

МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННО БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ОБРАЗОВАНИЯ
НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ
ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

Учебный Центр Информационных Технологий «Информатика»



Лабораторная работа № 1
по дисциплине «Информатика и программирование 2 часть»

Выполнил слушатель: Пешков Е.В.

Вариант: 5

Дата сдачи:

Преподаватель: Юшманов А.А.

Новосибирск, 2019г.

1. Цель

Научиться решать задачи с использованием строк.

2. Вариант задания

Вариант задания реализовать в виде функции, использующей для работы со строкой только указатели и операции вида ***p++**, **p++** и т.д. Если функция возвращает строку или ее фрагмент, то это также необходимо сделать через указатель.

Вариант 5

Функция находит в строке пары инвертированных фрагментов (например "123apг" и "гра321") и возвращает указатель на первый. С помощью функции найти все пары.

3. Теория

Указатели и ссылки

Объект, указатель и ссылка

Указатели совместно с адресной арифметикой играют в Си особую роль. Можно сказать, что они определяют лицо языка. Благодаря им Си может считаться одновременно языком высокого и низкого уровня по отношению к памяти.

Если говорить о понятиях указатель, ссылка, объект, то они встречаются не только в языках программирования, но в широком смысле в информационных технологиях. Когда речь идет о доступе к информационным ресурсам, то существуют различные варианты доступа к ним:

копия (значение, объект) – пользователь получает точную копию информационного ресурса в момент доступа к ней (например, копию файла, таблицы базы данных и т.п.). Он может как ему угодно изменять его содержимое, что не отражается на оригинале;

указатель – адресная информация о расположении информационного ресурса, через которую пользователь может обратиться к нему. При изменении содержимого объекта через указатель на него всегда возникает проблема **синхронизации (разделения)** ресурса между несколькими пользователями, имеющими адресную информацию о нем. Синонимом указателя в информационных технологиях является **ссылка**. Иногда она имеет все внешние признаки объекта, например, ярлык файла на рабочем столе, который внешне выглядит как файл, а на самом деле ссылается на файл-оригинал.

В языках программирования термины объект (значение), указатель и ссылка имеют примерно аналогичный смысл, но касаются способов доступа и передачи значений переменных.

- терминология **ссылка, значение** касается фундаментальных свойств переменных в языках программирования. Имя переменной в различных контекстах может восприниматься как ее значение (содержимое памяти), так и ссылка на нее (адрес памяти, указатель)(см. 1.3). Например, при присваивании левая часть рассматривается как ссылка, а правая – как значение (см. 1.4);
- при передаче формальных параметров при вызове процедур (функций) практически во всех языках программирования реализованы способы передачи **по ссылке и по значению**;
- в Паскале и Си определено понятие **указатель** как переменная особого вида, содержащая адрес размещения в памяти другой переменной. Использование указателей позволяет создавать динамические структуры данных, в которых элементы взаимно ссылаются друг на друга;
- и, наконец, в Си существует расширенная интерпретация указателя, именуемая **адресной арифметикой**, которая позволяет интерпретировать значение любого указателя как адрес не отдельной переменной, а памяти в целом, где она размещена.

Указатель в Си

Передавать данные между программами, данные от одной части программы к другой (например, от вызывающей функции к вызываемой) можно двумя способами:

- создавать в каждой точке программы (например, на входе функции) копию тех данных, которые необходимо обрабатывать;

· передавать информацию о том, где в памяти расположены данные. Такая информация, естественно, является более компактной, чем сами данные, и ее условно можно назвать указателем. Получаем «дилетантское» определение указателя: указатель - переменная, содержащая информацию о расположении в памяти другой переменной.

Наряду с указателем в программировании также используется термин **ссылка**. Ссылка – содержанием ссылки также является адресная информация об объекте (переменной), но внешне она выглядит как переменная (синоним оригинала).

