ФУНКЦИИ И УКАЗАТЕЛИ В СИ

Функции

- Функция в СИ имеет вид
- **имя (список аргументов, если они имеются)**
- **описания аргументов, если они имеются;**
- **{**
- **описания и операторы, если они имеются**
- **}**
- Описания аргументов помещаются между списком аргументов и открывающейся левой фигурной скобкой; каждое описание заканчивается точкой с запятой. Имена, использованные для аргументов конкретной функции, являются чисто локальными и недоступны никаким другим функциям: другие процедуры могут использовать те же самые имена без возникновения конфликта.

Функции

- Если функция возвращает что-либо отличное от целого значения, то перед ее именем может стоять указатель типа, например:
- **long func(int, int, int*);**
- Связь между функциями осуществляется через аргументы и возвращаемые функциями значения; ее можно также осуществлять через внешние переменные. Функции могут располагаться в исходном файле в любом порядке, а сама исходная программа может размещаться на нескольких файлах, но так, чтобы ни одна функция не расщеплялась.
- Оператор **return** служит механизмом для возвращения значения из вызванной функции в функцию, которая к ней обратилась. За **return** может следовать любое выражение:
- **return (выражение)**

Функции

- Вызывающая функция может игнорировать возвращаемое значение, если она этого пожелает. Более того, после **return** может не быть вообще никакого выражения; в этом случае в вызывающую программу не передается никакого значения. Управление также возвращается в вызывающую программу без передачи какого-либо значения и в том случае, когда при выполнении происходит переход на конец функции, достигая закрывающейся правой фигурной скобки. Если функция возвращает значение из одного места и не возвращает никакого значения из другого места, это не является незаконным, но может быть признаком каких-то неприятностей. В любом случае "значением" функции, которая не возвращает значения, несомненно, будет мусор.

Аргументы функции

- Аргументы функций передаются по значению, т.е. вызванная функция получает свою временную копию каждого аргумента, а не его адрес. Это означает, что вызванная функция не может воздействовать на исходный аргумент в вызывающей функции. Внутри функции каждый аргумент по существу является локальной переменной, которая инициализируется тем значением, с которым к этой функции обратились.
- При необходимости все же можно добиться, чтобы функция изменила переменную из вызывающей программы. Эта программа должна обеспечить установление адреса переменной (технически, через указатель на переменную), а в вызываемой функции надо описать соответствующий аргумент как указатель и ссылаться к фактической переменной косвенно через него.

Внешние переменные

- Если в качестве аргумента функции выступает имя массива, то передается адрес начала этого массива; сами элементы не копируются. Функция может изменять элементы массива, используя индексацию и адрес начала. Таким образом, массив передается по ссылке.
- Программа на языке "С" состоит из набора внешних объектов, которые являются либо переменными, либо функциями. Термин "внешний" используется главным образом в противопоставление термину "внутренний", которым описываются аргументы и автоматические переменные, определенные внутри функций.
- Внешние переменные определены вне какой-либо функции и, таким образом, потенциально доступны для многих функций. Сами функции всегда являются внешними, поскольку что правила языка "С" не разрешают определять одни функции внутри других. По умолчанию внешние переменные являются также и "глобальными".

Внешние переменные

- В силу своей глобальной доступности внешние переменные предоставляют другую, отличную от аргументов и возвращаемых значений, возможность для обмена данными между функциями. Если имя внешней переменной каким-либо образом описано, то любая функция имеет доступ к этой переменной, ссылаясь к ней по этому имени.
- В случаях, когда связь между функциями осуществляется с помощью большого числа переменных, внешние переменные оказываются более удобными и эффективными, чем использование длинных списков аргументов.
- Вторая причина использования внешних переменных связана с инициализацией. В частности, внешние массивы могут быть инициализированы, а автоматические нет.

Внешние переменные

- Третья причина использования внешних переменных обусловлена их областью действия и временем существования. Автоматические переменные являются внутренними по отношению к функциям; они возникают при входе в функцию и исчезают при выходе из нее. Внешние переменные, напротив, существуют постоянно. Они не появляются и не исчезают, так что могут сохранять свои значения в период от одного обращения к функции до другого. В силу этого, если две функции используют некоторые общие данные, причем ни одна из них не обращается к другой , то часто наиболее удобным оказывается хранить эти общие данные в виде внешних переменных, а не передавать их в функцию и обратно с помощью аргументов.

