ВВЕДЕНИЕ В ЯЗЫК С

Часть 2

Учебное пособие

МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ
ФЕДЕРАЛЬНОЕ АГЕНТСТВО ПО ОБРАЗОВАНИЮ
Томский государственный университет
Факультет прикладной математики и кибернетики

"УТВЕРЖДАЮ"
Декан ФПМК
профессор _____ А.М. Горцев
25 марта 2011 г.

ВВЕДЕНИЕ В ЯЗЫК С

Часть 2

Учебное пособие

Томск
2011

РАССМОТРЕНО и УТВЕРЖДЕНО методической комиссией
факультета прикладной математики и кибернетики

Председатель комиссии

профессор С.Э. Воробейчиков

Протокол 38 от 25 марта 2011 г.

В данном учебном пособии рассматриваются такие разделы языка программирования С как: работа с логическими операциями, структуры, работа с файлами. Здесь также подробно рассматривается структура даиных *список* и его частные случаи: стек, очередь, циклический и двунаправленный списки. Теоретический материал подкреплен примерами.
Пособие предназначено для студентов, изучающих язык программирования С.

Составители

доцент каф программирования О.И. Голубева,

старший преподаватель каф. программирования В.А. Сибирякова

1. Поразрядные логические операции

Поразрядные логические операции рассматривают данные, расположенные в памяти, как последовательности битов В языке С имеются следующие операции этого типа:

& – ***конъюнкция*** или *поразрядное умножение*,
| – ***дизъюнкция*** или *поразрядное сложение*,
^ – ***сложение по модулю 2*** или *исключающее или*,
~ – ***инверсия***,
<< – ***сдвиг влево***,
>> – ***сдвиг вправо***.

Поразрядные операции применяются только к целочисленным операндам.

Рассмотрим работу этих операций.

1. Операция &.

Операция & является *бинарной*, т.е , имеет два операнда Она выполняет логическое умножение одноименных разрядов двух целочисленных операндов одинаковой длины

Пример.

```
int a = 5, b = 12, c
c = a & b,
```

Результат операции будет следующим

```
      a = 0...000101
&     b = 0...001100
      ---------------
      c = 0...000100
```

Т.е., в результате выполнения операции переменная *c* примет значение 4.

2. Операция |.

Операции | выполняет поразрядное логическое сложение двух целочисленных операндов одинаковой длины

Пример.
int a = 5, b = 12, c,
c = a | b,

Результат операции будет следующим

```
    a = 0.. 000101
|   b = 0.. 001100
  ---------------------
    c = 0...001101
```

3. Операция ^.

Операция ^ выполняет поразрядное сложение по модулю 2 двух целочисленных операндов одинаковой длины.

Пример.
int a = 5, b = 12, c,
c = a ^ b,

Результат операции:

```
    a = 0...000101
^   b = 0...001100
  ---------------------
    c = 0.. 001001
```

4. Операция ~

Операция ~ является *унарной*, т е. имеет *один операнд*. Разряды операнда инвертируются – меняют значение на противоположное.

Пример.
int a = 5, c
c = ~a,

Результат операции:

```
    c = ~0. .000101 = 1...111010
```

5. Операция сдвига влево <<.

Операция имеет формат:

операнд << k

где операнд – сдвигаемое данное, а *k* – величина сдвига.

Операция сдвига << выполняет сдвиг влево всех разрядов целочисленного операнда на указанное количество позиций *k*. При этом левые *k* разрядов теряются, а правые *k* заполняются нулями.

Пример.

```
int a = 5;                  // a = 0  000101
c = a << 2,
```

Переменная *c* получит значение 0...010100.

6. Операция сдвига вправо >>.

Операция имеет формат:

операнд >> k

Операция сдвига вправо >> выполняет сдвиг разрядов операнда вправо, но имеет одну существенную особенность: при сдвиге знаковых данных разряд знака копируется вправо на *k* позиций

Например, при следующем сдвиге положительного числа

```
int b = 12;                 // b = 0...001100
b = b >> 2,
```

операция будет выполняться аналогично сдвигу влево. 2 правых разряда потеряются, а слева добавятся 2 нулевых разряда, т.е. *b* получит значение 0...000011

А при сдвиге отрицательного числа

```
int b = -12,
```

которое в дополнительном коде имеет значение 1111...110100 (жирным выделен разряд знака), вправо

```
b = b >> 2;
```

получим:

b = 1111...111101.

Здесь, разряд знака распространился на 2 позиции вправо, т.е. слева появились две новые 1

Такое выполнение операций сдвига позволяет применять их для целочисленного умножения (при сдвиге влево) и деления (при сдвиге вправо) на степень двойки. Т.е. при сдвиге влево на *k* позиций, выполняется умножение операнда на 2^k, а при сдвиге вправо на *k* позиций выполняется деление на 2^k. Так, в нашем примере, при сдвиге на 2 позиции вправо переменной *b* = -12, эта

переменная примет значение -3 в дополнительном коде, т.е, выполнится деление переменной на 4 (2^2).

Заметим, что в большинстве случаев, поразрядные операции используются для работы над данными, которые представляют собой последовательности нулей и единиц, т.е. логическими или булевыми векторами В этих задачах и знаковый разряд обычно рассматривается как компонента вектора При сдвиге знакового разряда вправо, он может копироваться нежелательным образом Поэтому при работе с булевыми векторами тип данных (как правило) надо определять как *беззнаковый*, т.е. с предшествующим словом *unsigned*

Прежде, чем рассмотреть примеры работы с булевыми векторами, рассмотрим понятие *маски*. Дело в том, что нет операции, которая бы позволила проанализировать значение одного бита данного Рассмотренные выше операции выполняются над всеми разрядами булева вектора одновременно. Поэтому, чтобы узнать или изменить значение одного или нескольких разрядов, используется вспомогательная переменная или константа, которую часто называют ***маской***

Например, чтобы узнать значение некоторой компоненты вектора, можно использовать *маску*, содержащую единицу только в том разряде, который нас интересует, а в остальных нули Выполнив операцию логического умножения булева вектора и *маски*, по результату операции можно определить значение интересуемого разряда Если разряд равнялся 1, то при умножении на *маску*, результат умножения будет отличен от нуля, а если равнялся 0, то результат будет нулевым во всех разрядах

Рассмотрим примеры работы с булевыми векторами

Замечание Для того чтобы сделать программу более ясной, можно дать типу новое имя с помощью ключевого слова ***typedef***:

typedef тип новое_имя

Введенное т о имя можно использовать также как и имена стандартных типов

Для решения логических задач определим тип данных *word* – *unsigned int*.

Пример. Необходимо определить вес булева вектора, т.е. колнчество его единичных компонент.

```
#include <stdio.h>
typedef unsigned int word;

void main()
{ word v = 0x5555,          // v – чередование 20 нулей и единиц
                            // 0x определяет шестнадцатеричную константу
  unsigned int maska = 1,
                            // maska содержит одну единицу в младшем
                            // разряде, все остальные – нули
  int w = 0,                // w – вес вектора
  // на каждой итерации цикла будем проверять, находится ли в
  // младшем разряде вектора 1 и затем сдвигать вектор на один
  // разряд вправо
  while (v)                 // пока в векторе есть единицы
                            // он будет отличен от 0
  { if (v & maska) w++      // если результат умножения отличен от нуля,
                            // то увеличим вес на 1
    v = v >> 1,             // сдвинем вектор вправо на 1 разряд
  }
  printf("\nВес вектор %d\n", w),
}
```

Булеву матрицу можно задать как массив булевых векторов.