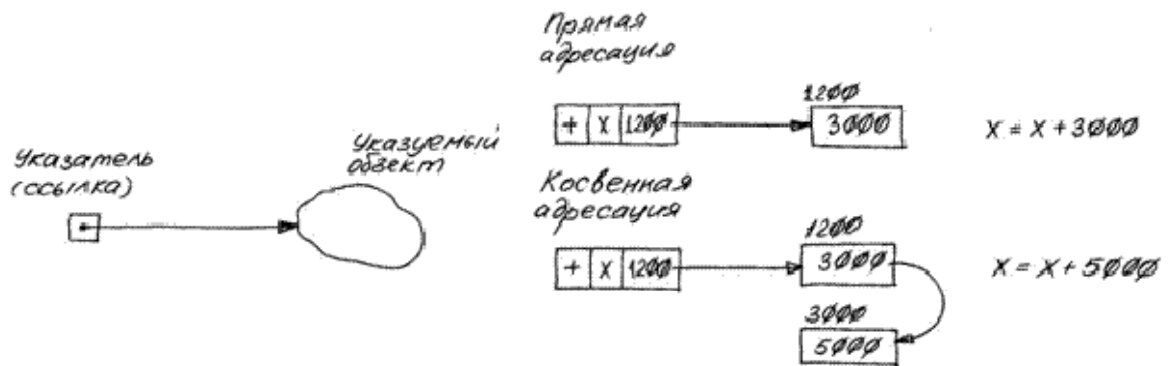


рис. 52-1. Указатель в информационных технологиях и в архитектуре

Указатель как элемент архитектуры компьютера. Указатели занимают особое место среди типов данных, потому что они проецируют на язык программирования ряд важных принципов организации обработки данных в компьютере. Понятие указателя связано с такими понятиями компьютерной архитектуры как адрес, косвенная адресация, организация внутренней (оперативной) памяти. От них мы и будем отталкиваться. **Внутренняя (оперативная) память** компьютера представляет собой упорядоченную последовательность байтов или машинных слов (ячеек памяти), проще говоря - массив. Номер байта или слова памяти, через который оно доступно как из команд компьютера, так и во всех других случаях, называется **адресом**. Если в команде непосредственно содержится адрес памяти, то такой доступ этому слову памяти называется **прямой адресацией**.

Возможен также случай, когда машинное слово содержит адрес другого машинного слова. Тогда доступ к данным во втором машинном слове через первое называется **косвенной адресацией**. Команды косвенной адресации имеются в любом компьютере и являются основой любого регулярного процесса обработки данных. То же самое можно сказать о языке программирования. Даже если в нем отсутствуют указатели, как таковые, работа с массивами базируется на аналогичных способах адресации данных.

В языках программирования имя переменной ассоциируется с адресом области памяти, в которой транслятор размещает ее в процессе трансляции программы. Все операции над обычными переменными преобразуются в команды с прямой адресацией к соответствующим словам памяти.

Таким образом, в компьютерной архитектуре **указатель - переменная, содержимым которой является адрес другой переменной**.

Соответственно, основная операция для указателя - это косвенное обращение по нему к той переменной, адрес которой он содержит. В Си имеется специальная операция * - звездочка, которую называют **косвенным обращением по указателю**. В более широком смысле ее следует понимать как переход от переменной-указателя к той переменной (объекту), на которую он ссылается. В дальнейшем будем пользоваться такими терминами:

- указатель, который содержит адрес переменной, **ссылается** на эту переменную или **назначен** на нее;
- переменная, адрес которой содержится в указателе, называется **указуемой** переменной.

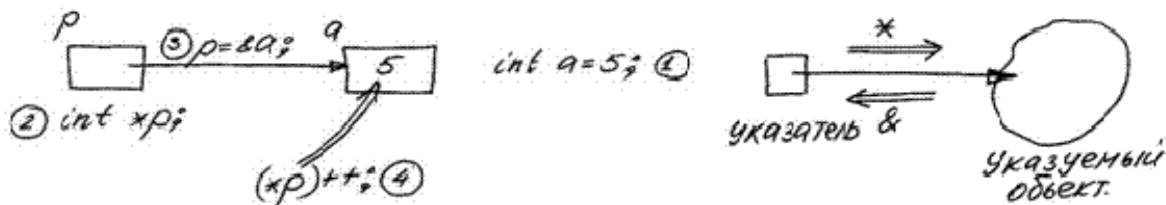


рис. 52-2. Определение указателя и операции над ним

Последовательность действий при работе с указателем включает 3 шага:

1. Определение указуемых переменных и переменной-указателя. Для переменной-указателя это делается особым образом.

```
int    a,x;    // Обычные целые переменные
int    *p;     // Переменная - указатель на другую целую переменную
```

В определении указателя присутствует та же самая операция косвенного обращения по указателю. В соответствии с принципами контекстного определения типа переменной (см. 5.5) эту фразу следует понимать так: переменная **p** при косвенном обращении к ней дает переменную типа **int**. То есть свойство ее – быть указателем, определяется в контексте возможного применения к ней операции *****. Обратите внимание, что в определении присутствует **указуемый тип данных**. Это значит, что указатель может ссылаться не на любые переменные, а только на переменные заданного типа, то есть указатель в Си **типизирован**.

2. Связывание указателя с указуемой переменной. Значением указателя является адрес другой переменной. Следующим шагом указатель должен быть настроен, или **назначен** на переменную, на которую он будет ссылаться.

```
p = &a; // Указатель содержит адрес переменной a
```

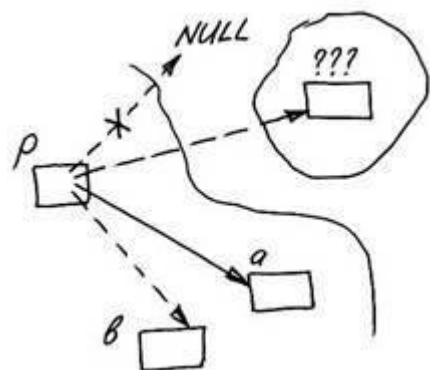
Операция **&** понимается буквально как адрес переменной, стоящей справа от нее. В более широкой интерпретации она «превращает» объект в указатель на него (или производит переход от объекта к указателю на него) и является в этом смысле прямой противоположностью операции *****, которая «превращает» указатель в указуемый объект. То же самое касается типов данных. Если переменная **a** имеет тип **int**, то выражение **&a** имеет тип – указатель на **int** или **int***.