Область действия

- Областью действия имени является та часть программы, в которой это имя определено. Для автоматической переменной, описанной в начале функции, областью действия является та функция, в которой описано имя этой переменной, а переменные из разных функций, имеющие одинаковое имя, считаются не относящимися друг к другу. Это же справедливо и для аргументов функций.
- Область действия внешней переменной простирается от точки, в которой она объявлена в исходном файле, до конца этого файла.
- С другой стороны, если нужно сослаться на внешнюю переменную до ее определения, или если такая переменная определена в файле, отличном от того, в котором она используется, то необходимо описание **extern**.

Область действия

- Важно различать описание внешней переменной и ее определение. описание указывает свойства переменной (ее тип, размер и т.д.); определение же вызывает еще и отведение памяти. Если вне какой бы то ни было функции появляются строчки
- **int sp;**
- **double val[maxval];**
- то они определяют внешние переменные **sp** и **val**, вызывают отведение памяти для них и служат в качестве описания для остальной части этого исходного файла. В то же время строчки
- **extern int sp;**
- **extern double val[];**
- описывают в остальной части этого исходного файла переменную **sp** как **int**, а **val** как массив типа **double** (размер которого указан в другом месте), но не создают переменных и не отводят им места в памяти.

Статические переменные

- Во всех файлах, составляющих исходную программу, должно содержаться только одно определение внешней переменной; другие файлы могут содержать описания **extern** для доступа к ней. Любая инициализация внешней переменной проводится только в определении. В определении должны указываться размеры массивов, а в описании **extern** этого можно не делать.
- Статические переменные представляют собой третий класс памяти, в дополнении к автоматическим переменным и **extern**.
- Статические переменные могут быть либо внутренними, либо внешними. Внутренние статические переменные точно так же, как и автоматические, являются локальными для некоторой функции, но, в отличие от автоматических, они остаются существовать, а не появляются и исчезают вместе с обращением к этой функции. это означает, что внутренние статические переменные обеспечивают постоянное, недоступное извне хранение внутри функции.

Статические переменные

- Внешние статические переменные определены в остальной части того исходного файла, в котором они описаны, но не в каком-либо другом файле.
- Статическая память, как внутренняя, так и внешняя, специфицируется словом **static**, стоящим перед обычным описанием. Переменная является внешней, если она описана вне какой бы то ни было функции, и внутренней, если она описана внутри некоторой функции.
- **static int bufp=0;**
- Нормально функции являются внешними объектами; их имена известны глобально. Возможно, однако, объявить функцию как **static**; тогда ее имя становится неизвестным вне файла, в котором оно описано.
- В языке "C" **static** отражает не только постоянство, но и степень того, что можно назвать "приватностью".

Регистровые переменные

- Внутренние статические объекты определены только внутри одной функции; внешние статические объекты (переменные или функции) определены только внутри того исходного файла, где они появляются, и их имена не вступают в конфликт с такими же именами переменных и функций из других файлов.
- Четвертый и последний класс памяти называется регистровым. Описание **register** указывает компилятору, что данная переменная будет часто использоваться. Когда это возможно, переменные, описанные как **register**, располагаются в машинных регистрах, что может привести к меньшим по размеру и более быстрым программам. Описание **register** выглядит как
- **register int x;**
- **register char c;**

Регистровые переменные

- Часть **int** может быть опущена. Описание **register** можно использовать только для автоматических переменных и формальных параметров функций.
- На практике возникают некоторые ограничения на регистровые переменные, отражающие реальные возможности имеющихся аппаратных средств. В регистры можно поместить только несколько переменных в каждой функции, причем только определенных типов. В случае превышения возможного числа или использования неразрешенных типов слово **register** игнорируется.
- Кроме того невозможно извлечь адрес регистровой переменной

Рекурсия

- В языке "С" функции могут использоваться рекурсивно; это означает, что функция может прямо или косвенно обращаться к себе самой. Когда функция вызывает себя рекурсивно, при каждом обращении образуется новый набор всех автоматических переменных, совершенно не зависящий от предыдущего набора.
- Рекурсия обычно не дает никакой экономии памяти, поскольку приходится где-то создавать стек для обрабатываемых значений. Не приводит она и к созданию более быстрых программ. Но рекурсивные программы более компактны, и они зачастую становятся более легкими для понимания и написания. Рекурсия особенно удобна при работе с рекурсивно определяемыми структурами данных, например, с деревьями.

