МИНИСТЕРСТВО ОБРАЗОВАНИЯ и НАУКИ РЕСПУБЛИКИ КАЗАХСТАН

КАЗАХСКИЙ НАЦИОНАЛЬНЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ имени К. И. САТПАЕВА

Б. С. Кубеков

УКАЗАТЕЛЬНЫЙ ТИП В ЯЗЫКАХ ПРОГРАММИРОВАНИЯ С / С ++

Рекомендовано Республиканским учебно-методическим объединением в качестве учебного пособия

Алматы 2008

УДК 681.3.06
ББК 32,973.26-018.1 я73

К 88 **Кубеков Б.С.**
Указательный тип в языках программирования С/С++: Учеб. пособие. – Алматы : КазНТУ, 2008. – 103 с.

ISBN 978-601-228-023-4

В учебном пособии приводится систематическое изложение вопросов, отражающих определение и применение объектов указательного типа.

В первой части пособия рассматриваются: природа статических и динамических объектов и их применение в практике программирования; понятие указателей и операции с ними; связь массивов с указателями; способ использования указателей в качестве параметров функций; понятие указатели на функции и их разновидности. В конце первого раздела приводятся полезные практические примеры программирования с указателями, выдержанные в духе технологии обобщенного программирования. Во второй части пособия рассматривается указательный тип и его применение для обработки динамических объектов сложной структуры, таких как односвязные и двунаправленные списки, очереди, стеки и, конечно же, двоичные деревья.

Учебное пособие предназначено для бакалавров техники и технологии, обучающихся по специальностям 050704 – " Вычислительная техника и программное обеспечение " и 050602 – " Информатика ", а также может быть полезно и для заинтересованных преподавателей высших учебных заведений.

Библиогр. – 7 назв.

ББК 32,973.26-018.1 я73

Рецензенты: *А.У.Тукеев*, зав.каф."Информационные системы " КазНУ им. Аль-Фараби, д-р.техн.наук, проф.
И.Т.Утепбергенов, зав.каф."Информационные системы" КазАТК им. М.Тынышпаева, д-.техн.наук, проф.
Б.И.Жумагалиев, проф. каф. "Техническая кибернетика" КазНТУ им.К.И.Сатпаева, канд.техн.наук.

Печатается по плану издания Министерства образования и науки Республики Казахстан на 2008 г.

ISBN 978-601-228-023-4

© Кубеков Б.С., 2008
© КазНТУ, 2008

ВВЕДЕНИЕ

Написанию учебного пособия «Указательный тип в языках программирования С/С++» способствовал ряд причин, среди которых следует отметить следующие:

- практика преподавания автором различных дисциплин, связанных с языками и технологиями программирования на начальных курсах обучения, показала достаточно трудное усвоение студентами навыков работы с динамическими объектами;

- анализ целого ряда популярных изданий научного и учебно – методического характера по языкам и технологиям программирования на языках С/С++ показывает, что тема «Указательный тип» представляется в них достаточно обще, без необходимой студентам систематизации и соответствующего методического сопровождения;

- в отличие от языка Паскаль ни язык С, ни язык С++ не были ориентированы на учебные цели, а создавались их авторами для программистов – профессионалов, поэтому преподавание (а следовательно, и изучение) языка С и особенно языка С++ не такое уж простое дело.

В цикле языковой и технологической подготовки студентов специальности 050704 – «Вычислительная техника и программное обеспечение» обязательным компонентом первого года обучения входит дисциплина « Технология программирования», в качестве пререквизита таких курсов, как «Объектно-ориентированное программирование», «Сетевые технологии программирования», «Основы компонентных технологий», «Технология разработки программного обеспечения». Тема «указательный тип» является базовой в дисциплине «Технология программирования», так как объекты указательного типа позволяют строить и оптимально обрабатывать все известные структуры данных в динамическом исполнении, поэтому большие возможности, которые несет указательный тип и его объекты, делают освоение темы «указательный тип» критичным.

В предлагаемом учебном пособии автор преследовал цель в первую очередь помочь студентам в систематизации всего того, что составляет базу для понимания природы указательного типа, и в показе техники программирования различных типовых структур данных с использованием объектов указательного типа. Успешное усвоение языков С/С++, являющихся базовыми языками академических курсов «Технология программирования» и « Объектно – ориентированное

программирование», читаемых автором для студентов специальности 050704, соответственно на первом и третьем курсах, невозможно без указательного типа, как и невозможно дальнейшее освоение технологии обобщенного программирования. При этом справедливым следует считать тезис о том, что задача учебного пособия заключается не только в том, чтобы помочь студентам, а также заинтересованным преподавателям более детально разобраться в деталях указательного типа и методике преподавания, но, как следствие, улучшить и сделать более интересным сам процесс обучения программированию.

Сопровождающий текст *примеры* в основном довольно просты и непохожи на реальные задачи программирования. Ценность таких демонстрационных примеров заключается в том, что, изучая их, студенты смогут понять основные иллюстрируемые ими концепции и механизмы указательного типа, не оказываясь под давлением объемных законченных исходных текстов.

Автор пособия является разработчиком Типовой программы по дисциплине «Технология программирования», а также соответствующего учебно – методического комплекса дисциплины студента, выполненных в соответствии с требованиями Государственного общеобязательного стандарта образования РК.

Учебное пособие «Указательный тип в языках программирования C/C++» состоит из введения и двух глав. В первой главе представляется понятие «указатель», «концепции» и «механизмы», лежащие в основе конструирования динамических объектов различных стандартных типов с помощью объектов указательного типа. Рассматриваются многочисленные примеры описания функций по обработке различных типовых структур данных, в том числе функций, реализованных в духе технологии обобщенного программирования. Вторая глава посвящена анализу механизмов применения объектов указательного типа при конструировании и обработке таких сложных динамических структур данных как списки, стеки, очереди, таблицы и двоичные деревья.

Учебное пособие может быть полезным в первую очередь студентам специальности 050704 – Вычислительная техника и программное обеспечение, а также студентам смежных специальностей и всем желающим при самостоятельном освоении основ программирования на языках C/C++.

1. УКАЗАТЕЛЬНЫЙ ТИП И ОСНОВЫ ПРОГРАММИРОВАНИЯ

1.1. Статические и динамические программные объекты

Термином *статические* программные объекты обозначаются такие объекты, которые порождаются непосредственно перед выполнением программы, существуют в течение всего времени её выполнения и размер значений которых (в смысле объема машинной памяти, необходимой для их размещения) не изменяется по ходу выполнения программы. Статические объекты порождаются с помощью соответствующих им описаний. Например, следующее описание переменной **x**

```
float x[10];
```

свидетельствует о том, что в программе вводится в употребление (порождается) статическая переменная регулярного типа (массив), значением которой может быть упорядоченная (в смысле размещения в памяти) последовательность из десяти вещественных чисел. В программе ссылка на программные объекты осуществляется путем указания их имен. На машинном же языке ссылка на объект осуществляется путем указания его места в памяти компьютера, т. е. место обычно задаётся адресом объекта – номером ячейки байта памяти, начиная с которой размещен этот объект. Таким образом, вся работа по размещению таких объектов в памяти компьютера и формирование ссылок на эти объекты в программе выполняется на этапе трансляции (раннее связывание), что и является положительной стороной использования статических объектов.

Однако использование при программировании только статических объектов может вызвать определенные трудности, особенно с точки зрения получения эффективной программы, что бывает чрезвычайно важной при решении не только ряда практических задач, но и при применении самых последних технологий программирования, развитых и введенных в широкую практику в конце XX века, и остающимися предельно актуальными в начале XXI-го с большими перспективами на будущее. Дело в том, что иногда мы заранее (т. е. на этапе составления программы) не знаем не только размер значения того или иного программного объекта, но даже и того, будет существовать этот объект или нет. Такого рода программные объекты, которые возникают и обрабатываются в процессе выполнения программы или размер значений которых определяется (или изменяется) при выполнении программы, называют *динамическими объектами.*

Если, например, в заданном тексте, состоящем из слов произвольной длины, требуется найти первое по порядку слово, обладающее заданным свойством, то переменная, значением которой является искомое слово, в процессе выполнения программы может вообще не появиться (нет слова с заданным свойством), а если оно появится, то заранее неизвестна длина этого слова, являющегося значением этой переменной. Такая переменная по своей сути является динамической, в противном случае для результирующей переменной пришлось бы зарезервировать максимально возможную длину слова, что приводит к нерациональному использованию памяти компьютера.

Кроме того, часто бывает так, что какой-либо программный объект, занимая значительный объем памяти, нужен не на всё время выполнения программы, поэтому для устранения указанных выше недостатков, связанных с использованием статических объектов, в современных языках программирования предусмотрена возможность использования динамических объектов, что влечёт за собой и определённые трудности. Одна из таких трудностей заключается в том, что возникают объекты динамически в процессе выполнения программы, а действие над ними необходимо задать уже при написании программы до её выполнения. Возникает вопрос: как же в программе сослаться на ещё не порождённые, то есть пока несуществующие объекты? Для преодоления этой трудности при работе с динамическими объектами предусмотрен специальный тип значений – так называемый *указательный* тип (тип данных «указатель»). Значениями переменных типа «указатель», которые для краткости будем называть указателями, служат адреса объектов конкретного типа. Другими словами, значением типа «указатель» является ссылка на какой-либо программный объект, и по этой ссылке осуществляется непосредственный доступ к нужному объекту.

В этом случае для описания действий над динамическими объектами каждому такому объекту в программе сопоставляется статическая переменная типа «указатель» - в терминах этих переменных и описываются действия над соответствующими динамическими объектами. Значения же переменных типа «указатель» как обычно определяются уже в процессе выполнения программы, а для достаточно удобного формирования таких значений предусмотрены соответствующие средства, которые мы и будем рассматривать.

1.2. Понятие указателей

Указатели – это одна из наиболее трудных для освоения возможностей языков C/C++. Наличие указателей делает эти языки очень удобными для системного программирования.

С помощью указателей, например, можно:

- обрабатывать символы, массивы, строки, структуры, массивы структур;
- динамически создавать новые переменные в процессе выполнения программы;
- передавать функциям адреса фактических параметров;
- передавать функциям адреса функций в качестве параметров.

Итак, ***указатель – это переменная или константа, содержащая адрес другой переменной.***

Значения указателей – это беззнаковые целые числа и их можно обрабатывать как целочисленные величины. Они сообщают, где размещены переменные, и не говорят ничего о самих переменных.

Концепция указателей состоит в том, что вы работаете с адресом ячейки памяти, получая лишь *косвенный доступ* к её содержимому, благодаря чему появляется возможность динамически создавать и уничтожать переменные.

Упрощенно, организация памяти компьютера представляет собой массив последовательно пронумерованных или проадресованных ячеек, с которыми можно работать по отдельности или связными кусками. Сам указатель – это группа ячеек, в которых может храниться адрес какого-либо объекта.

Чтобы получить адрес объекта *в явном виде* применяют унарную операцию ‘&’ (знак & читается *амперсанд*). Выражение & *имя – переменной* позволяет получить адрес участка, выделенного на машинном уровне для этой *переменной*. Операция ‘&’ применима только к объектам, имеющим имя и размещённым в памяти. Её нельзя применять к выражениям, константам, битовым полям структур, регистровым переменным или внешним объектам (файлам), с которыми может взаимодействовать программа.

Имея возможность с помощью операции ‘&’ определять адрес переменной или другого объекта программы, нужно уметь его *сохранять, преобразовывать* и *передавать*. Для этих целей и введены переменные типа «указатель», которые для краткости, как мы договорились, будем называть просто указателями.

Прежде всего, рассмотрим вопрос о том, как задаётся указательный тип. В этом задании обязательно должен фигурировать тип динамических программных объектов, на которые будут указывать (ссылаться) переменные указательного типа.

Синтаксически задание переменной указательного типа определяется следующим образом:

спецификация_типа *указатель;

где

спецификация_ типа – тип объекта, на который будет ссылаться указатель;

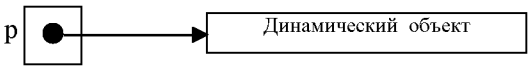
указатель – идентификатор переменной типа «указатель»; ‘*’ – разделитель, который является знаком унарной операции косвенной адресации, иначе называемой *операцией разыменования* или *операцией раскрытия ссылки* или *операцией обращения по адресу*.

Примеры определения указателей:

```
char *pc;          // указатель на литеру
int *pk, *pi;      // pk, pi - указатели на
                   // объекты целого типа
float *pf;         // pf - указатель на объект
                   // вещественного типа
```

У всех этих *статических переменных* типа «указатель» есть одна общая черта – их значения ссылаются на место в памяти, где размещаются *динамические объекты.*

Связь указателя с динамическим объектом в графической нотации можно изобразить следующим образом:



На этом рисунке точкой изображено *значение* указателя **p** (от английского слова **pointer** - указатель), а стрелка, исходящая из этой точки, отражает тот факт, что значением указателя является *ссылка на динамический объект,* посредством которой он и доступен в программе.

В некоторых случаях значением указателя может быть заведомо неравное никакому адресу значение, принимаемое за нулевой адрес. Для его обозначения в ряде заголовочных файлов, например в файле **stdio.h**, определена специальная константа **NULL**, которая принадлежит любому указательному типу.

Обратите внимание на то, что указатель может ссылаться на объекты только того типа, *который присутствует в определении указателя.* Исключением являются указатели, в определении которых использован тип **void** – отсутствие значения. Такие указатели могут ссылаться на объекты *любого типа,* однако к ним нельзя применять операцию разыменования, то есть операцию ‘*’.

Отметим ещё раз, что в определениях и описаниях указателей применяется разделитель ‘*’, который является в этом случае знаком унарной операции косвенной адресации, иначе называемой операцией

разыменования или операцией раскрытия содержимого того динамического объекта, на который существует ссылка (адрес) указателя. Операндом операции '*' всегда является указатель! Результат операции '*' – тот объект, который адресует указатель. Таким образом, выражение *pc обозначает объект типа **char** (символьная переменная), на который ссылается указатель pc. Другими словами, результат операции *pc – содержимое динамического символьного объекта, на который ссылается указатель pc, то есть какой-либо символ. Оператор *pc ='0'; заносит нулевое значение в участок памяти, адрес которого задан указателем pc.

Для иллюстрации особенности указателей приведем пример

```
*pk = *pk + *pi;
```

Здесь происходит суммирование целых значений участков памяти (содержимого динамических объектов) и присваивание суммы участку памяти, адрес которого содержит указатель pk.

Возможности и преимущества указателей *наиболее эффективно* проявляются при работе с функциями, массивами, строками, структурами и т. д., не говоря уже об объектно-ориентированном программировании и других парадигмах программирования, например, обобщенном программировании.

1.2.1. Объявление и использование указателей

В объявлении **int** *pi; создается указатель, который в дальнейшем будет адресовать переменную типа **int.** Указатели могут хранить адреса только конкретного типа. Если делается попытка присвоить указателю одного типа адрес переменной другого типа, возникает ошибка.

Пример:

```
int *pi;
float x = 1.5;
pi = &x;
```

Здесь делается попытка присвоить указателю адрес переменной, которая имеет тип **float** – ошибка!

В объявлении указателей звёздочка обычно относится непосредственно к имени указателя, поэтому при объявлении указателей требуется ставить этот знак непосредственно ***перед*** **именем** каждого из них. Например:

```
int *pa, *pb, *pc;
```

В приведенном примере все указатели помещаются в стековую память.

Рассмотрим следующий фрагмент программы:

```
            int ia = 15, ib = 37, ic;
                  int * pi;
    pi = &ia; ic = *pi; ia = ib;  ib = ic;
```

в котором производится обмен значений между переменными ia и ib.

Теперь определим, что будет выведено на экран в результате выполнения следующего фрагмента:

```
            char c1 = 's' , c2 = 't';
            char *pc1, *pc2, *pca;
    pc1 = &c1; pc2 = &c2;pca = pc1;
    pc1 = pc2; pc2 = pca;
    printf("%c%c\n", *pc1, *pc2);
```

Ответ: ts. Здесь осуществляется обмен адресами между указателями.

Замечания:

- в ходе выполнения операций исходные значения переменных с 1 и с 2 не изменялись;
- описанный способ может пригодиться в дальнейшем, так как в зависимости от размеров объектов часто бывает выгоднее копировать их адреса, чем перемещать содержимое.

Указатели можно инициализировать при их объявлении как любые другие переменные

```
                int ia;
              int *pi = &ia;
```

Здесь инициализирован указатель, то есть указателю присвоен адрес области памяти, в которой будет храниться целое число, и эта область памяти имеет идентификатор ia.

Рассмотрим программу с примером инициализации указателя

```
#include<stdio.h>
#include<string.h>
int main()
    {
          char *ps = "Dog Rex";
 //в указателе ps будет храниться адрес первого
 // символа строки
           int i;
for (i = strlen(ps)-1; i>=0; i--)
       printf("%c", ps[i]);
       printf("\n %s \n", ps);
         return 0;
      }
```

Результаты выполнения программы

```
xeR goD
Dog Rex
```

Замечание

Функция **strlen()** вычисляет «чистую» длину строки, которая в нашем случае, равна семи. В рассматриваемых языках программирования, индексы элементов массива всегда нумеруются с нуля.

1.2.2. Ограничения на использование операции ‘&’

Рассмотрим случай неправильного использования операции ‘&’. Напомним, что операция & (амперсанд) возвращает адрес программного объекта, и этот адрес становится значением указателя.

Определение

```
int *pi = &48;
```

приведет к ошибке, так как нельзя адресовать константу, поскольку она не связана ни с одной из ячеек памяти. Если

```
int k=5;
pi = &(k+15);
```

также ошибка, так как происходит попытка получения адреса числа. И наконец, если

```
register int i;
pi = &i;
```

также будет фиксирована ошибка, так как регистровая переменная располагается не в памяти, а непосредственно в регистрах процессора.

Таким образом, **указатель – это переменная, либо константа для хранения адреса *переменной* определенного типа.** Тип адресуемой переменной может быть стандартный, перечисляемый, структурный, объединение и **void.**

Указатель на тип **void** может адресовать значение любого типа. Размер памяти для самого указателя и формат хранимого адреса зависит от компьютера и выбранной модели памяти. Константа **NULL** из стандартного файла **stdio.h** предназначена для инициализации указателей нулевым (незанятым) значением адреса.

Указатель содержит адрес нулевого (самого первого) байта* области *памяти, отводимой для переменной. Тип адресуемой программной переменной определяет, сколько байт отводится для значений этой переменной. Например, при следующих объявлениях:

int k;
int *pi = &k;

для переменной k отводится участок в 4 байта памяти, начиная, допустим, с адреса 56534, тогда значением указателя pi будет **адрес первого байта** этого участка, то есть 56534.

В общем случае различают **три вида указателей**: указатель на программный объект, указатель на функцию и указатель на **void.**

Указатель на функцию содержит адрес, по которому передается управление при вызове функции. Указатель на функцию используется для косвенного вызова функции (не через её имя, а через обращение к указателю -переменной, хранящей её адрес – адрес первого байта тела функции), а так же для передачи имени функции в другую функцию, в качестве её параметра.

Например, декларация

int (*pf) (**double, double**);

обозначает указатель с именем pf на функцию, которая возвращает **int** и имеет два параметра типа **doubl**e.

Как мы уже знаем, простейшее объявление указателя на объект имеет вид

имя-типа *идентификатор указателя;

где тип может быть любым, кроме **ссылки и битового поля**.

Есть возможность определять указатель на указатель, указатель на указатель на указатель и т. д. Рассмотрим пример объявления указателей для многоуровневой косвенной адресации:

```
int  k = 123;  //  k - имя переменной
int *pi=&k;    //  pi - указатель на переменную
int **ppi=&pi;  // ppi- указатель на указатель на
                // переменную
int ***pppi=&ppi; //pppi-указатель на указатель на
                  // указатель на переменную
```

Указатель на **void** удобно применять в тех случаях, когда конкретный тип объекта, адрес которого требуется хранить, не определён (например, если в одной и той же переменной в разные моменты времени требуется хранить адреса объектов различных типов). В C++ такой механизм реализует полиморфизм.

Указателю на **void** можно присвоить значение указателя любого типа, а также сравнивать его с любыми указателями, но перед выполнением таких действий с областью памяти, на которую он ссылается, требуется преобразовать его в конкретный тип явным образом.

Еще раз повторим , что тип указателя **void*** строго предназначен лишь для хранения адресов (и операций сравнения). Если же нужно обработать участок памяти, адресуемый указателем типа **void***, то его нужно сначала явно привести к соответствующему типу указателя.

Указатель может быть константой или переменной, а также ссылаться на константу или переменную.

