениками в классе, то мы можем организовать узлы списка двумя способами: (i) Организовать узлы в алфавитном порядке (по имени) (ii) Организовать узлы в порядке убывания оценок, чтобы информация об ученике, получившем самые высокие оценки, была перед другими учениками.



# Многосвязанные списки – пример (разреженная матрица)

y/x	0	1	2
0	0	25	0
1	0	0	0
2	17	0	5
3	19	0	0





# Резюме

- Мы продолжили изучать структуры и познакомились с различными видами структур
- Мы рассмотрели перечислимые типы
- Кратко разобрали – что такое объединения
- Познакомились со словом `typedef`
- Рассмотрели подробно различные виды связанных списков