Рассмотрим два примера работы с булевыми матрицами

Для решения задач определим две функции: inbool – функция ввода булевого вектора и outbool – функция вывода булевого вектора.

```
#include <stdio h>
typedef unsigned int word;

word inbool()
{ char s[33],
  gets(s),
  word v = 0,
```

```
 for (int i = 0, s[i]; i++)
 { v <<= 1
   if (s[i] == '1') v++,
 }
 return v,
}

void outbool(word v, int n)
{ word m = 1 << n - 1,
  putchar('\n'),                          // Функция putchar выводит символ
  while(m)
  { if (v & m) putchar('1'),
    else putchar('0'),
    m >>= 1,
  )
}
```

Пример 1 Необходимо найти столбец с минимальным весом в булевой матрице m × n (n <= 32).

```
void main()
( word a[32], maska,
  int m  n, i  j, w  max_w = 0, max_i = 0
  printf("Введите m и n  \n")
  scanf("%d%d", &m, &n)
  fflush(stdin),
  for (i = 0, i < m, i++)
    a[i] = inbool(),
  for (i = 0, maska = 1 << n - 1, i < n, i++, maska >>= 1)
  { for (j = 0, w = 0; j < m, j++)
      if (a[j] & maska) w++,
    if (w > max_w) { max_w = w, max_i = i, }
  }
  printf("\nСтолбец  %d, максимальный вес  %d\n",max_i + 1, max_w)
}
```

Пример 2 Необходимо инвертировать элементы, расположенные выше главной диагонали квадратной булевой матрицы n × n (n <= 32)

Инвертирование элементов будем выполнять построчно, складывая по модулю два строкн с *маской*. Для первой строки *маска* должна состоять из одного 0 и (*n* – 1) единиц. Эта *маска* формируется по формуле (1 << n - 1) - 1. Для следующих строк *маску* получаем сдвигом *маски* предыдущей строки иа одну позицию вправо.

```
void main()
{ word a[32], maska,
  int n, i,
  printf("Введите n: "),
  scanf("%d", &n),
  fflush(stdin);
  for (i = 0, i < n, i++)
    a[i] = inbool();
  for (i = 0, maska = (1 << n - 1) - 1, i < n, i++, maska >>= 1)
  { a[i] = a[i] ^ maska,
    outbool(a[i], n);
  }
}
```

2. Локальные и глобальные переменные

Вспомним, что под *составным оператором* понимают последовательность операторов, заключенную в фигурные скобки Если среди операторов, находящихся в фигурных скобках, имеются определения и описания, то составной оператор называют *блоком*

Переменная, определенная внутри блока, называется ***локальной*** Переменная, определенная вне любого блока, называется ***глобальной***.

Если глобальная переменная явно не инициализирована при описании, то она автоматически инициализируются 0.

Локальные переменные, у которых не задан класс памяти, называются ***автоматическими*** Автоматические переменные создаются при каждом входе в блок, где они описаны, и уничтожаются при выходе из него. Эти переменные при описании автоматически ничем не инициализируются, то есть, если автоматическая переменная не инициализирована явно, то ее значение непредсказуемо

Областью действия идентификатора (имени переменной) называется часть текста программы, в которой идентификатор можно использовать для доступа к связанному с ним участку памяти

Область действия идентификатора глобальной переменной – от точки ее описания до конца файла, в котором она определена. Область действия идентификатора локальной переменной – от точки ее описания до конца блока, в котором она определена, включая все вложенные блоки.

Глобальные переменные удобно использовать в том случае, когда различные функции модуля программы должны работать с общими данными.

Рассмотрим еще один атрибут переменной.

Областью видимости идентификатора переменной называется часть текста программы, из которой допустим обычный доступ к этой переменной Область видимости идентификатора некоторой переменной x не совпадает с областью его действия тогда, когда во вложенном блоке, входящем в область действия этой переменной, описана переменная с таким же именем В этом случае

внешняя переменная *x* во вложенном блоке невидима. Если эта переменная глобальная, то к ней можно обратиться, используя операцию доступа к области видимости .::.

Пример.

Рассмотрим на примере области действия и области видимости идентификаторов переменных программы. Для удобства пронумеруем строки программы (нумерация строк не является частью текста программы).

```
1       # include <stdio h>
        // Область действия a – строки 2 – 17,
        // область видимости – строки 2 – 10 и 15 – 17;
2       int a;                  // переменная a – глобальная,
                                // она автоматически инициализируется 0
        // функция возводит в квадрат две переменные
3       int ab_sq(int b)
4       { a = a * a,
          b = b * b,
          return b;
5       }

6       int main()
        // Область действия идентификатора локальной автоматической
        // переменной b, определенной в функции main – строки 7 – 17,
        // область видимости – строки 7 – 10 и 15 – 17,
7       { int b = 4;
8         b = ab_sq(b);
9         printf("\na = %d, b = %d", a, b),              // a = 0, b = 16
10        a = 2;
          // Область действия и область видимости идентификаторов
          // локальных автоматических переменных a и b,
          // определенных в блоке, – строки 11 – 14,
11        { int b = 3, a = 1;
12          b = ab_sq(b),
13          printf("\na = %d,  a = %d, b = %d", a,  a, b),  // a = 1,  a = 4, b = 9
14        }
15        b = ab_sq(b);
16        printf("\na = %d, b = %d", a, b),              // a = 16, b = 256
17      }
```

3. Структуры и массивы структур

Структура - это производный тип данных Структура состоит из произвольного количества элементов – *полей*, в общем случае разного типа Формат описания структуры:

struct имя_типа { ***описание_полей_через_точку_с_запятой*** };
здесь, *имя_типа* – идентификатор, определяющий новый тип данных, отличающийся от базовых (*int*, *char*,..) и определенных ранее типов данных

Поля структуры описываются так же, как описываются переменные в программе Поля одного типа можно описать, указав тип данных один раз и далее перечислив имена всех полей этого типа через запятую.

Пример.
Зададим структуру для хранения данных абитуриентов

```
struct abit { char name[20], town[15], int age, },
```

Данное описание определяет новый тип даиных ***abit*** Он состоит из трех полей два поля (***name*** и ***town***) строкового типа *char** и одно поле (***age***) целого типа *int*

При определении переменной типа *abit* значения её полей могут быть проннициализнрованы в соответствин с типом каждого поля н его порядком в определении структуры.

Например,

```
void main()
{ abit a = { "Иванов И И ", "Москва", 17 },
  printf("%s %s %d\n", a.name, a.town, a age),
}
```

Здесь, определяется переменная *a* структурного типа *abit* и выполияется инициализация ее полей *name* и *town* строкового типа и *age* целого типа При инициалнзацни значения полей заключаются в фнгурные скобки

Действия со структурой выполияются иад отдельиыичи полями Так, вывод переменной структурного типа в примере выполияется по полям каждое поле выводится с указанием соответствующего спецификатора вывода

Обращение к полю структуры выполняется с помощью операции "." (точка) следующим образом.

имя_переменной_структурного_типа.имя_поля

Как и для других типов данных для структурного типа можно определять указатели

Например.

```
abit a = { "Иванов И И ", "Москва", 17 }, *b,
b = &a,
```

Так как *b* содержит адрес структуры, то получить значение структуры можно выполнив операцию *разадресации* *, т е **b*, а для обращения в структуре к конкретному полю используется операция "*точка*". Таким образом, обращение, например, к полю *age* структуры с адресом *b* будет иметь вид:

```
(*b).age
```

Для сокращения такой записи введена операция "->", используя эту операцию, к полю *age* можно обратиться следующим образом:

```
b -> age
```

Можно определить *массив структур* в статической или динамической области Статический массив может быть проинициализирован поэлементно. Каждый элемент инициализируется в фигурных скобках { }.

Рассмотрим решение задач с использованием структур

Пример. Задан список фамилий учеников нескольких классов школы и их рост Необходимо определить в каком классе учится самый высокий ученик

```
struct uchenik { char fio[20] klass[4]; int rost, },
void main()
{ uchenik p[10] = { { "Антонов А", "10а", 172 },
                    { "Петрова П", "10б", 160 },
                    { "Сергеев С", "11б", 181 },
                    { "Матвеева М", "10а", 165 } },      // определены 4
                                                         //элемента массива структур
  int n = 4, max, k, i;
  // за начальное значение переменной max берем рост первого ученика
```

```
 for (i = 1, max = p[0].rost, k = 0; i < n, i++)
   if (p[i].rost > max) { max = p[i].rost, k = i, }
 printf("max рост %d у ученика %s из класса %s\n", max, p[k].fio,
 p[k].klass),
}
```

Заметим, что в списке учеников проннициалнзироваио только 4 элемента, остальные будут проинициалнзированы по умолчанию: строковые данные – пустой строкой, числовые – нулями. Если иннцнализация отсутствует, все поля всех элемеитов локального массива p будут не определены.

Рассмотрим определение массива структур в динамической области. В этом случае ннициалнзацня невозможна.

Пример. Заданы два спнска результатов сдачи двух предметов абнтуриентамн. Не все абитуриенты сдавали оба экзамена. Необходимо сформировать упорядоченный по убыванию суммарного балла список абитурнентов, сдавших оба экзамена.