Рассмотрим некоторые примеры объявлений:

```
int k;              // целая переменная
const int ci=1;     // целая константа
int *pi;       //  указатель на целую переменную
const int *pci; // указатель на целую константу
int *const cp=&k; // указатель-константа на целую
                  // переменную
const int *const cpc=&ci; //указатель-константа
на целую
                    //  константу
```

Модификатор **const,** находящийся между именем указателя и звездочкой, относится к самому указателю и запрещает его изменение, а **const** слева от звездочки задает постоянство значения, на которое он ссылается. Для инициализации указателя использована операция *взятия адреса* - *&*.

Рассмотрим два объявления указателей и найдем различия между ними:

const type *p1; // указатель на константу типа type

type ***const** p2=&k; // константный указатель на переменную k типа type

Указателю p1, как и любой переменной, может быть присвоен адрес любой ячейки памяти, содержащей данные типа type, впоследствии содержимое этой ячейки не может быть изменено путем обращения к указателю.

*p1=(type)float_value; // недопустимая попытка изменить значение
// защищенной ячейки памяти

Переменная p2 – объявлена как константный указатель. Она может содержать адрес ячейки памяти с данными типа type, однако этот адрес не доступен для изменения. По этой причине инициализация константного указателя обязательно должна происходить одновременно с его объявлением. Содержимое же ячейки, на которую ссылается такой указатель, может быть свободно изменено.

p2=&k; // недопустимая попытка изменить защищенный адрес

*p2=(type) float_value; // допустимое изменение содержимого ячейки памяти

Следует иметь в виду, что *имя массива* представляет собой ***неизменяемый адрес*** его первого элемента, то есть **имя массива – это указатель - константа.**

Приведем еще несколько примеров объявления указателей:

```
int *pi; //указатель-переменная на данные типа int
char *pc; //указатель-переменная на данные типа char
float*pf; //указатель-переменная на данные типа float
int arr[5];  // arr- имя массива на пять значений
             // типа int, имя массива – указатель –
             //  константа
int *arr[10]; // arr-массив из десяти указателей
              //  на значения типа int;
int (*par)[10];//указатель на массив из
               // десяти элементов типа  int;
float (*pf)()  // pf – имя указателя-переменной на
               //функцию без параметров, возвращающую
               //значение типа float.pf- это
               // указатель на машинный код функции
float *f();   // f- имя указателя-константы на
              //функцию без параметров,
              // возвращающую указатель на значение
              // типа float.
```

1.2.3. Подробнее об инициализации указателей

Дополним сказанное ещё несколькими подробностями об инициализации указателей.

Указатель чаще всего используется при работе с динамической памятью компьютера, называемой **кучей** (от английского **heap**). Это свободная память, в которой можно во время выполнения программы выделять место в соответствии с потребностями. Доступ к выделенным участкам динамической памяти, называемыми динамическими переменными, производится **только через указатели**. Время жизни динамических переменных определяется от момента их создания до конца работы программы или до явного освобождении памяти через обнуление указателя.

Существуют два способа работы с динамической памятью:

- Использование семейства функций **malloc(), free**() – язык Си;
- Применение операторов **new** и **delete** – C++.

При определении указателя надо стремиться выполнять его инициализацию, то есть присваивать начальное значение, в противном случае возникает очень большое количество ошибок. Инициализирующее выражение записывается либо после имени указателя, либо после знака равенства, либо в круглых скобках.

Существует несколько способов **инициализации указателей.**

1. Присвоение указателю адреса существующего объекта:

а) с помощью операции получения адреса

```
int k = 11;        // объявление и инициализация переменной;
int *p = &k;    // в объявленный указатель p записывается адрес k;
```

б) с помощью другого инициализированного указателя

```
int  *q = p;
```

в) с помощью имени массива или функции, которые сами трактуются как указатели - константы

```
int arr[10];  // описание массива; arr - константный указатель,
              // значением, которого является адрес первого элемента
              // массива
int *p= arr;  // объявление указателя и присвоение ему адреса начала
              // массива, то есть адреса элемента с индексом 0
void f(int k) {...}          // определение функции
void (*pf) (int);            // объявление указателя на функцию
     pf = f;                 // присвоение указателю адреса функции.
```

2. Присвоение пустого значения

```
int  *p = NULL;
int  *q = 0;                 // рекомендуемый способ
```

3. Присвоение указателю адреса области памяти в явном виде

```
char  * pc = (char*) 0x0065FDFO;
```

Здесь 0x0065FDFO – шестнадцатиричная константа, а (**char***) – явное преобразование типа в "указатель на char".

4. Выделение участка динамической памяти и присвоение адреса выделенного участка указателю:

а) с помощью функции ***malloc*** (описана в заголовочном файле **malloc.h**)

```
int  *p1 =  (int *) malloc (sizeof (int));
```

где **malloc()** – функция выделения памяти, параметр которой задает количество выделяемой памяти в байтах, а конструкция (**int***) используется для приведения типа указателя – **void*,** возвращаемого функцией, к требуемому типу. Так как функция ***malloc()*** возвращает указатель типа **void ***, а нам нужен указатель типа **int***, то приходится явно приводить тип возвращаемого указателя к нужному нам типу. Если память выделить не удалось, то функция возвращает нуль

б) с помощью оператора **new** (С++);

int *p_2 = **new int;**

Выполняется выделение достаточного для размещения значений типа **int** участка памяти и адрес начала этого участка записывается в переменную p_2. Память же под сам указатель p_2 выделяется в стеке на этапе компиляции

int *p_3 = **new int** (20);

Здесь кроме описанных выше действий, производится ещё инициализация выделенной динамической памяти значением 20:

int * p_4 = **new int** [20];

Здесь происходит выделение памяти под 20 величин типа **int** (массива из 20 элементов) и записывается адрес начала (адрес элемента массива с индексом, равным 0) в указатель p_4 , который теперь будет трактоваться как имя массива. Через это имя можно обращаться к любому элементу массива с помощью операции индексации, например p_4 [18].

Освобождение памяти, выделенной функцией ***malloc()*, должно выполняться с помощью функции *free*** (идентификатор указателя), а память, выделенная с помощью операции ***new***, - посредством оператора ***delete***. При этом *переменная – указатель* сохраняется (так как она статическая и хранится в стеке) и может инициализироваться повторно.

Приведенные выше динамические переменные уничтожаются следующим образом:

```
free(p1);
delete p2;
    delete p3;
    delete[] p4;
```

Ещё лучше после применения функции ***free()*** присвоить указателю значение **NULL**, которое затем, если его проверять в программе, будет сигнализировать о том, что память уже была освобождена:

```
#include<malloc.h>
int *p = NULL;
p = (int *) malloc(sizeof(int)*20);
            . . .
     *p=34;
            . . .
       free(p);
   p = NULL;
```

Вызывать функцию ***free*(p)** нужно в точности с тем же самым значением указателя p, по которому ранее и была выделена функцией ***malloc ()*** память. Если же в промежутке между двумя этими событиями (выделение и освобождение памяти) мы изменим значение указателя p, то операция осво-

бождения памяти не сможет быть корректно выполнена, а мы уже на работающей программе получим сообщение об ошибке.

Если все же требуется с помощью указателя обращаться к внутренним подобластям выделенной памяти, то делать это нужно не с помощью того же указателя, по которому и была выделена память, а с помощью другого, вспомогательного указателя

```
P =(int*)malloc(20);
      int  *q = p;
         q++;
           . . .
        free(p);
```

Если память выделялась для массива с помощью операции ***new*** , то для её освобождения необходимо применить конструкцию ***delete[].*** Размерность массива при этом не указывается. Если квадратные скобки не указать, ошибки не будет, но помечен как свободный будет только первый элемент массива, а остальные окажутся недоступными для дальнейших операций. Такие ячейки памяти называются *мусором.*

Отметим, что операцию '*' следует применять только после того, как значение указателя определено путем присваивания ему адреса ранее объявленной переменной или выделением памяти с помощью функции ***malloc()***, или операцией ***new***.

Отметим также, что имя *массива* или *строки* является константным указателем на начало массива или строки.

Покажем в качестве примера нетипичное использование буфера в динамической памяти на 12 байт, заказанного с помощью операции ***new*** на указатель p типа **char***. В первых четырёх байтах этого буфера мы разместим целое значение 5, а в оставшихся 8 байтах разместим значение 7.8 типа **double**. Когда буфер больше не требуется, его следует удалить операцией ***delete.***

```
                int n =5;
            double d=7.8;
            char *p = NULL;
                p = new char[12];
             char *q = p;
*(int*)q=n;  // приведение типа 'указатель на
             //char'к типу 'указатель на int' и
             //присвоение объекту q значения 5
q=q+4;     // смещаем указатель на 4 байта
           // для размещения числа 7.8
*(double*)q=d; // приведение типа указатель на
           // char к типу double и присвоение
```

```
объекту q значение объекта d
        . . .
      delete[]p;
```

Намного более типичным является использование операции ***new*** для выделения буферов под однотипные массивы

```
char *pc = NULL;
    pc = new char[2];
      for (int i=0; i<2; i++) *(pc+i)=i;
          . . .
        delete []pc;
```

Для сравнения следует заметить, что и потенциальные возможности обоих способов работы с динамически выделяемой памятью (с помощью ***malloc/free*** и с помощью ***new/delete***), и приёмы работы, и подводные камни у них одинаковые. Не следует использовать смешанные пары ***malloc/delete*** **и** ***new/free***, так как корректное поведение программы в этом случае не гарантируется.

Закрепим изученный материал, рассмотрев несколько учебных примеров. Определите, что будет выведено в результате выполнения следующей программы:

```
#include<stdio.h>
int main()
{
char c ='s';
const char lit ='r';  //объявление константы
const char *pcc=&lit;//указатель на константу
char *const cpc=&c;  //константный указатель
    //lit='n';-неправильно, так как lit-const
    //cpc = &lit; - неправильно, так как это
       константный   указатель
        pcc = &c;
//*pcc ='q'; - неправильно, так как это
             указатель на const
        *cpc ='q';
   printf("%c%c%c\n",lit, *cpc, *pcc);
          return 0;
    }
```

Ответ: rqq

Объясните все операции в следующем фрагменте:

```
int *pi;
 int K=5;
*pi=K;  //неправильно, почему?
```

```
pi=&K;
*pi=*pi*k;
printf("%d", *pi);
```

Ответ: 25

Указателю можно присвоить другой указатель, если они оба ссылаются на один тип. Для приведения к одному типу нужно использовать операцию явного преобразования типов. Например:

```
char *pc;
long l;
(long*) pc = &l;
double *pd;
int K;
pd = (double*)&K;
```

Исключением является указатель на тип **void** *(универсальный тип указателя),* которому можно присвоить указатели на все типы. Например,

```
void *pv; int K;
int *pi;
pi = &K;
pv = pi; или pv = &K;
```

Однако сам указатель на **void** не может быть присвоен указателю на другой тип без предварительного приведения. Например:

```
pi=(int*)pv;
```

Кроме того, указатель на **void** до операции приведения не может быть ни изменён, ни разыменован, так как он содержит ячейку памяти для не определенного типа данных и компилятору неизвестно, сколько байтов зарезервировано. Поэтому:

```
pv++;              //недопустимо
K =*pv;            //недопустимо
(int*) pv++;       //а так допустимо
```

Пример учебной программы по работе с указателями:

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
int main(void)
{ char *pc = NULL; int *pi = NULL;
double * pd = NULL;
char c='A';
int K=3;
double d=3.14;
pc =&c; pi =&K; pd =&d;
*pc ='F'; (*pi)++; *pd =1.23;
```

```
    printf ("Pointer pc=%p\n", pc);
    printf ("Pointed by pc value is %d\n", *pc)
    printf ("Pointed by pc value is %c\n", *pc)
                getch();
                return 0;
  }
```

На экран выводим значения, указуемые через указатель pc, то есть цифровой код литеры и саму литеру, а также адрес самого этого указателя, для чего в функции ***printf()*** *применяют* спецификацию формата %p.

Сделаем несколько замечаний по стилю записи указателей, а затем перейдем к рассмотрению операций с указателями.

- Хотя это и необязательно, но имена (идентификаторы) указателей принято начинать с буквы **p** (от английского слова **pointer**-указатель.)
- Возможны два варианта определения указателей:

int* pi; или **int** *pi;

То есть звездочку принято прижимать либо к идентификатору, либо к ключевому слову, определяющему базовый тип, на который ссылается данный указатель. Для компилятора оба стиля записи – «по барабану». Есть легкая тенденция к увеличению доли первого варианта в общем объеме написанного и опубликованного кода, но нужно иметь в виду следующее. Если применяется первый стиль, то для определения двух (и более) указателей нужно писать два и более оператора определения

int* pi1;
int* pi2;

а не один оператор определения

int* pi1, pi2;

Так как здесь второй идентификатор pi2 задает переменную типа **int**, а вовсе не указатель на **int**.

При втором же стиле записи в одной строке можно сразу определить несколько идентификаторов указателей:

int *pi1, *pi2;

- Необязательно инициализировать указатели сразу же в операторах определения, однако это очень полезная практика. Инициализация специальным значением **NULL** говорит о том, что указатель не показывает ни на что разумное, поэтому лучше с его помощью не обращаться ни к каким ячейкам памяти.

Вопросы и упражнения

1. Назначение и отличие статических и динамических программных объектов?
2. Дайте определение указательному типу. Что определяет множество значений этого типа и каковы операции над ним?
3. Почему наличие указателей делает языки С /С++ удобными для системного программирования?
4. Что позволяет получить операции `&` и `*`. С какими программными объектами они могут работать? Ограничения на использование операции `&`.
5. Каким образом определяется возможность работы указателя с объектами?
6. Какие три вида указателей вы знаете?
7. В каких случаях удобно применять указатель на **void**?
8. Что собой представляет имя массива (числового, строкового)?
9. Какие существуют способы работы с динамической памятью?
10. Какие существуют способы инициализации указателей?
11. Требуется ли явное преобразование типов при смешивании в выражении указателей разных типов? Если да, то применяется ли это правило для **void***?
12. Преобразуется ли значение 0 к указателю на любой тип?
13. Можно ли присваивать значения указателям-константам?
14. Могут ли ссылаться на одну и ту же область памяти несколько указателей различного типа? Если да, то применимая к ним операция разыменования даст разные или одинаковые результаты?
15. Пример, где конструкция **имя-указателя* может использоваться не только в правой, но и в левой части оператора присваивания:

```
char lit;
char *pc = new char;
    *pc = `a`; lit = *pc;
```

16. Будут ли иметь следующие два выражения разный результат и, если да, то почему?

```
* pc ++ = getchar();
* ++ pc = getchar();
```

17. Напишите программу, в которой операция **new** используется для динамического размещения переменных типа **float, long и char**. Задайте этим динамическим переменным начальные значения и выведите эти значения на экран. Произведите произвольную перестановку содержимого переменных и выведите эти значения на экран. В завершение с помощью операции **delete** освободите выделенную область памяти.

18. Какими двумя способами операция **new** может известить вас о неудачной попытке выделения памяти?

19. Каковы преимущества операций **new** и **delete** перед функциями **malloc()** и **free()?**

20. Переделайте данный фрагмент так, чтобы в нем использовалась операция **new**:

```
char *pc;
    pc = (char*) malloc(100);
        . . .
    strcpy (pc, "Это проверка");
```

21. Можно ли инициализировать динамический массив?

1.3. Операции с указателями

В языках C /C++ допустимы следующие (основные) операции над указателями: присваивание; получение значения того объекта, на который ссылается указатель (синонимы операции: косвенная адресация, разыменование, раскрытие ссылки); получение адреса самого указателя; унарные операции изменения значения указателя; аддитивные операции и операции сравнений. С частью операций мы с вами познакомились и использовали ранее, другие рассмотрим подробнее.

Но прежде еще раз отметим, что указатель – это переменная в стеке, содержащая ***адрес, по которому хранится значение определенного типа.*** Если к указателю прибавить число 1, то он будет ссылаться на следующую ячейку памяти. Если к указателю прибавить 5, то он будет ссылаться на ячейку, отстоящую от текущего адреса на пять ячеек.

Рассмотрим пример. Значение указателя, допустим, равно 1000, если к указателю прибавить 1, то результат (то есть насколько изменится значение указателя) будет зависеть от типа объекта, с которым он связан. Если объект имеет тип **char** , то результат сложения будет равен 1001, но если прибавить 1 к указателю типа **int**, то значение будет уже 1004 (так как тип **int** требует четыре байта памяти). Аналогично и для типа **float** (1004).

Поэтому при выполнении арифметических действий над указателями надо иметь в виду тип объекта, с которым указатель связан при своем определении. Пусть имеется массив

```
int arr [3] = { 1, 2, 3 };
```

и указатель p, типа **int** *, адресован на элемент этого массива с индексом 0:

```
p = & (arr [0]);      либо    аналогично  p = arr;
```

Возникает вопрос: на что будет ссылаться p после операции:

```
p = p + 1;
```

Ответ: он будет ссылаться на *элемент массива с индексом 1.* Так как тип **int** требует четыре байта памяти, значит и значение указателя, то есть его адрес, изменится на четыре. Это свойство указателей очень удобно, так как избавляет программиста от лишней работы.

Из только что рассмотренного ясно, что операция (pd)++; увеличивает значение указателя pd (имеющего тип **double***) на 8, и т. д.

Также вполне очевидна и операция вычитания *целого* числа из указателя, и операция вычитания одного указателя из другого. Например, разность между двумя указателями типа **int***, ссылающимися на *соседние* элементы массива целых, будет равна 4.

Когда целое число N складывается с указателем (или вычитается из него), указатель перемещается по ячейкам памяти на столько байт, сколько занимает N переменных того типа, на который он ссылается. Вычитание двух указателей определяет, *сколько переменных* данного типа размещается между указанными ячейками памяти. Правила сложения и вычитания указателей совпадают с арифметическими только для указателей на символьные переменные, так как такая переменная занимает "*только один байт памяти*".

Арифметика указателей имеет смысл только применительно к структурам данных, последовательно размещённым в памяти, например, к массивам, так как *только в этом случае* компьютер гарантированно записывает переменные одного типа в ячейки подряд друг за другом.

Сами же указатели любого типа всегда занимают в памяти 4 байта, то есть ими теоретически можно адресовать около 4 Гигабайта (миллиардов байт) памяти. Значением указателей всегда является адрес первого из входящих в блок памяти байтов. То есть, например, указатель типа **int *** логически адресует сразу 4 байта памяти, но непосредственное его значение является адресом самого первого из этих байтов. Отсюда следует вывод о том, что должен существовать *универсальный тип указателя*, предназначенного лишь для хранения адреса некоторого байта в памяти компьютера, но не предназначенного для манипулирования с числовыми значениями, хранящимися в этой памяти. Такой тип указателя специфицируется с помощью уже известного нам (и применяемого для других целей) ключевого слова **void**:

```
void *pv = NULL;    int K=5;    pv = &K;
```

Здесь указатель pv хранит адрес целой переменной K, а точнее он ссылается на первый байт из группы байтов, отведённых под переменную K. Попытка применить к этому указателю операцию «*» вызывает самое резкое возражение со стороны всё понимающего компьютера.

Ещё раз повторим, что этот тип указателей строго предназначен *только для хранения адресов* (и операций сравнения), а не для косвенного обращения к памяти. Если же нужно обработать участок памяти, адресуемый указателем типа **void***, то его нужно сначала явно привести к соответствующему типу указателей, что мы и делали выше.

С помощью указателя типа **void** можно передавать в функции адреса переменных любого типа, а затем, в теле функции, приводить указатель к требуемому типу в соответствии с полученной дополнительной информацией. Используя такой подход, можно создавать универсальные (обобщённые) функции, подходящие для обработки данных разного типа. В этом случае ключевое слово **void** в списке параметров функции или в качестве типа возвращаемого функцией значения означает *"ничего"*.

Следующая программа демонстрирует операцию приращения указателя

```
#include<stdio.h>
 int main(void)
{     double D[]={1.1, 2.2, 3.3, 4.4, 5.5};
     double *pd;
      pd = &D[0];    // можно и так:  pd=D;
       pd = pd+4;
     printf("%f\n", *pd);    // Ответ: 5.5
         return 0;
}
```

Как мы уже знаем, при выполнении арифметических действий с указателями нужно учитывать размер объекта, адресуемого данным указателем. В нашем случае приращение будет 4*8 = 32. В результате указатель pd сместится на четыре ячейки, каждая размером 8 (размерность типа **double**).

Инкремент перемещает указатель к следующему по порядку элементу массива, а декремент – к предыдущему. Фактически получается, что значение указателей изменяется на величину ***sizeof***(тип). Если указатель на определённый тип увеличивается или уменьшается на константу, то его значение изменяется на величину этой константы, умноженной на размер объекта данного типа. Например:

```
          int *pi=new  int[5];
pi++ //значение pi увеличивается на 4 (байта)
          double  *pd=new  double[5];
       pd++;  //значение pd увеличивается на 8.
```

Разность двух указателей – это разность их значений, делённая на размер типа в байтах (для массивов, например, разность указателей на третий и шестой элемент равна 3).

А вот *суммирование двух указателей не допускается*! Попробуйте дать объяснение этому запрету.

После выполнения операторов:

```
int   arr[5],*pi,*qi,K;
  pi = &arr[0]; qi = &arr[4];  K = qi-pi;
```

переменная K принимает значение 4.

Следует заметить, что тип разности указателей (в примере это тип **int**) определяется по-разному, в зависимости от особенностей компилятора. В языке **Си** с целью конкретизации типа разности указателей определено имя **ptrdiff_t** заголовочного файла **stddef.h,** использование которого позволяет однозначно задавать тип разности указателей.