3. И наконец, в любом выражении косвенное обращение по указателю интерпретируется как переход от него к указуемой переменной с выполнением над ней всех далее перечисленных в выражении операций.

```
*p=100;    // Эквивалентно a=100
x = x + *p; // Эквивалентно x=x+a
(*p)++;    // Эквивалентно a++
```

Замечание: при обращении через указатель имя указуемой переменной в выражении отсутствует. Поэтому можно считать, что обращение через указатель производится к «безымянной» переменной, а операцию «*» называются также операцией **разыменования указателя**.

Указатель дает «степень свободы» или универсальности любому алгоритму обработки данных. Действительно, если некоторый фрагмент программы получает данные непосредственно в некоторой переменной, то он может обрабатывать ее и только ее. Если же данные он получает через указатель, то обработка данных (указуемых переменных) может производиться в любой области памяти компьютера (или программы). При этом сам фрагмент может и «не знать», какие данные он обрабатывает, если значение самого указателя переда-



но программе извне.

Адресная арифметика и управление памятью

Способность указателя ссылаться на «отдельно стоящие» переменные не меняет качества языка, поскольку нельзя выйти за рамки множества указуемых переменных, определенных в программе. Такая же концепция указателя принята, например, в Паскале. Но в Си существует еще одна, расширенная интерпретация, позволяющая через указатель работать с массивами и с памятью компьютера на низком (архитектурном) уровне без каких-либо ограничений со стороны транслятора. Это «свобода самовыражения» обеспечивается одной дополнительной операцией **адресной арифметики**. Но сначала определим свойства указателя в соответствии с расширенной интерпретацией.

Любой указатель в Си ссылается на неограниченную в обе стороны область памяти (массив), заполненную переменными указуемого типа с индексацией элементов относительно текущего положения указателя.

Такие свойства указателя обеспечиваются **адресной арифметикой**, которая базируется на нестандартной интерпретации операции **указатель+целое** и других, производных от нее операциях:

- любой указатель потенциально ссылается на неограниченную в обе стороны область памяти, заполненную переменными указуемого типа;
- переменные в области нумеруются от текущей указуемой переменной, которая получает относительный номер 0. Переменные в направлении возрастания адресов памяти нумеруются положительными значениями 1,2,3..., убывания - отрицательными -1,-2..;
- результатом операции **указатель+i** является адрес i-ой переменной (значение указателя на i-ую переменную) в этой области относительно текущего положения указателя.

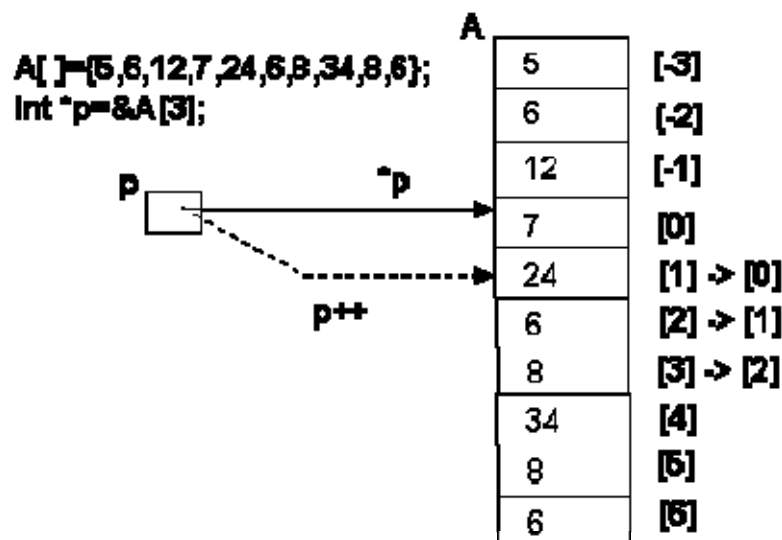


рис. 52-3. Адресная арифметика

	Смысл
*p	Значение указуемой переменной
p+i	Указатель на i-ю переменную после указуемой
p-i	Указатель на i-ю переменную перед указуемой

<code>*(p+i)</code>	Значение <i>i</i> -й переменной после указуемой
<code>p[i]</code>	Значение <i>i</i> -й переменной после указуемой
<code>p++</code>	Переместить указатель на следующую переменную
<code>p--</code>	Переместить указатель на предыдущую переменную
<code>p+=i</code>	Переместить указатель на <i>i</i> переменных вперед
<code>p-=i</code>	Переместить указатель на <i>i</i> переменных назад
<code>*p++</code>	Получить значение указуемой переменной и переместить указатель к следующей
<code>*(--p)</code>	Переместить указатель к переменной, предшествующей указуемой, и получить ее значение
<code>p+1</code>	Указатель на свободную память вслед за указуемой переменной

В операциях адресной арифметики транслятором автоматически учитывается размер указуемых переменных, то есть `+i` понимается не как смещение на *i* байтов или слов, а как смещение на *i* указуемых переменных. Другая важная особенность: при перемещении указателя нумерация переменных в памяти остается относительной и всегда производится от текущей указуемой переменной.