Заметнм, что исходиые спнски и результирующий список будут иметь одинаковую структуру.

```
struct abit { char fio[20], int bali; };

void main()
{ abit *l1, *l2, *l3, t,
  // l1, l2 – указатели на исходные списки, l3 – на итоговый,
  // t – структура для обмена
  int n, m, i, j, k = 0,
  scanf("%d%d", &n, &m);          // n – длина первого списка,
                                  // m – длина второго списка
  l1 = new abit[n],       // запрашиваем память под первый список
  l2 = new abit[m],       // запрашиваем память под второй список
  l3 = new abit [n < m ? n : m], // память для итогового списка берем
                                 // по минимальной длине исходных списков
  puts("\nВведите первый список:");
  for (i = 0, i < n, i++)
  { fflush(stdin),                // очистка буфера ввода
    gets(l1[i].fio);              // ввод Ф И.О.
```

```
  scanf("%d", &l1[i].ball),            // ввод балла
}
puts("\nВведите второй список "),
for (i = 0, i < m, i++)
{ fflush(stdin);                       // очистка буфера ввода
  gets(l2[i].fio),                     // ввод Ф.И О
  scanf("%d", &l2[i].ball),            // ввод балла
}

// абитуриенты, сдавшие оба экзамена имеются в обоих списках
// ищем таких
for (i = 0; i < n, i++)
{ for (j = 0, j < m && strcmp(l1[i] fio, l2[j].fio), j++);   // сравниваем
                                       // фамилии функцией strcmp!
  if (j < m)                           // нашли,
  { strcpy(l3[k].fio, l1[i].fio),      // записываем в новый список Ф.И.О.
    l3[k++].ball = l1[i].ball + l2[j].ball;   // и суммарный балл
  }
  // Заметьте, присваивание фамилий выполняется не операцией '=',
  // а функцией strcpy!
}

// Упорядочиваем итоговый список по убыванию суммарного балла
// методом "пузырька"
for (i = 0; i < k - 1; i++)
  for (j = 0, j < k - i - 1, j++)
    if (l3[j].ball < l3[j + 1].ball) { t = l3[j], l3[j] = l3[j + 1]; l3[j + 1] = t, }
puts("\nОба экзамена сдали абитуриенты: ");
for (i = 0; i < k, i++)
  printf("\n %20s %3d", l3[i] fio. l3[i] ball);
)
```

Замечание Целиком к структуре применима только операция присваивания '='; оба операнда операции должны быть одного структурного типа При применении операции присваивания происходит поэлементное копирование. В примере она использовалась для обмена значений двух соседних элементов массива через вспомогательную переменную t структурного типа

Другие операции применимы только к полям структур.

Полем структуры может быть и другая нзвестная структура.

Пример. Задаи спнсок студентов: фамилия, группа, дата рождения Необходимо вывести список студеитов, деиь рождения которых в феврале

```
// date - структура для даты рождения
struct date { int day, char month[10]; int year, };
struct student { char name[20], int gr, date bd, };        // данные о студенте
```

Так как дата рождення является структурой, то прн ннницнализацин даты её поля будут заключаться в скобки {}. Таким образом, инициализация массива данных о студентах будет содержать трн уровня скобок {}

- для всего массива;
- для каждого элемента массива,
- для даты рождення каждого студента

```
void main()
{ student fpmk[3] = { { "Антонов А.", 1155, { 10, "март", 1990 } },
                      { "Алексеева В.", 1165, { 14, "февраль", 1991 } },
                      { "Матвеева С.", 1172, { 25, "июль", 1991 } }
                    },
  int i, n = 3,
  puts("\nРодились в феврале ");
  for (i = 0; i < n, i++)
    if (strcmp(fpmk[i].bd.month, "февраль") == 0)
      printf("\n%s %d\n", fpmk[i] name, fpmk[i].bd.day),
}
```

Замечаиие Прн сравнении месяца рождення каждого студента использовалнсь 2 операцни "точка"

- первая – для выделення всего поля ***bd*** даты рождення в элементе массива,
- вторая – для обращения к полю месяца ***month*** в дате рождеиия

Аналогнчио прн выводе дня рождения

Замечаиие. Поле структуры не может иметь тнп определяемого структуриого типа

Полем структуры может быть указатель.

В рассмотрением выше примере в структуре для фамилии определялась строка длиной в 20 байт, хотя фактически в строках размещались фамилии меньшей длины. Используя указатели и операцию *new*, для каждой фамилии можно выделять необходимую для ее хранения память. Поскольку до ввода фамилий их длины неизвестны, будем использовать дополнительную строку – *буфер ввода*:

- вводим фамилию в *буфер ввода*,
- затем, определив длину фамилии, берем память для поля *name* структуры;
- копируем в поле *name* из *буфера* введенную фамилию

Пример. Задан список абитуриентов: Ф.И.О , город откуда прибыл, возраст. Необходимо вывести фамилии томичей

```
struct abit { char *name, *town, int age; };

void main()
{ abit *fpmk,              // определяем указатель динамического массива
  char buf[30],            // и буфер для ввода
  int i, n,
  printf("\nВведите количество абитуриентов: ");
  scanf("%d", &n),
  fpmk = new abit[n]; // берем память под массив в динамической области
  puts("Введите данные об абитуриентах Ф.И О , город, возраст"),
  // ввод данных
  for (i = 0, i < n, i++)
  { puts("ФИО?");
    fflush(stdin);
    gets(buf),                                  // вводим фамилию в буфер
    fpmk[i].name = new char[strlen(buf) + 1]; // берем память по длине ФИО
                                                // (+1 для нуль-кода)
    strcpy(fpmk[i].name, buf), // копируем ФИО из буфера в поле структуры

    puts("Город?"),          // действуем аналогично для ввода города
    gets(buf);
    fpmk[i].town = new char[strlen(buf) + 1],
    strcpy(fpmk[i].town, buf);
```

```
    puts("Возраст?");
    scanf("%d", &fpmk[i].age);
  }
  puts("\nТомичи ");
  for (i = 0, i < n; i++)
    if (strcmp(fpmk[i].town, "Томск") == 0)     // Томич?
      puts(fpmk[i].name);                       // Да - выводим фамилию
}
```

4. Линейные списки

4.1. Понятие списка. Операции добавления и удаления элементов списка

Одни из недостатков представления данных в виде массива – расположение их в непрерывном участке оперативной памяти. При таком представлении, для вставки числа в массив, необходимо элементы массива переписать на одну позицию вправо начиная с конца массива до места вставки (сдвинуть массив вправо); для удаления – элементы массива переписываются на одну позицию влево, начиная от элемента, стоящего справа от удаляемого (сдвиг массива влево). Например, если в массиве из 1000 элементов удаляется 3-ий, то 997 элементов, начиная с 4-го, переписываются на одну позицию влево. Если операции вставки и удаления выполняются часто, то они приводят к большим затратам времени. Представление данных в виде ***связного списка*** позволяет исключить этот недостаток.

Пример связанного списка представлен на рисунке 1

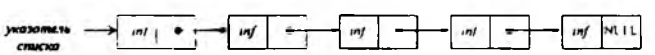


Рисунок 1. Пример связанного списка

Здесь прямоугольником представлен один элемент списка, ***inf*** – информационное поле (поле данных) любого типа (символ, число, массив, структура, .); второе поле элемента – поле связи со следующим таким же элементом, оно содержит *адрес* этого элемента, т.е. является указателем. Назовем второе поле – ***next***. ***Указателем списка*** называется адрес начала списка, т.е. его первого элемента. Поле ***next*** у последнего элемента не содержит адреса следующего элемента и равно нулю (NULL). Элементы списка часто называют ***узлами***. Узел может содержать и несколько полей данных и указателей. Если указатель один, как представлено на рисунке 1, то список называется ***однонаправленным*** (или ***односвязным***).

Рассмотрим, каким образом выполняются удаления и добавления элементов в однонаправленном списке.

Если потребуется удалить какой-либо узел, то, находясь на предшествующем узле с адресом *p*, достаточно в поле связи *next* этого узла занести адрес связи из поля *next* удаляемого узла

Операцию удаления узла из списка иллюстрирует рисуиок 2.

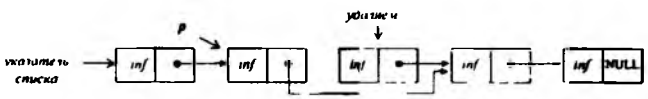


Рисунок 2. Удаление узла из списка

Таким образом, удаление узла списка, расположеииого после узла с адресом *p*, выполняется следующим образом

p -> next = p -> next -> next;

Для добавлеиия иового узла с адресом *t* после узла с адресом *p*, в поле связи *next* нового узла заносим зиачение поля связи *next* из узла с адресом *p*. Затем, в поле связи *next* узла с адресом *p* заиосим адрес иового узла *t*.

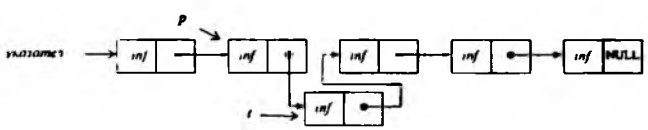


Рисунок 3. Добавлеиие узла в спнсок

Таким образом, вставка элемента выполняется всего за 2 действия.

t -> next = p -> next;

p -> next = t;

Операцию добавления узла в связаниый список иллюстрирует рисуиок 3.