```
#include <stdio.h>
#include <stddef.h>
   int main (void)
    {  int  arr[5];
       int  *pi,*qi;
            ptrdiff_t   K;
              pi = arr;
                qi = &arr[4];
                   K = qi-pi;
     printf("\n K=%d", (int)K); // K=4
            return 0;
     }
```

1.3.1. Арифметические операции и указатели

Унарные адресные операции `&` и `*` имеют более высокий приоритет, чем обычные арифметические операции. Рассмотрим пример, иллюстрирующий это правило:

```
        float  x=4.0,  *pf, y;
pf=&y; // указатель pf адресовали переменной y
*pf=5;   // с помощью операции разыменования
       //через указатель присвоили переменной
       // y значение 5
x=x+*pf+1; //4.0+5+1=10.0 Значение x стало 10.0
```

При использовании адресной операции `*` в арифметических выражениях следует остерегаться случайного сочетания знаков операций деления `/` и разыменования `*`, так как комбинацию `/*`

компилятор воспринимает как начало комментария. Например, выражение a/*b следует заменить таким: a/(*b).

Унарные операции `*` и `++` или `--` имеют одинаковый приоритет и при размещении рядом выполняются **справа налево.** Унарные операции инкремента - `++` и декремента - `--`, в зависимости от того, когда должно быть изменение значения указателя, до его использования в выражении, либо после, бывают префиксные и, соответственно, постфиксные.

Рассмотрим следующий фрагмент программы:

```
        int arr[4]={0,2,4,6}, *pi, k;
pi = arr;  // pi равно адресу элемента arr[0]
k = *pi;         // k равно 0; pi равно &arr[0]
k = *pi++;       // k равно 0; pi равно &arr[1]
k = ++*pi;       // k равно 3; pi равно &arr[1]
k = *++pi;       // k равно 4; pi равно &arr[2]
k = (*pi)++;  // k равно 4; pi равно &arr[2], a
                 //  arr[2] стало равно 5-ти
k = ++( *pi);  // k равно 6; pi равно &arr[2],
            // значение arr[2] увеличилось на 1
```

Работая с адресами и указателями нужно внимательно относиться к последовательности выполнения операций `*`, `++`, `--` и `&`, так как они в выражениях могут употребляться в самых разных сочетаниях. Пропагандировать такой стиль программирования не стоит, однако нужно уметь понимать смысл запутанных выражений с адресами и указателями. Следующая программа иллюстрирует особенности и приоритеты этих операций:

```
#include<stdio.h>
int main (void)
{ int i1 = 10, i2 = 20, i3 = 30;
          //  все переменные помещаются в стек,
          //  i1 имеет наибольший адрес
      int *pi = &i2;
     printf ("\nЗначение i2=%d", *&i2);
     // значение i2,сначала увеличенное на 1
       printf ("\nЗначение i2 = %d",*&++i2);
printf ("\nЗначение i2 = %d", *pi);
      // значение указателя увеличивается на 1
printf("\nЗначение i2=%d", *pi++);

printf("\nЗначение i1=%d", *pi);
```

```
        // значение i1 сначала увеличенное на 1
printf("\nЗначение i1=%d", ++*pi);
         // значение i2, сначала уменьшается pi
printf("\nЗначение i2=%d", *--pi);
// значение i3, сначала уменьшается pi, затем
// полученное значение i3 увеличивается на 1
printf("\nЗначение i3=%d", ++*--pi);
            return 0;
  }
```

Результат выполнения программы:

```
Значение i2=20
Значение i2=21
Значение i2=21
Значение i2=21
Значение i1=10
Значение i1=11
Значение i2=21
Значение i3=31
```

Результаты иллюстрируют "левонаправленность" выполнения расположенных рядом унарных операций `*`, `++`, `--` и `&`. Однако выражение *pi++ вычисляется в таком порядке: вначале выполняется разыменование (обращение по адресу), и полученное значение (21) служит значением выражения в целом. Затем выполняется постфиксная инкрементная операция pi++, и значение указателя увеличивается на 1. Тем самым в стеке он "устанавливается" на переменную i1. Последовательно уменьшая на 1 значение pi, переходим от i1 к участкам памяти, отведённым для i2 и i3 соответственно.

Недопустимыми будут следующие записи:

```
++ & i2;        -- &i2 ++;
```

так как смысл ошибки очевиден, ведь адрес участка памяти не есть леводопустимое выражение, **адрес – это константа** и его нельзя изменять.

1.3.2. Указатели и отношения

К указателям применяются операции сравнения: `>`, `>=`, `<`, `<=`, `= =`, `!=`. Таким образом, указатели можно использовать *в отношениях*. Но сравнить указатели допустимо только с другими указателями того же типа или с константой **NULL**, которая обозначает значение условного нулевого адреса.

Пример:

```
#include<stdio.h>
  float x[]={1.0,2.0,3.0,4.0,5.0};
  int main (void)
 {     float *pf1, *pf2; int i;
printf("\nАдресауказателей:&pf1=%p,&pf2=%p",
&pf1,&pf2);
  printf("\nАдреса элементов массива: \n" );
    for(i=0; i<5; i++);
       {if(i==3) printf("\n");
           printf("&x[%d]=%p", i, &x[i]);
       }
 printf("\nЗначения элементов массива: \n");
    for(i=0; i<5; i++)
     { if ( i== 3) printf ( "\n");
          printf ( "x[%d] = % 5.1 f",i, x[i]);
     }
      for(pf1=&x[0], pf2=&x[4]; pf2>=&x[0]; pf1++, pf2--)
       {
printf("\n pf1=%p *pf1=%5.1f pf2=%p *pf2=%5.1f", pf1, *pf1, pf2, *pf2);
    printf("\n pf2-pf1=%d", pf2-pf1);
       }
      return 0;
    }
```

При печати значений разностей указателей и адресов в функции **printf**() использована спецификация преобразования %d – вывод знакового десятичного целого.

Изучите приведённое решение, введите в компьютер и посмотрите результаты. Не бойтесь экспериментировать , вносите небольшие изменения в программу и анализируйте происходящие при этом процессы.

Вопросы и упражнения

1. Перечислите доступные операции над указателями.
2. С чем связан размер изменяемой памяти указателя при сложении (вычитании) с целым значением?

3. Когда целое число N складывается с указателем (или вычитается из него), на сколько байт указатель перемещается по ячейкам памяти?

4. “Арифметика” указателей имеет смысл только применительно к каким структурам данных?

5. Как программный объект, указатель сам имеет: адрес, длину и значение. Поясните эти свойства указателя?

6. Какими полезными свойствами обладает указатель **void*?**

7. Что нужно обязательно сделать, если необходимо обработать участок, адресуемый указателем типа **void*?**

8. Что означает ключевое слово **void** в списке параметров функции, либо в качестве типа возвращаемого ею результата?

9. Что определено в языке Си с целью конкретизации типа разности указателей, использование которого позволяет однозначно задавать тип разности указателей?

10. Какова разница использования унарных префиксных и постфиксных операций инкремента (декремента) в “арифметике” указателей?

11. Почему недопустимы следующие записи:

++ & k; -- & k ++; где k – переменная целого типа?

12. Что показывает спецификация преобразования - % p в операторе **printf()?**

13. Пусть определены следующие объекты:

```
int a = 10, b = 20, c = 30;  //переменная  a в стеке имеет
   int  *p = &b;             //максимальный адрес
      int  k ;
```

Что будет напечатано:

```
k = *&b;          printf(“*&b = %d \n”, k);
        k = *& ++ b;        printf(“*& ++b = %d \n”, k);
k = *p;                   printf(“*p = %d \n”, k);
        k = *p ++;              printf(“*p ++ = %d \n”, k);
k = *p;                   printf(“*p = %d \n”, k);
        k = ++ * p;             printf(“++*p = %d \n”, k);
k = *-- p;                printf(“*--p = %d\n”, k);
        k = ++*--p;             printf(“++*--p = %d”, k);
```

14. Постройте логическую связь между следующими фразами:

1) &p – адрес указателя.
2) Указатель p. Значение указателя.
3) Адрес объекта. Значение указателя p.
4) *p – объект, адресуемый указателем p. Значение объекта.

Нарисуйте схему этих связей в графической нотации.

15. Пусть определены следующие переменные:

```
int  k = 10;
int  *p = &k;
```

Допустимы ли следующие операции: ++ & k; -- & k ++ ; ?

1.4. Связь массивов и указателей

Из материала предыдущих разделов видна тесная связь указателей с массивами, у которых элементы располагаются подряд в непрерывных участках памяти компьютера. На эту мысль наталкивает и величина разности адресов последовательных элементов массива, и специфическая арифметика указателей, когда в зависимости от типа указателя операция '++' может увеличивать значение указателя (то есть адрес адресуемой им ячейка памяти) на 1, 2, 4 и 8 единиц соответственно. Это позволяет с помощью чрезвычайно лаконичной операции '++' продвигаться последовательно по элементам массива любого типа.

Однако эта связь ещё теснее, чем можно предположить из анализа представленных выше данных. Дело в том, что ***имя массива, используемое без квадратных скобок, представляет собой адрес первого элемента массива*!**

Адреса элементов массива вычисляются во время компиляции и не могут быть изменены в дальнейшем во время работы программы. В итоге, имя массива представляет собой фактически константу – указатель (инициализированную адресом элемента массива с индексом, равным 0).

Получается, таким образом, что с массивом можно работать двойственным образом, используя либо более привычный синтаксис операций индексации, либо синтаксис указателей! Выбор одного из двух возможных способов целиком зависит от вкуса программиста. В качестве экзотики можно даже чередовать их в рамках одной программы.

Продемонстрируем оба способа работы с массивами:

```
#include<stdio.h>
#include<conio.h>
  int main(void)
  {       int  arr[8] = {1, 6, 3, 8, 2, 8, 1, 3};
          int  sum, i ;
 for( i = 0, sum = 0; i < 8;  i++)   sum = sum + arr[i];
                                  printf("\n сумма = %d\n", sum);
                      sum = 0;
                for( i = 0; i < 8; i++)   sum = sum + *(arr+i);
                printf("сумма = %d", sum);
```

```
        getch();
        return 0;
    }
```

Такая же двойственность синтаксиса имеет место и по отношению к динамически выделяемым буферам памяти. Если выделен буфер

```
int *p = (int*) malloc(sizeof(int)*10);
```

на 10 целых значений (40 байт), то к отдельным целым значениям можно обращаться как по указательному синтаксису

```
*( p+4) //  обращаемся к значению  5-го элемента
```

так и с помощью операции индексации – p [4].

Рассмотрим задачу *"инвертирования"* массива символов и различные способы её решения с применением указателей.

Первое решение:

```
char  str[80],  lit;
char  *pc,  *qc;
    for (pc = str, qc  = &str[79]; pc < qc; pc++, qc--)
            { lit = *pc;   *pc = *qc;   *qc = lit; }
```

В заголовке цикла инициализируем указатели, то есть адресуем их первым и последним элементами массива; сравниваем оба указателя для управления циклом, и модифицируем их операциями инкремента и декремента. Смысл операций тела цикла очевиден.

Второе решение:

```
                    char  str[80], lit, *pc, *qc;
for (pc = str, qc = &str[79]; pc < qc;)   // модификацию указателей
                                          // перенесли в тело цикла
        {lit = *pc;   *pc++ = *qc;   *qc-- = lit;}
```

Третье решение:

```
char str[80], lit, *pc, *qc;
            pc = str;
              qc = &str[79];
  while(pc < qc)
     {  lit = *pc;   *pc++ = *qc;   *qc-- = lit;}
```

Четвертое решение - имитация индексированных переменных указателями со смещениями:

```
char  str [80], lit;
 int i;
    for( i=0; i < 40; i++)
        {     lit = *(str+i);   *(str+i) = *(str+(79-i));
              *(str+(79-i)) = lit;
        }
```

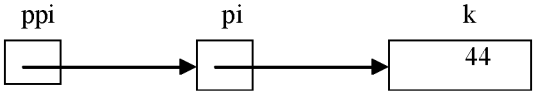
Таким образом, &str[i] и (str+i) – это две формы определения **адреса** одного и того же элемента массива, отстоящего на i позиций от его начала

Ещё раз, (str+i) – это **адрес** i-го элемента массива, а *(str+i) – это **значение** i-го элемента, т. е. в нашем случае – это символ. Так как имя массива есть константа, содержащая адрес начала массива, поэтому к имени массива не применимы операции '++', '--'; имени массива нельзя присвоить значение, т. е. имя массива не может использоваться в левой части операции присваивания.

Рассмотрим указательный синтаксис работы с двумерными массивами. Для этого прежде рассмотрим указатель на указатель, т. е. производный тип данных более высокого порядка, чем указатели на простые типы данных.

```
          int k;
int *pi = NULL;     int ** ppi = NULL;
    pi = &k;          ppi = &pi;
  **ppi = 44;        // теперь значением К стало число 44.
```

Графически такую связь изобразим следующим образом:



В определении указателя на указатель (двойной указатель) требуется двойное присутствие знака '*'. Чтобы воздействовать на простой тип, данных через двойной указатель, требуется в свою очередь два раза применить операцию '*'.

Если в косвенном обращении к памяти через двойной указатель ppi ***однократно*** применить операцию '*', то таким образом осуществится доступ непосредственно к промежуточному указателю pi, и, в частности, можно будет изменить его значение, т. е. изменить адрес указателя pi, так что он теперь перестанет ссылаться на целую переменную К. Таким образом, * ppi = = pi.

Теперь рассмотрим "указательный" синтаксис работы с двумерными массивами. Указательный синтаксис в этом случае основан на том, что имя двумерного массива представляет собой "константу – двойной указатель", однократное применение к которой операции '*' обеспечивает доступ к адресу блока памяти, по которому располагаются элементы *первой* строки двумерного массива. Фактически блок памяти – это массив указателей на строки матрицы. Если теперь к выражению

(имя_матрицы +1)

применить операцию '*', то получим доступ к адресу блока памяти, по которому располагаются элементы *второй* строки двумерного массива.

Для двумерного массива

int arr [5] [5];

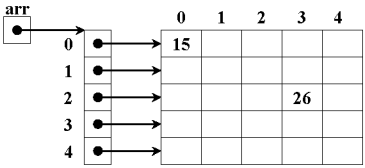
из пяти строк и пяти столбцов выражение *arr представляет собой указатель на первую (с нулевым индексом) строку этого двумерного массива, а выражение *(arr+1) является указателем на вторую его строку и т. д.

Выражение

* (* (arr + 2) + 3)

дает целое значение элемента массива arr из третьей строки и четвертого столбца, т. е. число 26. Выражение ****arr** – это целое число из первой строки и первого столбца, т. е. число 15. Значением выражения *** arr** является *указатель на первую строку*, а значением выражения **** arr** - *содержимое первого элемента этой строки.*

Поясним сказанное графически:



Напомним, что значением указателя является *адрес первого байта блока байтов,* отводимых под данные.

Ниже приведен учебный пример, в котором используются два способа работы с двумерным массивом:

```
# include<stdio.h>
# include<conio.h>
 int main(void)
   {  int  arr[2][3]  = {{1, 6, 3}, {8, 2, 8}};
          int sum  = 0, i, j;
             for ( i = 0; i < 2; i++)
                  for( j = 0; j < 3; j ++)
                          sum = sum + arr[i][j];
               printf("\n сумма = %d \n", sum);
             sum = 0;
            for ( i = 0; i < 2; i ++)
                  for( j = 0; j < 3; j ++)
                          sum = sum + *(*(arr + i) +j);
```

```
        printf("сумма = %d", sum);
        getch();
        return 0;
}
```

Заметим, что выражение arr[0] – представляет собой адрес начала самой первой строки двумерного массива arr, а выражение arr[2] адресует его третью строку и т. д.

Если мы хотим динамически выделять память под двумерный массив целых, что означает, что мы хотим обращаться к целым значениям в выделенном блоке памяти по синтаксису двумерных массивов, то следует определить указатель на массив. Такой тип указателя языков C/C++ обеспечивает возможность доступа к значениям в динамически выделенном буфере памяти как к строкам и столбцам в соответствии с операциями индексации статических двумерных массивов:

```
int ( *par )[4];
  par = new  int [12][4];
for( i = 0;  i < 12;  i++)
    for( j = 0;  j < 4;  j++)      par[i][j] = i + j;
    delete [ ] par;
```

Здесь указатель par имеет тип

```
int (*)[4] ,
```

что читается как "*указатель на массив из четырех элементов типа* ***int***". Это пример *абстрактного описания типа данных.*

Предложим еще один способ динамического выделения памяти под двумерный массив размерности n*m:

```
int  **par;
par = ( int**)malloc(sizeof( int* )* n);
```

Двойной указатель par адресуем блоком памяти из (4*n) байт. Блок – это массив указателей. Так как функция **malloc**() возвращает указатель типа **void** *, поэтому явное преобразование обязательно.

На следующем шаге выделим **n** блоков памяти для **m** целых значений двумерного массива и адресуем каждый элемент массива указателей адресами этих блоков:

```
for( i = 0;  i < n;  i++)
  par[i] = (int*)malloc(sizeof(int)*m);
```

и наконец, инициализируем полученный двумерный массив

```
for( i = 0;  i < n;  i++)
    for( j = 0;  j < m;  j++)
            par [i][j] = i + j;
```

Таким образом, par – это указатель на указатель, par[i] – i - й элемент массива указателей, т. е. это указатель на i- ую строку матрицы.

Освобождение памяти заключается в процедуре последовательного удаления указателей в массиве указателей, а также и самого указателя на массив указателей

```
for( i = 0;  i < n;  i++)
free ( par[i]);
free ( par);
```

1.4.1. Массивы указателей

Массивы указателей служат для динамического управления памятью при работе со сложными структурами данных и позволяют эффективнее использовать память компьютера.

Массив указателей фиксированного размера вводится одним из следующих определений:

тип * *имя_массива* [размер] ;
тип * *имя_массива* [ ] = *инициализатор*;
тип * *имя _массива* [размер] = *инициализатор*;

Например:

```
int data[6];   // обычный массив
int *p1[6];   // массив указателей
int *p2[ ] =  { &data[0],  &data[4],  &data[2] };
```

Здесь каждый элемент массивов **p1 и p2** является указателем на объекты типа **int**, а в последнем примере три элемента, кроме того, инициализированы адресами конкретных элементов массива data.

Интересен случай применения массива указателей для сортировки строк матрицы различными способами. Количество способов сортировки будет определять количество необходимых массивов указателей. Рассмотрим простую программу одновременного упорядочения как по возрастанию, так и по убыванию одномерного массива без перестановки его элементов:

```
#include<stdio.h>
#include<conio.h>
int main (void)
{       float arr[ ] = {5.0,2.0,3.0,1.0,6.0,4.0};
        float  *pmin[6];
        float  *pmax[6];
```

```
        float  *p;
      int  i, j ;
            for ( i = 0; i < 6; i++)
                pmin [i] = pmax [i] = &arr[i];
          for ( i = 0;i < 5;i ++)
            for ( j = i + 1; j < 6; j ++)
                {
                  if ( *pmin[i] < *pmin[j] )
                        {
                p = pmin[i];  pmin[i] = pmin[j]; pmin[j] = p;
                        }
                         if (*pmax[i]  > *pmax[j] )
                              {
                p = pmax[i]; pmax[i] = pmax[j]; pmax[j] = p;
                              }
                }
                 printf("\nПо убыванию:\n");
  for ( i = 0;i < 6;i ++)    printf("\t%5.1f",*pmin[i]);
                 printf("\nПо возрастанию:\n");
   for (i = 0;i < 6; i++)    printf("\t%5.1f",*pmax[i]);
                                getch( );
                               return 0;
}
```

Рассмотрим теперь использование массивов указателей и средств динамического выделения памяти для обработки объектов разного размера, оформленных в виде двумерного массива.

Пусть необходимо ввести и распечатать в обратном порядке набор строк числовых значений произвольных длин:

```
#include<stdio.h>
#include<conio.h>
int main (void)
{
    int  **p;       //  p - указатель  на блок  памяти, выделяемого
                          //     для массива указателей
        int  i, j, k, n;
        int  *q;       \\   q –указатель для блока памяти
            printf("\nВведите количество строк n=");
            scanf("%d",&n);
                  p = ( int **)calloc(n,sizeof(int *));
                  q = ( int *)malloc(sizeof(int) *n);
        for( i = 0; i < n; i++)
                {
```

```
                printf("Введите длину строки q[%d]=",i);
                scanf("%d",&q[i]);
            p[i] = ( int *) calloc(q[i],sizeof(int));
                for( j = 0; j < q[i]; j ++)
                    {
                        printf("p[%d][%d] = ", i, j);
                        scanf("%d",&k);
                            p[i][j] = k;
                    }
            }
        printf("\nРезультаты обработки:");
         for( i = n-1; i >= 0; i--)
            {
    printf("\nСтрока %d,  элементов%d:\n",i,q[i]);
                    for( j = 0;j < q[i]; j++)
                        printf("\t%d", p[i][j]);
    free(p[i]);         \\Освободить память строки
                }
    free(p);    \\Освободить память от массива указателей
    free(q);    \\Освободить память массива int:
                    return 0;
}
```

1.4.2. Строки и указатели

Ранее мы рассмотрели взаимосвязь указателей с числовыми массивами. Строка – это тоже массив и, поэтому нет ничего удивительного во взаимосвязи такого типа данных и указателя. Строковая константа – это символьный массив с конечным нулевым символом (‘\0’). Указатель на строку можно объявить и инициализировать в синтаксисе указателей следующим образом:

```
char  *ps  =  " My string";
```

В конце строки будет добавлен символ ‘\0’.

Указателю ps типа **char *** будет присвоен адрес первого символа строки, то есть ‘M’.

Эту строку можно записать по – другому:

```
char  *ps;
 ps = "My string";
```

Существует также и привычный формат определения строки, т. е. в синтаксисе операции индексирования

```
char  str[20];
```

- массив, в который можно записать до 20 символов. Массив str получает память в стеке ***автоматически*** *при обработке данного определения*. Строке, с которой мы хотим связать указатель ps, память при обработке определения указателя ***не выделяется***. Поэтому если далее следуют операторы

```
scanf ("% s", str);    // оператор верен
scanf ("% s", ps);     // оператор не корректен, так как
                       // памяти под строку нет,
```

то первый из них допустим, а второй приводит к ошибке во время выполнения программы – попытка ввода в неопределенный участок памяти. До выполнения второго из этих операторов ввода, с указателем ps нужно связать некоторый участок памяти. Для этого существует несколько возможностей. Во – первых, переменной ps можно присвоить адрес уже определенного в стеке символьного массива, т. е. ps= str. Во – вторых, указатель ps можно «настроить» на участок памяти в «куче», выделяемый с помощью средств динамического распределения памяти. Например, следующая операция присваивания

```
ps = (char*)malloc(80);
```

выделит 80 байт и свяжет этот блок памяти с указателем ps. Вот только теперь применение второго оператора ввода корректно.

Рассмотрим две следующие строки:

```
char *ps  =  "My string";

char str[] = "My string";
```

Основное различие заключается в том, что значение указателя ps может быть изменено, так как указатель – это переменная, а значение имени массива str не может быть изменено, потому что имя массива – константный указатель.

Поэтому следующие выражения ошибочны:

```
char  arr[15];
  arr = "My string";     // ошибка,
```

так как операция присваивания пытается скопировать адрес первой ячейки строки "My string" в объект arr. Однако имя массива arr, как мы уже знаем, является константой, а не переменной – указателем, поэтому будет выдано сообщение об ошибке.

Следующий фрагмент также ошибочен:

```
char  *ps;
scanf("% s", ps);
```

потому что указатель ps объявлен, но ***не инициализирован***.

Для решения проблемы можно поступить следующим образом:

```
char  str[10];
```

```
char  *ps = str;
```

Теперь чтение потока символов длиной не более десяти будет выполнено успешно, поскольку в функцию **scanf()** передается существующий адрес массива

```
scanf("% s", ps);
```

Для размещения строки в динамической памяти надо описать указатель **char *** , а затем выделить память с помощью операции **new** , либо функцией **malloc** (первый способ предпочтительнее):

```
char *ps = new char[N];
char *ps = (char *)malloc(sizeof(char)*N);
```

В этом случае длина строки может быть переменной и задаваться на этапе выполнения программы. Динамические строки, как и другие динамические массивы, нельзя инициализировать при создании, так как операция присваивания

```
char  *ps = "It is a constant";
```

создает не строковую переменную, а указатель на строковую константу, изменить которую невозможно.

С помощью указателей типа **char *** удобно получать доступ к строкам в виде массивов символов. Типичная задача – обработка слов или предложений, каждое из которых представлено в виде строки (в конце предложения или слова находится нуль – символ '\0'). Использование указателей, а не массивов с фиксированными размерами особенно целесообразно, когда предложения или слова должны быть разной длины. В следующей программе определен, а после инициализации связан с набором строк, одномерный массив указателей типа **char** *. Функция **printf()** и спецификатор преобразования % s допускают использование в качестве параметра функции указатель на строку, при этом на экран выводится не значение указателя arp[i] , а ***все содержимое адресуемой им строки.***

Текст программы:

```
#include<stdio.h>
 int main( void)
{  char  *arp[] = { "One", "Two","Three","Four","Five"};
        int i, n;
    n = sizeof (arp)/sizeof(arp[o]); // длина массива
            for ( i = n; i > o; i--)
                printf("\n % s", arp[i]);
        return o;
 }
```

Результат выполнения программы:

Five
Foure
Three
Two
One

Все стандартные функции языков C/C++ при обработке строк работают с указателями на строку. Среди множества этих функций нам представляется полезным рассмотреть использование функции **strtok()**, с помощью которой можно разбивать строки на лексемы – неделимые последовательности литер

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
int   main (void)
     {   char str[] = “This sentence;has-5 words.”;
          char *p;
               printf(“\nИсходная строка:”);
               printf ( “%s \n”, str);
               printf(“раскладывается на слова:\n”);
                  p = strtok(str,”  ;-.”);
```

/* Параметрами функции **strtok()** являются указатель на исходную строку, а также строковая константа, состоящая из разделителей: пробел, точка с запятой, дефис и точка.

Функция **strtok()** возвращает из исходной строки лексему, отделенную от других строк любым из указанных разделителей. Указатель p – это есть указатель, ссылающийся на следующий после разделителя символ, начиная с которого должен осуществляться поиск следующей лексемы. Каждый последующий вызов функции, но теперь уже с нулевым указателем (**NULL**) в качестве значения первого аргумента, продолжает поиск лексем, начиная с того места в исходной строке, на которое ссылается возвращенное в результате последнего вызова значение указателя p .

Библиотечные функции не должны вызывать функцию **strtok()**. Функция **strtok()** возвращает нулевой указатель, если лексемы в строке не обнаружены. */

```
while( p != NULL)
   {   printf(“% s\n”, p);
          p =  strtok( NULL, “  ;-.”);
   }
```

```
        getch();
    return 0;
}
```

Результаты выполнения программы:

Исходная строка: This sentence;has-5 words.
раскладывается на слова:
This
sentence
has
5
words

Вопросы и упражнения

1. Что собой представляет имя массива, используемое без квадратных скобок?

2. Что такое синтаксис операции индексации и синтаксис указателей?

3. Покажите две формы определения адреса одного и того же элемента массива, отстоящего на i позиций от его начала.

4. В чём суть указательного синтаксиса работы с двумерными массивами?

5. Применимы ли к имени массива операции '++' , '--'? Может ли имя массива использоваться в левой части операции присваивания?

6. Ввести и напечатать в обратном порядке набор вещественных чисел, количество которых заранее не фиксировано, а вводится до начала ввода самих числовых значении.

7. Перечислите все определения, с помощью которых вводится массив указателей фиксированных размеров.

8. Корректна ли запись: имя_массива = = & имя_массива = = & имя_массива[0]?

9. Пусть имеются следующие определения:

```
int m[ ] = {10,20,30,40};
int i = 1, k;
```

Какие значения будет принимать переменная k:

1) k=m[i]; **printf**("\n m[i]=%d\n",m[i]);
2) k=*(m+i++); **printf**("*(+i++)=%d\n",*(m+i++));
3) k=*(++i+m); **printf**("*(++i+m)=%d\n",*(++i+m));
4) k=i[m]; **printf**("i[m]=%d\n",i[m]);
5) k=*(i--+m); **printf**("*(i--+m)=%d\n",*(i--+m));
6) k=i--[m]; **printf**("i--[m]=%d\n",i--[m]);

7) k=*(i--+m); **printf**("*(i--+m)=%d\n",*(i--+m));

8) k=--i[m]; **printf**("--i[m]=%d",--i[m]);

10. Пусть даны следующие определения:

```
char  mas[ ] = "top";
char *pc = &mas[2];
```

Вывести на экран слово "top" в обратном порядке. Можно ли это сделать с использованием вспомогательной переменной.

1.5. Указатели в параметрах функций

Как мы знаем, существуют различные механизмы передачи параметров при вызове функций: ***по значению, по указателю и по ссылке.*** Схема передачи параметра *по значению* означает передачу в функцию копии фактического параметра. Поэтому объект вызывающей программы, использованный в качестве фактического параметра, не может быть изменён из тела функции. Однако существует косвенная возможность изменять значения объектов вызывающей программы действиями в вызванной функции. Эту возможность обеспечивает аппарат указателей. С помощью указателя в вызываемую функцию передается *адрес* любого объекта из вызывающей программы, а с помощью выполняемой в тексте функции операции разыменования указателя, мы получаем доступ к адресуемому указателем фактическому объекту вызывающей программы.

Таким образом, не изменяя самого параметра функции, поскольку *указатель-параметр* ***постоянно*** содержит только адрес одного и того же объекта, можно изменять содержимое объекта вызывающей программы.

Продемонстрируем изложенный механизм на простом примере

```
#include<stdio.h>
   void  positive (int *p)
         {
           *p = (*p>0)?  *p :  -*p;
   }

    int main (void)
         {
            int  k = -3;
                    positive(&k);
              printf("\n k =%d", k);
                return 0;
         }
```

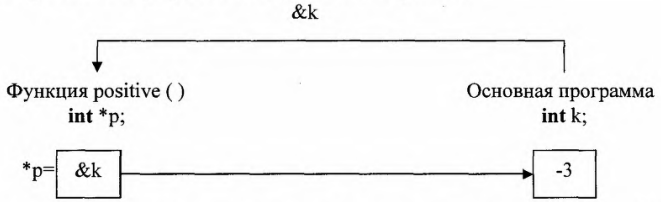
Ответ: k=3

Параметром функции positive() является указатель типа **int***. При вызове этой функции в качестве фактического параметра используется адрес переменной k, который и присваивается указателю p. Разыменование *p обеспечивает доступ к тому участку памяти, на который ссылается указатель p. Тем самым в выражении

*p = (*p > 0) ? *p : -*p;

все действия выполняются над значениями переменной основной программы, адрес которой использован в качестве аргумента.

Графически взаимосвязь функции positive() и основной программы проиллюстрируем следующим образом:



p – адрес переменной к; *p – содержимое переменной к; *(&k) – содержимое переменной k.

Другим популярным примером эффективности механизма передачи параметров в функцию по указателю является задача обмена значениями между двумя объектами основной программы. Для этого в определении функции формальные параметры, с помощью которых результаты должны передаваться из функции в точку вызова основной программы, описываются как указатели. Теперь с помощью этих указателей может быть реализован доступ из тела функции к тем объектам вызывающей программы, которые адресуются параметрами - указателями.

```
#include<stdio.h>
int main(void)
 {
        int m, n;
          void change( int *, int *);   // прототип функции
                printf ("\nВведите m=");
                scanf ("% d", &m);
                    printf ("\n Введите n=");
                    scanf ("% d", &n);
             change (&m, &n );
printf("\nРезультат обмена: m=%d n=%d",m,n);
        return 0;
```

```
}
// определение функции, которая меняет местами значения переменных
//m и n, на которые указывают формальные параметры функции p,q
void change ( int *p, int *q)
{
        int term;
            term = *p;   *p = *q;   *q = term;
}
```

В основной программе описаны переменные m и n , значения которых пользователь вводит с клавиатуры. Формальными параметрами функции change() служат указатели типа **int** *. Задача функции – поменять местами значения тех переменных, на которые ссылаются её параметры. При вызове функции change() адреса переменных m и n используются в качестве фактических параметров.

1.6. Массивы и строки как параметры функций

Часто требуется передать массив для обработки в какую-либо функцию. Это сделать очень легко, так как имя массива, взятое само по себе, представляет, как мы уже знаем, адрес первого его элемента. Таким образом, фактически в функцию передается лишь адрес начала массива, и далее с ним работают по синтаксису указателей (отсюда вытекает, что ***массив передать в функцию «по значению» нельзя).***

Кроме того, нужно в виде отдельных аргументов передавать размеры массива (имеются в виду многомерные массивы), чтобы внешняя функция могла получить всю необходимую информацию.

Приведем пример передачи одномерного массива в функцию для вычисления суммы всех его элементов:

```
#include<stdio.h>
#include<conio.h>
                // прототип функции
int Mysum (int *p, int size);
        int main (void)
         {    int ar[3] = {1,6,3};
          int sum;
              sum = Mysum (ar,3);
           printf ("сумма = %d\n", sum);
             getch ();
              return 0;
          }
```

```
int Mysum (int *p, int size)
 {  int result = 0;
for (int i=0; i < size; i++)
      result = result + *(p+i);
// можно и так :  result = result + p[i];
   return result;
 }
```

Теперь рассмотрим передачу в функцию многомерных массивов. Для этого поставим учебную задачу: надо передать в функцию двумерный массив, целые элементы которого будут суммироваться в этой функции. Приведем самое прямое и поэтому самое правильное решение

```
#include<stdio.h>
int Fun (int ar[2][3], int n , int m );
int main (void)
 {       int ar[2][3] = {1,2,3,4,5,6};
int sum;
      sum = Fun (ar, 2, 3);
 printf ("\n сумма = %d\n", sum);
   return 0;
 }
  int Fun (int a[2][3], int n, int m)
 {
        int s=0;
   for (int i=0; i<n; i++)
 for (int j=0; j<m; j++)  s=s + a[i][j];
     return 0;
 }
```

В определении функции можно убрать число 2 из первых квадратных скобок, но убирать все числа из квадратных скобок – нельзя.

Любая попытка перейти на двойные указатели с явным преобразованием

```
sum = Fun(( int **) ar, 2, 3);
```

приводит к ошибке во время выполнения программы.

Правильным решением является описание двумерного массива в качестве входного параметра функции Fun в следующем виде:

```
int Fun (int (*a)[3], int n, int m);
```

с последующим вызовом этой функции:

```
sum = Fun (ar, 2, 3);
```

В этом случае и компилируется все безошибочно, и программа выполняется правильно.

Усложним нашу учебную задачу: в функции Fun будем инициализировать двумерный массив AR тех же самых размеров, что и массив ar, причем его элементы будем получать удвоением элементов массива ar. Кроме этого, вернем из функции Fun и сумму всех элементов проинициализированного массива AR . Вот решение этой задачи:

```
    #include <stdio.h>
    #include <conio.h>
  int Fun (int a[2][3],int n,int m,int b[2][3], int *);
    int main (void)
    {
      int ar[2][3] = {1,2,3,4,5,6};
     int AR[2][3];
     int ret, sum;
          ret = Fun (ar, 2, 3, AR, &sum);
   printf ("ret = %i sum = %d\n", ret, sum);
       getch();
       return 0;
    }
int Fun(int a[2][3],int n,int m,int b[2][3],int *p)
    {
      int s=0, sum =0;
    for (int i=0; i<n; i++)
       for (int j=0; j<m; j++)
    {          s = s + a[i][j];
           b[i][j]= 2*a[i][j];
            sum = sum + b[i][j];
    }
        *p = sum;
       return s;
    }
```

Ответ: ret=21, sum=42.

Сделаем некоторые замечания по поводу массивов как параметров функций .

Заголовки функций:

```
int f ( int n, int A [] ) . . .
int f ( int n,  int *p)   . . .
```

Совершенно равноправны в спецификациях формальных параметров, однако в первом случае роль имени A как указателя не так явственна.

В теле функции обращение к элементам массивов – параметров возможно как с помощью операции индексации, т. е. A[i] , так и с помощью операции разыменования соответствующих значений адресов, т. е. используя синтаксис указателей в выражении: *(p+1).

Так как массив всегда передается в функцию через указатель, то внутри функции можно изменять значения элементов массива. Это возможно как при использовании обычной операции индексирования, так и при разыменовании указателя на элементы массива!

Для иллюстрации указанных возможностей рассмотрим функцию, возводящую в квадрат значения элементов одномерного массива

```
#include<stdio.h>
#include<conio.h>
  void  quart ( int, float*);
int  main (void)
{    float z[] = { 1.0,2.0,3.0,4.0};
     int  i;
              quart (4,z);       // вызов функции
       //   печать измененного массива
         for ( i = 0; i < 4; i ++)
               printf ("\n z[%d] = %f" , i , z[i] );
                getch();
                   return  0;
}
       //  определение функции
         void  quart( int  n, float  *pf)
{    int  i;
                  for( i=0; i < n; i ++)
                       *( pf + i)  *=  *(pf + i);
}
```

Результат выполнения программы:

```
z [0] = 1.000000
z[1] = 4.000000
z[2] = 9.000000
z[3] = 16.000000
```

Обратите внимание на то, что заголовок функции мог быть и таким:

void quart (**int** n, **float** ar []).

В теле функции разыменовано выражение, содержащее имя массива – параметра, т. е. вместо индексированной переменной pf[i]

используется выражение *(pf + i). Кстати, в теле функции справедливо будет и так: pf[i].

Более интересная возможность состоит в том, что можно изменять внутри тела функции значение указателя, например, в теле цикла записать такой оператор:

```
* pf * = * pf ++;
```

Каким образом будет организована работа функции guart() в случае такого варианта записи этого оператора ? :

```
* pf ++  *= * pf ++;
```

Следует отметить, что имя массива внутри тела функции не воспринимается как константный указатель, т. е. не допускающий изменений, а вот в основной программе, т. е. в **main()**, имя массива всегда указатель – константа! Поэтому, если в теле основной программы, например в функции **printf()** , вместо значения z[i] попытаться использовать выражение *z++, то получим сообщение об ошибке.

Подведем итоги, отметив, что с одной стороны, имя массива является константным указателем со значением, равным адресу нулевого элемента массива. С другой стороны, имя массива, использованное в качестве параметра функции, играет в теле этой функции роль обычного (неконстантного) указателя, то есть может использоваться в качестве леводопустимого выражения. Именно поэтому в спецификациях формальных параметров функции эквивалентны такие формы:

int ar[] и **int** *ar

Строки как параметры функции

Строки в качестве фактических параметров могут быть специфицированы либо как одномерные массивы **char**[], либо как указатели типа **char*.** В обоих случаях с помощью фактического параметра в функцию передается адрес начала символьного массива, содержащего строку. В отличие от обычных массивов для параметров – строк нет необходимости явно указывать их длину. Эскейп – последовательность ‘\0’ в конце строки, означающая конец строки, позволяет перебирать элементы строки и не выходить таким образом за ее пределы.

В качестве полезных примеров рассмотрим некоторые стандартные функции решения типовых задач обработки строк, используя различные способы задания строк – параметров.

Функция вычисления длины строки

```
int  len (char s [])              // заголовок по стандарту   ISO
{   int  n;
  for (n = 0;  s[n] ! = '\0';  n++) ;      // цикл без тела
                return  n;
}
```

Компилятор всегда воспринимает массив как указатель на его начало, а индекс как смещение относительно начала массива. Тот же вариант функции, но явно реализующий механизм работы с указателями:

```
int len ( char * ps)
 {  int n;
        for ( n=0; *ps ++; n++) ;
          return  n;
 }
```

Еще один вариант – через разность двух указателей на строку:

```
 int len ( char * ps)
 {    char  *q;
        for ( q = ps;  *q ++;  );
         return  (q - p);
 }
```

Функция инвертирования строки

```
 void   invert ( char * ps)
{    char lit;
 int  i, j, m;          //  m - номер позиции  '\0'  в строке
       for ( m=0; *ps ++; m++);
          for ( i = 0, j = m-1;  i < j;  i++, j++)
              {    lit = *( ps + i);
                    * ( ps + i )  = * ( ps + j);
                        * ( ps + j) = lit;
              }
}
```

Символ '\0' после инвертирования остается на своем месте, в конце строки.