Указатели и массивы. Нетрудно заметить, что указатель в Си имеет много общего с массивом. Наоборот, труднее сформулировать, чем они отличаются друг от друга. Действительно, разница лежит не в принципе работы с указуемыми переменными, а в способе назначения указателя и массива на ту память, с которой они работают. Образно говоря, указателю соответствует массив, «не привязанный» к конкретной памяти, а массиву соответствует указатель, постоянно назначенный на выделенную транслятором область памяти. Это положение вещей поддерживается еще одним правилом: имя массива во всех выражениях воспринимается как указатель на его начало, то есть имя массива **A** эквивалентно выражению `&A[0]` и имеет тип «указатель на тип данных элементов массива». Таким образом, различие между указателем и массивом аналогично различию между переменной и константой: указатель - это ссылочная переменная, а имя массива - ссылочная константа, привязанная к конкретному адресу памяти.

Если **МАССИВ=ПАМЯТЬ+УКАЗАТЕЛЬ** (начальный адрес), то **УКАЗАТЕЛЬ=МАССИВ-ПАМЯТЬ**, т.е. указатель это «массив без памяти», «свободно перемещающийся по памяти» массив.

Массив	Указатель	Различия и сходства
<code>int A[20]</code>	<code>int *p</code>	
<code>A</code>	<code>p</code>	Оба интерпретируются как указатели и оба имеют тип <code>int*</code>
<code>---</code>	<code>p=&A[3]</code>	Указатель требует настройки «на память»
<code>A[i]</code> <code>&A[i]</code> <code>A+i</code> <code>*(A+i)</code>	<code>p[i]</code> <code>&p[i]</code> <code>p+i</code> <code>*(p+i)</code>	Работа с областью памяти как с обычным массивом, так и через указатель полностью идентична вплоть до синтаксиса
<code>----</code>	<code>p++</code> <code>*p++</code> <code>p+=i</code>	Указатель может перемещаться по памяти относительно своего текущего положения

Указатели и многомерные массивы. Двумерный массив реализован как «массив массивов» - одномерный массив с количеством элементов, соответствующих первому индексу, причем каждый элемент представляет собой массив элементов базового типа с количеством, соответствующим второму индексу. Например, `char A[20][80]` определяет массив из 20 массивов по 80 символов в каждом и никак иначе.

Идентификатор массива без скобок интерпретируется как адрес нулевого элемента нулевой строки, или указатель на базовый тип данных. В нашем примере идентификатору **A** будет соответствовать выражение **&A[0][0]** с типом **char***.

Имя двумерного массива с единственным индексом интерпретируется как начальный адрес соответствующего внутреннего одномерного массива. **A[i]** понимается как **&A[i][0]**, то есть начальный адрес **i-го** массива символов.

От такого многообразия возможностей работы с указателями нетрудно прийти в замешательство: как вообще с ними работать, кто за что отвечает? Действительно, при работе с указателями легко выйти «за рамки дозволенного», т.е. определенных самим же программистом структур данных. Поэтому попробуем еще раз обсудить принципиальные моменты адресной арифметики.

Границы памяти, адресуемой указателем. Если любой указатель ссылается на неограниченную область памяти, то возникают резонные вопросы: где границы этой памяти, кто и как их определяет, кто и как контролирует нарушение этих границ указателем. Ответ на него неутешителен для начинающего программиста: транслятор принципиально исключает такой контроль как в процессе трансляции программы, так и в процессе ее выполнения. Он не помещает в генерируемый программный код каких-либо дополнительных команд, которые могли бы это сделать. И дело здесь прежде всего в самой концепции языка Си: не включать в программный код ничего, не предусмотренного самой программой, и не вносить ограничений в возможности работы с данными. Следовательно, ответственность ложится целиком на работающую программу (точнее, на программиста, который ее написал).

На что ссылается указатель? Синтаксис языка в операциях с указателями не позволяет различить в конкретной точке программы, что подразумевается под этим указателем - указатель на отдельную переменную, массив (начало, середину, конец...), какова размерность массива и т.д.. Все эти вопросы целиком находятся в ведении работающей программы. Все же даже поверхностный взгляд на программу позволяет сказать, с чем же работает указатель – с отдельной переменной или массивом.

- наличие операции инкремента или индексации говорит о работе указателя с памятью (массивом);
- использование исключительно операции косвенного обращения по указателю свидетельствует о работе с отдельной переменной.

Типичные ошибки при работе с указателями. Основная ошибка, которая периодически возникает даже у опытных программистов – указатель ассоциируется с адресуемой им памятью. Память – это прежде всего ресурс, а указатель – ссылка на него. Здесь же отметим наиболее грубые ошибки:

- неинициализированный указатель. После определения указатель ссылается «в никуда», тем не менее программист работает через него с переменной или массивом, записывая данные по случайным адресам;
- несколько указателей, ссылающихся на общий массив – это все-таки один массив, а не несколько. Если программа работает с несколькими массивами, то они должны либо создаваться динамически, либо браться из двумерного массива;
- выход указателя за границы памяти. Например, конец строки отмечается символом `'\0'`, начало же формально соответствует начальному положению указателя. Если в процессе работы со строкой требуется возвращение на ее начало, то начальный указатель необходимо запоминать, либо дополнительно отсчитывать символы.