Замечаиие Переменные *p* и *t* в рассмотренных примерах являются вспомогательиыми указателями

Рассмотрим пример создания списка целых чисел и фуикций работы с ним

Узел списка будет определяться структурой, состоящей из двух полей k и next

Рассмотрим список, для которого определяется только указатель p на начало списка.

```
struct Node { int k, Node *next; };

// Ф-ия помещает число k в конец списка и возвращает
// (возможно новый) указатель на начало списка
Node *Add(Node *p, int k)
{ Node *t = new Node, *q = p;       // берем память под новый узел
  t -> k = k;                       // в информационное поле помещаем число k
  t -> next = 0;                    // т.к  t – последний узел, в поле связи заносим 0
  if (!p) return t,                 // если список был пустым, то возвращаем адрес
                                    // нового узла
  while (q -> next)                 // передвигаем указатель q на последний
    q = q -> next,                  // узел списка
  q -> next = t;                    // в поле связи бывшего последнего узла
                                    // заносим адрес нового узла t
  return p,                         // указатвль на начало списка остается прежним
}

// Ф-ия выводит числа, находящиеся в списке
void Pnnt(Node *p)
{ for ( ; p; p = p -> next)         // идем по узлам списка
    cout << p -> k << " ",
}
```

Рассмотрим операцию удаления из списка произвольного элемента с каким-либо свойством. Как было рассмотрено выше для удаления необходимо иметь адрес узла предшествующего удаляемому Удаляемый узел может быть любым, в частности первым. У первого узла нет предшестающего Поэтому удаление произвольного узла из односвязного списка можно выполнить следующим образом

1 Создается ***фиктивный узел*** с адресом f Он связывается с первым узлом списка, то есть, в его поле связи next заносится указатель на начало списка (адрес начала списка) Информацнонное поле узла с адресом f игнорируется

2. Вспомогательный указатель, например, t, устанавливается на фиктивный узел.

3 Организуется поиск узла удаления путем анализа следующих за t узлов

4 Если узел для удаления найден, выполняется его удаление из списка уже описанным ранее способом

```
t -> next = t -> next -> next
```

Если удаляемый узел будет первым, то в поле next фиктивного узла с адресом f занесется адрес второго узла списка. Таким образом, указатель списка может измениться, и его новое значение будет находиться в поле next фиктивного узла. Заметим, что перед удалением узла из списка его адрес запоминается во вспомогательной переменной для последующего освобождения памяти от этого узла.

Ниже приведена функция удаления узла из списка целых чисел с заданным значением информационного поля.

```
// Ф-ия удаляет из списка число m и возвращает (возможно новый)
// указатель на начало списка
Node *Del_m(Node *p, int m)
{ Node *f = new Node, *t,          // создаем фиктивный узел с адресом f
  f -> next = p,                   // связываем фиктивный узел с
                                   // началом списка p
  // передвигаемся по узлам списка и ищем среди них такой,
  // у которого значение поля k следующего за ним узла равно m
  for (t = f; t -> next && t -> next -> k != m, t = t -> next),
  if (t -> next == 0)
  { puts("\nЧисло для удаления не найдено"),
    delete f; return p,
  }
  p = t -> next,             // запоминаем адрес удаляемого узла
  t -> next = t -> next -> next,     // удаляем узел из списка
  delete p;                          // освобождаем занятую им память
  t = f -> next,                     // запоминаем адрес начала списка
                                     // (возможно он изменился)
  delete f,                  // освобождаем память от фиктивного узла
  return t;                  // возвращаем адрес начала списка
}
```

Функция Del_m удаляет из списка первый узел со значением m. Следующая функция удаляет из списка все узлы с заданным значением m

Замечание. Пусть в следующем за узлом с адресом t узле находится число m, которое надо удалить. После удаления указатель t надо оставить на том же месте, т.к. новый узел, подсоединенный к t, тоже может содержать число m

```
// Ф-ия удаляет из списка все вхождения числа m и
// возвращает (возможно новую) вершину
Node * Del_all_m(Node *p, int m)
{ Node *f = new Node, *t, *q.        // создаем фиктивный узел
  f -> next = p,                     // связываем его с началом списка
  t = f                              // t устанавливаем на фиктивный узел
  while (t -> next)                  // пока есть следующий узел
    if (t -> next -> k == m)  // в следующем узле значение поля k равно m ?
    { q = t -> next;                 // да, запоминаем адрес удаляемого узла
      t -> next = t -> next -> next.  // удвляем его из списка
      delete q.                      // освобождаем занятую им память
    }
    else t = t -> next               // если в следующем узле не число m,
                                     // переходим дальше по списку
  q = f -> next;                     // запоминаем адрес начала списка
                                     // (возможно измененный)
  delete f;                          // освобождаем память от фиктивного узла
  retum q;                           // возвращаем адрес первого узла списка
}
```

Приведем пример использования определенных выше функций.

Создадим список из n случайных чисел Удалим из него первое вхождение числа k и все нули

```
void main()
{ int n, k, i;
  srand(time(0));
  Node *beg = NULL,.                 // список еще пуст
  pnntf("n = "),
  scanf("%d", &n);
  for (i = 0; i < n; i++)
  { k = rand()%5;                    // генерируем случайные числа и
```

```
  beg = Add(beg, k),                  // заносим их в список
 }
 Pnnt(beg);                           // выводим список
 printf("\nk = "), scanf("%d",&k),    // вводим число k
 beg = Del_m(beg, k);                 // удаляем первое вхождение числа k
 pnntf("\nСписок после удаления числа %d. \n", k).
 Print(beg);
 beg = Del_all_m(beg, 0),             // удаляем все вхождения числа 0
 puts("\nСписок после удаления иулей ");
 Print(beg).
}
```

В примере мы рассмотрели список с одиим указателем иа начало списка, однако для удобства работы со списком можно ввести также указатель на коиец списка.

Существуют разные способы формирования связаиных списков и работы с иими. Рассмотрим некоторые виды списков

4.2. Стеки

Стек - частиый случай связанного списка. Добавлеиие и удалеиие элементов в стек выполияется по принципу **LIFO (Last In – First Out** последним пришел – первым ушел). Это озиачает, что добавлеиие и удалеиие выполняется с одного конца списка, который называется ***вершиной стека*** Адрес вершины хранится в специальиой переменной – ***указателе стека*** Указатель стека содержит адрес последиего занесениого в стек элемента

Рассмотрим пример создания стека целых чисел и функций работы с ним.

Узел стека будет определяться структурой, состоящей из двух полей k и next.

```
struct Node { int k; Node *next, },

// Ф-ия помещает число k в вершину стека p и возвращает
// адрес новой аершины
Node *Push(Node *p, int k)
{ Node *t = new Node,    // берем память под новый узел
  t -> k = k;            // в информационное поле помещаем число k
  t -> next = p,         // в поле связи заносим адрес старой вершины p
```

```
 return t;                  // возвращаем адрес нового узла t
                            // в качестве новой вершины стека
}

// Ф-ия удаляет элемент из вершины стека p и возвращает адрес
// новой вершины
Node *Pop(Node *p)
{ if(!p) return 0;          // если стек пуст, возвращаем 0
  Node *t = p -> next,      // иначе новой вершиной
                            // будет следующий за p узел
  delete p,                 // освобождаем память от узла старой вершины
  return t;                 // возвращаем адрес новой вершины
}

// Ф-ия возвращает значение из вершины стека p
int Top(Node *p)
{ if (p) return p -> k,     // если стек не пуст, возвращаем число
                            // из вершины стека не удаляя узел
  puts("\nСтек пуст"), getchar(), exit(1),      // иначе выводим сообщение
                                                // и выходим из программы
}

// проверка стек пуст?
bool Empty(Node *p)
{ if (p) return false,
  return true,
}
```

Замечание Для того чтобы при работе функции Top не возникало прерывание программы при пустом стеке, корректнее перед обращением к функции Top проверить стек пуст или нет функцией Empty Например, следующим образом:

```
if (!Empty(p)) k = Top(p),
```

Используя стек, решим задачу.