Функция поиска в строке ближайшего слева вхождения другой строки

```
          // поиск строки  s2   в строке  s1
          //  аналог стандартной функции   strstr()
int  index ( char  * ps1, char * ps2)
{   int   i , j, d1, d2;        //  d1,d2 – длины строк
```

```
                // вычисление длин строк
            for ( d1 = 0;  * (ps1 + d1);  d1++ );
             for ( d2 = 0; * (ps2 + d2); d2 ++) ;
                 if ( d2 > d1)    return  -1;
                    for (i =0; i <= d1-d2;  i ++)
        {  for ( j=0; j < d2; j++)      // цикл сравнения
                              if ( * ( ps2 + j) != * (ps1 + ( i+j)))
                                break;
                                 if ( j == d2)   return  i;
         }
              return   -1;
         }
```

Функция возвращает номер позиции, начиная с которой фрагмент строки от ps2 полностью совпадает с частью строки, начиная с ps1.

Функция сравнения строк (стандартная функция **strcmp()**)

```
          int row ( char s1 [] , char s2[] )
    {   int  i, d1, d2;              // d1,d2 - длины строк
              for (d1 = 0; * (s1 + d1 ) != '\0'; d1++);
                for ( d2 = 0; * ( s2 + d2) != '\0'; d2++);
                     if ( d1 != d2)    return  -1;
                        for ( i=0; i< d1; i++)
                       if(  *s1++ != *s2++)   return ( i+1);
               return 0;
    }
```

В теле функции обращение к элементам массивов – строк реализовано через разыменование указателей. За счет операций s1++ и s2++ изменяются начальные «настройки» указателей на массивы. Одновременно в той же функции к тем же массивам – параметрам выполняется обращение с помощью выражений *(s1 + d1) и *(s2 + d2) , при вычислении которых значения s1 и s2 не меняются.

Функция соединения строк

```
        void conc (  char * ps1 , char *ps2)
          {    int  i, d1;    //  d1 –  длина первой строки
                for ( d1= 0; *( ps1 + d1)  != '\0';  d1++ );
                   for ( i=0; *( ps2 + i) != '\0'; i++)
                            *( ps1 + d1+ i)  = *( ps2 + i );
               *( ps1 + d1 + i) = '\0';
          }
```

Результат - значение первого аргумента ps1 . Второй аргумент ps2 не изменяется. Естественно требование достаточности длины строки ps1 для приема результирующей строки.

Функция копирования содержимого строки

Так как в языке Си отсутствует операция присваивания для строк, поэтому приведем тексты разных вариантов копирования:

```
void copi ( char  *ps1 , char* ps2)
  {       // ps1 - оригинал , ps2 – копия
      int i;
          for ( i = 0;  ps2[i] != '\0'; i++)
                    ps1[i] = ps2[i] ;
          ps1[i] = '\0';
  }
       void copi ( char c1[], char c2[])
         {    int i =0;
                   do
                      {   c1 [i] = c2[i];
                      }
                 while ( c1[i++] )  != '\0');
          }
void copi ( char *ps1,  char *ps2)
  {                 // операция присваивания перенесена
                    //    в выражение  - условие
        int i =0;
              while ( ( ps1[i] = ps2[i] ) != '\0')
                        i ++;
   }
```

Вообще в приведенных примерах явное сравнение с '\0' необязательно, так как ненулевое значение при сравнении считается всегда ***истинным.*** Поэтому

```
void copi ( char  * ps1 ,  char  * ps2)
{    int i =0;
          while ( ps1[i] = ps2[i] )
                    i++;
}
```

И, наконец , наиболее короткий вариант:

```
 void copi ( char  *ps1, char  * ps2)
   {      while ( *ps1++  =  *ps2++);
   }
```

Пример использования функции copi ():

```
#include<stdio.h>
#include<conio.h>
  int main ( void)
{     char str[] = "My string";
      void copi ( char [], char []);     // прототип функции
       char s[80];
              copi (s, str);             // вызов функции
      printf ( "\n% s" , s);
          getch();
        return 0;
}
```

Рассмотрим еще один учебный пример использования строки в качестве параметра функции.

Опишем как один из вариантов функцию, которая политерно выводит на экран содержимое строки в прямом и в обратном порядках. В этом случае нужно рассмотреть: каким образом передавать в функцию строку, есть ли необходимость создать для этой строки локальный объект – строка с дальнейшей его обработкой.

Для передачи строки в функцию используем синтаксис указателей

```
void  Print_of_stroka ( const char * s)
```

В таком описании заголовка функции мы вложили следующие свойства:

– функция ничего не возвращает, о чем говорит тип **void;**

– параметром функции является s - указатель на строку. При вызове этой функции указателю s будет присвоен адрес первого байта строки – аргумента;

– модификатор **const** определяет работу функции со строкой – оригиналом только в режиме чтения, что гарантирует защиту строки – оригинала от возможных несанкционированных действий функции через ее параметр.

```
          void  Print_of_stroka ( const char * s)
{   char * term,  * p;
    short  len , i ;
          // найдем длину строки –оригинала, чтобы выделить
         // необходимую память в "куче" для локального объекта-
          // строки
            len = strlen (s);
```

```
        term = (char *)malloc( sizeof(char) * (len + 1));
    strcpy (term , s);          // скопировали строку – оригинал в
                                // локальный объект - строку
  // адресуем вспомогательный указатель p на объект – строку
                    p = term;
  // выведем в столбик содержимое объекта – строка в прямом порядке
            while ( *p )        // пока строка не закончилась
                printf (" % c \n ",  *p ++ );
```

/* выражение ***p++*** включает две операции: **'*'**- операция разыменования, которая применяется только к указателям, и возвращает содержимое области памяти, на которую ссылается указатель p; **'++'** - постфиксная инкрементная операция – выполняется над указателем p после операции '*' и реализует арифметику с указателями, которая заключается в изменении адреса указателя p на величину, кратную числу байт типа (**char** - как в нашем случае) того объекта, с которым связан указатель. Это означает сдвиг указателя p на единицу, т. е. на следующую справа литеру объекта – строка.

Работа цикла завершится как только указатель выйдет на литеру **'\0'** - признак конца строки.

Теперь, чтобы вывести содержимое объекта – строки в обратном порядке, нужно вернуть указатель p на последнюю значащую литеру строки. Для этого следует уменьшить его адрес на одну единицу, а затем повторить цикл, но применяя к указателю теперь уже постфиксную декрементную операцию */

```
            for ( i = len, p -- ;  i > 0 ; i --)
                printf ( "% c \n " , * p --);
}
```

Вопрос: Допустимо ли следующее условие цикла и что будет выведено в этом случае : **while** (p != term) ?

Приведем текст тестирующей программы.

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

```
int main ( void)              // головная функция возвращает 0
                               // при успешном завершении работы
{
        char  buf [80];        // резервируем память в стеке под
                               //  строку - оригинал
                clrscr();
                   puts ( "\n Введите строку:  \n");
                   gets ( buf );
puts ("\n Содержимое строки в прямом и обратном порядке : \n");
                  Print_of_stroka( buf );          // вызов функции
                    getch ();                      // задержка экрана
               return 0;
}
```

Резюме по строкам - параметрам функции:

В приведенных примерах функций, кроме последнего, происходит изменение объектов вызывающей программы, что стало возможным только потому, что параметрами являются указатели, оформленные либо с операцией индексирования, либо непосредственно в синтаксисе указателей. В теле же функции доступ к объектам программы осуществляется как с помощью индексированных переменных, так и с помощью операции разыменования указателей.

Компилятор языка Си, встречая, например, такое обращение к элементу массива str[i] , ***всегда заменяет*** его выражением ***(str + i)***, т. е. указателем str со смещением на i !. Такую замену, то есть использование указателей со смещениями вместо индексированных переменных, можно явно использовать в определениях функций. Иногда это делает нагляднее связь между функцией и вызывающей ее программой.

1.7. Указатели на функции

Из всего рассмотренного становится ясно, что роль указателей в языках C/C++ чрезвычайно велика. Она покажется вам ещё более значительной, когда мы познакомимся ещё с одним типом указателей – с *указателями на функции.* Указатели в языках C/C++ могут ссылаться не только на данные любого типа, но даже и на машинный код – на *машинный код функции.* Указатель на функцию определяется следующим образом:

тип результата (*имя_указателя) (параметры функции);

Как видим, здесь явно отсутствует *имя функции*, а вызов функции происходит косвенно, с помощью указателя на функцию.

После определения указателя на функцию ему можно присваивать адреса функций, но только тех, у которых *сигнатура*, т. е. список и тип параметров, а также тип возвращаемого значения в точности совпадают с таковыми, как в определении указателя на функцию.

Например:

```
int f1( int, double);
double f2( double, int);
int ( *pf) (int, double);
pf  = f1;          // корректно
pf  = f2;          // ошибка, сигнатуры разные.
```

Как видим, *имя функции* – это *константа*-указатель на *начало* её машинного кода.

Вызов функции по указателю осуществляется так же, как и обычно с помощью имени функции

```
pf ( int_value,  double_ value),
```

поскольку и определенный нами указатель на функцию и собственно имя функции имеют один и тот же тип.

Указатели на функции позволяют одинаковым кодом вызывать разные функции. Вот соответствующий пример:

```
#include<stdlib.h>
#include<conio.h>
        // прототипы функций
int   f1(int, double);
int   f2(int, double);

  int main(void)
{
      int  ( *pf )( int, double) = NULL;
      int  k;
          pf  = f1;
              k  = pf (2, 4.1);
          pf  = f2;
              k  = pf (2, 4.1);
       getch();
            return 0;
}
        //    определение f1 и f2
      int   f1(int a, double b)   { return  a * (int)b;}
      int   f2(int a, double b)   { return a + (int)b;}
```

Этот пример имеет целью показать лишь технику использования указателя на функцию. Более осмысленной, с точки зрения практической пользы, применение указателя на функцию будет дано в разделе 1.8 методического пособия.

1.7.1. Массивы указателей на функции

Массивы указателей на функции отличаются от массивов других объектов только форматом определения:

*тип (* имя_массива [размер])(спецификация_параметров);*

Например:

int (*par[4])(**char);**

где par – массив из четырех указателей на функцию, каждому из которых можно присвоить адрес функции, определенной в тексте программы. Напомним, что в качестве инициализирующего выражения при определении указателя на функцию должен использоваться адрес функции того же типа, что и тип, при определении указателя.

Массив в соответствии с синтаксисом языка является производным типом наряду с указателями и функциями. Массив функций создать нельзя, но можно определить ***массив указателей на функции***, что очень удобно для организации различного рода ветвлений с возвратом по результатам анализа некоторых условий. Для этого все допустимые ветви обработки оформляются в виде однотипных функций (с одинаковым типом возвращаемого значения и одинаковой спецификацией параметров). Затем определяется массив указателей, где каждому элементу присваивается адрес конкретной функции обработки. Далее формируются условия, на основе которых должна выбираться та или иная функция (ветвь) обработки. И, наконец, каждому условию ставится в соответствие конкретное значение индекса, по которому выполняется обращение к элементу массива указателей на функции

имя_массива [индекс] (список_фактичеких_параметров);

а вызов соответствующей функции обработки будет иметь вид

*(*имя_массива [индекс]) (список_фактических_параметров).*

Описанную схему использования массива указателей на функции удобно применять для организации **меню**, точнее, программ, которыми управляет пользователь с помощью меню.

Самую общую схему реализации такого подхода иллюстрирует следующая программа для обработки следующих файлов:

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>          // для функции exit()
```

```
                    //определения функций для обработки файлов
        void  sozd_f (char *name)
          {printf("\nДействия по созданию файла - % s",name);}
        void  del_f (char *name)
    {printf("\n,Действия по уничтожению файла - %s",name);}
        void  read_f(char *name)
        {printf("\nДействия по чтению файла - %s",name);}
        void  mod_f(char *name)
    {printf("\n Действия по модификации файла - %s",name);}

        void  close_f(char *name)
       {printf("\n Действия по закрытию файла - %s",name);
                          exit(0);
            }
typedef void (*MENU)(char*); // MENU – это тип указателей на функции
                                    //  типа  void(char*)
              //инициализация таблицы адресов функций меню
     MENU actions[5] = {sozd_f,  del_f,  read_f,  mod_f,  close_f};
       int main(void)
       {
          int n;                        // n-номер пункта меню
          char file_name[30];
                printf("\n 1 – Создание файла");
                     printf("\n 2 – Уничтожение файла");
                printf("\n 3 – Чтение файла");
                printf("\n 4 – Модификация файла");
                printf("\n 5 – Выход из программы");
           while(1)
        {
     while(1)  //цикл продолжается до ввода правильного номера
               {
                    printf("\n Введите номер пункта меню: ");
                    scanf("%d",&n);
                        if (n>=1 && n<=5)  break;
      printf("\n Ошибка в номере пункта меню. Повторить выбор!");
               }
                  if (n!=5)
       {       printf("\n Введите имя файла для обработки: ");
                          scanf("% s", file_name);
       }
```

```
                    //вызов функции по указателю на эту функцию
                            ( *actions [n-1] ) ( file_name );
        }
                getch();      return 0;
}
```

1.7.2. Указатели на функции как параметры

Указатели на функции – незаменимое средство языков C/C++, когда объектами обработки должны служить сами функции. Например, создавая процедуру для вычисления определенного интеграла задаваемой пользователем функции или для построения таблицы значений произвольной функции, нужно иметь возможность передавать в программы соответствующие функции. Удобнее всего организовать связь между функцией, реализующей какой либо метод обработки, и той функцией, для которой этот метод нужно применить, через аппарат формальных параметров, в число которых входят указатели на функции.

Указатели на функции, как формальные параметры, позволяют создавать функции, реализующие тот или иной метод обработки другой функции, которая заранее не определена.

Для иллюстрации особенностей применения указателя на функцию, в качестве параметра функции, рассмотрим задачу вычисления корня функции f(x) на интервале локализации [A;B] с точностью EPS. Численный метод оформим в виде следующего заголовка функции:

float root(указатель_на_функцию, **float** A, **float** B, **float** EPS);

Введем имя типа – "*указатель на функцию*":

typedef float (*pFunc)(**float**);

Теперь можно определить функцию для вычисления корня заданной функции с указателем типа pFunc. Прототип функции будет таким:

float root (pFunc F, **float** A, **float** B**, float** EPS);

Приведем тестовую программу для определения с помощью функции root() корня функции $y = x^2 - 1$ на отрезке [0,2]:

```
 #include <stdio.h>
#include <stdlib.h>            //для функции exit()
#include <conio.h>
          //определение имени типа указателя на функцию:
                  typedef float (*pFunc)(float);
         //определение функции для вычисления корня:
        float  root ( pFunc F, float A, float B, float EPS)
```

```
    {       float  x, y, c,  Fx, Fy, Fc;
                   x =A;   y = B;
        Fx = (*F)(x);          //значение функции на левой границе
        Fy = (*F)(y);          //значение функции на правой границе
                if (Fx* Fy > 0.0)
                    printf("\n Неверен отрезок локализации");
        exit(1);       //аварийное завершение программы
          }
            do
              {
                    c = (y-x) / 2;
        Fc = (*F)(c);    //значение функции в центре отрезка
                              if ( Fc* Fx > 0)
                              { x = c;  Fx = Fc; }
                              else
                              { y = c; Fy = Fc;}
              }
           while ( Fc != 0 && y-x > EPS);
                 return c;
      }
#include <math.h>
          //определение тестовой функции – y = x*x-1:
                   float f (float x)
                   {  return  x*x-1;  }
     int main (void)
      {
              float root (pFunc, float, float, float);
              float result;
                   result = root (f, 0.0, 2.0, 1e-5);
  printf("\n Корень тестовой функции = %f", result);
                   getch();
                   return 0;
      }
```

Результат выполнения программы:

Корень тестовой функции =1

1.7.3. Указатель на функцию как возвращаемое функцией значение

При организации меню в тех случаях, когда количество вариантов и соответствующее количество действий определяются не в точке исполнения, а в *«промежуточной»* функции, удобно возвращать из этой промежуточной функции адрес той конкретной функции, которая должна быть выполнена. Этот адрес можно возвращать в виде значения указателя на функцию.

Рассмотрим программу, демонстрирующую особенности такой организации меню. В программе определены три функции: первые две функции f1() и f2() с прототипом вида

int f(**void**);

т. е. пустой список параметров, а возвращаемое значение типа **int,** и третья функция menu() с прототипом

int (*menu(**void**))(**void**);

возвращающая значение указателя на функцию, которая в свою очередь имеет пустой список параметров и возвращает значение типа **int.**

При выполнении функции menu() пользователю дается возможность выбора из двух пунктов меню.

Пунктам меню соответствуют определенные выше функции, указатель на одну из которых является возвращаемым значением. При неверном выборе номера пункта возвращаемое значение становится равным **NULL**.

В основной программе определен указатель pFunc, который может принимать значения адресов функций f1(), f2(). В цикле выполняются обращения к функции menu(), и если результат равен **NULL**, то программа печатает «конец» и завершает выполнение. В противном случае вызов

int k = (*pFunc)();

обеспечивает исполнение той из функций f1() или f2(), адрес которой является значением указателя pFunc.

```
#include<stdio.h>
#include<conio.h>
  int f1(void)
      { printf("\n Первое действие :");      return 1; }
  int f2 ( void)
      { printf("\n Второе действие: ");   return 2;}
  int ( *menu (void)) (void)
{      int n;                          //n- номер пункта меню
          // массив указателей на функции  :
```

```
        int (*menu_items[] ) () = { f1, f2};
              printf("\n Выберите пункт меню(1 или 2): ");
              scanf("% d", &n);
        if ( n < 3  &&  n > 0)
              return  menu_items[ n-1];
          else
              return  NULL;
}
 int   main(void);
  {
 int ( *pFunc) (void);              //указатель на функцию
       int k;
     while (1)
  {    pFunc = menu ();             //обращение к меню
         if ( pFunc = = NULL)
              {   printf("\n Конец!");
                    return;
               }
          // вызов выбранной функции
             k = ( *pFunc) ();
               printf("\t k = % d", k);
  }
           getch();     return 0;
  }
```

Результат выполнения программы :

```
Выберите пункт меню (1 или 2):2
Второе действие: k=2
Выберите пункт  меню (1 или 2):1
Первое  действие: k=1
Выберите пункт меню (1 или 2):5
Конец!
```

Наглядность определения массива указателей можно обеспечить с помощью вспомогательных имен, вводимых спецификатором **typedef.** Например, определение

```
int ( *menu_items[] ) () = { f1, f2};
```

можно представить так:

```
typedef  int ( *Menu_action) (void);
 Menu_action   menu_items [] = {f1, f2};
```

Здесь **typedef** вводит обозначение имени Menu_action - как типа *«указатель на функцию с пустым списком параметров, которая возвращает значение типа* ***int****»*.

1.8. Полезные практические примеры программирования с указателями

В предыдущих разделах мы с вами достаточно основательно изучили указательный тип, синтаксис определений объектов такого типа, не упустив при этом формальные аспекты использования объектов, а также операции над ними. Теперь настала пора хоть немного коснуться некоторых практически важных задач. Сочетание слов «хоть немного» означает, что спектр применения указательного типа в информационных технологиях огромен, поэтому просто нереально в этом издании коснуться всех тем, да это и не входило в нашу задачу. Главная наша цель заключалась в том, чтобы показать всем желающим, что указатель – *это просто*, поймите, а дальше все сложится хорошо!

В качестве полезных примеров ограничимся тремя задачами:

- Выделение динамической памяти фиксированными порциями (для уменьшения накладных расходов, связанных со слишком частым выделением новых порций памяти).
- Динамическое создание двумерных числовых массивов произвольного размера (неизвестного во время компиляции).
- Сортировка массива (пример обобщенного программирования).

Начнем с первой задачи.

При изучении массивов мы отмечали, что при отсутствии информации о размере массива, приходится выделять память во время выполнения программы. Но и такая гибкая схема иногда не срабатывает, если информация о количестве элементов в наборе поступает не сразу целиком, а постепенно – порциями, а хранить их надо компактно, как в обычных массивах. Поэтому будем выделять память фиксированными крупными порциями и размещать непрерывно поступающие данные внутри выделенного большого буфера. При каждом переполнении этого буфера нужно снова выделять буфер памяти, что будет происходить реже, с размером на некоторую фиксированную величину больше предыдущего буфера. Можно говорить о *гранулированном методе* динамического выделения памяти в отличие от прямоугольного мелкозернистого точного выделения памяти под текущую порцию данных.

В итоге программа становится более эффективной, так как удается избавиться от лишней работы. Вот соответствующий пример:

```
#include <stdio.h>
#include ><malloc.h
#include <conio.h>
                // глобальные переменные
int BigGrain = 12;
int CurrLimit = 0;
int CurrSize = 0;
bool FirstTime = true;
bool PlaceInBuffer (int **, int);
void FreeBuffer (int *);
     int main (void)
{ int *pBuf = NULL;
     char answer;
     int value;
     bool ret;
    for ( ; ; )
{printf("Желаете вводить новые целые значения?");
while (scanf ("%c", &answer), (answer! = 'n' && answer! = 'y'));
if ( answer == 'n') break;
printf ("\n Введите очередное значение:");
scanf ("%d", &value);
ret = PlaceInBuffer (&pBuf, value);
if (ret == false) break;
}
             FreeBuffer (pBuf);
          getch ();
               return 0;
}
  bool PlaceInBuffer (int **p, int val)
{ static int *pCur = NULL;
if (FirstTime)
{ *p = (int *)malloc(sizeof(int)*BigGrain);
           if (p == NULL)
                    return false;
CurrLimit = BigGrain;
pCur = *p;
FirstTime = false;
}
if (CurrSize *sizeof(int) == CurrLimit)
```

```
    // буфер заполнен
    { CurrLimit += BigGrain;
    realloc((void*)(*p),size of(int)*CurrLimit);
                if (*p== NULL)
                    return  false;
    // не удается выделить память
             pCur = *p + CurrSize;
    }
               *pCur ++ = val;
                 CurrSiz += 1;
                     return   true;
    }
         void FreeBuffer(int *q)
           {if (q!= NULL) free (q);}
```

Здесь мы применили функцию **realloc()** – повторное выделение памяти по фиксированному указателю с *автоматическим освобождением* ранее выделенной памяти, так что отпадает необходимость в явном вызове функции ***free()***.

Теперь перейдем ко второй практической задаче – задаче о динамическом создании целого двумерного массива произвольных размеров. Если неизвестным является лишь размер по первому измерению (это может быть значение переменной), а по второму размер известен во время компиляции, тогда все просто:

```
int (*A)[100] = new int[N][100];
```

Таким образом, динамически создается целый двумерный массив N*100, элементы которого адресуются обычным выражением A[i][j], где i и j – индексы (номера) элемента массива. Имя, т. е. идентификатор массива A, имеет тип **int**(*)[100] .

Если же оба размера неизвестны во время компиляции, то требуется воспользоваться набором из двух указателей – один типа **int****, а другой типа **int***. Вот решение проблемы:

```
    #include <stdio.h>
    #include <malloc.h>
    #include <conio.h>
    int main (void)
    { int i, j, nRow, nCol;
    int **A = NULL;
    int *p= NULL;
printf ("Введите количество строк и столбцов:");
    scanf ("%d % d", &nRow, &nCol);
```

```
      // буферы для указателей на строки матрицы
     //      и все ее элементы
    A=(int **)malloc(sizeof(int *) * nRow);
    p=(int *)malloc(sizeof(int) * (nRow * nCol));
      // инициализация указателей на строки
    for (i=0; i < nRow; i++) *(A+i) = p+i*nCol;
       // инициализация матрицы
         for (i=0; i<nRow; i++)
         for (j=0; j<nCol; j++)    A[i][j] = i+j;
         for (i=0; i<nRow; i++)
            for (j=0; j<nCol; j++)
       printf("A[%d][%d]=%d\n", i, j, A[i][j]);
            //освобождение памяти
            if (p!=NULL)     free(p);
             if (A!=NULL) free (A);
                 getch();
                 return 0;
    }
```

Ну и, наконец, рассмотрим популярную и важную на практике задачу о сортировке числовых массивов (по возрастанию или по убыванию). Алгоритм сортировки выберем «пузырьковый» (по-английски – bubble sort), который позволяет делать сортировку не более чем за N проходов (N - количество элементов в массиве).

```
    #include<stdio.h>
    #include<stdlib.h>
    #include<memory.h>
    #include<conio.h>
      #define N 10
            //прототипы функций
Void BubbleSort(void*,int,int,int((*)(void*,void*)));
    int MyCompare (const void*, const void*);
    void MySwap (void*, void*, int);
    int main (void)
    { int A[N];
    for (int i=0; i<N; i++) A[i]= rand();
    BubbleSort ((void*)A,N,sizeof(int),MyCompare);
          getch();
            return 0;
    }
    void      BubbleSort(void*p,int      ArrSize,int
ElemSize, int CompFun (void*, void*))
```

```
    { for(int i=0; i<ArrSize; i++)
      for(int j=1, j<ArrSize - i; j++)
    { void *p1= ((char*)p + (j-1) * ElemSize);
           void *p2= ((char*)p + j * ElemSize);
    if(CompFun(p1,p2)>0) MySwap(p1,p2,ElemSize);
        }
    }
int MyCompare(const void* pElem1, const void* pElem2)
{ return (int)(*(int*) pElem1 - *(int*) pElem2); }

    void MySwap (void *p1, void *p2, int Size)
    { void *p = malloc (sizeof (char) * Size);
                memcpy (p, p1, Size);
                 memcpy (p1, p2, Size);
                   memcpy (p2, p, Size);
          free (p);
      }
```

Данная программа спроектирована в духе так называемого «обобщенного программирования», т. е. в достаточно универсальной манере. Эта универсальность обеспечивается применением указателей типа **void***, что позволяет создавать повторно используемые (с минимальной переделкой кода, а еще лучше без оной) программные компоненты.

В результате мы имеем абсолютно универсальные функции BubbleSort () и MySwap (), потому что они без переделки годятся для сортировки массивов разных типов. Единственной функцией, «чувствующей» конкретный тип данных сортируемого массива, является функция MyCompare(), которая сравнивает между собой два элемента массива по величине. Именно эту функцию и нужно приспосабливать под конкретный тип данных. Универсальность реализации функции BubbleSort () обеспечивается еще и за счет применения указателя на функцию сравнения элементов массива. Для реализации сортировки по убыванию достаточно лишь изменить знак у возвращаемого значения в функции MyCompare ():

```
    int MyCompare (void * pElem1, void * pElem2)
 {return - (int)(*(int*) pElem1 - *(int*) pElem2);}
```

2. УКАЗАТЕЛЬНЫЙ ТИП И ДИНАМИЧЕСКИЕ ОБЪЕКТЫ СЛОЖНОЙ СТРУКТУРЫ

2.1. Необходимые сведения

В первой части методического пособия достаточно подробно рассмотрены основополагающие вопросы определения и использования указательного типа, что дало возможность овладеть основами программирования объектов этого типа. Как известно, для обеспечения простоты и надежности разрабатываемого алгоритма решения задачи важное значение имеет выбор наиболее подходящих структур данных, которые используются в процессе решения.

Набор типов языков С/С++ настолько универсален и гибок, что на его базе можно вводить в употребление и использовать самые различные структуры данных, которые в явном виде в языке и не фигурируют. Особенно большие возможности в этом отношении дает указательный тип в сочетании с массивами и комбинированными типами данных.

Во второй части пособия мы рассмотрим приемы создания *динамических структур* произвольной природы, отражающие объекты, в терминах которых наиболее просто формулируется алгоритм решения задачи.

Такими *динамическими* структурами, широко используемыми в практике программирования, являются стеки, очереди, различного вида списочные структуры, деревья и т. д.

Динамический объект имеет структуру, состоящую из, по крайней мере, двух полей: одно поле – содержит атрибуты, другое – является указателем, значение которого адресует связанный с данным объектом другой динамический объект. Таким образом, динамический объект под одним именем содержит набор разнородных данных и определяется как структурный тип с ключевым словом ***struct***. Теперь из таких динамических объектов можно конструировать сложные динамические объекты: стеки, очереди, списки, таблицы и т. д.

Мы не будем подробно рассматривать структурный тип, а только коснемся тех вопросов, которые будут полезны для понимания представления и способов обработки сложных динамических объектов.

Итак, приведем пример определения структурного типа *tagName*:

```
struct tagName
  {   int k;
      char str [6];
      double d;
      int ar[2][3];
  };
```

Переменные этого типа в C++ объявляются следующим образом:

```
tagName  st1,  st2;
```

а в языке Си обязательно нужно повторять ключевое слово **struct**:

```
struct  tagName  st1, st2;
```

Использование ключевого слова **typedef** упрощает в языке Си синтаксис объявления переменных структурного типа:

```
typedef  struct  tagName
  {          int k;
             char  str [6];
             double d;
             int  ar[2][3];
  }  STRUCTNAME;
```

после чего можно определять структурные переменные следующим образом:

```
STRUCTNAME  st1, st2;
```

Структурные переменные можно передавать в функции в качестве фактических параметров «*по значению*», т. е. копировать. Другой способ – это определить указатель типа tagName и инициализировать его переменной st1:

```
tagName  *pst  = & st1;
```

Операция «стрелка» реализует доступ через указатель к полям структурной переменной, например: pst → k.

С таким же успехом можно использовать выражение с операцией «точка»:

```
( * pst ) . k.
```

В этом примере мы видим действие операции разыменования '*' над указателем pst , которая "раскрывает" содержимое структуры, а операция '.' - обеспечивает доступ к переменной к. Наличие круглых скобок в выражении обязательно, так как приоритет операции '.' выше , нежели операции '*'.

К полям структурных переменных можно еще "подбираться", используя указатели *иных типов*, чем указатели на структурный тип данных. Действительно, определим указатель типа **char ***, тогда операция инкремента означает увеличение его значения (то есть адреса памяти) всего лишь на единицу. Значит с помощью такого указателя можно «пробежать» любой диапазон памяти подряд, а с проблемой совместимости всегда можно справиться с помощью явного приведения типов данных

```
char  *p = (char*) &st1;
```

Передавая во внешнюю функцию указатель на структурную переменную, мы обеспечиваем для этой функции возможность изменить

саму эту переменную, а не ее копию. В определении тела структуры возможно применение полей, имеющих тип указателя на определяемую структуру:

```
struct tagName
 {          …
      tagName *p;
 };
```

2.2. Односвязный список

Односвязный список – гибкая динамическая структура общего назначения.

Для создания односвязных списков применяются ранее полученные знания об указателях и структурах языков С/С++.

Односвязные списки, как и массивы, предназначены для хранения в них набора значений. В отличие от массивов, односвязные списки принципиально не требуют никаких знаний о количестве информационных элементов, т. е. они являются *динамическими по своей природе*.

Односвязные списки состоят из *«узлов»* (по английский - *nodes*), в каждом из которых хранится информационное значение данного узла, и указатель на следующий узел. Очевидно, что должен существовать самый первый узел, стартуя с которого, можно пройти всю цепочку узлов. Таким образом, можно подобраться к любому узлу, чтобы извлечь хранящееся в нем информационное значение. Последний узел характерен нулевым значением своего “указательного “ поля.

Создать односвязный список – значит создать самый первый узел с некоторым «полезным» значением, например, информацией о количестве узлов списка в его информационном поле и с нулевым «указательным» значением (**NULL**).

Затем возможен постепенный динамический процесс прикрепления все новых узлов к концу существующей цепочки, каждый из которых сначала создается динамически в памяти компьютера, после чего соответствующим образом прописываются значения информационных и указательных полей вновь созданного узла и последнего, из ранее прикрепленных к односвязному списку.

Естественно, нужно иметь возможность удалять созданный односвязный список узлов, освобождая зарезервированную под это память компьютера.

Теперь представим пример программы, реализующей все обязательные работы по конструированию односвязного списка, но пока без обработки списка.

Узел списка представляет из себя переменную структурного типа

```
struct Node
{  int val;                    //информационное значение
Node  *pNode;                  //указатель на очередной узел
};
```

Добавление новых узлов к существующему списку будет выполнять функция Add(), вывод содержимого списка – функция Print(), а удаление списка – функция Remove ( ):

```
#include <stdio.h>
#include <conio.h>
#include <malloc.h>
        struct Node
          {    int val;
               Node * pNode;
          };
        // прототипы функций
   void Add (Node  **ppStart, int newVal);
   void Print (Node  * pStart);
   void Remove (Node  **ppStart);
      int main (void)
      {          int ret;
         Node  *pN = NULL;
               Add (&pN, 5);
               Add ( &pN, 6);
               Add ( &pN, 7);
                    Print ( pN);
                    Remove (& pN);
                  getch ( );
                  return 0;
        }
  void  Add (Node  **ppStart, int newVal)
   { Node  *p = *ppStart;
if (p==NULL)       // создаем первый элемент списка
          { *ppStart = (Node*)malloc (sizeof(Node));
               ( *ppStart) → val = newVal;
               (*ppStart) → pNode  = NULL;
               return;
          }
     while ( p != NULL)
          {  if (p →  pNode == NULL)   break;
                    p = p → pNode;
          }
```

```
                p → pNode = (Node*)malloc (sizeof(Node));
                (p → pNode) → val =  newVal;
                (p → pNode) → pNode  = NULL;
        }
            void Print ( Node  *pStart)
                {Node  *p = pStart;
                    while (p != NULL)
                      { printf ("Элемент = %d \n", p → val );
                        p = p → pNode;
                      }
                    prinrf ("\n Список полностью выведен!  \n");
                }
    void  Remove (Node  **ppStart)
      {
          Node *p;
                p = *ppStart;
           if ( *ppStart == NULL)          // список пуст
                return;
 while ( p→ pNode != NULL)  // не последний элемент списка
        {
if (p →pNode)→pNode==NULL)   // два последних элемента
                                  // адресуются  указателем p.
            {    free (p → pNode);
                    p → pNode = NULL;
                      p = *ppStart;
                        continue;
             }
                  else
                    p = p → pNode;
            }
 // уничтожаем последний элемент, и  указателю
//  на начало списка присваиваем    NULL
                    free (* ppStart);
                    *ppStart = NULL;
      }
```

В данной программе передача «указателя на голову списка» в функции Add() и Remove() производится косвенно, т. е. передается *его адрес* с той целью, чтобы функция могла изменять сам этот указатель, а не его копию в стеке. Например, таким образом удается после вызова

функции Remove (& pN) получить нулевое значение этого указателя, что сигнализирует явным образом о том, что список пуст.

Заметим также, что вместо пары функций **malloc**() и **free**(), для динамической работы с памятью с тем же практическим успехом можно использовать пару операций **new/delete** языка C++.

2.3. Представление строки в виде односвязного списка

Кроме привычного, так называемого векторного представления строк, в котором понятие «следующий элемент» жестко связано с местом расположения этого элемента по отношению к предыдущему, существует представление строк в виде списков (односвязных, двунаправленных и кольцевых). Способ связи элементов в одну цепочку реализуется через механизм указателей.

Для иллюстрации техники работы со строками, представленными в виде цепочки, рассмотрим следующий пример:

Подсчитать число вхождений символа «A» в заданное непустое слово, состоящее из литер, не содержащих точки, и завершающееся точкой.

Мы не ставим целью получение эффективной программы, поскольку хотим просто проиллюстрировать технику работы с цепочками. Для этого примем следующую схему:

1. Ввести исходное слово и представить его в виде односвязного списка.
2. Определить число вхождений заданного символа в слово.
3. Вывести результаты на печать.

В программе придется иметь дело с динамическим объектом, поэтому введем в употребление следующий структурный тип:

```
struct  Node
{     char lit;
      Node  *next;
};
```

Для ссылки на список как целое, а точнее на его первый узел, введем в употребление соответствующий указатель

```
Node   *first;
```

В процессе формирования звеньев для присоединения к последнему, из уже сформированных, очередного звена, потребуется вспомогательный указатель. Введем его в употребление с помощью описания

```
Node *p;
```

Указатель p будем использовать для организации перехода по звеньям списка с помощью операции присваивания

```
p = p → next;
```

Еще одно примечание, связанное с организацией таких динамических структур. Речь идет о так называемом заглавном звене списка: нужен ли он или можно обходиться без него.

Если не вводить заглавное звено, то формирование первого звена приходится делать нестандартным способом – иначе, чем формирование остальных звеньев. И вообще при таком представлении цепочки обработка первого звена будет отличаться от обработки других звеньев.

Приведем текст программы без заглавного звена:

```
#include <stdio.h>
#include <conio.h>
#include <malloc.h>
#define sum 'A'
  struct  Node
  {
        char  lit;
          Node *next;
  };

      int main (void)
{  Node *first, *p;          //first-указатель на первый узел списка
                             //  p  -  рабочий указатель
          char ch;
      int  к;        //  к– счетчик искомых символов в строке
                          // формирование первого узла
          printf (" \n Введите символ:  ");
          scanf ("% c", &ch);
              first  = (Node*) malloc (sizeof (Node));
                 first → lit  =  ch;
                 first → next  = NULL;
// подготовка к циклу  формирования остальных узлов
                      p = first;
// цикл обработки последовательных литер строки
          while (ch!='.')
            {   printf ("\n Введите очередной символ: ");
                scanf ("% c", &ch);
        p =  p → next = (Node*) malloc (sizeof (Node));
                p →  lit = ch;
                p → next = NULL;
            }
```

```
/* Исходное слово, представлено в виде односвязного списка. Теперь
распечатаем содержимое звеньев списка, т. е. введенную строку */
                    p = first;
                        putchar('\n');
                  while (p)
                    {  printf("% c", p → lit);
                         p = p → next;
                    }

          // подсчет числа вхождений символа sum в слово
                     к=0;  p=first;
                   while (p)
                     { if (p → lit = = sum) k++;
                             p = p→ next;
                      }
                     // печать результата

    printf("\n Символ % с входит в строку - %d раз", sum, k);
                     getch( );
                     return 0;
              }
```

Теперь включим в список заглавное, «нулевое» звено, в поле *next* которого будет содержаться ссылка на первое звено, а поле lit можно использовать для хранения некоторой дополнительной информации о строке

```
#include <stdio.h>
#include<conio.h>
#include<malloc.h>
#define sym 'A'
     struct Node
       {    char lit;
              Node * next;
       };
int main (void)
{
        Node  *first, *p;
            char ch;
           int к;
         // формирование заглавного звена
     first = (Node*)malloc(sizeof(Node));
           p = first;
```

```
                    p → next = NULL;
                printf("\n  Введите первую букву строки:");
                scanf("% c", &ch);
//  циклическая обработка символов, если слово не пусто
                while (ch !='.')
                {
                    p = p → next = (Node*)malloc(sizeof(Node));
                      p → lit = ch;
                      p → next = NULL;
            printf ("\n Введите очередную букву строки:");
                    scanf("%c, &ch");
                }
```

/*Обратите внимание, что в первом варианте решения задачи в последнем узле списка формировалась точка, во втором же – она отсутствует */

```
                // вывод содержимого списка
              putchar ('\n');
    p = first → next;   // адресуем указатель p на первый узел
              while ( p)
                  {   printf ("% c", p → lit);
                      p = p -> next;
                  }
//  подсчет числа вхождений в слово заданного символа
                  k = 0;        p = first → next;
                  while (p)
                      {if ( p → lit == sym) k++;
                              p = p → next;
                      }
                          //  печать результата
printf("\n Символ % c  входит в строку % d раз", sym, к);
                      getch ( );
                  return 0;
          }
```

2.4. Реализация операций над строками – цепочками

Рассмотрим реализацию основных операций над строками в случае их представления в виде односвязного списка. Везде далее будем считать, что любая (в том числе и пустая) строка имеет заглавное звено (у пустой строки поле next заглавного звена содержит **NULL**).

Поиск заданного элемента в строке

Алгоритм выполнения этой операции зададим в виде описания функции, которая определяет первое вхождение заданного элемента в указанную строку и возвращает ссылку (адрес) на соответствующее звено списка. Пример описания такой функции может иметь вид

```
Node *search (char ch, Node *beg)
{ Node  *q = beg;
//  q - указатель на структуру, как локальный объект  функции,
//    его адресуем  значением указателя на начало списка,
//   то есть значением указателя first
     Node  *result = NULL;
          q = q → next;
            while (q)                          // пока список  не пуст
              { if (q → lit == ch)
                        { result = q;
                                  break;
                        }
                   q = q → next;
              }
        return  result;
  }
```

Удаление из строки заданного элемента

Здесь, прежде всего, необходимо решить вопрос о том, как следует задавать элемент строки, подлежащий удалению. Наиболее удобный вариант – исключаемый элемент задать с помощью указателя на то звено, за которым следует этот элемент. Этот указатель мы найдем с помощью функции поиска search (). Механизм исключения узла в списке достаточно простой: достаточно изменить значение указателя у предшествующего ему элемента, указатель **former** (предшествующий), который мы найдем функцией search(), причем в качестве нового значения указателя у этого элемента надо принять значение указателя, содержащееся в исключаемом элементе.

Пример описания функции исключения элемента из списка может иметь такой вид:

```
void  dispose (Node  *former )
{ if (former → next != NULL)       //    проверка, не ссылается ли
                                   //    указатель на последний узел списка?
        {     Node *q;
q = former → next;  // адресовали удаляемый узел указателем q
```

```
                    //  для последующего его уничтожения
                        former → next = q → next;
                    free(q);  // освобождаем память
            }
    }
```

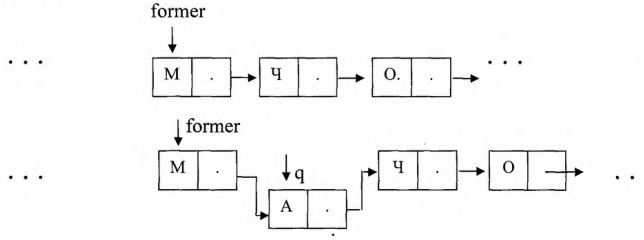
Вставка в список заданного элемента

Вставка в список нового узла основывается на объединении отдельных звеньев в единую цепочку с помощью указателей.

Механизм вставки нового звена после заданного следующий:

1. Создать новый динамический объект, которым будет представлено вставляемое звено.
2. В поле *lit* этого объекта занести заданный символ.
3. Поле *next* этого объекта адресовать значением поля next того звена, за которым должно следовать вставляемое звено.
4. Поле *next* того звена, за которым должно следовать вставляемое звено, адресовать вставляемым звеном.

Схематично на фрагменте списка это будет выглядеть следующим образом:



Таким образом, пример описания функции вставки в строку заданного элемента может выглядеть следующим образом:

```
void  insert (char ch, Node *former)
{
        Node *q = (Node *)malloc (sizeof (Node));
          q → lit = ch;
              q → next = former → next;
                    former → next = q;
}
```

Напомним, что указатель former, который адресует предшествующий вставляемому узел, мы находим с помощью функции search (). Мы не стали переусложнять учебный пример и включать функции вставки узла в произвольную внутреннюю позицию, и удаление отдельного узла списка.

Мы рассмотрели эти функции отдельно. Разберитесь самостоятельно в учебном примере, а потом включите недостающие функции в основную программу.

Рассмотрим программу создания списка и вывода его содержимого на консоль.