Указатели и адреса

- Указатель - это переменная, содержащая адрес другой переменной. Указатели очень широко используются в языке "С". Это происходит отчасти потому, что иногда они дают единственную возможность выразить нужное действие, а отчасти потому, что они обычно ведут к более компактным и эффективным программам, чем те, которые могут быть получены другими способами.
- Так как указатель содержит адрес объекта, это дает возможность "косвенного" доступа к этому объекту через указатель. Предположим, что **x** - переменная, например, типа **int**, а **px** - указатель, созданный неким еще не указанным способом.
- Унарная операция **&** выдает адрес объекта, так что оператор
- **px = &x;**
- присваивает адрес **x** переменной **px**; говорят, что **px** "указывает" на **x**.

Указатели и адреса

- Операция **&** применима только к переменным и элементам массива, конструкции вида **&(x-1)** и **&3** являются незаконными. Нельзя также получить адрес регистровой переменной.
- Унарная операция * рассматривает свой операнд как адрес конечной цели и обращается по этому адресу, чтобы извлечь содержимое. Следовательно, если **y** тоже имеет тип **int**, то
- **y = *px;**
- присваивает **y** содержимое того, на что указывает **px**. Так последовательность
- **px = &x;**
- **y = *px;**
- присваивает **y** то же самое значение, что и оператор
- **y = x;**

Указатели и адреса

- Переменные, участвующие во всем этом необходимо описать:
- **int x, y;**
- Описание указателя
- **int *px;**
- должно рассматриваться как мнемоническое; оно говорит, что комбинация ***px** имеет тип **int**. Это означает, что если **px** появляется в контексте ***px**, то это эквивалентно переменной типа **int**. Фактически синтаксис описания переменной имитирует синтаксис выражений, в которых эта переменная может появляться. Это замечание полезно во всех случаях, связанных со сложными описаниями.

Указатели и адреса

- Указатели могут входить в выражения. Например, если **px** указывает на целое **x**, то ***px** может появляться в любом контексте, где может встретиться **x**. Так оператор
- **y = *px + 1**
- присваивает **y** значение, на 1 большее значения **x**;
- Ссылки на указатели могут появляться и в левой части присваиваний. Если **px** указывает на **x**, то
- ***px = 0**
- полагает X равным нулю, а
- *** px += 1**
- увеличивает его на единицу, как и выражение
- **(*px)++**
- Круглые скобки в последнем примере необходимы; если их опустить, то поскольку унарные операции, подобные * и **++**, выполняются справа налево, это выражение увеличит **px**, а не ту переменную, на которую он указывает.

Указатели и аргументы функций

- И наконец, так как указатели являются переменными, то с ними можно обращаться, как и с остальными переменными. Если **py** - другой указатель на переменную типа **int**, то
- **py = px**
- копирует содержимое **px** в **py**, в результате чего **py** указывает на то же, что и **px**.
- Так как в Си передача аргументов функциям осуществляется "по значению", вызванная процедура не имеет непосредственной возможности изменить переменную из вызывающей программы. Все же имеется возможность получить желаемый эффект. Вызывающая программа передает указатели подлежащих изменению значений:
- **swap(&a, &b);**

Указатели и аргументы функций

- Так как операция **&** выдает адрес переменной, то **&a** является указателем на **a.** В самой **swap** аргументы описываются как указатели и доступ к фактическим операндам осуществляется через них.
- **swap(px, py) /* interchange *px and *py */**
- **int *px, *py;**
- **{**
- ** int temp;**
- ** temp = *px;**
- ** *px = *py;**
- ** *py = temp;**
- **}**