Задан массив целых чисел. Необходимо его "обернуть".

```
int main()
{ int n = 7, i, a[7] = { 1, 2, 3, 4, 5, 6, 7 }, k;
```

```
Node *beg = NULL,                  // Первоначально стек пуст и
                                   // адрес вершины beg равен 0
for (i = 0; i < n, i++)
( printf("%d ", a[i]),             // выводим на экран исходный массив
  beg = Push(beg, a[i]),           // заносим очередное число в стек
                                   // в указателе beg запоминается адрес
                                   // новой вершины
)
for (i = 0, !Empty(beg), i++)      // пока стек не пуст
( k = Top(beg),                    // берем число из вершины стека
  a[i] = k;                        // и записываем его в массив
  beg = Pop(beg),                  // удаляем число из стека
)
puts("\nЧисла в обратном порядке  "),
for (i = 0, i < n, i++) printf("%d ", a[i]);
)
```

4.3. Очереди

Добавление и удаленне элементов в списке "***очередь***" выполияется по прнницпу **FIFO** (First In – First Out· первым пришел – первым ушел). Добавление элементов выполияется в конец очереди, а удаление с начала Для работы с таким списком обычно формируют **2 указателя** – на иачало очереди и на коиец

Определим функции для работы с очередью целых чисел

```
struct Node { int k, Node *next, },

// Ф-ия помещает число k в конец очереди и возвращает, возможно
// измененный, адрес конца очереди, e – указатель на конец очереди
Node *Add_e(Node *e, int k)
{ Node *t = new Node,              // берем память под новый узел
                                   // с адресом t
  t -> k = k; t -> next = 0,       // определяем его поля
  // если очередь пустая, то возвращаем адрес нового узла
  if (e == 0) return t;
  // если не пустая  связываем последний узел с новым узлом
  e -> next = t,
```

```
  e = t;                          // новый узел становится последним
  return e;
}

// Ф-ия удаляет из очереди первый элемент и
// возвращает адрес нового начала очереди
Node *Del(Node *b)
{ if (b)
  { Node *t = b;                  // запоминаем адрес первого элемента
                                  // для освобождения памяти
    b = b -> next;                // указатель на начало очереди устанавливаем
                                  // на второй элемент очереди
    delete t;                     // освобождаем память от удаленного узла
  }
  return b;                       // возвращаем адрес нового начала очереди
}

// Ф-ия выводит числа, находящиеся в очереди
void Print(Node *b)               // b – адреса начала очереди
{ for (; b; b = b -> next)
    cout << b -> k << " ";
}
```

Для примера работы с очередью определим 2 функции

- создания очереди из n случайных упорядоченных по возрастанию чисел;
- слияния двух упорядоченных очередей в третью упорядоченную.

```
// Ф-ия создает упорядоченную по возрастанию очередь из n чисел
Node *Create(int n)
{ Node *b = 0, *e;
  int k, i;
  b = Add_e(b, k = rand()%10);    // начальный узел создаем до цикла
  e = b;                          // он же пока последний
  for (i = 1; i < n; i++)         // остальные узлы добавляем в конец очереди
    e = Add_e(e, k += rand()%10);

  return b;                       // возвращаем адрес начала очереди
}
```

Функция слияния двух упорядоченных очередей в третью не будет формировать узлы новой очереди, а будет на каждом шаге присоединять к новой очереди узел с наименьшим числом из одной из двух исходных очередей. Два вспомогательных указателя p1 и p2 будут двигаться по исходным очередям. Одна из очередей закончится раньше. Тогда оставшиеся числа другой очереди присоединятся к новой очереди. Первый узел новой очереди зададим *фиктивным*

```
// Ф-ия сливает две упорядоченных очереди в третью упорядоченную
Node * twointo1(Node *b1, Node *b2)
// создаем фиктивный узел с адресом f для новой очереди
// определяем два вспомогательных указателя p1  и p2
{ Node *p1, *p2, *f = new Node, *t = f;
  // пока числа есть в обеих очередях
  for (p1 = b1  p2 = b2, p1 && p2, t = t -> next)
    if (p1 -> k <= p2 -> k)
    // присоединяем к новой очереди узел из первой очереди
    { t -> next = p1, p1 = p1 -> next, }
    else { t -> next = p2, p2 = p2 -> next, }     // или узел из второй очереди
  if (p1) t -> next = p1,     // если не закончилась первая очередь или
  else t -> next = p2,        // не закончилась вторая очередь присоединяем
                              // оставшиеся узлы к новой очереди
  t = f -> next,              // записываем в t адрес начала новой очереди
  delete f;                   // освобождаем память от фиктивного узла
  return t;                   // возвращаем адрес нового начала очереди
}

Рассмотрим пример использования определенных функций.
void main( )
{ srand(time(0));
  Node *beg1, *beg2, *beg3;
  beg1 = Create(10),              // создаем первую упорядоченную очередь
  Print(beg1);  puts(""),
  beg2 = Create(7),               // создаем вторую упорядоченную очередь
  Print(beg2);  puts("");

  beg3 = twointo1(beg1, beg2);  // сливаем их в третью упорядоченную
  Print(beg3),
}
```

Заметим следующее

1 В рассмотренном примере недостатком является то, что очереди создавались в функции

Node * Creale(int n)

которая возвращает адрес начала очереди. Указатель конца очереди остался определенным только в функции и недоступен в других функциях (в частности в main( )).

2. В функции слияния

Node * twointo1(Node *b1, Node *b2)

исходные очереди преобразуются в третью упорядоченную очередь путем изменения связей (значений полей *next*) Функция возвращает адрес начала результирующей очередн Указатель конца очереди также не определяется

Таким образом, функция формирования исходных очередей и функция слияния не формируют указатель коица очереди, поэтому добавить новые элементы в конец очереди нет возможности Это связано с тем, что функция может вернуть только одно значение, а структура данных *очередь* имеет два указателя.

Устранить этот недостаток можно несколькими способами. Приведем три варианта решения проблемы, основанные на изученном материале

Первый вариант Указатель конца очереди end, вычисляемый в функции Create, определяем как глобальную переменную В функции main( ) переписываем значение указателя end в указатель иа конец соответствующей очереди Напомним, что глобальная переменная определяется вне функций в тексте программы и областью ее видимости являются все функции, расположенные ннже, если в них не определены переменные с тем же именем. Таким образом, функция Create может быть определена так.

```
Node *end;     // глобальная переменная – указатель на конец очереди
.
Node *Create(int n)
{ Node *b = 0;
  int k, i;
  b = Add_e(b, k = rand()%10),      // начальный узел создаем до цикла
  end = b,                          // он же пока последний, указатель на
                                    // него определен как глобальная переменная
  for (i = 1, i < n; i++)           // остальные узлы добавляем в конец очереди
```

```
    end = Add_e(end, k += rand()%10);
  return b;
}
```

В функции слияния добавим формирование указателя на конец результирующей очереди также в глобальной переменной **end**, задав в аргументах функции не только указатели на начала очередей (**b1** и **b2**), но и на концы очередей (**e1** и **e2**).

```
// Ф-ия сливает две упорядоченных очереди в третью упорядоченную
Node * twointo1(Node *b1, Node *b2, Node *e1, Node *e2)
// создаем фиктивный узел с адресом f для новой очереди,
// определяем два вспомогательных указателя p1 и p2
{ Node *p1, *p2, *f = new Node, *t = f,
  // пока числа есть в обеих очередях
  for (p1 = b1, p2 = b2, p1 && p2, t = t -> next)
    if (p1 -> k <= p2 -> k)
    // присоединяем к новой очереди узел из первой очереди
    { t -> next = p1, p1 = p1 -> next  }
    else { t -> next = p2, p2 = p2 -> next,)      // или узел из второй очереди
  // если не закончилась первая очередь или
  if (p1) { t -> next = p1  end = e1; }
  // не закончилась вторая очередь,
  // присоединяем оставшиеся узлы к новой очереди
  else { t -> next = p2; end = e2; }
  t = f -> next;              // записываем в t адрес начала новой очереди
  delete f;                   // освобождаем память от фиктивного узла
  return t;                   // возвращаем адрес нового начала очереди
}
```

Теперь в функции main( ) можно использовать эти функции.

```
void main()
{ srand(time(0)),
  Node *beg1, *end1, *beg2, *end2  *beg3, *end3,

  beg1 = Create(10),      // создаем первую упорядоченную очередь
  end1= end               // указатель на конец очереди определен
                          // функцией Create в глобальной переменной
                          // end, перепишем его значение в end1
```

```
  Print(beg1);  puts("");
  beg2 = Create(7),        // создаем вторую упорядоченную очередь
  end2 = end,              // указатель на конец второй очереди также
                           // определен в глобальной переменной end,
                           // перепишем его значение в end2
  Print(beg2),  puts("");
  beg3 = twointo1(beg1, beg2, end1, end2),
  end3 = end;
  Print(beg3),
}
```

Второй вариант. Будем передавать в функцию указатели на начало и конец очереди по адресу.