```
#include<stdio.h>
#include<conio.h>
 struct List
        {        int dat;
                 List  *next;
        };
 int main (void)
 {        int i
   List *p, *heap = new List;              // начало списка
            for ( p = heap, i=0; i < 10; i++)
                 {   p → dat = i;
                     // порождаем следующий элемент списка
                     p = p → next = new List;
                 }
        for ( p = heap, i=0; i<10; i++)
        {
            printf ("%4d", p → dat );
             p = p → next;
        }
                 getch ( );
                     return 0;
  }
```

Здесь мы описали два указателя: heap – для указания начала списка, p – для передвижения по списку; и простую переменную, как счетчик цикла. В отличие от массива, список «разбросан» по памяти в куче, что не имеет никакого значения, так как мы передвигаемся по списку, используя сохраненные в самом списке адреса

$$p = p \rightarrow next.$$

Если вопросы разработки сложных структур данных вызывают у вас интерес, то можно порекомендовать вам ознакомиться с книгами [6,7].

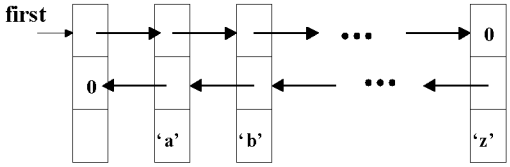
2.5. Двунаправленные списки

Использование однонаправленных списков при решении ряда задач может вызвать определенные трудности. Это в первую очередь

возможность перемещения только в одну сторону, что ведет за собой усложнение алгоритма обработки таких структур. Обойти это неудобство можно путем добавления в звено еще одного поля, которое будет адресовать предыдущее звено. В этом случае структура звена будет иметь вид

```
struct Node
{      char lit;       // информационный элемент
Node *pred;            // указатель на предыдущее звено
Node *next;            // указатель на следующее звено
};
```

Динамическая структура, состоящая из звеньев такого типа, называется двунаправленным списком, который схематично можно изобразить следующим образом (как список с заглавным звеном):



Если в качестве значения поля *next* последнего звена принять адрес заглавного звена, а в качестве значения поля *pred* заглавного звена принять адрес последнего, тогда получим двунаправленный ***кольцевой*** список.

Над списками определены те же основные операции, что и над строками:

- поиск элемента в списке;
- вставка заданного элемента в указанное место списка;
- удаление из списка заданного элемента.

Рассмотрим реализацию этих операций применительно к двунаправленному кольцевому списку с заглавным звеном.

Вставка элемента

Действия по вставке элемента зададим в виде описания функции с именем insert(), которая имеет два параметра: один из них представляет вставляемый элемент, а другой – указатель former (предшествующий), значением которого является адрес звена списка, после которого необходимо вставить новый элемент

```
void  insert (char ch, Node  *former)
{
        Node  *q = (Node*) malloc (sizeof (Node));
           q  -> lit = ch;
         q → next = former → next;
              former → next = q;
                 q → pred = q → next → pred;
                       q → next → pred = q;
}
```

Для вставки нового элемента *в начало списка* в качестве второго *фактического параметра* (аргумента) функции *insert()* надо задать указатель на заглавное звено списка, то есть указатель *first* на этот список.

Удаление элемента

Для исключения из списка какого-либо элемента достаточно изменить адрес в поле next у предшествующего ему звена и адрес в поле pred у звена, следующего за исключаемым. Возможность двигаться по списку в любом направлении позволяет задавать исключаемое звено через указатель на само это звено. Естественно необходимо удалять звено с помощью функции **free().**

Функция удаления имеет единственный параметр – указатель на удаляемое звено

```
void dispose (Node *pDel)
{
// полю  next  предшествующего исключаемому звену присвоим
// значение поля  next  следующего за исключаемым
            pDel → pred → next = pDel → next;
// полю  pred  последующего звена присвоим адрес поля  pred
   // удаляемого звена
            pDel → next → pred = pDel → pred;
               free (pDel);
}
```

Поиск элемента в двунаправленном кольцевом списке

Процедура поиска заданного элемента в списке практически не отличается от аналогичной процедуры для строки – цепочки. Если двунаправленный список кольцевой, то надо учитывать ту особенность, что у такого списка формально нет последнего элемента, поскольку фактический последний элемент также имеет ссылку на «следующий» элемент, каковым является заглавное или первое звено

списка (в зависимости от способа его реализации). Для принятого нами способа, т. е. с заглавным звеном функция поиска заданного в явном виде элемента будет использовать два формальных параметра: указатель на заглавное звено и значение искомого элемента.

Результатом функции является указатель на первое по порядку звено, содержащее заданный элемент.

Такую функцию можно описать следующим образом:

```
Node *Search(char ch, Node *beg)
{   Node *q = beq;   // локальный указатель адресуем указателем
                     //  на заглавный элемент списка
    Node *result = NULL;
q = q → next; // теперь q - имеет адрес первого элемента списка
        while (q)
          {if (q → lit == ch)
              {       result = q;
                      break; // выходим из цикла  в случае успеха
              }
            q  = q -> next;
           }
              return  result;
 }
```

Еще раз обратим внимание на тот факт, что единообразие обработки всех звеньев (включая первое и последнее звенья списка) достигается за счет введения в употребление заглавного звена. Все приведенные выше описания функций используют факт наличия в списке заглавного звена.

Теперь рассмотрим пример использования списков при решении конкретной задачи.

Пример. Задана строка литер, признаком конца которой является первая по порядку точка. Требуется удалить из строки все цифры, непосредственно предшествующие каждому вхождению буквы t, и напечатать результирующую строку.

Предлагается следующий метод решения задачи. Сначала создадим двунаправленный кольцевой список, элементами которого будут литеры заданной во внешнем текстовом файле строки, а затем обработаем элементы этой структуры в соответствии с требованиями задачи, т. е. исключим все звенья, содержащие цифры, непосредственно предшествующие каждому звену с буквой t. Обработку элементов будем осуществлять при их последовательном переборе от первого звена к последнему.

```
#include <stdio.h>
#include <conio.h>
#include <malloc.h>
        struct Node
          { char lit;
            Node  *next,  *pred;
          };
                  //  прототипы функций
void In List (char, Node *);          // включение звена в список
void  From  List (Node *);            // исключение звена из списка
              int main (void)
{     char ch;
                    Node *p;
Node *ring;       // ring указатель на заглавное звено списка
                // создание двунаправленного кольцевого списка
                // создание заглавного звена списочной структуры
                  p = (Node *)malloc(sizeof (Node));
                      p → next = p;
                        p → pred = p;
                          p → lit ='a';
                                  ring = p;
                    printf (" \n Введите первую литеру строки: ");
                    scanf ("%c", &ch);
//  последовательное чтение литер строки и включение их в список ring
                    while ( ch != '.')
                     {       In  List (ch, ring → pred);
                             printf ("\n Введите очередную литеру: ");
                             scanf("%c", &ch);
                     }
                  // установка указателя p на первое звено
                    p = ring → next;
                  //  последовательная обработка элементов списка
                    while (p != ring)
                    {       // поиск очередного звена списка с буквой 't'
                            while (p → lit != 't' && p != ring)
                                    p = p → next;
if (p != ring)          //  удаление предшествующих звеньев цифрами
              {  while (p → pred → lit >= '0'&& p → pred → lit <= '9')
                      From List (p → pred);
                        p = p → next;
              }
}
```

```
            // печать результата
        p = ring → next;
              putchar ('\n');
                  while (p != ring)
                   {
                        printf("%c", p → lit);
                          p = p → next;
                           putchar('\n');
                   }
                return o;
}          // Конец main()
          // определение функции  InList()
    void  InList ( char ch, Node * former)
     {    Node * q = ( Node *) malloc ( sizeof ( Node));
                q -> lit = ch;
                  q -> next = former -> next;
                     former -> next = q;
                        q - > pred = q -> next -> pred;
                          q -> next -> pred = q;
     }
        //  определение функции  FromList()
          void  FromList ( Node * pDel)
         {      pDel -> pred -> next = pDel -> next;
                pDel -> next -> pred = pDel -> pred;
                      free( pDel);
         }
```

Реализованный здесь алгоритм существенно использует тот факт, что в заглавном звене в поле lit находится литера – буква. Действительно, возможен случай, когда строка представляет собой, например, такую последовательность литер: 123t …s45; т. е. начинается и заканчивается любым числом литер- цифр. Если в этом случае в заглавном звене нет литеры – буквы, алгоритм не даст правильного результата. Анализ причины этого оставляем читателю.

2.6. Очереди и стеки

В программировании имеется структура данных, которая называется *очередь*. Эта структура данных используется при моделировании задач массового обслуживания при заданном законе поступления заказов и дисциплине их обслуживания.

Очередь по своему существу является сугубо динамическим объектом – с течением времени и длина очереди, и набор образующих её элементов изменяются.

Над очередью определены две операции: занесение элемента (заказа на обслуживание) в очередь и выбор элемента из очереди (для его обслуживания). При этом выбранный элемент, естественно, исключается из очереди. В очереди доступны две позиции: её начало (из этой позиции выбирается элемент из очереди) и конец (в эту позицию помещается заносимый в очередь элемент).

Существуют две дисциплины обслуживания:

FIFO (first in-first out, т. е. первый в очередь – первый из очереди);

LIFO (last in-first out, т. е. последний в очередь – первый из очереди);

В программировании чаще используется вторая дисциплина, а очереди такого вида принято называть ***стеком*** *(магазином).* Стек – одна из наиболее употребляемых структур данных, которая оказывается весьма удобной при решении различных задач.

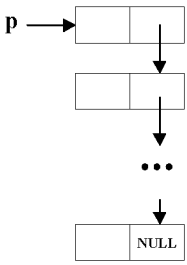
Вершина стека – это позиция, в которой находится последний по времени поступления в стек элемент. Выбрать элемент можно ***только из вершины стека,*** при этом такой элемент исключается из стека, а в его вершине оказывается элемент, который был занесён в стек перед выбранным из него элементом.

Наиболее подходящей структурой данных языков C/C++, обеспечивающей наиболее быстрое выполнение основных операций над стеком, является динамическая цепочка звеньев. Будем исходить из того, что вершиной стека является первое звено цепочки. Значением указателя, который представляет в программе стек как единый объект, является адрес на вершину стека и только. Как обычно в случае цепочки, каждое звено стека содержит ссылку на следующее звено цепочки, причем «дно» стека, т. е. элемент, занесенный в стек раньше всех, имеет значение ссылки, равное **NULL**.

Тип элемента стека зададим следующим образом:

```
struct Stack
  {
        int val;
        Stack * next;
  };
```

Схематично стек p можно изобразить следующим образом:



Если стек пуст, то значением указателя p является ссылка **NULL**. К началу использования стека его необходимо сделать пустым, т. е.

p = **NULL**;

Рассмотрим реализацию основных операций над стеком.

Занесение элемента в стек

Поскольку при решении задачи может использоваться несколько различных стеков, например, будем иметь такие программные объекты:

Stack *st1, *st2;

и поэтому функция занесения в стек должна иметь два параметра: один из них задаёт нужный стек, а другой – заносимое в него значение.

Пример описания такой функции может иметь вид

```
void InStack (Stack *p, int newval)
 {
      Stack  *q;
         //создание нового звена
      q = (Stack *)malloc (sizeof(Stack));
          q -> val = newval;
        //созданное звено сделать вершиной стека
          q -> next = p;
       p = q;
 }
```

Выбор элемента из стека

При выполнении этой операции элемент вершины стека должен быть присвоен в качестве значения некоторой переменной, а звено, в котором был представлен этот элемент, должно быть исключено из стека.

Для контроля корректности использования стека необходимо в программе предусмотреть проверку стека на случай, если он пустой.

Опишем функцию, реализующую операцию выбора элемента из стека:

```
int FromStack (Stack *p)
  {
        Stack  *q;
            int k;
          //выбор  элемента  из  вершин
        k  = p→ val;
            //запоминание ссылки на старую вершину
          q  = p;
// исключение и уничтожение использованного звена
              p = p→ next;
          free(q);
            return k;
  }
```

Покажем возможность работы со стеком , как с простым массивом данных:

```
  const int  N = 10;
   int  stack[N];
   int top;        //индекс вершины стека

        // Функция заполнения стека:
 void  push( )
  {
        int k;
        for (top = -1; top  <  N-1;)
          {
printf("Введите очередное значение стека - ");
              scanf("%d", &k);
                  stack [++top] = k;
          }
  }
```

После выполнения функции push() значением переменной top будет значение N-1. Другими словами, переменная top будет «ссылаться» на последний элемент массива, как это и должно происходить в соответствии с дисциплиной **LIFO.**

Функция извлечения элементов из стека

```
int pop ( )  { return stack [ top--]; }
```

Например:

```
for ( i=0; i<N; i++)        printf ("% d \n", pop ());
```

Метод push() обеспечивает добавление одного целого числа, используя индекс top, который всегда показывает на самое «верхнее» число в стеке. Метод pop() обеспечивает извлечение числа из стека и уменьшает индекс top на 1. Таким образом, реализуется стековый принцип организации хранения данных – «первым пришел - последним вышел».

2.7. Таблицы

Широко распространенным видом услуг в информационных технологиях является информационно - справочное обслуживание, которое подразумевает хранение сведений, прием новых сведений и выдачу хранимых сведений по запросам. Хранимые сведения в общем случае представляются записями.

Типичной операцией информационно-справочных систем является поиск и выдача затребованной записи. Для пользователей такой системы было бы обременительно знать и в каждом запросе к системе указывать место хранения нужной записи – тем более, если в системе хранится очень большое количество записей.

Избежать этих неудобств позволяет структура данных, называемая таблицей, в которой каждой записи соответствует определенное имя.

При этом в заказе на выдачу нужной записи достаточно указать только её имя, а реализация такой структуры данных должна сама обеспечивать достаточно быстрый поиск записи с указанным именем.

Итак, таблица – это некоторый (вообще говоря, неупорядоченный) набор именованных записей. Имена записей могут выбираться достаточно произвольным образом. Однако, чтобы иметь возможность организовать эффектный поиск записи по заданному её имени, нужно иметь возможность сравнивать любые два имени записей и устанавливать, какое из них «меньше», а какое «больше». При этом, естественно, подразумевается, что записи имеют разные имена. Имя записи в таблице часто называют также *ключом* записи.

В качестве ключей чаще всего используются целые положительные числа. Удобно также использовать и строки литер одинаковой длины, поскольку строки, с одной стороны, позволяют давать записям достаточно естественные имена, а с другой стороны – над ними определены операции сравнения.

Таким образом, каждая запись, входящая в таблицу, содержит свой ключ и некоторую информацию, связанную с этим ключом (текст записи).

Над таблицей, как структурой данных определены следующие операции:

- поиск в таблице записи с заданным ключом;
- включение в таблицу записи с заданным ключом (если в таблице уже есть запись с таким ключом, то это означает замену старой записи на новую);
- исключение из таблицы записи с заданным ключом.

Существует много разных способов организации таблиц, каждый из которых имеет и свои преимущества, и свои недостатки, так что выбор того или иного способа должен определяться характером использования данной таблицы.

2.7.1. Способы представления таблиц

Простейший способ представления таблицы – это однонаправленный список, который мы рассмотрели ранее. Каждое звено цепочки, которая и образует список, содержит ключ записи, текст записи и ссылку на следующее звено. Такой способ представления таблицы, имея несомненные достоинства, обладает существенным недостатком при поиске требуемой записи. Действительно, для поиска записи приходиться последовательно перебирать звенья списка, пока не встретится запись с заданным ключом или не исчерпается список.

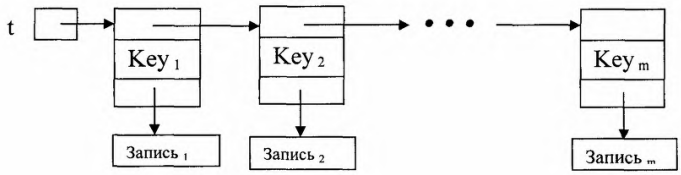
Другой недостаток рассматриваемого способа состоит в том, что если в таблице нет записи с заданным ключом, то для установления этого факта придётся перебрать все записи, поскольку новые записи добавляются, каждый раз в начало списка, поэтому записи следуют в списке в произвольном порядке.

Для устранения второго недостатка представления таблицы в виде простой цепочки можно поддерживать в списке определенный порядок следования записей, например, по возрастанию их ключей. При этом усложнится процедура включения новой записи в таблицу. Определенных затрат потребует процесс поддержания упорядоченности записей в списке, так как включение новой записи в таблицу становится более длительной операцией. Однако эти затраты оправдаются, если включение записей производится сравнительно редко, а поиск отсутствующих записей является достаточно типичным случаем.

2.7.2. Бинарный поиск в таблице

Для ускорения процедуры поиска требуемой записи в таблице можно использовать следующий прием. Прежде всего заметим, что тексты записей можно хранить отдельно от ключей, а при ключе хранить только ссылку на текст записи. В этом случае представление таб-

лицы схематически можно изобразить следующим образом (через key_i обозначим ключ i -й записи):



Как видно, все элементы списка в этом случае одинаковы, т. е. имеют один и тот же тип значений, а тексты записей могут иметь разную длину и даже разную структуру. Каждый элемент списка является записью, содержащей два поля: в одном из них содержится ключ записи, а в другом - ссылка на текст записи.

Так как все элементы однотипны, поэтому их можно объединить не в список, а в одномерный массив (вектор). Введем соответствующие для этого описания:

```
        struct  Record
{       // в качестве ключей будем использовать
            int  key;       //  целые числа
    char * text;            //  указатель на текст записи
};
Record   T[N];   // массив из  N  элементов – записей
```

Будем исходить из того, что компоненты массива упорядочены по возрастанию содержащихся в них ключей, т. е. по значению переменной T[i].key. Это обстоятельство позволяет применить эффективный способ поиска записи по заданному ключу, который называется *дихотомическим поиском.* Идея этого способа состоит в том, что задача поиска записи с заданным ключом сводится к задаче нахождения элемента массива T, в котором содержится этот ключ, а если определен индекс этого элемента, то искомая запись является значением переменной T[i].text.

При поиске компоненты массива, содержащей заданный ключ k, сначала зоной поиска является весь вектор. Возьмем серединную компоненту с индексом i = N/2. Если k = T[i].key, то поиск закончен и искомой записью является значение переменной T[i].text (значением переменной T[i].text является указатель на символьный массив, который, например, можно вывести следующим образом**:** **printf**(“%s”, T[i].text)); . Если k != T[i].key, то процесс поиска продолжается. При k < T[i].key требуемую компоненту следует искать в левой половине

массива, в противном случае – в правой. Таким образом, на каждом следующем шаге зона поиска становится вдвое меньше. Процесс завершается, когда серединная компонента текущей зоны поиска содержит заданный ключ, либо зона поиска оказалась пустой, что означает отсутствие в таблице записи с этим ключом.

Для реализации операции поиска введем в употребление функцию DXTM_search() - дихотомического поиска записи по заданному ключу. Функция возвращает индекс компоненты массива, содержащей заданный ключ **к,** либо нуль при неуспешном поиске.