Указатели и массивы

- Указатели в качестве аргументов обычно используются в функциях, которые должны возвращать более одного значения. (Можно сказать, что **swap** возвращает два значения, новые значения ее аргументов).
- В языке "С" существует сильная взаимосвязь между указателями и массивами , настолько сильная, что указатели и массивы действительно следует рассматривать одновременно. Любую операцию, которую можно выполнить с помощью индексов массива, можно сделать и с помощью указателей. вариант с указателями обычно оказывается более быстрым, но и несколько более трудным для непосредственного понимания, по крайней мере для начинающего. Описание
- int a[10];
- определяет массив размера 10, т.е. набор из 10 последовательных объектов, называемых a[0], a[1], ..., a[9].

Указатели и массивы

- Запись a[i] соответствует элементу массива через i позиций от начала. Если pa - указатель целого, описанный как
- int *pa;
- то присваивание
- pa = &a[0];
- приводит к тому, что pa указывает на нулевой элемент массива a; это означает, что pa содержит адрес элемента a[0]. Теперь присваивание
- x = *pa
- будет копировать содержимое a[0]в x.
- Если **pa** указывает на некоторый определенный элемент массива **a**, то по определению **pa+1** указывает на следующий элемент, и вообще **pa-i** указывает на элемент, стоящий на **i** позиций до элемента, указываемого **pa**, а **pa+i** на элемент, стоящий на **i** позиций после.

Указатели и массивы

- Таким образом, если **pa** указывает на **a[0]**, то ***(pa+1)** ссылается на содержимое **a[1]**, **pa+i** - адрес **a[i]**, а ***(pa+i)** - содержимое **a[i]**.
- Эти замечания справедливы независимо от типа переменных в массиве **a**. Суть определения "добавления 1 к указателю", а также его распространения на всю арифметику указателей, состоит в том, что приращение масштабируется размером памяти, занимаемой объектом, на который указывает указатель. Таким образом, **i** в **pa+i** перед прибавлением умножается на размер объектов, на которые указывает **pa**.
- Очевидно существует очень тесное соответствие между индексацией и арифметикой указателей. В действительности компилятор преобразует ссылку на массив в указатель на начало массива. В результате этого имя массива является указательным выражением. Отсюда вытекает несколько весьма полезных следствий.

Указатели и массивы

- Так как имя массива является синонимом местоположения его нулевого элемента, то присваивание **pa=&a[0]** можно записать как **pa = a.**
- Ссылку на **a[i]** можно записать в виде ***(a+i)**. При анализе выражения **a[i]** в языке Си оно немедленно преобразуется к виду ***(a+i)**; эти две формы совершенно эквивалентны. Если применить операцию **&** к обеим частям такого соотношения эквивалентности, то мы получим, что **&a[i]** и **a+i** тоже идентичны: **a+i** - адрес **i** -го элемента от начала **a**. С другой стороны, если **pa** является указателем, то в выражениях его можно использовать с индексом: **pa[i]** идентично ***(pa+i)**. Короче, любое выражение, включающее массивы и индексы, может быть записано через указатели и смещения и наоборот, причем даже в одном и том же утверждении.

Указатели и массивы

- Имеется одно различие между именем массива и указателем, которое необходимо иметь в виду. Указатель является переменной, так что операции **pa=a** и **pa++** имеют смысл. Но имя массива является константой, а не переменной: конструкции типа **a=pa** или **a++**, или **p=&a** будут незаконными.
- Когда имя массива передается функции, то на самом деле ей передается местоположение начала этого массива. Внутри вызванной функции такой аргумент является точно такой же переменной, как и любая другая, так что имя массива в качестве аргумента действительно является указателем, т.е. переменной, содержащей адрес.
- **strlen(s) /* return length of string s */**
- **char *s;**
- **{ int n;**
- **for (n = 0; *s != '\0'; s++) n++;**
- **return(n); }**

Указатели и массивы

- Операция увеличения **s** совершенно законна, поскольку эта переменная является указателем; **s++** никак не влияет на символьную строку в обратившейся к **strlen** функции, а только увеличивает локальную для функции **strlen** копию адреса. Описания формальных параметров в определении функции в виде
- **char s[];**
- **char *s;**
- совершенно эквивалентны, какой вид описания следует предпочесть, определяется в значительной степени тем, какие выражения будут использованы при написании функции. Если функции передается имя массива, то в зависимости от того, что удобнее, можно полагать, что функция оперирует либо с массивом, либо с указателем, и действовать далее соответствующим образом. Можно даже использовать оба вида операций, если это кажется уместным и ясным.