В функции обращение к адресу начала и конца очереди выполняется через операцию разадресации *

В функции Create добавляется два параметра: Node **beg – указатель на адрес начала создаваемой очереди и Node **end – указатель на адрес конца создаваемой очереди.

```
void Create(Node **beg, Node **end, int n)
{ int k, i,
  *beg = 0;
  *beg = Add_e(*beg, k = rand()%10);
  *end = *beg;
  for (i = 1; i < n, i++)
    *end = Add_e(*end, k += rand()%10),
}
```

В функции twointo1 также добавляется два параметра: Node **b – указатель на адрес начала новой очереди и Node **e – указатель на адрес конца новой очереди.

```
void twointo1(Node *b1, Node *b2, Node **b, Node *e1, Node *e2, Node
**e)
{ Node *p1, *p2, *t = new Node, *t = t,
  for (p1 = b1, p2 = b2; p1 && p2, t = t -> next)
    if (p1 -> k <= p2 -> k)
    { t -> next = p1, p1 = p1 -> next, }
    else { t -> next = p2; p2 = p2 -> next, }
  if (p1) { t -> next = p1, *e = e1; }
```

```
  else { t -> next = p2, *e = e2; }
  *b = f -> next,
  delete f,
}

void main()
{ srand(time(0));
  Node *beg1 = 0, *end1 = 0, *beg2 = 0, *end2 = 0, *beg3, *end3,
  Create(&beg1, &end1, 10);        // в функцию передаем адреса
                                   // указателей очереди
  Print(beg1), puts("");
  Create(&beg2, &end2, 7),
  Print(beg2), puts("");
  twointo1(beg1, beg2, &beg3, end1, end2, &end3),
  Print(beg3),
}
```

Третий вариант. Для очереди определим вспомогательную структуру pointers с двумя полями – указателями на начало и конец очереди

В функции Create будем формировать оба указателя структуры pointers и возвращать её

```
struct pointers { Node *beg, *end, },

pointers Create(int n)
{ pointers uk = { 0, 0 };  // определяем переменную uk – структуру
                           // с двумя указателями очереди
  int k, i;
  uk.beg = Add_e(uk.beg  k = rand()%10),  // создаем начальный узел
  uk.end = uk.beg,         // он же пока последний, указатель на
                           // него определен в поле end структуры uk
  for (i = 1, i < n; i++)  // остальные узлы добавляем в конец очереди
    uk.end = Add_e(uk.end, k += rand()%10);

  return uk,               // возвращаем структуру с двумя указателями
}
```

Преобразуем функцию слияния twointo1.

```
// Ф-ия сливает 2 упорядоченные очереди в 3-ю упорядоченную
pointers twointo1(pointers l1, pointers i2)
{ Node *p1, *p2, *f = new Node, *t = f,
  pointers i3 = {0, 0};                  // определяем указатели новой очереди
  // пока числа есть в обеих очередях
  for(p1 = l1.beg, p2 = i2.beg, p1 && p2, t = t -> next)
    // присоединяем к новой очереди узел из первой очереди
    if (p1 -> k <= p2 -> k)  { t -> next = p1, p1 = p1 -> next, }
    else { t -> next = p2, p2 = p2 -> next, }     // или узел из второй очереди
  // если не закончилась первая очередь или
  if (p1) { t -> next = p1, i3.end = i1.end; }
  // не закончилась вторая очередь,
  // присоединяем оставшиеся узлы к новой очереди
  else { t -> next = p2, i3.end = i2.end; }
  // записываем в указатель beg адрес начала новой очереди
  i3.beg = f -> next,
  delete f,                  // освобождаем память от фиктивного узла
  return i3;                 // возвращаем указатели новой очереди
}
```

Приведем пример использования этих функций для формирования и слияния двух упорядоченных очередей.

```
void main( )
{ srand(time(0)),
  pointers u1, u2, u3;
  u1 = Create(10),          // создаем первую упорядоченную очередь
  Print(u1.beg), puts(""); // функцию вывода можно оставить прежней
                            // или определить с аргументом pointers
  u2 = Create(7);           // создаем вторую упорядоченную очередь
  Print(u2.beg),  puts("");
  u3 = twointo1(u1, u2),
  Print(u3.beg),
}
```

4.4. Циклические списки

Хранить 2 указателя списка – на начало и конец, не всегда удобно. В циклическом списке хранится один указатель – на конец

списка, а в поле *next* последнего узла формируется адрес первого узла списка Таким образом, в циклическом списке доступеи как адрес первого элемента, так и последнего Циклический список представлеи иа рисуике 4.

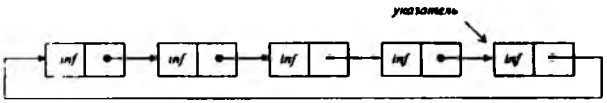


Рисунок 4. Циклический список

Определим фуикции для работы с циклическим списком строк и решим задачу с их использоваиием

```
struct Node { char *name, Node *next, };

// Функция добавляет строку s в конец циклического списка
Node *Add_e(Node *e, char *s)
{ Node *t = new Node,              // создаем новый узел t
  // берем память размера длины строки s + 1 байт для нуль-кода
  t -> name = new char [strlen(s) + 1],
  strcpy(t -> name, s),            // копируем строку s в
                                   // поле name новой структуры
  // если список пуст, то новый элемент – последний и он же первый,
  // в поле next заносим адрес этого же узла, возвращаем его,
  // как новый указатель на конец списка
  if(!e) { t -> next = t, return t, }
  // узлы в списке были – в поле next нового узла заносим адрес первого
  // узла списка из поля next прежнего последнего узла
  t -> next = e -> next,
  // связываем прежний последний узел с новым узлом
  e -> next = t,
  return t,                        // возвращаем в качестве указателя списка
                                   // адрес нового узла
}

// Функция удаляет из списка первый узел
Node *Del_b(Node *e )
{ if(!e) return NULL;              // список пуст – возвращаем NULL
  Node *q,
```

```
  q = e -> next,              // запоминаем адрес первого узла списка
  e -> next = e -> next -> next,    // или e -> next = q -> next – удаляем
                                    // узел с адресом q из списка
  delete [] q -> name,              // освобождаем память от поля
                                    // name удаляемого узла
  delete q,      // освобождаем память от удаленного из списка узла
  // если в списке был один узел – возвращаем NULL
  if(q == e) return NULL;
  return e,      // иначе значение указателя списка не изменилось
}
```

Функция вывода циклического списка отличается от функции вывода не циклического списка, так как в нем нет узла с нулевым значением поля next. При выводе, сначала, до цикла, выводится информация первого узла, а в цикле выполняется переход по узлам списка от второго до предпоследнего узла включительно

```
// Функция выводит циклический список
void Print(Node *e)               // e – указатель на последний узел
{ Node *p  *q,
  if (!e) return,                 // список пуст – выходим из функции
  q = e -> next,                  // q – адрес первого узла
  puts(q -> name),                // вывод данных из первого узла
  // устанавливаем указатель p на узел, следующий за первым,
  // в цикле выводим данные из узла p и переходим к следующему узлу,
  // пока указатель p не окажется на первом узле
  for (p = q -> next, p != q, p = p -> next)
    puts(p -> name),
}
```

Приведенных функций достаточно для решения следующей задачи

Задача о призе. По кругу стоят n человек, каждый m-ый удаляется; последнему оставшемуся вручается приз. Выведите его имя

```
char * Priz(Node *e, int m)   // e – указатель на конец циклического списка
{ Node *p, *q, int k,
```

```
  // становимся на первый узел списка;
  // передвигаемся по узлам списка, пока в нем не оствнется один узел
  // (пока поле next узла не покажет на сам узел)
  for (p = e -> next; p != p -> next, p = p -> next)
  { for (k = 0; k < m - 2; k++, p = p -> next),    // шагаем m - 2 раза, чтобы
                                                   // остановиться перед удаляемым узлом
      q = p -> next;                               // запоминаем адрес удаляемого узла
      p -> next = p -> next -> next;               // удаляем его из списка
      delete [] q -> name,                         // освобождаем память от поля name
      delete q,                                    // освобождаем память от узла
  }
  return p -> name,                                // возвращаем имя из единственного
                                                   // оставшегося узла
}

void main()
{ char *name[5] = { "Юля", "Вера", "Ира", "Саша", "Олег" },
  int i, n = 5, m = 3,
  Node *end = 0,
  for (i = 0, i < n, i++)
    end = Add_e(end, name[i]);
  Print(end);
  printf("\nПризер  %s\n", Priz(end, m) ),
}
```

4.5. Двунаправленные списки

Узел ***двунаправленного списка*** содержит два поля связи одно указывает на предшествующий узел, назовем его ***prev***, второе – на следующий, назовем его по-прежнему ***next***. Двунаправленный список представлен на рисунке 5

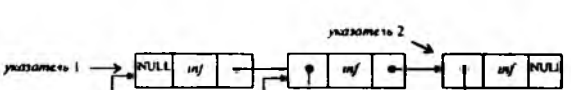


Рисунок 5. Двунаправленный список

При использовании двунаправленного списка обычно удаление и добавление выполняется с обоих концов списка. Поэтому при работе со списком формируются и используются *два указателя – на начало и конец списка*. Для их хранения будем использовать вспомогательную структуру из двух полей, как рассматривалось выше.