Другие операции над указателями

В процессе определения указателей мы рассмотрели основные операции над ними:

- операция присваивания указателей одного типа. Назначение указателю адреса переменной **p=&a** есть один из вариантов такой операции;
- операция косвенного обращения по указателю (разыменования указателя);
- операция адресной арифметики «указатель+целое» и все производные от нее.

Кроме того, имеется еще ряд операций, понимание которых не выходит за рамки уже имеющейся интерпретации указателя.

Сравнение указателей на равенство. Равенство указателей однозначно понимается как совпадение адресов, то есть назначение их на одну и ту же область памяти (переменную).

Пустой указатель (NULL-указатель). Среди множества адресов выделяется такой, который не может быть использован в правильно работающей программе для размещения данных. Это значение адреса называется **NULL-указателем** или «пустым» указателем. Считается, что указатель с таким значением не является корректным (указывает «в никуда»). Обычно такое значение определяется в стандартной библиотеке ввода-вывода в виде **#define NULL 0**.

Значение NULL может быть присвоено любому указателю. Если указатель по логике работы программы может иметь такое значение, то перед косвенным обращением по нему его нужно проверять на достоверность:

```
int    *p,a;
if (...) p=NULL; else p=&a; ...
if (p !=NULL) *p = 5;  ...
```

Сравнение указателей на «больше-меньше»: при сравнении указателей производится сравнение соответствующих адресов как беззнаковых переменных. Если оба указателя ссылаются на элементы одного и того же массива, тогда соотношение «больше-меньше» следует понимать как «ближе-дальше» к началу массива:

```
//--- Симметричная перестановка символов строки
void F(char *p){
for (char *q=p; *q!=0; q++);           // Указатель q до конца строки
for (q--; q>p; p++, q--)                // Пока p левее q
    { char c; c=*p; *p=*q; *q=c; }     // 3 стакана над переменными под указателями
}
```

Разность значений указателей. В случае, когда указатели ссылаются на один и тот же массив, их разность понимается как «расстояние между ними», выраженную в количестве указываемых переменных.

Преобразование типа указателя. Отдельная операция преобразования, связанная с изменением типа указываемых элементов при сохранении значения указателя (адреса), используется при работе с памятью на низком (архитектурном) уровне (см. 9.2). Отдельный разговор о преобразовании типов указателей при наследовании (см. 11.3, 11.4). Сюда же относится **преобразование вида «целое-указатель»**.

Указатель типа void*. Если фрагмент программы «не должен знать» или не имеет достаточной информации о структуре данных в адресуемой области памяти, если указатель во время работы программы ссылается на данные различных типов, то используется указатель на неопределенный (пустой) тип **void**. Указатель понимается как адрес памяти как таковой, с неопределенной организацией и неизвестной размерностью указываемой переменной. Его можно присваивать, передавать в качестве параметра и результата функции, менять тип указателя, но операции косвенного обращения и адресной арифметики с ним недопустимы.

```
extern int fread(void *, int, int, FILE *);
int    A[20];
fread(A, sizeof(int), 20, fd);
```

Функция **fread** выполняет чтение из двоичного файла **n** записей длиной по **m** байтов, при этом структура записи для функции неизвестна. Поэтому начальный адрес области памяти передается формальным параметром типа **void***. При подстановке фактического параметра **A** типа **int*** производится неявное преобразование его к типу **void***.


```
extern void *malloc(int);
int *p = (int*)malloc(sizeof(int)*20);    // Явное преобразование void* к int*
```

Функция **malloc** возвращает адрес зарезервированной области динамической памяти в виде указателя **void***. Это означает, что функцией выделяется память как таковая, безотносительно к размещаемым в ней переменным. Вызывающая функция явно преобразует тип указателя **void*** в требуемый тип **int*** для работы с этой областью как с массивом целых переменных.

Вывод: преобразование указателя **void*** к любому другому типу указателя соответствует «смене точки зрения» программы на адресуемую память от «данные вообще» к «конкретные данные» и наоборот (см. 9.2) и должно быть сделано явно. Преобразование указателя к типу **void*** не требует явного подтверждения.

Указатель как формальный параметр и результат функции

В Си при передаче параметров в функцию по умолчанию используется **передача по значению (by value)**. Формальные параметры представляют собой аналог собственных локальных переменных функции, которым в момент вызова присваиваются значения фактических параметров. Формальные параметры, представляя собой копии, могут как угодно изменяться - это не затрагивает соответствующих фактических параметров.