Указатели и массивы

- Можно передать функции часть массива, если задать в качестве аргумента указатель начала подмассива. Например, если **a** - массив, то как **f(&a[2])** как и **f(a+2)** передают функции **f** адрес элемента **a[2]**, потому что и **&a[2]**, и **a+2** являются указательными выражениями, ссылающимися на третий элемент **a**. Внутри функции **f** описания аргументов могут присутствовать в виде:
- **f(arr)**
- **int arr[];**
- **...**
- или
- **f(arr)**
- **int *arr;**
- **...**

Адресная арифметика

- Если **p** является указателем, то операция **p++** увеличивает **p** так, что он указывает на следующий элемент набора этих объектов, а операция **p+=i** увеличивает **p** так, чтобы он указывал на элемент, отстоящий на **i** элементов от текущего элемента. Эти и аналогичные конструкции представляют собой формы арифметики указателей или адресной арифметики.
- Язык Си последователен и постоянен в своем подходе к адресной арифметике; объединение в одно целое указателей, массивов и адресной арифметики является одной из наиболее сильных сторон языка. Проиллюстрируем некоторые из соответствующих возможностей языка на примере элементарной (но полезной) программы распределения памяти. Имеются две функции: функция **alloc(n)** возвращает в качестве своего значения указатель **p**, который указывает на первую из **n** последовательных символьных позиций, которые могут быть использованы вызывающей функцию **alloc** программой для хранения символов.

Адресная арифметика

- Функция **free(p)** освобождает приобретенную таким образом память, так что ее в дальнейшем можно снова использовать. программа является "элементарной", потому что обращения к **free** должны производиться в порядке, обратном тому, в котором производились обращения к **alloc**.
- Таким образом, управляемая функциями **alloc** и **free** память является стеком или списком, в котором последний вводимый элемент извлекается первым. Стандартная библиотека языка Си содержит аналогичные функции, не имеющие таких ограничений.
- Между тем, однако, для многих приложений нужна только тривиальная функция **alloc** для распределения небольших участков памяти неизвестных заранее размеров в непредсказуемые моменты времени.

Адресная арифметика

- Простейшая реализация состоит в том, чтобы функция раздавала отрезки большого символьного массива, которому присвоили имя **allocbuf**. Этот массив является собственностью функций **alloc** и **free**. Так как они работают с указателями, а не с индексами массива, никакой другой функции не нужно знать имя этого массива. Он может быть описан как внешний статический, т.е. он будет локальным по отношению к исходному файлу, содержащему **alloc** и **free**, и невидимым за его пределами. При практической реализации этот массив может даже не иметь имени; вместо этого он может быть получен в результате запроса к операционной системе на указатель некоторого неименованного блока памяти.
- Другой необходимой информацией является то, какая часть массива **allocbuf** уже использована. Мы пользуемся указателем первого свободного элемента, названным **allocp**.

Адресная арифметика

- Когда к функции **alloc** обращаются за выделением **n** символов, то она проверяет, достаточно ли осталось для этого места в **allocbuf**. Если достаточно, то **alloc** возвращает текущее значение **allocp** (т.е. начало свободного блока), затем увеличивает его на **n**, с тем чтобы он указывал на следующую свободную область. Функция **free(p)** просто полагает **allocp** равным **p** при условии, что **p** указывает на позицию внутри **allocbuf**.
- Определим константы и глобальные переменные:

```
define null 0  /* pointer value for error report */
define allocsize 1000  /* size of available space */
static char allocbuf[allocsize];/* storage for alloc */
static char *allocp = allocbuf;/*next free position */
```