Приведем в качестве примера работу с двунаправленным списком, содержащим в информационном поле указатель на строку, например, на фамилию.

```
struct Node { char *fam; Node *prev, *next, };

// будем использовать структуру для указателей
// на начало и конец списка
struct pointers { Node *beg, *end, },

// Функция добаавляет узел в начало списка
pointers Add_b(pointers p, char *f)
{ Node *t = new Node;                    // создаем новый узел
  t -> fam = new char[strlen(f) + 1],    // и
  strcpy(t -> fam, f);                   // задаем значения его полей,
  t -> prev = 0;                         // он будет первым,
                                         // поэтому у него нет предшественника
  // если список был пустым, то у нового узла t нет и последователя,
  // он будет первым и последним в списке
  if (!p.beg)
  { t -> next = 0; p.beg = p.end = t; return p, }
  p.beg -> prev = t,                     // иначе у прежнего первого узла
                                         // появился предшественник,
  t -> next = p beg,                     // а у нового первого узла t
                                         // последователь – бывший первый
  p beg = t,                             // узел t стал первым
  return p,                  // возвращаем новые значения указателей
}

// Функция добавляет узел в конец списка
pointers Add_e(pointers p, char *f)
{ Node *t = new Node,                    // создаем новый узел
  t -> fam = new char[strlen(f) + 1],    // и
```

```
 strcpy(t -> fam, f),                    // задаем значения его полей
 t -> next = 0;                          // узел последний,
                                         // у него нет последователя
 // если список был пустым, то новый узел t
 // будет первым и последним в списке
 if (!p.end)
 { t -> prev = 0, p.beg = p end = t, return p, }
 p.end -> next = t,                      // иначе прежний последний узел
                                         // связываем с новым последним
 t -> prev = p.end,                      // бывший последний узел списка
                                         // становится предшественником
                                         // нового последнего узла
 p end = t,                              // новое значение конца списка
 return p,                  // возвращаем измененные значения указателей
}

// Функция удаляет узел из начала списка
pointers Del_b(pointers p)
{ if (!p.beg) return p,     // если список пуст, выходим из ф-ии
  Node *q = p.beg,          // запоминаем адрес первого узла для
                            // последующего освобождения памяти
  p beg = q -> next,        // началом списка становится второй узел
  // у второго узла (если он есть) нет предшественников
  if(p beg) p.beg -> prev = 0;
  // если нет второго узла, то после удаления список становится пустым
  else p end = 0;
  delete [] q -> fam,                   // освобождаем память от информ-нной
                                        // части удаляемого узла
  delete q;                 // освобождаем память от прежнего первого узла
  return p,                 // возвращаем измененные значения указателей
}

// Функция удаляет узел из конца списка
pointers Del_e(pointers p)
{ if (!p end) return p,                 // список пуст, выходим
  Node *q = p.end;                      // запоминаем адрес удаляемого узла
  p end = p.end -> prev;                // указатель конца списка
                                        // устанавливается на предпоследний
                                        // элемент списка
```

```
 // если в списке было больше одного узла,
 // то удаляем из списка последний узел
 if(p.end) p end -> next = 0;
 // иначе список становится пустым
 else p beg = 0,
 delete [] q -> fam,                    // освобождаем память от информ-нной
                                        // части удаляемого узла
 delete q,              // освобождаем память от прежнего последнего узла
 return p;              // возвращаем измененные значения указателей
}

// Функция удаляет узел со значением информационного поля равным f,
// этот узел может быть в списке любым
pointers Del_f(pointers p, char *f)
{ Node *t, *q
 for (t = p beg, t && strcmp(t -> fam, f), t = t -> next),   // ищем узел
 if (!t) return p,              // если узел не найден то выходим из функции
 // если удаляемый узел первый
 // то используем для его удаления функцию Del_b
 if (t == p.beg) { p = Del_b(p), return p, }
 // если удаляемый узел последний,
 // то используем для его удаления функцию Del_e
 if (t == p end) { p = Del_e(p), return p, }
 // если удаляемый узел не первый и не последний то
 t -> next -> prev = t -> prev,         // следующий узел связываем с
                                        // предшествующим удаляемому, а
 t -> prev -> next = t -> next;         // предшествующий удаляемому узел
                                        // связываем со следующим за ним
 delete [] t -> fam,
 delete t,
 return p,
}

void Pnnt(pointers p)                   // вывод списка
{ Node *q,
 puts("");
 for (q = p.beg  q, q = q -> next)  puts(q -> fam)
}
```

Пример использования двунаправленного списка. В примере показано, что по такому списку можно «ходить» и с начала, и с конца

```
int main()
{ char *fam[5] = { "Иванов", "Петров", "Алексеев", "Сидоров", "Антонов" },
  pointers d = { 0, 0 };
  Node *q,
  int i, m = 5,
  for (i = 0; i < m, i++)              // создаем список
    d = Add_e(d, fam[i]);
  Print(d),                            // выводим его
  d = Add_b(d, "Сергеев");             // добавляем узел с фамилией
  Print(d);                            // "Сергеев" в начало списка
  puts("\nСписок наоборот. ");         // выводим список, передвигаясь с
  for (q = d end, q, q = q -> prev)    // конца списка к его началу
    puts(q -> fam);
  d = Del_b(d),                        // удаляем первый узел списка
  Print(d),
  d = Del_e(d),                        // удаляем последний узел списка
  Print(d),
  d = Del_f(d, "Сидоров");             // удаляем узел с фамилией "Сидоров"
  Print(d),
}
```

Рассмотрим еще два примера работы с двунаправленными списками

Пример 1. Из заданного списка фамилий, упорядоченного по алфавиту, необходимо удалить все фамилии, расположенные между первой и последней фамилиями на букву, заданную в переменной c.

```
// Функция удаляет фамилии между первой и последней фамилиями,
// начинающимися на букву, заданную в переменной c
void Del_fam(pointers u, char c)
{ Node *q, *t, *p;
  // ищем первую фамилию, начинающуюся на букву c
  for (p = u.beg, p && p -> fam[0] != c, p = p -> next),
  if (!p)                              // если не нашли, то выходим из функции
  { printf("Нет фамилий, начинающихся буквой %c", c), getchar(); return, }
```

```
// если нашли, то начинаем поиск последней фамилии на букву c,
// передвигаясь с конца списка до p, используя поле prev
for (q = u.end; q != p && q -> fam[0] != c; q = q -> prev),
if (q == p)       // фамилия на c одна, выводим сообщение, выходим
{ printf("В списке одна фамилия на букву %c", c), getchar(), return, }
t = p -> next;              // в пременной t запоминаем адрес
                            // первого из удаляемых узлов
p -> next = q,              // связываем переую фамилию на букву c
q -> prev = p,              // с последней и последнюю с первой
// освобождаем память от удаленных из списка узлов
for (p = t; t != q; p = t)
( t = t -> next,
  delete [] p -> fam,     // не забываем освобождать память и от поля fam
  delete p                // освобождаем память от узла
 )
}
```

Пример 2. Необходимо упорядочить список методом прямого выбора

Будем формировать новый упорядоченный список следующим образом:

- находим в исходном списке наименьшую фамилию (в алфавитном порядке);
- добавляем её в конец нового списка,
- из исходного списка соответствующий узел исключаем

Определим функцию, которая формирует новый список и возвращает его указатели.

```
pointers Sort_fam(pointers p)       // p – указатели исходного списка
( Node *q, *t;                      // q – указатель на текущий узел
                                    // t – указатель на минимум,
 pointers newl = {0, 0};
 while(p.beg)             // пока исходный список на опустеет
 ( q = p.beg, t = q,      // запоминаем в q и t адрес первого узла
   for (q = q -> next, q, q = q -> next)        // цикл поиска минимума
     if (strcmp(q -> fam, t -> fam) < 0)
       t = q,             // в узле q фамилия меньше, запомнили его в t
   // добавляем в конец нового списка узел с минимальной фамилией
   newl = Add_e(newl, t -> fam),
```

```
    // если "минимальный" узел первый, удаляем
    // его из списка функцией Del_b
    if (t == p.beg) { p = Del_b(p), continue, }
    // если последний, то функцией Del_e
    if (t == p.end) { p = Del_e(p), continue, }
    // "минимальный" узел не первый и не последний – меняем связи
    t -> prev -> next = t -> next,      // узел, предшествующий удаляемому,
                                        // соединяем со следующим за ним,
    t -> next -> prev = t -> prev,      //  а следующий за ним – с
                                        // предшествующим ему
    delete [] t -> fam;       //  освобождаем память, занятую полем fam
    delete t,         // освобождаем память от "минимального" узла
  }
  return newl,      // возвращаем указатели нового упорядоченного списка
}
```

Пример использования функций.

```
void main( )
{ char *fam[5] = { "Иванов", "Петров", "Алексеев", "Сидоров", "Антонов" },
  pointers d = {0, 0}, Node *q,
  int i, m = 5,
  for (i = 0; i < m; i++)                 // формируем новый список
    d = Add_e(d, fam[i]),
  Print(d);                               // выводим его
  d = Sort_fam(d),
  puts("\nУпорядоченный по алфавиту список  ");
  Print(d),
  d = Add_e(d, "Алин"),
  Del_fam(d, 'A');
  Print(d),
}
```

5. Работа с файлами

Рассмотрим возможности работы с файлами, предоставляемые стандартной библиотекой функций с заголовочным файлом stdio.h.