```
            int  DXTM_search( int k)
{
 int   left, right;              // левая и правая границы зоны поиска
            int  i, yes;                    // yes = 1 – запись найдена
            left = 1;  right = N;
   yes = 0;              // записи с ключом   к  в массиве нет
                do
                   {   i = ( left + right) / 2;
                       if ( k = T[i].key)
                             { yes = 1; break; }
                          else if ( k < T[i].key)  right = i – 1;
                                  else  left = i + 1;
                    }
                while ( left <= right);
                      if ( yes)   return i;
                         else  return 0;
}
```

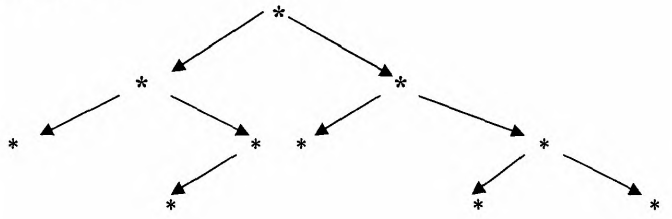
2.7.3. Двоичное дерево

Рассмотренный выше способ представления таблицы разумно использовать в том случае, когда таблица изменяется сравнительно редко, а чаще всего осуществляется поиск в ней, так как быстрота поиска обеспечивается за счет упорядоченности компонента массива.

Рассмотрим представление таблицы в виде двоичного дерева, которое позволяет одинаково эффективно реализовать все три операции над таблицей, причем эта эффективность близка к эффективности дихотомического поиска.

Двоичное дерево схематично можно определить следующим образом: имеется набор вершин, соединенных стрелками. Из каждой вершины выходит не более двух стрелок (ветвей), направленных влево – вниз, либо вправо – вниз. Должна существовать единственная

вершина, в которую не входит ни одна стрелка – эта вершина называется ***корнем*** дерева. В каждую из оставшихся вершин входит в точности одна стрелка:



При представлении таблицы в виде двоичного дерева будем считать, что тексты записей хранятся отдельно. Тогда каждое звено (вершина дерева), соответствующее элементу таблицы, будет записью, содержащей четыре поля, значениями которых будут соответственно ключ записи, указатели на вершину влево – вниз, на вершину вправо – вниз и указатель на текст записи.

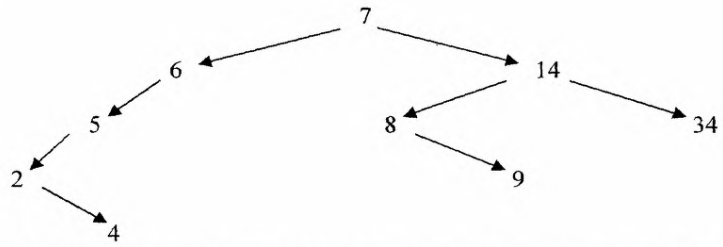
Принцип построения двоичного дерева при занесении записей в таблицу следующий. Пусть имеем последовательность записей с ключами 7, 6, 14, 5, 2, 8, 4, 9, 34. Первую из записей с ключом $К_1 = 7$ делаем корнем дерева, при этом значениями указателей на соседние вершины положим равными **NULL (** указатель на *текст записи* изображать пока не будем, поскольку он сейчас роли не играет).

NULL		K_1	**NULL**

Если ключ следующей записи $К_2$ окажется меньше ключа $К_1$, то этой записи поставим в соответствие левую вершину. В нашем случае $К_2 = 6 < К_1 = 7$, поэтому очередную формируемую вершину дерева делаем левой для корня. Если ключ следующей записи окажется больше ключа предыдущей, то этой записи поставим в соответствие правую вершину и т. д.

Таким образом, если поступает очередная запись с ключом К, то – ***начиная каждый раз с корня дерева*** – в зависимости от сравнения ключа К с ключом в очередной вершине идем влево или вправо от нее, до тех пор, пока не найдем подходящую вершину, к которой можно будет присоединить новую вершину с ключом К.

Двоичное дерево для нашей последовательности записей будет иметь следующий вид:



Теперь рассмотрим вопрос о том, как ввести в употребление нужное нам для представления таблицы двоичное дерево. Для этого отметим, что каждый элемент дерева является записью, в которой содержится: ключ записи, указатели на левую и правую вершины дерева, т. е. на такие же элементы дерева, а также указатель на текст записи.

Двоичное дерево введем в употребление с помощью следующих описаний:

```
struct   Node
{       int  key;
        Node * left ,  * right;
         char  * text;
};
        Node  * tree;
```

Поиск записи в дереве

Дадим описание функции, осуществляющей поиск вершины дерева с заданным ключом. Функция возвращает указатель на найденную вершину (если поиск был успешным), либо **NULL** – в противном случае. Параметрами функции являются ключ поиска записи и дерево, в котором ведется поиск.

```
        Node * Search_in_tree ( int k, const Node * t)
{  Node *p = t;   // заводим указатель p  на корневую вершину дерева
             int   yes = 0;        //  yes - флаг успешности поиска
               if ( p)             //   дерево не пусто
                do
                 {
                    if ( p -> key == k)              // вершина найдена
                              { yes = 1; break; }
                     else
      if ( k < p -> key)  p = p -> left;  //переходим к левой вершине
    else  p = p -> right;            //переходим к правой вершине
                 }
             while ( p);
```

```
                    if ( yes )   return p;
    else  return   NULL;   // вершины с ключом К нет
        }
```

Длительность операции поиска (число вершин, которые надо перебрать для этой цели) зависит от структуры дерева. Наибольший эффект от использования дерева достигается в том случае, когда дерево « сбалансировано», т. е. когда все его вершины – кроме конечных – имеют как левого, так и правого преемника. В этом случае поиск осуществляется так же быстро, как и при дихотомии.

Включение записи в дерево

Рассмотрим случай, когда в таблице нет записи с тем же ключом, что и у включаемой записи (т. е. не рассматриваем случай замены записи).

Для включения новой записи в таблицу прежде всего нужно найти в дереве ту вершину, к которой можно « подвесить » (присоединить) новую вершину, соответствующую включаемой записи. Эта вершина будет найдена в тот момент, когда в качестве очередного значения указателя, определяющего ветвь дерева, в которой надо *продолжить* поиск, окажется значение **NULL**. Для этой цели модифицируем описанную выше функцию Search_in_tree() так, чтобы указатель фиксировался на ту вершину дерева, после обработки которой поиск прекращен.

```
    Node * Search ( int k, Node * t)
{   Node  * p, result;
            p = t;
              if ( p)                    // дерево не пусто
                do
                  {  result = p;
                       if ( k < p -> key)  p = p -> left;
                         else   p = p -> right;
                  }
                while ( p);
        return   result;
}
```

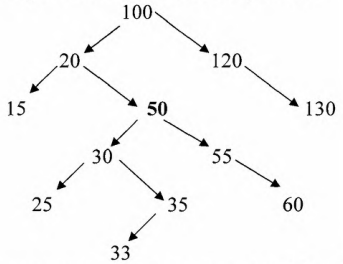
При наличии в программе такого описания функции, процедура включения записи в таблицу, представленную в виде двоичного дерева, описывается достаточно просто:

```
                void  Insert_to_tree ( int k, Node *t, char *s)
               {   Node * p, * result;
                   int len;
result = Search (k,t);  // result – указатель на вершину дерева t после
                        // которой будет включена новая вершина с
                                        // записью  s
                       //  формирование новой вершины в дереве
                           p = new Node;
                                p -> key = k;
                                p -> left = NULL;   p -> right = NULL;
                        // занесение в таблицу текста записи
                                  len = strlen (s);
                              p -> text =  new char [ len + 1];
                              strcpy ( p -> text, s);
    if ( t == NULL)    // если дерево пусто, то созданное звено
      t = p;              // сделать вершиной дерева
              else    // подсоединить новую вершину к дереву
            if ( k < result -> key)    result -> left = p;
                        else    result -> right = p;
                  }
```

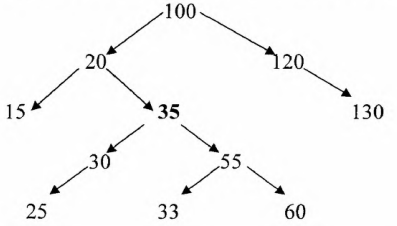
Удаление записи из дерева

Теперь приступим к описанию операции удаления из двоичного дерева звена с заданным ключом. Если вершина является конечной или из нее выходит только одна ветвь – для этого достаточно скорректировать соответствующую ссылку (адрес) у вершины предшественника. Основная трудность состоит в удалении вершины, из которой выходят две ветви, поскольку в удаляемую вершину входит одна стрелка, а выходят две. В этом случае нужно найти подходящее звено дерева, которое можно было бы вставить на место удаляемого, причем это подходящее звено должно просто перемещаться. Такое звено всегда существует: это либо самый правый элемент левого поддерева (для достижения этого звена необходимо перейти в следующую вершину по левой ветви, а затем переходить в очередные вершины только по правой ветви до тех пор, пока очередная такая ссылка не будет равна **NULL**), либо самый левый элемент правого поддерева (для достижения этого звена необходимо перейти в следующую вершину по правой ветви, а затем переходить в очередные вершины только по левой ветви до тех пор, пока очередная такая ссылка не будет равна **NULL**). Очевидно, что такие подходящие звенья могут иметь не более одной

ветви. Изобразим схему исключения из двоичного дерева вершины с ключом К = 50, из которой выходят две стрелки:



Вид дерева до удаления звена с ключом К = 50



Вид дерева после удаления вершины с ключом К = 50

Таким образом, процедура исключения из двоичного дерева звена с заданным ключом должна различать три случая:

1) звена с заданным ключом в дереве нет;
2) звено с заданным ключом имеет не более одной ветви;
3) звено с заданным ключом имеет две ветви.

Приведем описание функции исключения вершины с заданным ключом, автором алгоритма которой является Никлаус Вирт.

```
Node *p;          // вспомогательная переменная – указатель
                  //  на вершину
void Delete ( Node * pointer);     // прототип рекурсивной функции
       void  Delete_from_tree ( int k, Node * t)
       {
    // удаление  вершины с ключом  k  из дерева  t
if ( t == NULL)         // первый случай процедуры исключения
      puts ( " \n Вершина с заданным ключом не найдена ");
 else                   // поиск вершины с заданным ключом
         if ( k < t -> key)  Delete_from_tree ( k, t -> left);
        else  if ( k > t -> key)  Delete_from_tree ( k, t -> right);
```

```
            else
    {    // вершина найдена, необходимо ее удалить
p = t;       // второй случай процедуры исключения
        if ( p -> right == NULL)     t = p -> left;
           else if ( p -> left == NULL)   t = p -> right;
           else    // третий случай исключения
                                Delete ( p -> left );
                        }
    }

// вспомогательная рекурсивная функция, вызываемая лишь
          //  в третьем случае процедуры исключения
       void  Delete ( Node * pointer)
       {     if ( pointer -> right == NULL)
                {   p -> key = pointer -> key;
                      p -> text = pointer -> text;
                         p = pointer ;
                            pointer = pointer -> left;
                }
                   else  Delete ( pointer -> right);
    }
```

Вспомогательная рекурсивная функция Delete () " спускается" до самой правой вершины левого поддерева удаляемого элемента *p , а затем заменяет значения полей key и text в *p соответствующими значениями полей key и text вершины * pointer. После этого вершину, на которую ссылается указатель pointer, можно исключить (это делается операцией присваивания pointer = pointer -> left;). Заметим, что можно освободиться и от памяти, занимаемой удаленной вершиной, используя стандартную функцию **free** () .

В заключение рассмотрим в качестве упражнения, применение функций работы с динамической памятью и динамическими информационными структурами на примере сортировки слов с помощью бинарного дерева [1]. В этом упражнении демонстрируются операции построения бинарного дерева и его обход для печати содержимого его вершин. В качестве метода сортировки использована бинарная сортировка включением, при которой, если новое слово выше в алфавитном порядке предыдущего, то оно располагается в вершине нижнего уровня, к которой ведет *левая ветвь*.

Так как заранее неизвестны длина каждого конкретного слова и количество слов в исходном файле, поэтому целесообразно динамиче-

ски выделять память как для каждой вершины дерева, так и для собственно массива, хранящего слово в виде строки.

Приведем тексты функций программы

```
#include<stdio.h>
#define SIZE 80
        // определение структурного типа
        // соответствующего вершине дерева
                struct Node
                    {  char * str;
                       Node * left, * right;
                    };
  int main( int argc, char * argv[] )
  {
        Node * Add_tree ( Node *root, char * line);
             void  output ( Node * ptr);
        Node * root = NULL;
         FILE * fp;
         char line [ SIZE];
          if (( fp = fopen( argv[1], "r")) != NULL)
             {    while ( fgets( line,SIZE,fp))
                  root = Add_tree ( root, line);
             }
          else
            {  perror ( argv[1]);   return -1;  }
              output  ( root);
                 return o;
   }

 //Add_tree() – функция  включения новой вершины в дерево,
 // параметры функции:  line -строка файла,root -указатель на
        // корневую вершину в дереве
    #include<string.h>
#include<stdio.h>
        struct Node
        {  char * str;
           Node *left, *right;
        };
```

```
Node * Add_tree ( Node * root, char * line)
  {
     Node ** prior;
     Node * ptr;                  // указатель на текущую вершину
     Node * New_node ( char * line);
       if ( root != NULL)
        {    ptr = root;

             // цикл спуска по ветвям дерева до вершины,
             // содержащей нулевой адрес

          while ( ptr != NULL)
            {
               if ( strcmp( line, ptr -> str) <=0)

               {       // идем по левой ветви
                    prior = & ptr -> left;
                      ptr = ptr -> left;
               }
                else
               {    // новая строка ниже – идем по правой ветви
                       prior = & ptr -> right;
                         ptr = ptr -> right;
                }
            } //   конец цикла

             // создание новой вершины и возврат ее в
             // функцию  Add_tree()
                  * prior = New_node(line);
            return root;

        }       //  конец  if

return  New_node(line);     // вызывается функция New_node(),
                            // если дерево  пустое
  }
```

// Функция New_node() , куда передается вновь введенная строка из файла

```
#include<alloc.h>
#include<string.h>
#include<stdio.h>
        struct Node
          {   char * str;
               Node * left, *right;
          };
         Node *  New_node (char * line)
          {
                Node * ptr;
                  ptr = (Node *)malloc( sizeof( Node));
                  ptr -> str =( char*)malloc( strlen(line) +1);
                              strcpy( ptr -> str, line);
                  ptr -> left = ptr -> right = NULL;
                        return ptr;
          }

      //  печать строк из вершин дерева в алфавитном порядке

       #include<stdio.h>
                    struct Node
                     {    char * str;
                          Node * left, * right;
                     };
            void  output( Node * ptr)
            {
                 int i = 0;
        Node * stack[10];   // список из 10-ти указателей на структуру
                    while (1)
                       {    while ( ptr != NULL)
                                 { stack [ i++] = ptr;
                                   ptr = ptr -> left;
                                 }
                            ptr = stack[--i];
                               if ( i<0)  break;
                               printf ( "%s", ptr -> str);
                                  ptr = ptr -> right;
                       }
              }
```

Контрольные вопросы и задания

1. Каков механизм и операции размещения динамических данных в оперативной памяти?

2. Типизированный указательный тип–описание, операции, использование. Доступ к значению динамического данного.

3. Динамические данные и динамические структуры данных, базовые алгоритмы при работе с ними?

4. Что собой представляет связный список? Почему такая структура данных является динамической? Каковы основные операции, которые можно выполнять над списками?

5. Напишите функцию Invert-List(), обращающую список, т. е. изменяющую направление указателей на противоположное.

6. Что представляет собой стек, очередь и каким образом возможен к ним доступ?

7. Напишите функцию создания стека и вывода его на экран.

8. Напишите функцию считывания и удаления элемента из очереди.

9. Какого соглашения придерживаются при построении бинарного дерева целочисленных значений?

10. Напишите функцию нахождения самого правого узла бинарного дерева.

11. Представить двоичное дерево списочной структурой и описать операцию обхода дерева снизу вверх (в высоту).

12. Описать и протестировать рекурсивные функции, которые для заданного двоичного дерева с целочисленными элементами выполняют следующие действия:

- поиск элемента с заданным значением;
- вычисление суммы значений его элементов;
- подсчет числа вершин на N -м уровне дерева.

13. Во внешнем файле задана информация о студентах университета, включающая в себя: фамилию, номер группы, оценки, полученные в последнюю сессию. Требуется:

- представить исходную информацию в виде двоичного дерева, в вершинах которого содержатся сведения об отдельных студентах, причем фамилия является ключом;
- напечатать фамилии студентов, сдавших все экзамены на 5;
- напечатать фамилии тех студентов, которые получили по всем трем экзаменам неудовлетворительные оценки.

Методические указания к упражнению

Схематично алгоритм обхода двоичного дерева может выглядеть, например, следующим образом.

1. В качестве очередной вершины взять корень дерева. Перейти к пункту 2.

2. Произвести обработку очередной вершины в соответствии с требованиями задачи. Перейти к пункту 3.

3. А) Если очередная вершина имеет обе ветви, то в качестве новой очередной вершины выбрать ту вершину, на которую ссылается левая ветвь, а вершину, на которую ссылается правая ветвь, поставить в очередь; перейти к пункту 2;

Б) если очередная вершина является конечной, то выбрать в качестве новой очередной вершины вершину из очереди, если она не пуста, и перейти к пункту 2; если же очередь пуста, то это означает, что обход всего дерева окончен, перейти к пункту 4;

В) если очередная вершина имеет только одну ветвь, то в качестве новой очередной вершины выбрать ту вершину, на которую эта ветвь указывает. И перейти к пункту 2.

4. Конец алгоритма.

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. *Подбельский В.В., Фомин С.С.* Программирование на языке Си: Учеб. Пособие. – 2- е изд. – М .: Финансы и статистика, 2002.

2. *Подбельский В.В.* Язык С ++: Учеб. пособие. – 3- е изд. - М.: Финансы и статистика, 1998.

3. *Культин Н.Б.* С/ С++ в задачах и примерах. – СПб.: БХВ – Петербург, 2004.

4. *Мартынов Н.Н.* Программирование для Windows на С/ С++. Т.1.- М.: ООО " Бином – Пресс", 2004

5. *Шиманович Е.А.* С/ С++ в примерах и задачах. – М.: Новое знание, 2004.

6. *Ахо А.В., Хопкрофт Д.Э., Ульман Д.Д.* Структуры данных и алгоритмы М.: Изд. "Вильямс", 2000.

7. *Седжвик Р.* Фундаментальные алгоритмы на С++. Ч. 1-4, М.: Изд. "ДиаСофт", 2001.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ ……..……3

1. УКАЗАТЕЛЬНЫЙ ТИП И ОСНОВЫ ПРОГРАММИРОВАНИЯ....5
1.1 Статические и динамические программные объекты ……....5
1.2 Понятие указателей ……..……6
1.2.1 Объявление и использование указателей ……....9
1.2.2 Ограничения на использование операции……..11
1.2.3 Подробнее об инициализации указателей ……....14
1.3 Операции с указателями ……..……22
1.3.1 Арифметические операции и указатели ……..……...25
1.3.2 Указатели и отношения ……27
1.4 Связь массивов и указателей …….. ...30
1.4.1 Массивы указателей ……..……...35
1.4.2 Строки и указатели ……..……..37
1.5 Указатели в параметрах функций ……...42
1.6 Массивы и строки как параметры функций ……..……44
1.7 Указатели на функции ……54
1.7.1 Массивы указателей на функции ……56
1.7.2 Указатели на функции как параметры ……58
1.7.3 Указатель на функцию, как возвращаемое функцией значение..…… ……....60
1.8 Полезные практические примеры программирования с указателями ……..…...62

2 УКАЗАТЕЛЬНЫЙ ТИП И ДИНАМИЧЕСКИЕ ОБЪЕКТЫ СЛОЖНОЙ СТРУКТУРЫ …….. ……67
2.1 Необходимые сведения ……....67
2.2 Односвязный список ……....69
2.3 Представление строки в виде односвязного списка ……72
2.4 Реализация операций над строками – цепочками ……. ……75
2.5 Двунаправленные списки …….. ...78
2.6 Очереди и стеки …….. ...83
2.7 Таблицы …….. 87
2.7.1 Способы представления таблиц ……88
2.7.2 Бинарный поиск в таблице ……..…88
2.7.3 Двоичное дерево ……... ……90

БИБЛИОГРАФИЧЕСКИЙ СПИСОК ……..……102

Учебное издание