Если же фактический параметр должен быть изменен, то формальный параметр можно определить как явный указатель. Тогда фактический параметр должен быть явно передан в виде указателя на ту переменную (с использованием операции **&**).

```
void inc(int *p)
{ (*p)++; }           // аналог вызова: pi = &a
void main()
{ int a;
  inc(&a); }           // *(pi)++ эквивалентно a++
```

В Си имеется единственное исключение: формальный параметр - массив передается в виде неявного указателя на его начало, то есть **по ссылке**. С помощью адресной арифметики это также можно сделать явно с использованием указателя на его начало.

```
int sum(int A[],int n)    // Исходная программа
{ int s,i;
  for (i=s=0; i<n; i++) s+= A[i];
  return s;}
int sum(int *p, int n)    // Эквивалент с указателем
{ int s,i;
  for (i=s=0; i<n; i++) s+= p[i];
  return s; }
```

```
int x,B[10]={1,4,3,6,3,7,2,5,23,6};
void main()
{ x = sum(B,10); }       // аналог вызова: p = B, n = 10
```

В вызове фигурирует идентификатор массива, который интерпретируется как указатель на начало. Поэтому типы формального и фактического параметров совпадают. Совпадают также оба варианта функций вплоть до генерируемого кода.

Указатель - результат функции. Функция в качестве результата может возвращать указатель. Формальная схема функции обязательно включает в себя:

- определение типа результата в заголовке функции как указателя. Это обеспечивается добавлением пресловутой ***** перед именем функции - **int *F(...;**

- оператор **return** возвращает объект (переменную или выражение), являющееся по своей природе (типу данных) указателем. Для этого можно использовать локальную переменную - указатель.

Содержательная сторона проблемы состоит в том, что функция либо **выбирает** один из известных ей объектов (переменных), либо **создает** их в процессе своего выполнения (динамические переменные), возвращая в том и другом случае указатель на нее. Для выбора у нее не так уж много возможностей. Это могут быть:

- глобальные переменные программы;
- формальные параметры, если они являются массивами, указателями или ссылками, то есть «за ними стоят» другие переменные.

Функция не может вернуть указатель на локальную переменную или формальный параметр-значение, поскольку они разрушаются при выходе из функции. Это приводит к ошибке времени выполнения, не обнаруживаемой транслятором.

4. Анализ задачи и алгоритм

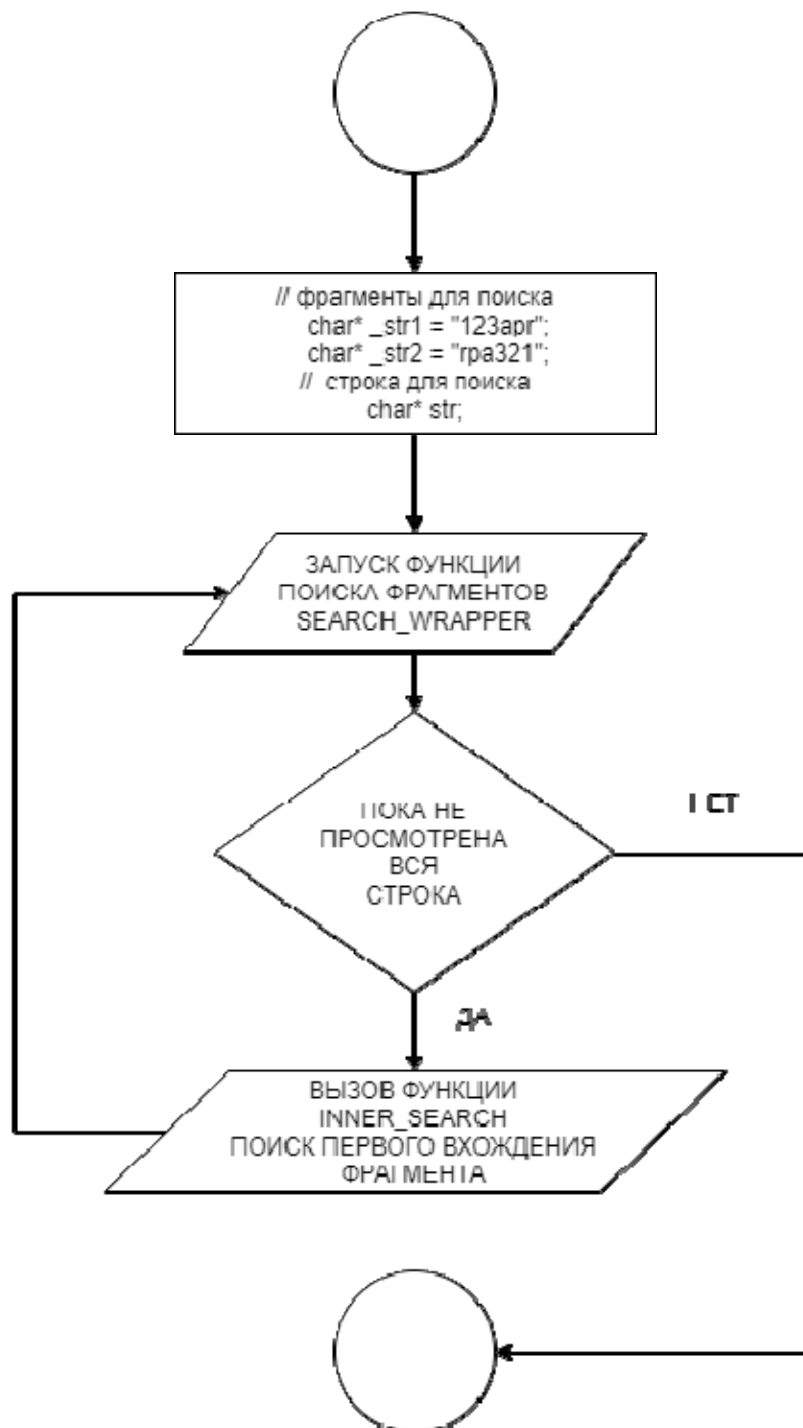
1) Анализ задачи

Входные данные: строка символов содержащая буквы и цифры.