Для работы с файлом определяется файловая переменная - указатель на структуру типа FILE

*FILE *имя_файловой_переменной*

Например,
FILE *f

Для того чтобы начать работу с файлом, необходимо его открыть. Открытие файла выполняется функцией ***fopen***, которая имеет следующий формат

fopen(имя_файла, режим_открытия_файла)

Эта функция возвращает указатель на структуру типа FILE. ***Имя_файла*** задается строковой переменной или строковой константой. Если имя файла задано без указания пути, то файл открывается в текущей директории

Рассмотрим некоторые режимы открытия файла

"w"	– открывается новый файл для записи, если файл с таким именем был, то его содержимое теряется,
"r"	– открывается существующий файл для чтения,
"a"	– файл открывается для добавления информации в конец файла Если такого файла нет, то он создается;
"w+"	– создается новый файл для чтения и записи,
"r+"	– открывается существующий файл для чтения и записи,
"a+"	– открывается файл для чтения и добавления информации в конец файла

Файлы можно открывать в текстовом режиме (по умолчанию) В текстовом режиме, при чтении из файла выполняются некоторые преобразования из символьного во внутренний формат, при записи в файл – из внутреннего в текстовый. Файл можно также открыть в двоичном режиме, в котором такие преобразования не выполняются Текстовый режим задается буквой **t**, двоичный – **b**

При открытии файла устанавливается значение *указателя текущей позиции* чтения или записи в зависимости от режима открытия файла. Если файл открыть не удалось, например, файл ввода не найден или недостаточно места на диске, функция *fopen* возвращает значение *NULL*. При открытии файла желательно всегда контролировать правильность открытия.

Например,

```
FILE *f;
if ((f = fopen("a.txt", "r")) == NULL)
{ puts("Ошибка при открытии файла. Выход"), return; }
```

Указатель *f* используется далее в операциях с файлами.

Для чтения из файла можно использовать функцию *fscanf* форматированного ввода. Функция имеет следующий формат:

***fscanf*(*имя_файловой_переменной*, "*формат*", *список_адресов_вводимых_данных*)**

Например,

```
fscanf(f, "%d%d", &x, &y),          // чтение из файла 2-х чисел
```

Формат и *список_адресов_вводимых_данных* определяется также как и в функции *scanf*.

Для записи в файл используется функция форматированного вывода *fprintf* формата:

***fprintf*(*имя_файловой_переменной*, "*формат*", *список_вывода*)**

Например,

```
fprintf(f, "%d %d", x, y),   // также как и при выводе на экран
                             // между выводимыми числами надо
                             // поставить пробел, иначе они сольются
```

Формат и *список_вывода* определяется также как и в функции *printf*.

При чтении или записи *указатель текущей позиции* передвигается по файлу Для проверки достигнут или нет конец файла, может использоваться функция-предикат·

feof(имя_файловой_переменной),

которая возвращает *истину*, если файл закончился и *ложь* в противном случае

После работы с файлом его следует закрыть функцией

fclose(имя_файловой_переменной)

После закрытия файла работать с ним нельзя до нового открытия

Можно закрыть все файлы одновременно функцией

fcloseall( )

Для чтения строковых данных может использоваться функция

fgets(имя_строки, число_символов, имя_файловой_переменной)

В строковую переменную читается указанное *число_символов* или символы до символа '\n'.

Например,

```
char s[30];

fgets(s, 20, f);
```

Замечание

При чтении строка читается вместе с '\n' и затем добавляется нуль-код. При использовании функции *fscanf* строка читается до первого пробела или символа '\n'

Для вывода строки может использоваться функция

fputs(имя_строки, имя_файловой_переменной)

Например,

```
fputs("пришла весна-красна", f),
```

Рассмотрим два примера

Пример 1. Необходимо создать файл со случайными целыми числами, затем выбрать из него во второй файл все четные числа и вывести его содержимое на экран

Заметим, что если пробел стоит после последнего числа в файле и цикл организуется с использованием функции *feof*, то он будет работать лишний раз. Эту проблему можно обойти следующим образом будем читать один раз до цикла, а затем в конце цикла

Зададим 2 функции: создание файла чисел и функцию вывода

```
#include <stdio h>
#include <stdlib h>
#include <time.h>

// namef - имя файла, n - количество чисел  d – диапозон
void createF(char *namef, int n, int d)
{ int i, k,
  FILE *f;
  f = fopen(namef, "w");
  if (f == NULL)
  { puts("Ошибка при открытии файла\n"), exit(1); }
  for (i = 0, i < n, i++)
  { k = rand()%d, fprintf(f, "%d ", k)  }
  fclose(f),
}

void OutF(char *name)
{ int k,
  FILE *f;
  f = fopen(name, "r"),
  if (f == NULL)
  { puts("Ошибка при открытии файла\n"), exit(1), }
  fscanf(f, "%d", &k),
  while (!feof(f))
  { printf("%d ",k); fscanf(f,"%d",&k), )
  fclose(f),
}
```

```
void main()
{ FILE *f, *f2,
  int a;
  char *name = "ch.txt", *name2 = "ch2.txt";
  srand(time(0));
  createF(name,15,50);
  OutF(name);
  puts("\n----------");
  f = fopen(name, "r");
  f2 = fopen(name2, "w");
  if (f == NULL || f2 == NULL)
  { puts("Ошибка при открытии файла\n"), exit(1), }
  fscanf(f, "%d", &a),
  while (!feof(f))
  { if (a && a%2 == 0) fprintf(f2, "%d ", a),
    fscanf(f,"%d",&a),
  }
  fcloseall(),
  puts("\nЧетные числа "); OutF(name2),
  puts("\n");
}
```

Пример 2. Пусть в файле хранится информация о результатах вступительных экзаменов абитурнентов в виде фамилия, номер паспорта, оценки по трем экзаменам Данные о каждом абитуриенте находятся в отдельной строке. Необходнмо в новом файле сформировать список поступивших абитуриентов, т е. таких абитурнентов, суммарный бал которых выше заданного Одна строка этого файла должна содержать следующие данные одного абнтурнеита. порядковый номер, фамилия абнтурнента и его суммарный бал

```
#include <stdlib.h>
#include <stdio h>

void main()
{  FILE *f_in, *f_out;
   char name_in[30], name_out[30], s_in[100], fam[30];
   int pas, pb, b1, b2, b3, b, i = 0,
```

```
  puts("Введите имя входного файла. ");
  gets(name_in);
  puts("Введите имя выходного файла  ");
  gets(name_out);
  f_in = fopen(name_in, "r");
  f_out = fopen(name_out, "w"),
  if (f_in == NULL || f_out == NULL)
  { puts("Ошибка при открытии файла"), exit(1), };
  puts("Введите проходной бал  "),
  scanf("%d", &pb);
  fputs("Список поступивших абитуриентов\n\n", f_out),
  while (!feof(f_in))
  {
    fgets(s_in, 100, f_in),
    sscanf(s_in,"%s%d%d%d%d",&fam, &pas, &b1, &b2, &b3),
    b = b1 + b2 + b3,
    if (b >= pb)
      fprintf(f_out, "%-3d %-30s %d\n", ++i, fam, b),
  }
  fcloseall(),
}
```

Литература

1 В В Подбельский Язык Си++ – М : Финансы и статистика, 2000 560 с
2 Т А. Павловская С/С++ Программирование на языке высокого уровня – СПб · Лидер, 2010 – 461 с
3 М Уэйт, С Прата, Д Мартин Язык Си – М Мир, 1988 – 512 с

Содержание

1. Поразрядные логические операцин 3
2. Локальные н глобальные переменные 10
3. Структуры н массивы структур 12
4. Линейные списки.. 19
4.1. Понятие списка Операцни добавления н удаления элементов списка..... 19
4.2 Стеки. 24
4.3. Очереди.. 26
4.4. Циклические списки.. 33
4.5. Двунаправленные списки. 36
5. Работа с файламн..... 43
Литература. 49

Отпечатано на участке оперативной полиграфии
редакционно-из ательского отдела ТГУ

Заказ № 103 от « 18 » 04 2011 г Тираж 150 экз