Адресная арифметика

```
char *alloc(n)  /* return pointer to n characters */
int n;
{
if (allocp + n <= allocbuf + allocsize) {
   allocp += n;
   return(allocp - n); /* old p */
} else         /* not enough room */
   return(null);
}
free(p)    /* free storage pointed by p */
char *p;
{
 if (p >= allocbuf && p < allocbuf + allocsize)
    allocp = p;
}
```

Адресная арифметика

- Указатель может быть инициализирован точно так же, как и любая другая переменная, хотя обычно единственными осмысленными значениями являются **NULL** или выражение, включающее адреса ранее определенных данных соответствующего типа. Описание
- **static char *allocp = allocbuf;**
- определяет **allocp** как указатель на символы и инициализирует его так, чтобы он указывал на **allocbuf**, т.е. на первую свободную позицию при начале работы программы.

Адресная арифметика

- Так как имя массива является адресом его нулевого элемента, то это можно было бы записать в виде
- **static char allocbuf[allocsize];/* storage for alloc */**
- **static char *allocp = &allocbuf[0];**
- Можно использовать ту запись, которая кажется более естественной. С помощью проверки
- **if (allocp + n <= allocbuf + allocsize)**
- выясняется, осталось ли достаточно места, чтобы удовлетворить запрос на **n** символов.
- Если достаточно, то новое значение **allocp** не будет указывать дальше, чем на последнюю позицию **allocbuf**. Если запрос может быть удовлетворен, то функция возвращает обычный указатель. Если же нет, то функция должна вернуть некоторый признак, говорящий о том, что больше места не осталось, например, **return(NULL).**

Адресная арифметика

- В языке Си гарантируется, что ни один правильный указатель данных не может иметь значение нуль, так что возвращение нуля может служить в качестве сигнала о ненормальном событии, в данном случае об отсутствии места. Однако вместо нуля пишут **NULL**, с тем чтобы более ясно показать, что это специальное значение указателя.
- Вообще говоря, целые не могут осмысленно присваиваться указателям, а нуль - это особый случай.
- Проверки вида
- **if (allocp + n <= allocbuf + allocsize)**
- и
- **if (p >= allocbuf && p < allocbuf + allocsize)**
- демонстрируют несколько важных аспектов арифметики указателей.

Адресная арифметика

- Во-первых, при определенных условиях указатели можно сравнивать. Если **p** и **q** указывают на элементы одного и того же массива, то такие отношения, как **<, >=** и т.д., работают надлежащим образом. Например, **p < q** истинно, если **p** указывает на более ранний элемент массива, чем **q**. Отношения **= =** и **!=** тоже работают. Любой указатель можно осмысленным образом сравнить на равенство или неравенство с **NULL**. Но ни за что нельзя ручаться, сравнивая указатели, указывающие на разные массивы.
- Во-вторых, указатель и целое можно складывать и вычитать. Конструкция **p + n** подразумевает **n**-ый объект за тем, на который **p** указывает в настоящий момент. Это справедливо независимо от того, на какой вид объектов **p** должен указывать; компилятор сам масштабирует **n** в соответствии с определяемым из описания **p** размером объектов, указываемых с помощью **p**.

Адресная арифметика

- Вычитание указателей тоже возможно: если **p** и **q** указывают на элементы одного и того же массива, то **p-q** - количество элементов между **p** и **q**.
- За исключением упомянутых выше операций (сложение и вычитание указателя и целого, вычитание и сравнение двух указателей), вся остальная арифметика указателей является незаконной. Запрещено складывать два указателя, умножать, делить, сдвигать или маскировать их, а также прибавлять к ним переменные типа **float** или **double**.

Массивы указателей, указатели указателей

- Так как указатели сами являются переменными, то можно ожидать использования массива указателей. Проиллюстрируем это написанием программы сортировки в алфавитном порядке набора текстовых строк, предельно упрощенного варианта утилиты **sort** операционной систем UNIX.
- Если подлежащие сортировке сроки хранятся одна за другой в длинном символьном массиве, то к каждой строке можно обратиться с помощью указателя на ее первый символ. Сами указатели можно хранить в массиве. Две строки можно сравнить, передав их указатели функции **strcmp**.
- Если две расположенные в неправильном порядке строки должны быть переставлены, то фактически переставляются указатели в массиве указателей, а не сами тексты строк. Этим исключаются сразу две связанные проблемы: сложного управления памятью и больших дополнительных затрат на фактическую перестановку строк.