Результат: вывод строки начиная с позиции найденных пар инвертированных фрагментов.

Метод решения: находим пары инвертированных фрагментов и выводим строку начиная с адреса возвращенного из функции поиска.

2) Алгоритм решения задачи



5. Описание программной реализации

1) Используемые переменные

`char*` `str` – исходная строка для поиска
`char*` `_str1` – фрагмент для поиска
`char*` `_str2` – инвертированный фрагмент для поиска

2) Используемые функции

Функция **`char* inner_search`** (`char* str`, `char* str1`, `char* str2`)

Аргументы функции:

`char* str` – исходная строка для поиска.
`char* _str1` – фрагмент для поиска.
`char* _str2` – инвертированный фрагмент для поиска.

Возвращаемый результат: указатель на начало найденного фрагмента.

Принцип работы: находит первое вхождение пары фрагментов строки.

Функция **`void search_wrapper`** (`char* str`, `char* _str1`, `char* _str2`)

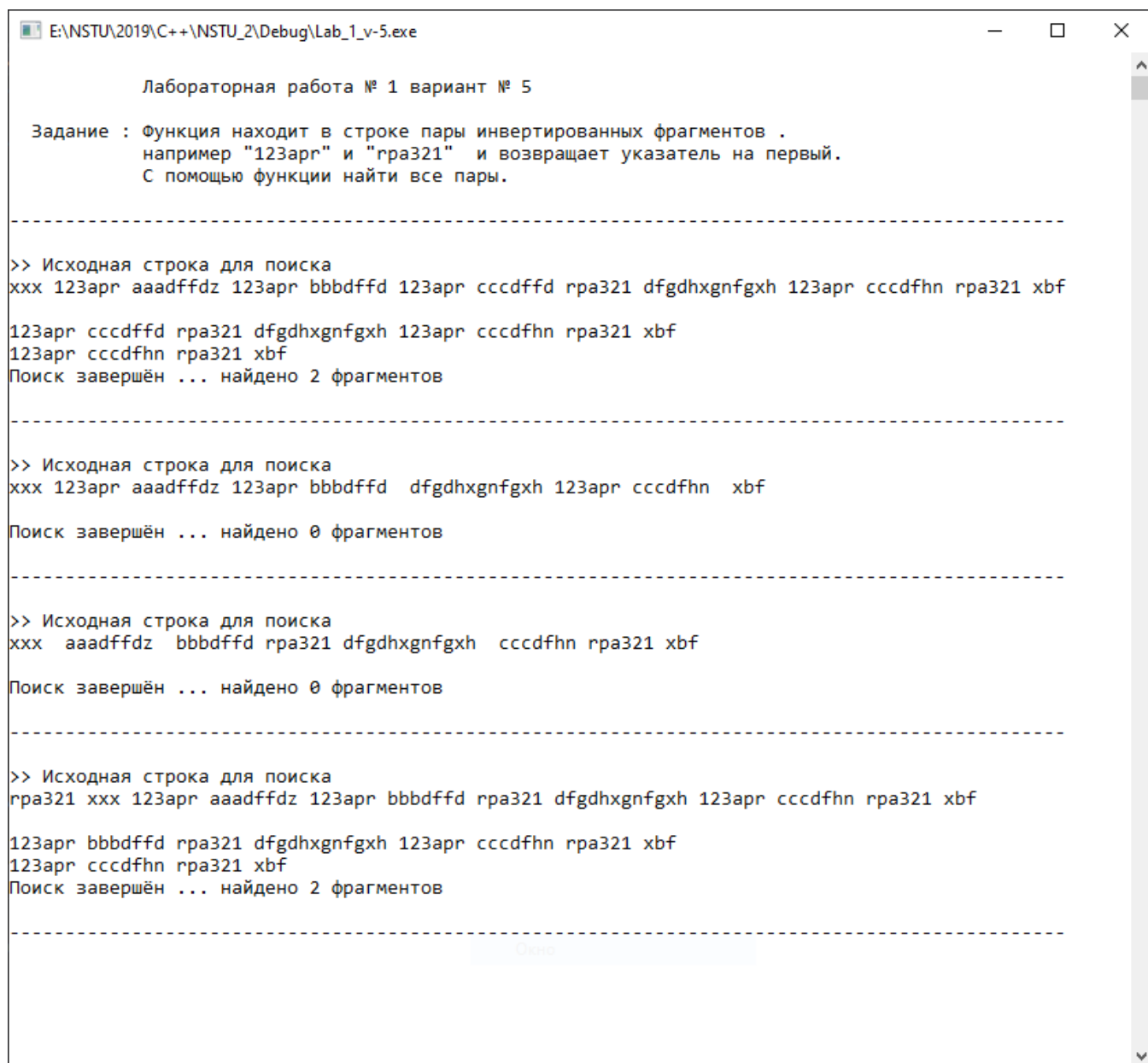
Аргументы функции:

`char* str` – исходная строка для поиска.
`char* _str1` – фрагмент для поиска.
`char* _str2` – инвертированный фрагмент для поиска.