Массивы указателей, указатели указателей

- Процесс сортировки включает три шага:
  - **чтение всех строк ввода**
  - **их сортировка**
  - **вывод их в правильном порядке**
- Лучше разделить программу на несколько функций в соответствии с естественным делением задачи.
- Функция, осуществляющая ввод, должна извлечь символы каждой строки, запомнить их и построить массив указателей строк. Она должна также подсчитать число строк во вводе, так как эта информация необходима при сортировке и выводе. Так как функция ввода в состоянии справиться только с конечным числом вводимых строк, в случае слишком большого их числа она может возвращать некоторое число, отличное от возможного числа строк, например -1.
- Символ новой строки в конце каждой строки удаляется, так что он никак не будет влиять на порядок, в котором сортируются строки.

Массивы указателей, указатели указателей

```
#define maxlen 1000
readlines(lineptr, maxlines) /* read input lines */
 char *lineptr[];       /* for sorting */
 int maxlines;
{
  int len, nlines;
  char *p, *alloc(), line[maxlen];
  nlines = 0;
  while ((len = getline(line, maxlen)) > 0)
     if (nlines >= maxlines)    return(-1);
     else if ((p = alloc(len)) == null)  return (-1);
     else {
          line[len-1] = '\0'; /* zap newline */
          strcpy(p,line);
          lineptr[nlines++] = p;
      }
   return(nlines);   }
```

Массивы указателей, указатели указателей

- Функция, осуществляющая вывод, должна печатать строки в том порядке, в каком они появляются в массиве указателей.

```
writelines(lineptr, nlines) /* write output lines */
char *lineptr[];
int nlines;
{
 int i;
 for (i = 0; i < nlines; i++)
    printf("%s\n", lineptr[i]);
}
```

- Существенно новым в этой программе является описание
- **char *lineptr[lines];**
- которое сообщает, что **lineptr** является массивом из **lines** элементов, каждый из которых - указатель на переменные типа **char**.

Массивы указателей, указатели указателей

- Это означает, что **lineptr[i]** - указатель на символы, а ***lineptr[i]** извлекает символ.
- Так как сам **lineptr** является массивом, который передается функции **writelines**, с ним можно обращаться как с указателем. Тогда последнюю функцию можно переписать в виде:

```
writelines(lineptr, nlines) /* write output lines */
char *lineptr[];
int nlines;
{
 int i;
 while (--nlines >= 0)
    printf("%s\n", *lineptr++);
}
```

- здесь ***lineptr** сначала указывает на первую строку; каждое увеличение передвигает указатель на следующую строку, в то время как **nlines** убывает до нуля.

Массивы указателей, указатели указателей

- Справившись с вводом и выводом, мы можем перейти к сортировке.

```
sort(v, n)   /* sort strings v[0] ... v[n-1] */
char *v[];   /* into increasing order */
int n;
{
 int gap, i, j;   char *temp;
 for (gap = n/2; gap > 0; gap /= 2)
    for (i = gap; i < n; i++)
   for (j = i - gap; j >= 0; j -= gap) {
      if (strcmp(v[j], v[j+gap]) <= 0) break;
      temp = v[j];
      v[j] = v[j+gap];
      v[j+gap] = temp;
   }
}
```

Массивы указателей, указатели указателей

- Так как каждый отдельный элемент массива **v** (имя формального параметра, соответствующего **lineptr**) является указателем на символы, то и **temp** должен быть указателем на символы, чтобы их было можно копировать друг в друга.
- И, в заключение, функция **main():**

```
#define null 0
#define lines 100 /* max lines to be sorted */
main()   /* sort input lines */
 {
  char *lineptr[lines]; /*pointers to text lines */
  int nlines;     /* number of input lines read */
  if ((nlines = readlines(lineptr, lines)) >= 0) {
     sort(lineptr, nlines);
     writelines(lineptr, nlines);
  }
  else     printf("input too big to sort\n");  }
```