Возвращаемый результат: ничего не возвращает.

Принцип работы: является обёрткой для поиска пар фрагментов.

6. Пример работы программы



```
Лабораторная работа № 1 вариант № 5

Задание : Функция находит в строке пары инвертированных фрагментов .
           например "123apr" и "rpa321" и возвращает указатель на первый.
           С помощью функции найти все пары.

-----

>> Исходная строка для поиска
xxx 123apr aaadffdz 123apr bbdbffd 123apr cccdfhd rpa321 dfgdhxgnfgxh 123apr cccdfhn rpa321 xbf

123apr cccdfhd rpa321 dfgdhxgnfgxh 123apr cccdfhn rpa321 xbf
123apr cccdfhn rpa321 xbf
Поиск завершён ... найдено 2 фрагментов

-----

>> Исходная строка для поиска
xxx 123apr aaadffdz 123apr bbdbffd dfgdhxgnfgxh 123apr cccdfhn xbf

Поиск завершён ... найдено 0 фрагментов

-----

>> Исходная строка для поиска
xxx aaadffdz bbdbffd rpa321 dfgdhxgnfgxh cccdfhn rpa321 xbf

Поиск завершён ... найдено 0 фрагментов

-----

>> Исходная строка для поиска
rpa321 xxx 123apr aaadffdz 123apr bbdbffd rpa321 dfgdhxgnfgxh 123apr cccdfhn rpa321 xbf

123apr bbdbffd rpa321 dfgdhxgnfgxh 123apr cccdfhn rpa321 xbf
123apr cccdfhn rpa321 xbf
Поиск завершён ... найдено 2 фрагментов

-----

Окно
```

7. Выводы

В ходе выполнения лабораторной работы были изучены строки и указатели и применены в решении задачи.

Приложение. Текст программы

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

char* inner_search(char*, char*, char*);
void search_wrapper(char*, char*, char*);

int main()
{
    setlocale(LC_ALL, "RU-ru");
    printf("-----\n\n");
    printf("Лабораторная работа № 1 вариант № 5\n\n");
    printf("Задание : Функция находит в строке пары инвертированных фрагментов .\n");
    printf("например \"123apr\" и \"rpa321\" и возвращает указатель на пер-
вый.\n");
    printf("С помощью функции найти все пары.\n\n");
    printf("-----\n\n");
    // тестовые строки для поиска
    char* str = "xxx 123apr aaadffdz 123apr bbdbffd 123apr ccdfdd rpa321 dfgdhxgnfgxh 123apr
cccdfhn rpa321 xbf";
    char* str1 = "xxx 123apr aaadffdz 123apr bbdbffd dfgdhxgnfgxh 123apr ccdfhn xbf";
    char* str2 = "xxx aaadffdz bbdbffd rpa321 dfgdhxgnfgxh ccdfhn rpa321 xbf";
    char* str3 = "rpa321 xxx 123apr aaadffdz 123apr bbdbffd rpa321 dfgdhxgnfgxh 123apr ccdfhn
rpa321 xbf";

    // фрагменты для поиска
    char* _str1 = "123apr";
    char* _str2 = "rpa321";

    // тестирование работы функции
    search_wrapper(str, _str1, _str2);
    search_wrapper(str1, _str1, _str2);
    search_wrapper(str2, _str1, _str2);
    search_wrapper(str3, _str1, _str2);
    getchar();

    return 0;
}

void search_wrapper(char* str, char* _str1, char* _str2)
{
    printf(">> Исходная строка для поиска \n%s\n\n", str);

    int count = 0;
    char* start_next_point = str;

    while (strstr(start_next_point, _str1) && strstr(start_next_point, _str2))
    {
        if (strstr(start_next_point, _str2) < strstr(start_next_point, _str1))
        {
            start_next_point = strstr(start_next_point, _str1);
        }
        inner_search(start_next_point, _str1, _str2);
        printf("%s\n", inner_search(start_next_point, _str1, _str2));
        count++;
        start_next_point = inner_search(start_next_point, _str1, _str2) + strlen(_str1);
        inner_search(start_next_point, _str1, _str2);
    }
    printf("Поиск завершён ... найдено %d фрагментов \n\n", count);
}
```

```

        printf("-----\n\n");
    }

char* inner_search(char* str, char* str1, char* str2)
{
    // указатель на фрагмент
    char* pointer1 = (strstr(str, str1));
    // указатель на инвертированный фрагмент
    char* pointer2 = (strstr(str, str2));
    // указатель на точку продолжения поиска
    char* start_next_point = pointer1 + strlen(str1);

    while (pointer1 != NULL && strstr(start_next_point, str1) && (strstr(start_next_point, str1)
< pointer2))
    {
        pointer1 = strstr(start_next_point, str1);
        start_next_point = pointer1 + strlen(str1);
    }
    return pointer1;
}

```